Design and Modelling of a Parallel Data Server for Telecom Applications

Mikael Ronström

Ericsson Utveckling AB

Acknowledgements

The reason that this thesis is written is the inspiration from the Lord and it would not have been possible to conclude without a constant flow of inspiration from the Lord. The inspired teachings of the Church of Jesus Christ of Latter-Day Saints have helped me find a balance in life between family, church activities and research work. The constant support of my wife Marianne and her love made it worthwhile to perform this work.

I am grateful to Ericsson Telecom AB and Ericsson Utveckling AB and especially then to my department manager Göran Ahlforn who also had the vision of the future importance of databases in the telecom network as well as to my section managers Martin Hänström, Jan Gunnelin and Hans Alberg and my department manager Klaus Wildling, for the time I was granted to do this work. Also to NUTEK for the economic support that made this possible. A large part of this work was performed within the RACE MONET and RACE MONETII projects during 1992-1995.

I am grateful to my advisor Professor Tore Risch at the Department of Computer and Information Science at Linköping University and also to Professor Anders Martin-Löf at the Mathematical Statistics-institution at the University of Stockholm.

Others who have provided help and inspiration in this work that deserve mentioning are Svein-Olaf Hvasshovd, Oddvar Risnes and Øystein Torbjørnsen at TeleNor Research in Trondheim, Lars Gauffin at the Royal Institute of Technology in Stockholm and Mats Olstedt and Thomas Muth at Ericsson. The different DBMS design teams at Ericsson have provided ideas that have been helpful.

The master thesis students I have worked with as part of this research, Micael Larsson, Shahid Malik, Daniel Johansson, Houshang Shirvanpour, Massoud Mowla, Björn Avnborg, Patrik Johansson, Arne Eriksson, Joakim Larsson, Ataullah Dabaghi, Gustaf Andersson, and Christer Buskas are also worthy of mention. Also Daniel Högberg and Andreas Hersen have contributed to parts of this research. David Jakobsson made valuable contribution to the structure of the tuple storage. Also I have had help of a number of practising students in 1995 and 1996.

Finally an acknowledgement to my mother who helped me take my first steps into this life and my father who taught me to reach for the stars and never be satisfied until I reach them. I am also grateful to my sister, Ann-Charlotte Oden who made the illustration on the cover, and to all of my family both living and dead to whom I dedicate this work.

Abstract

Telecom databases are databases used in the operation of the telecom network and as parts of applications in the telecom network. The first telecom databases were Service Control Points (SCP) in Intelligent Networks. These provided mostly number translations for various services, such as Freephone. Also databases that keep track of the mobile phones (Home Location Registers, HLR) for mobile telecommunications were early starters. SCPs and HLRs are now becoming the platforms for service execution of telecommunication services. Other telecom databases are used for management of the network, especially for real-time charging information. Many information servers, such as Web Servers, Cache Servers, Mail Servers, File Servers are also becoming part of the telecom databases.

These servers have in common that they all have to answer massive amounts of rather simple queries, that they have to be very reliable, and that they have requirements on short response times. Some of them also need large storage and some needs to send large amounts of data to the users.

Given the requirements of telecom applications an architecture of a Parallel Data Server has been developed. This architecture contains new ideas on a replication architecture, two-phase commit protocols, and an extension of the nWAL concept writing into two or more main memories instead of writing to disk at commit. The two-phase commit protocol has been integrated with a protocol that supports network redundancy (replication between clusters).

Some ideas are also described on linear hashing and B-trees, and a data structure for tuple storage that provides efficient logging. It is shown how the data server can handle all types of reconfiguration and recovery activities with the system on-line. Finally advanced support of on-line schema change has been developed. This includes support of splitting and merging tables without any service interruption.

Together these ideas represent an architecture of a Parallel Data Server that provides non-stop operation. The distribution is transparent to the application and this will be important when designing load control algorithms of the applications using the data server. This Parallel Data Server opens up a new usage area for databases. Telecom applications have traditionally been seen as an area of proprietary solutions. Given the achieved performance, reliability and response time of the data server presented in this thesis it should be possible to use databases in many new telecom applications.

I: Introduction

1 Introduction

The aim of this research is to study the DBMS of telecom databases. Many times in the text of the thesis, this DBMS will be referred to as the telecom database.

The definition of telecom databases in this thesis is: Data storage nodes used for operation of the telecom network or used in applications that are part of the telecom network.

1.1 Research Method

When the research for this thesis started it had a very general objective: Find ways to build efficient database servers for current and future telecom applications. It should be efficient at storing, searching and reading data. It was not decided where the focus of the research would be.

To find a research focus two work items were started. The first was an application study; this was beneficial to understand the requirements of telecom databases. The second was a technical study to understand where the current systems have their major bottlenecks.

As a result of these studies the actual research work was performed. This started with development of the essential algorithms needed in databases with very high reliability. During the development of these algorithms a platform was also developed. After developing most of the essential algorithms a system architecture was developed.

Currently a platform is being developed based on the system architecture and most of the algorithms developed in this thesis.

1.2 Application Study

A large part of the application study was performed by participating in a RACE project called MONET. This project had the objective to perform pre-standard research on how to build third generation systems (UMTS). The main use of databases in the project was in Service Control Points that handles services, locations and authentication of users.

Studies were also performed on future real-time charging applications and future Information services. In this area it was more difficult to acquire estimates on the use of databases. The reason is the uncertainty of what future applications will be. The idea that was used then was to perform “guestimates” based on that use of future information services will reflect current use of mail, TV, newspapers, radio and so forth.

It was evident during the application study that requirements on very high reliability and response times in the order of 1-20 ms was needed.

The application study was also converted into a set of benchmarks that will reflect a number of important telecom applications.

1.3 Requirements on Architecture of Telecom Databases

From the study of various applications of telecom databases we can draw certain conclusions about the requirements on a telecom database.

1.3.1 Reliability

The availability class of the telecom databases should be 6. This means that the unavailability must be less than 30 seconds per year. This means that no planned downtime of the system is allowed.

1.3.2 Performance

Typical future performance requirements on telecom databases are 10-20,000 requests per second. Most of these requests are either write or read a single record. Many applications also need network redundancy to handle earthquakes and other catastrophes.

1.3.3 Delays

A short delay of read operations is particularly crucial, but also write operations must have a short delays. Typical future delay requirements are 5-15 ms [Ronstrom93].

The impact of the delay requirement is that disk writes can not be used in the transactions. This means that the availability requirements must be handled in some other way. In this thesis we will study a method where several main memories are used to ensure high availability. The probability of two processor nodes failing at the same time is very small. General power failures can be handled by using battery backups to ensure that log information can be sent to disk before the battery runs out of energy.

1.3.4 Storage Types

Most of the data studied can be kept in main memory. There are also applications with great needs of larger data sets and these will need disks and might even need some form of data juke-box. Disk storage is needed for the documents in the news-on-demand application, the email bodies with attached files, and the set of scanned objects in the genealogy application. These are all examples of BLOBs. It is furthermore necessary to store event records on disk. This is actually the only exception in the applications of small tuples that need to be stored on disk. If BLOBs are attributes then obviously it must be possible to separate attributes that are stored in main memory from attributes that are stored on disk in a table. Another solution could be that BLOBs are stored in a special BLOB table that is stored on disk.

Main memory data can also be found in the email server, the genealogy application and the news-on-demand application. In these cases they are used to store control structures, indexes and descriptive information about BLOBs.

A requirement in the charging database is the possibility to insert a tuple into main memory and then specifying that it should move to disk at some later time, either by a timer or by specific request. This could also be achieved by an insert into a main memory table, followed by a delete and an insert into a disk table.

1.3.5 Object Orientation

From the charging database and genealogy database we can draw conclusions that many of the requirements on object-oriented databases are very relevant. We need the possibility to perform queries on tables that contains several types of objects based on a common base class. We also need methods as first class attributes. The genealogy database needs many of the complex data structures provided by the object-oriented model. The primary keys of a table can be both an application key, like a telephone number, or an object identifier.

1.3.6 Triggers

Almost all applications have a need of being informed when relevant events take place in the database. This means that a good support of a trigger mechanism should be included. To simplify application development there should also be good support for referential integrity constraints that should be automatically maintained.

1.3.7 Indexing

A good support of indexes is needed, especially indexes for various telecom addresses, http-addresses, file names and so on. Both primary indexes and secondary indexes are needed.

1.3.8 Persistent Locks

From another study of an application of a telecom database, a Cache Server for WWW[JOH97], some requirements for persistent locks were found. The problem is that sometimes the application needs to lock tuples for longer time than the transaction. This means that the database must be able to handle locks on tuples which are not involved in transactions. These locks must also be persistent over crashes.

1.3.9 Counters

The Cache Server also needs to maintain persistent counters and this is most likely also needed by many of the other applications. It is important that these counters are not handled by the normal concurrency control protocol. This would create a hotspot which would degrade performance.

1.3.10 Complex Queries

The support of complex queries is not found to be among the most important aspects. The charging database does, however, require some of this. The charging database has the benefit of not being continously updated, thereby concurrency control for large queries is not a problem.

1.3.11 Create-Only Tables

It can also be noted that many disk-based applications are create-only applications. Deletes can occur but then usually whole tables or large parts of a table are deleted at a time. This can be used to simplify recovery and transaction handling for these applications.

1.4 Major Features of Parallel Telecom Data Server

The requirements lead to the following significant features of a Telecom Database:

1) Scalability

Telecommunication systems are sometimes very small (e.g. 10-300 users) and sometimes very large (several million users). A database for telecommunication applications must therefore be scalable both upwards and downwards. Scalability downwards is achieved by portability to various platforms. This is not covered in this thesis but is the aim of the distributed and real-time run-time system described in

[Ronstrom97], [Ronstrom97a]. Since the data server described in this thesis can be implemented on a portable run-time system, the data server is as portable as the run-time system is.

2) Security against different kinds of catastrophes

In times of catastrophes such as earthquakes, storms and so forth it is actually even more important that the telecommunication network does not fail. Therefore a database system for telecommunication applications should be secure against different kinds of catastrophes.

3) Very high availability

A telecommunication network must be always available. A lost telephone call can sometimes be a very serious matter. Databases are often centralised and it is even more crucial for them to have a very high level of availability.

4) Very high reliability

Transactions which have been committed must not be lost. A transaction could be very critical to the operator and must not be lost. Also users of the telecom network are not likely to accept that the system forgets transactions.

5) Load Regulation and Overload Control

All telecom networks are built with the assumption that not all subscribers are active at the same time. This means that overload situations can always occur. It is important that databases also have a scheme to handle overload situations. This is not covered in this thesis and is an item for future work.

6) Openness

This will not be discussed in this thesis but is certainly a very important requirement. It is necessary to be able to reuse code, tools, platforms, and so forth developed by others to enable quick development of new services.

1.5 Technical Study

The technical study consisted of finding the bottlenecks in the computer and communication systems, and to find solutions of how to overcome them. Bottlenecks may occur anywhere in the design of a system. They can occur in some software parts where code can be unoptimised and data usage can be such that utilisation of cache memories is bad. There can even be problems with instruction caches where the instructions of the DBMS do not fit well in the cache memories. The operating system can also become a bottleneck if it is used in the wrong way. E.g. the overhead to switch processes, disk I/O handling and communication handling can easily become the main source of performance bottlenecks. Actually in a DBMS executing TPC-B and TPC-C benchmarks, it has been shown that they spend 40-50% of their time executing in the operating system [FLAN96]. Yet other bottlenecks can occur in the electronic systems packaging, which could be due to insufficient number of I/O pins in a package, or because of too high delays in communications on a Printed Wiring Board.

To find solutions to these problems it is necessary to study software optimisations, data structures using cache memories in a careful manner and ways to achieve better usage of instruction caches. How to avoid excessive usage of the operating system is another important feature.

All these parts were studied initially in this research. The conclusion was that the major bottlenecks are found in:

1) Excessive usage of the operating system for communication between processes in the operating system.

2) Excessive usage of communication protocols implemented in operating system software.

3) Excessive usage of disk I/O implemented in operating system software.

4) Bad behaviour of instruction caches[CVET96].

5) Bad behaviour of data caches[CVET96].

6) Excessive use of disk writes in update transactions.

An interesting observation is that most of the major problems were not found in the DBMSs. The major bottlenecks were found in the heavy use of operating system services. These operating systems were not designed with databases as the prime application. This is not a new result, it was also found in [Stonebraker81].

A result of this observation was that a platform development was started. This is not reported in this thesis [Ronstrom97a], [Ronstrom97]. The main idea in this platform development is to build a platform for the database such that the database could be developed using a modular structure without interfering with the operating system any more than absolutely needed. This platform development solves the problems of items 1), 2), and 4), while 3) is not attacked. There are currently no products that provide the possibility to handle 3).

Bottleneck 5) have been covered by development of two new index structures and development of a data structure for tuple storage which is reported in this thesis. Bottleneck 2) and 6) have been covered by development of a new two-phase commit protocol. The solution to bottleneck 2) has inspired the development of the stand-by replica which is reported in this thesis.

The requirements on very high reliability using DBMSs in the application study found a number of open research areas. The reliability support in the DBMS was an important research item which is covered in this thesis.

The requirement on response time found in the application study have also been handled by the platform development.

The hardware development and in particular the processor development is progressing with rapid speed. This was not a bottleneck and thus was not included in the research. This area was mainly studied to find parameters of future performance modelling of telecom databases. Also the hardware development have an effect on the software design and therefore it is also important to study this area.

1.6 Summary of Contributions

The major results are the following:

- A new data structure in pages that makes it possible to avoid logging the logical UNDO parts without sacrificing any flexibility in writing pages to disk. It uses the fact that an action-consistent checkpoint is produced and thus a copy of changed parts can be saved in the tuple storage as UNDO information.
- For reliability even in extreme situations a two-level replication scheme was developed. The first level of replication is between machines that are located in physical vicinity of each other. The second level is used between systems that are located at a large distance from each other. [KING91] provided much input to this work. The major new work here is to integrate the algorithms in [KING91] into a new two-phase commit algorithm in an efficient manner. The proposed handling of replication and consistency within a system and between systems is very flexible and can sometimes even differ on an attribute level.
- Another example which is covered in this thesis is the algorithm used for on-line schema changes. This algorithm makes it possible

to split and merge tables both horisontally and vertically without stopping the system. These changes are also maintained with two levels of replication. It is also shown how these changes can occur even in presence of of foreign keys on the tables that are split/merged.

- A new indexing technique, LH3, has been developed that extends earlier work in scalable data structures, especially LH [Litwin80], LH* [Litwin93a] and LH*LH [KARL96]. LH* and LH*LH are distributed data structures which have been modified to exist in an environment with replication and transactions. Also a new compressed and distributed B+tree has been developed.
- The assumption of cheap communication cost achieved by new technological development has consequences on the system architecture and therefore a set of new ideas have been developed in this thesis on transaction protocols and replication strategies.

The research presented in this thesis focuses on the development of a reliable distributed DBMS. Much of this research is based on earlier research in distributed databases and has extended a number of ideas from research at NTH in Trondheim (Svein-Olaf Hvasshovd, Prof. Bratbergsengen among others). nWAL is an algorithm developed in Trondheim that has been used as an inspiration in developing a new transaction protocol. nWAL removes the need to flush the log buffer to disk at commit, thereby making it possible to handle very many updates per second.

1.7 System Design

The next step after this thesis is to use the ideas presented here in a design. This work has already started and is presented in [Ronstrom97a]. The design consists of platform development, AXE Virtual Machine, and development of NDB Cluster, a parallel data server for telecom applications.

1.8 Thesis Outline

Chapter 2 describes a set of telecom applications and their use of databases. It also includes a study of reliability and availability requirements.

Chapter 3 contains a set of benchmarks which has been developed based on the application descriptions in chapter 2.

Chapter 4 contains an explanation of basic concepts which have been used to make the architecturial decisions that underly the algorithms developed in this thesis. An example of such a architecturial decision is to use a hybrid shared-nothing model. It includes explanations of how databases can be used for more applications than currently.

Chapter 5 describes the system architecture. This includes a description of the replication structure, a description of all protocols used in the database and how they relate to each other and a discussion on what schema entities are handled. It includes a new result in the area of replication structure. A stand-by replica is introduced to be able to handle multiple failures cheaply.

Chapter 6 describes a new two-phase commit protocol and protocols for handling reads, updates of indexes and so forth.

Chapter 7 describes a number of protocols needed for on-line recovery and reorganisation of the database. It includes automatic creation of new replicas after node failures and automatic fragment split/merge when nodes are added/removed. Many ideas are based on [CHAMB92].

Chapter 8 describes how to support network redundancy in conjunction with the new two-phase commit protocol. It is based on ideas from [KING91].

Chapter 9 describes the crash recovery protocols.

Chapter 10 describes how to handle on-line schema changes. It introduces soft schema change which previously only have been reported in conjunction with software change. It also extends the possibilities of on-line schema changes to handle split and merge of tables. Most results in this section are new.

Chapter 11 describes an extension of LH [Litwin80], LH* [Litwin93a] and LH*LH [KARL96], the LH^3 index. It is shown how this index is integrated with the handling of replicas and also how it provides very few cache misses in accessing the index data.

Chapter 12 describes a new compressed and distributed B+tree, the HTTP-tree. It is useful for storing indexes of file names, http-addresses, telephone numbers and so forth. It is based on ideas developed in [Bayer77]. The method is favourable with processors that have the capability to process hundreds of instructions in the same time as accessing a buffer in main memory.

Chapter 13 describes the data structure used by the tuple storage. The tuple storage is based on pages to enable one structure that both handles main memory and disk data. Main memory data does however have much faster access to its pages through a very fast page index. It is shown how ideas from main-memory databases can be used to avoid the use of a logical UNDO-log.

Chapter 14 reports simulations on the new two-phase commit protocol and the new replica structure. The parameters used in these simulations were based on an early design effort for NDB Cluster [Ronstrom97a].

Chapter 15 reports the conclusions, the related work and future work.

II: Applications of Telecom Databases

The aim of this part is to understand the requirements that future applications put on telecom databases. To find this information it is necessary to investigate the possible applications and their use. It is also necessary to make some predictions on how services are used, as well as how often and in what manner. From this investigation the load on the telecom databases can be deduced. This knowledge is of great value when designing the telecom database and also when analysing the performance of the design.

The characteristics of some of the applications described in this part are studied. Traffic models are derived and based on the traffic models, the flow of requests to the telecom databases are estimated. From this, requirements on telecom databases are found. Based on these findings we also go on to define a suit of benchmarks for telecom databases.

The major difficulty in finding requirements on the information services is that these are new applications. Of course, email, for example, has been around for a long time. It is, however, not used so much by private persons and also email will be used to transfer much more multimedia files in the future. Therefore many of the requirements on these services must be drawn from extrapolations of the way we currently use similar services, such as ordinary mail, newspapers, television news and radio news.

2 Applications and their Characteristics

The application that will be studied in this thesis are future mobile telecom services and new information services. Both of those services need introduction of on-line billing services. This is necessary to enable payment in an electronic world, also to be informed of the price of services, in real-time. The services of telecom databases will be described; for some services we will also describe their database behaviour and the conceptual schema of the application and some more details on how it works. By doing this we will better understand how the database should be structured, which data types that it needs to support and what types of interaction it should be optimised for.

2.1 Telecom Applications

As a first step towards finding the characteristics of the application we will look at the network architecture that the telecom databases will be a part of. In Figure 2-1 a generic network architecture of a future telecom network is shown. There must be an access network; this can be an access network for fixed terminals or an access network for mobile terminals. The access network structure can be very complex in itself, which is not an issue in this thesis. There is also a Communication network that connects the various access networks together. This is also highly complex; there could be different types of transport networks, based on PDH (Plesichronous Digital Hiearchy), SDH (Synchronous Digital Hierarchy) or any other transport network. On top of the transport network there could be an ATM network, circuit-switched connections, an IP network and many other packet-switched networks and circuit-switched networks. In this communication network there are routing servers to assist in routing, there are also name servers to translate between addresses in the various networks. The name servers can also be used to translate between logical names and physical names inside a network.

The service network assists in providing communication services to the end-users. This service network is a logical network. This means that it contains a number of nodes that are connected through the communication network. Basically all nodes in this network are telecom databases. There are email servers, SCPs (Service Control Points), HLRs (Home Location Registers) and directory servers. The Charging Network is very similar to this network, it contains a special type of telecom databases, providing charging services and event data services. These services are used by other nodes in the telecom network and by various external systems (e.g. network planning, market research).

The Information Servers will be connected, either directly through the communication network or through an access network. Those nodes provide information services using the communication services of the telecom network.

In Figure 2-2 an example of a Physical Network Architecture is provided, though it does not show all details of the access network and the transport network. The user can be connected either through a mobile terminal or through a fixed terminal. Local Info Server represents an Info Server provided as part of the network operators services to subscribers. This service could include email, personal homepages on WWW, a storage function to store personal files and many other applications that could be provided by a network operator. For mobile users, it is likely that this Server is situated in connection with the HLR. The Local Info Servers could, however, be placed anywhere in the network and furthermore it is not necessary that the service is provided by a network operator. In Figure 2-2 a gateway between the ATM and the IP network is also shown. There is a routing and name server that support this gateway function.

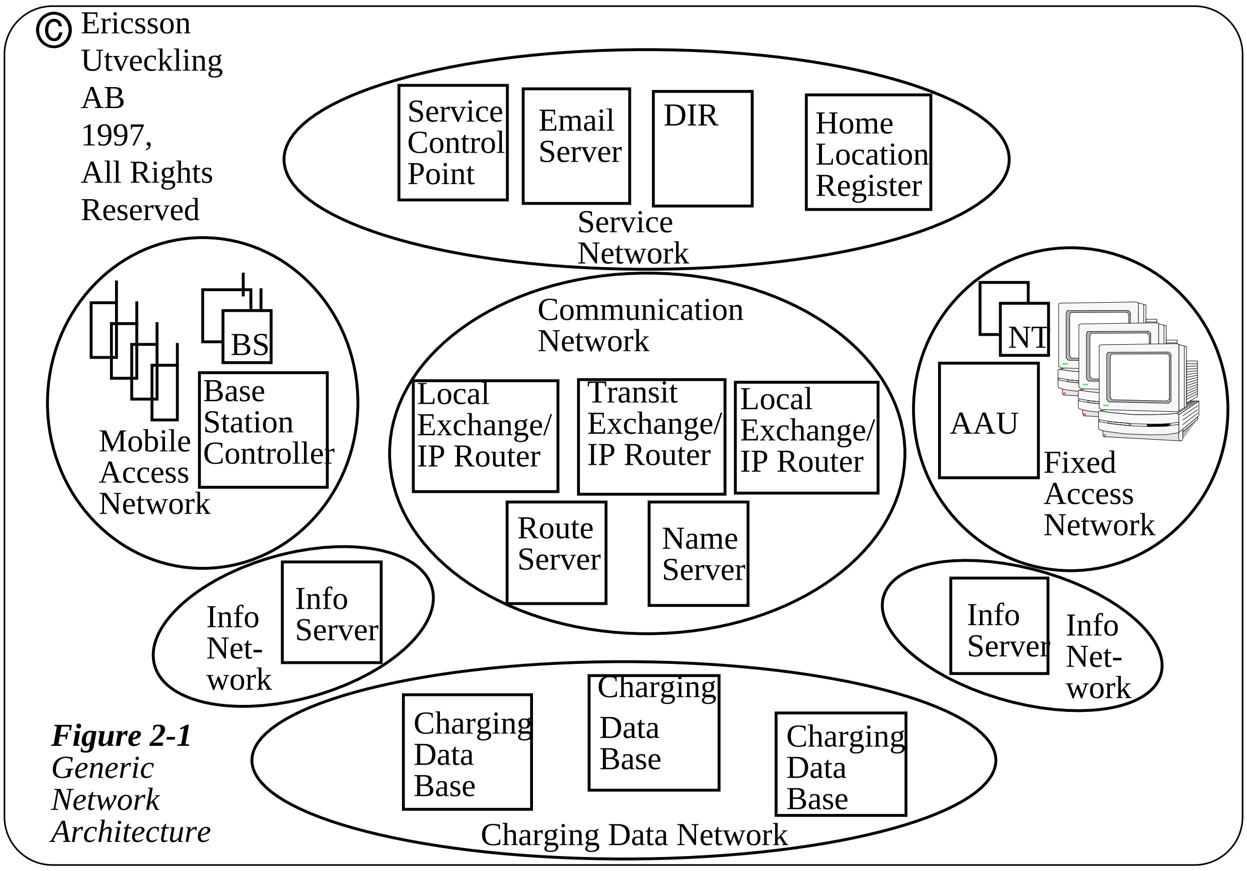


Figure 2-1 *Generic Network Architecture*

2.1.1 Service Network

There are many telecom databases that will be used to provide services for the network; the most common service comprises various kinds of address translation services. Another important type of service are directory services, where it is possible to translate a user name to a user address, also directories of the available services in the telecom network; this will be an essential service in an information network.

2.1.1.1 Number Portability

In most networks the telephone numbers of fixed telephones are connected with a particular local exchange, this has made it easy to route the calls since there is a mapping from the number to the physical location of the telephone. This does, however, create several problems, the operator meets problems whenever it becomes necessary to change the network configuration, the customer also has to change number when he permanently moves away from the area of his local exchange. It also prevents the subscribers from keeping their numbers when they change operator.

To solve this problem many operators are going to make the numbers portable. In EU this will be a requirement on the operators from 2003. This means that the number is no longer connected to the physical location of the subscriber. It could, however, still contain some directions as to where the subscriber resides, such as country code or area code. In UPT, one option for numbering schemes is a global number, in this scheme there is not even a country code.

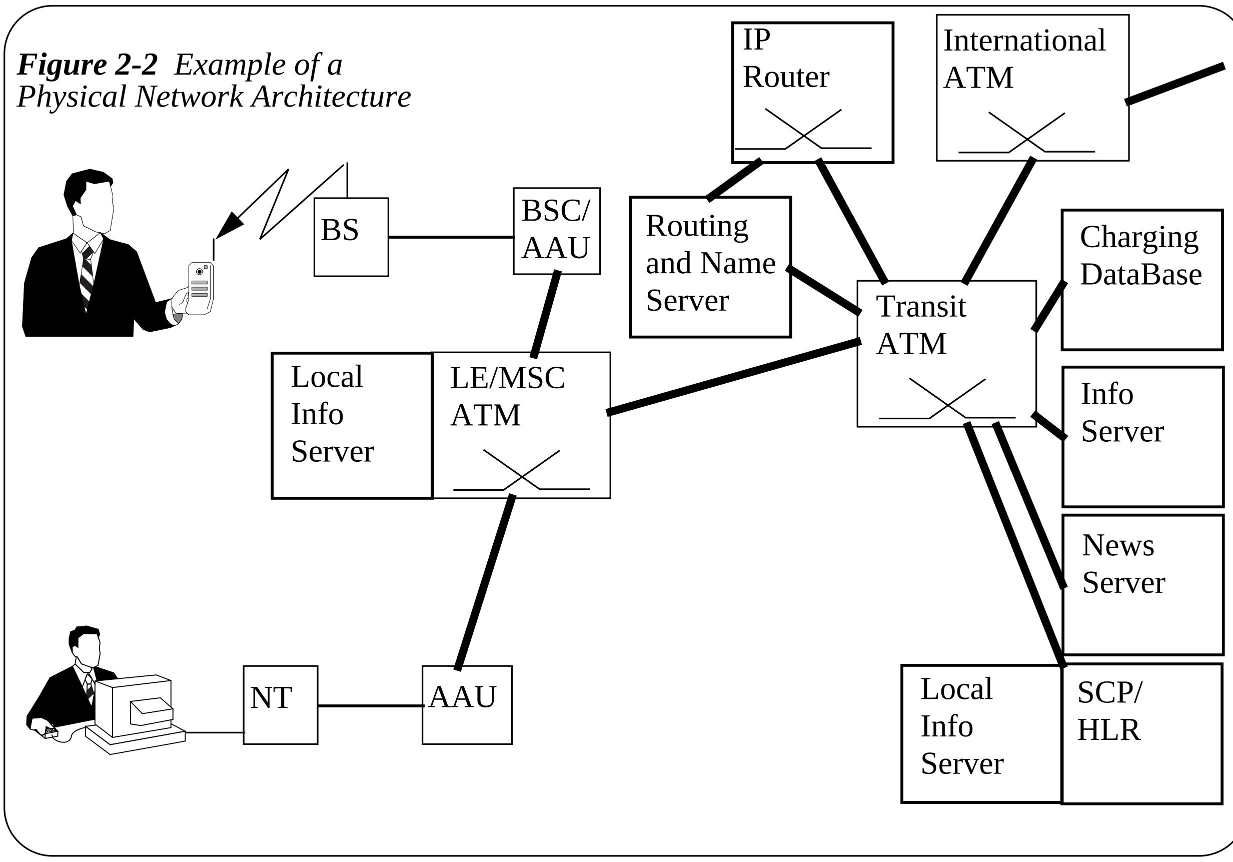


Figure 2-2 *Example of a Physical Network Architecture*

To make this possible, some scheme involving number translation is needed. To use the same routing principles in the telecom network as at present, it is still necessary to have numbers tied to physical locations. These numbers are then only used for routing in the network, the diallable number is the portable number. At some point this number is then translated to a number used in the network for routing. There are many proposals on how to implement this in the network, most of them using some kind of intelligent network solution, where the local exchange sends a translation request to a SCP (Service Control Point) where the number is translated. This is the same technique as is used today for freephone services (800-numbers in US).

This particular service uses the telecom database mostly for reading, the request are simple table look-ups. There is however a high number of them; it is necessary to translate both the calling number and the called number; therefore each telephone call consists of two requests to the telecom database. The number of updates is small; it is only necessary to update when the operator changes the structure in the network, and when the subscriber permanently moves so that it is necessary to change his routing number. The requirements on reliability are high, the function is necessary for the network to operate.

The conceptual schema of this service is simple, it basically contains a few numbers. There is only one table where those numbers are stored and the queries used translate from one number to another, which could look like this:

```
SELECT NETWORK_NUMBER
FROM NUMBER_TABLE
WHERE USER_NUMBER = user_number;
```

Two new index structures have been developed in this thesis that are useful for this particular query. One of them is optimised for speed and uses a hash-based structure. The other is optimised for memory usage and uses a compressed B-tree structure.

2.1.1.2 Routing Server

Currently most routing decisions are taken in the control processor of the switches and routers. This has been necessary since it is a very common operation and it also requires information that is up-to-date about network performance. This has required the use of distributed routing algorithms. When the capabilities of the telecom databases increase, it becomes possible to also put some of the routing control in telecom databases; this database could, of course, also be a part of the switch or router.

This requires that the telecom database is updated frequently on the network performance. Then the switches and routers ask the routing server for information on how to route the calls and the messages, which can be performed on a call-by-call basis or on a periodic basis.

In Internet there exist specific routing protocols; these specify that each router have a routing table. A router normally use routing protocols that inform neighbours of what destinations it can serve for the moment. This updating is performed each 30 seconds.

In a routing server there are two common actions. The first is to make a routing decision; this could involve a number of retrievals of records from the routing table. The second is to periodically update the routing table with new information. Since these two actions can occur concurrently, there is also a need to examine concurrency control. It should be possible to make routing decisions even when the routing table is updated and therefore some version-based scheme is needed. One could either simply keep two copies and switch between them at each routing table update, another solution is to make sure that reading can always be performed on a consistent copy of the routing table, which can be implemented using a multi-version concurrency scheme.

This service requires extensive use of both updates and reads. The updates in the routing server do not need to be as reliable as in most other cases. Routing data is not perfectly consistent since each routing server performs its updates independently of the other routing servers. Also each routing server can find each IP-address on Internet and thereby erroneous routing decisions are quickly recovered by other routers.

2.1.1.3 Name Server

The name server functionality is used in translating between logical addresses and physical addresses (as in the Number Portability case); it is also used in translating from an address in one network to an address in another network. As an example a user could have one telephone number, an email-address, an IP-address and an ATM-address. When passing through gateways between networks it is necessary to translate the addresses.

An example of translating from logical to physical addresses could be translation from a URN-address, that is an address on WWW that is a logical address, which needs to be translated to an URL-address. The URL-address specifies the physical location of where the message is directed (e.g. to fetch a file in a specific directory, in a specific machine, using WWW).

An example of translating between different networks is when an IP-message is sent on the ATM-network. There the IP destination address is used to find the ATM destination address, which could either be the final destination or another gateway node.

This service uses the telecom database mostly for simple table look-ups where the number of updates is very small compared to the number of reads. The requirements on reliability are, however, very high since the network does not operate without this function.

This conceptual schema, too, is very simple, with one table for each type of translation necessary, the queries could look like this:

```
SELECT ATM_ADDRESS
FROM ATM_IP_TABLE
WHERE IP_ADDRESS = ip_address;
```

or like this for WWW address translation:

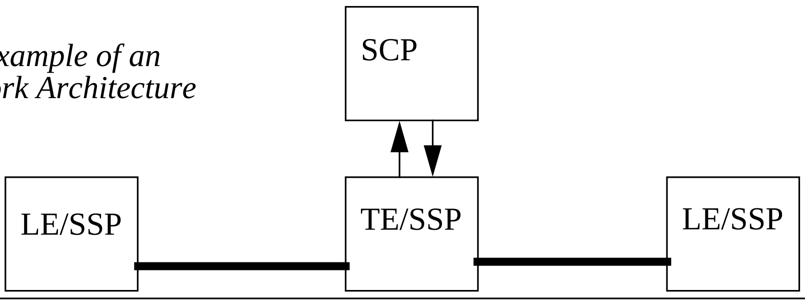
```
SELECT URL_ADRESS
FROM URL_URN_TABLE
WHERE URN_ADDRESS = urn_address;
```

2.1.1.4 Intelligent Network Services

Intelligent Network development started with the use of 800-numbers in USA. Companies needed more flexible billing strategies when calling their offices; and the routing had to be more flexible. One type of requirement was to route the calls to different offices dependent on the time. All these requirement led to the development of Intelligent Network Services. Most of these services are based on the idea of translating numbers and redirecting the charging. However some functions also involve other functionality, such as Televoting. There have also been many studies, experiments and implementations on providing supplementary services (e.g. Call Waiting, Call Redirection), using intelligent network techniques.

In Figure 2-3 an example architecture of Intelligent Networks is shown; the SSP (Service Switching Point) is triggered when an action is performed that needs interrogation of the SCP (Service Control Point). The SSP interrogates the SCP and takes action upon delivery of the response from the SCP. The SCP can be co-located with the SSP (SSCP), can be an adjunct processor to the SSP (ACP) and various other scenarios are possible. The SSP can be part of any type of switch, in most implementations it is part of a transit or international switch. It can also be located in a local exchange.

Figure 2-3 *An example of an Intelligent Network Architecture*



The use of the telecom database in this case is diverse; there are both transactions that update the telecom database and services that read various data. It is similar to the usage reported on mobile telecommunications. The study on mobile telecommunications also used a network where the infrastructure is based on intelligent network concepts. The use of this application is, however, not a mass service in the same way as the other services presented here, so this type of service will not be considered any further. If a telecom database can handle the other applications investigated in this thesis, then it should also be able to handle this application.

2.1.1.5 Directory Service

This service can be used directly by an end-user or by another network node. It translates names recognisable by an end-user into network addresses.

An example could be to translate from “Joe Shannon” into his telephone number. If interacting with an end-user, it is not necessary that the name is unique; the directory service can provide all the records that match the description. If used by network nodes the name must be unique however, since the network node does not have any intelligence to select from the possible persons.

This service is also mainly a table look-up function, only a little bit more complex table look-up since it is not always unique. The requirement on reliability of this service is dependent on how important this service is to the operator and the users. This service is not absolutely necessary for the network to continue its operation. Therefore a lower reliability of the system is sufficient.

This service will not be further studied in this thesis as it is not envisioned to become a mass service. This would require that people call by name, rather than by number and such an evolution is not foreseen. However it will still be used often by many people. Slightly more complex queries could also be used by the directory services. As an example similarity searches and searches on partial names could be performed. This is of great value when only partial information about a subscriber is known and also when spelling errors have occurred.

2.1.1.6 Mobile Telecommunications

The mobile telecom network needs telecom databases to handle both the location of users and mobile telephones and also the services of the users. The requirements here will be very tough and are therefore studied in greater detail later in this thesis.

2.1.1.7 Characteristics of the Telecom Applications in the Service Network

The characteristic of a Number Portability service is that it should handle about 4-5 table lookups per user per hour. This requirement is easier to handle than the requirements from mobile telecommunications and is therefore not considered further. The requirements on the name server are very much dependent on the network architecture, the size of the messages, the bandwidth and so forth. Basically the more table lookups a database can do, the better. Therefore we do not specify any particular requirements for this type of application.

2.1.2 Information Network

In 1994 the usage of Internet based on World Wide Web (WWW) exploded and also the usage of email, ftp and various other Internet services are quickly increasing. The network is, however, limited in bandwidth and there are bottlenecks in many places. It is possible to transfer very large files over this network but the bottlenecks in the network still preclude the usage of video, HIFI-sound and other broadband services on a large scale. To design a network that can handle the load of millions, possibly billions, of concurrent users that need both text, sound, video, pictures of various qualities, it is necessary to analyse both the need of network nodes, the network links and network switches. Therefore it is necessary that the usage is modelled in some sense. The aim in this thesis is to find requirements on a general information server that contain data objects of sizes up to about 1 GByte. Objects of greater size are mostly long video sequences and other specialised applications such as virtual reality. It is assumed that these are solved by specialised servers that can handle many continous streams of information, while the server that we focus on in this thesis requires fast access to small and medium-sized objects used by many concurrent users.

Since the global information system is a distributed system, it will not be seen from the user that there exist different types of servers. These different servers could even be parts of one general multimedia server, since general servers are in themselves distributed systems. So one part of the system could be a number of signal processors, another part could be a number of data servers, another part could contain video servers, another part could contain query servers and yet another part could contain application servers.

In this thesis three applications will be analysed in greater detail. The first is a general email-server, the second is a News-on-demand server for a very large newspaper and the last one is a genealogy application. It is believed that the other applications will have smaller requirements on throughput compared to those server applications. The email application is a mass service and must also provide large storage volumes. News-on-demand is also a mass service; it needs a higher bandwidth since the objects that will be fetched will be much larger than email objects. The storage of the News-on-demand service can, however, use caching to a much larger extent than the email service can do, since pages of the newspaper will be read by many users. The genealogy database is a service with very high requirements on the storage volume since it needs to store scanned images with very high resolution. A genealogist will also need to look at many such images in one session. The genealogy service also provides an interesting application of a global information application. In this application millions of people can work concurrently on a similar problem, to find their ancestors. Since many have common ancestors, a global database can be used to spread information on research results immediately to all other genealogists.

2.1.3 Charging Network

Charging is something needed by all telecom applications and the requirements on charging applications are tougher than before due to the many events in the telecom network and also due to the fact that much more of the billing has to be performed immediately after usage of a service and not several months after as has previously been the case. This is studied in greater detail later in this thesis.

2.2 Mobile Telecommunications

The first Mobile Telecommunication systems were developed a long time ago. The first systems that generated a mass market were, however, the analog systems. These systems are generally called first generation systems. Examples of these are NMT (Nordic standard), TACS (English standard) and AMPS (American standard). These still exist and in some countries still form a substantial part of the market.

The second generation system development was started when the first generation system was beginning to catch a mass market. Second generation systems were digital and made better use of frequencies and also tried to achieve a better sound quality. The better use of frequencies and also the smaller size of the cells made it possible to sell second generation systems to an even larger market. This market is still expanding very quickly. There are three major systems in the second generation. These are GSM (European standard), D-AMPS (American standard) and PDC (Japanese standard).

In parallel with the development of second generation systems, new systems were also developed for use in houses and in urban areas and in the vicinity of a person's home. A large number of such systems were developed; the major system that survived in this market is DECT (European standard) and PHS (Japanese standard).

Already in 1987 a project was defined in the RACE-program (Research on Advanced Communications) in EU to define UMTS (Universal Mobile Telecommunications Systems). The aim of this project was to study third generation mobile telecommunication networks. An essential part of this study was to look at integration of the B-ISDN network and the mobile network. In the RACE II-program (1992-1995) a follow-up project was performed.

In this phase the reality of an exploding second generation system (e.g. GSM) changed the surrounding world. In 1993-1995 also WWW took giant steps into defining the next generation broadband applications. Therefore third generation mobile systems must be based on the second generation mobile systems and the network must be able to interwork with Internet. Therefore many of the initial aims will be very difficult to reach. However, many of the studies performed in the RACE-programs can also be used to develop the second generation systems and so a standardisation of UMTS is proceeding with the aim of having systems available in 2002-2003.

The explosion of the second generation systems and the development of DECT and PHS which has also attracted a large market have driven the development of generation 2.5. In the USA this development led to PCS that was started recently. In this generation the existing systems are further developed for higher bandwidth, better use of radio resources by even more efficient use of frequency and even smaller cells. Also packet switched data is an important issue in this generation. Integration of second generation systems and DECT and PHS is also likely to occur in this generation.

In this thesis Mobile Telecommunications are studied from a future perspective with as many users as in the current fixed network and also lower prices and therefore also much higher usage than in current systems. The telecom databases found in this application are used to keep track of users and mobile telephones. It also provides a platform for many network based services; it will also provide an interface to intelligent terminals and thereby make it possible to implement terminal-based services. The databases will store information on these services, location information and directory information. This information is rather compact and possible to store in main memory. Most procedures require rapid responses; in [MONET075] a delay (without queueing) of 1-10 ms (dependent on query type) in telecom databases gave a delay of 0.6 s in the fixed network for call set-up in simulations (on top of this there is a delay in the access network). Therefore telecom database response time must be in the order of 5-50 ms in a loaded system. The toughest delay requirements are however on retrievals; these are used in call set-up while most other procedures are not as delay sensitive. Therefore 20-200 ms delay time on updates can be acceptable.

Network studies were performed in this work and were based on an estimation that 50% of the users will be mobile users. These users will be charged as for calls from fixed phones and therefore their usage is similar to the fixed network and traffic models were derived from mathematical models on movement of people in cities. These results were reported mainly in [MONET075]. Other deliverables in the RACE-program also took some of these issues into consideration. In

[Ronstrom93] the same basic assumptions were used. The network architecture used was, however, an architecture based on an evolved second generation mobile telecommunication network. In [MICL94], later results from the RACE-program were used and a message service was also included in the study, and these results were used to derive results on loads on nodes and links.

The results in this thesis were derived from an analysis of research in a third generation system. These figures are, however, also applicable in evolved second generation systems. The main difference with current systems is the assumption on the traffic model where mobile phones are used in a similar manner to current fixed phones, the assumption that more functions are moved to the telecom databases and also that there are more communication services in the future systems.

The main differences between current second generation systems and third generation systems will be the bandwidth and also that the type of calls will be more diverse. Data communication calls, fax calls and message service calls will be much more common in third generation systems than in current second generation systems where speech connections are the dominating feature.

2.2.1 Conceptual Schema of UMTS

In Figure 2-4 a conceptual schema of the UMTS system is described[MONET061]. Of course in a real system, the schema must be supplied with much more details. This schema does, however, mention the basic entities and their relationships.

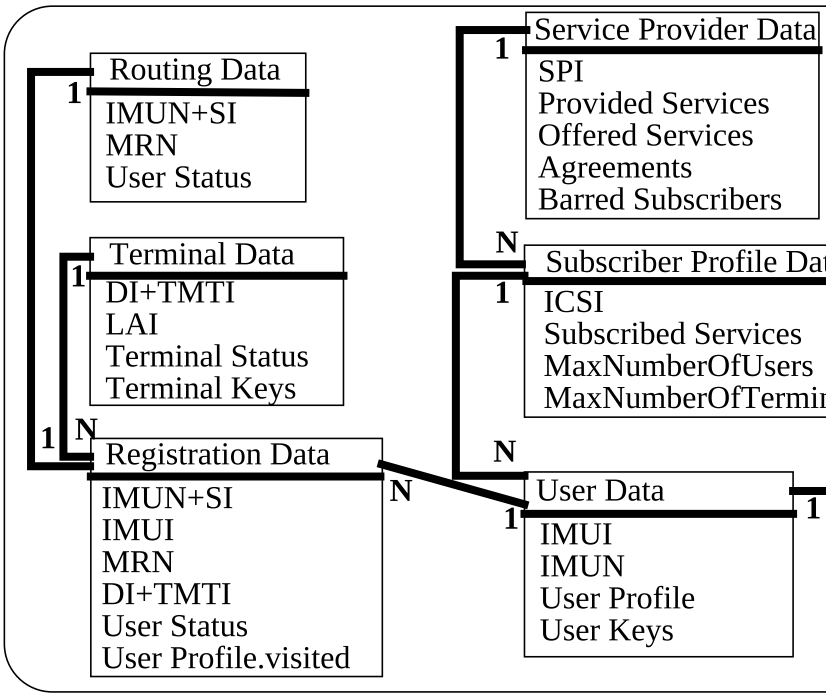
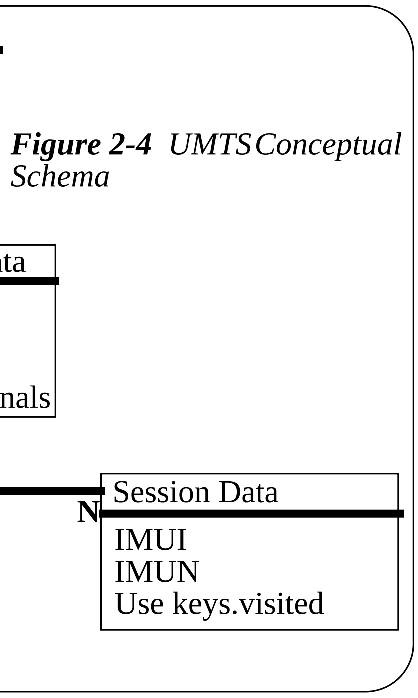


Figure 2-4 *UMTS Conceptual Schema*



There is a set of identifiers and numbers that is used in a UMTS network. SPI is a Service Provider Identity, identifying the Service Provider. ICSI is the International Charged Subscriber Identity: this is the identity of the subscriber, the entity that pays the bill and has the business relationship with the Service Provider. IMUI is the International Mobile User Identifier, which identifies a certain UMTS user, the IMUN is the International Mobile User Number, which is the diallable number of the UMTS user. SI is the Service Identity of the registered service. A terminal address is composed of DI, which is a domain identifier (identifies the network operator) and TMTI, which is the Temporary Mobile Terminal Identifier, a temporary identifier of the terminal in this particular network. LAI is the Location Area Identifier, which identifies a location area used at paging of the terminal. Then we have the MRN, which is the Mobile Roaming Number, which is used for routing purposes, to set-up the connection to the exchange where the user currently resides.

Service Provider Data consists of the information on the Service Provider that is needed for execution of UMTS Procedures. This consists of the services offered to home users, services offered to visiting users, agreements with other operators and a list of barred subscribers. Subscriber Profile Data consists of the information on the subscription that is relevant to execution of UMTS procedures, this includes a list of the services subscribed to (at the home operator) and maximum number of users and terminals. User Data contains information on a specific user, his service profile and his authentication and encryption keys. Each user can then have a number of registrations, one for each service the user has registered for. The registration is connected to a specific terminal. The terminal data also contains terminal status and some keys for security procedures. It is necessary to have Routing Data connected to the registration to be able to find the Terminal to which the user has registered. When a user is active he also has sessions connected to it. There could be several simultaneous sessions, but there is only one session per service provider, which means that each UMTS database contains no more than one session record per user.

The data in the UMTS network is distributed in databases of the home operator and databases of the visited operator. There are various ways to handle this distribution. A likely way is that the home database stores the Service Provider Data, Subscriber Data and User Data. Routing Data is also needed in the home network to be able to route the call to an exchange in the network the user is visiting. The Session Data, Registration Data and Terminal Data are always created in the visited network; parts of the Registration Data will also be stored in the home network. Parts of the User Data can be transferred to the visited network to make the execution of the UMTS procedures more efficient.

More information on the UMTS conceptual schema and distribution of data in the UMTS network can be found in [MONET061], [MONET075], [MONET108], [MONET109].

2.2.2 UMTS Procedures

The procedures used in an UMTS network will be presented. For each procedure, the impact on the telecom database will be analysed and the queries generated will be shown. Many of these procedures are general and necessary also in another mobile network and most of them also in a fixed network. The information flows of the procedures are based on [MONET099]; in this document these information flows have been mapped to a physical architecture.

The major differences of these information flows compared to information flows of current systems is the use of sessions in all communications between the system and the user, the separation of terminal and the user (as in UPT, Universal Personal Telecommunications) and many more services. Finally the telecom database is more involved in the call set-up process where it is involved in both the originating and the terminating part of the call set-up.

The network architecture used in the information flows is shown in Figure 2-5. The SCP (Service Control Point) and SDP (Service Data Point) are normally collocated; they are chosen as separate entities in this presentation, since databases are the focus of this thesis. The interface to the SDP could therefore be an internal interface in the SCP or a network interface. In this way we can derive the number of requests that the SCP database must handle internally.

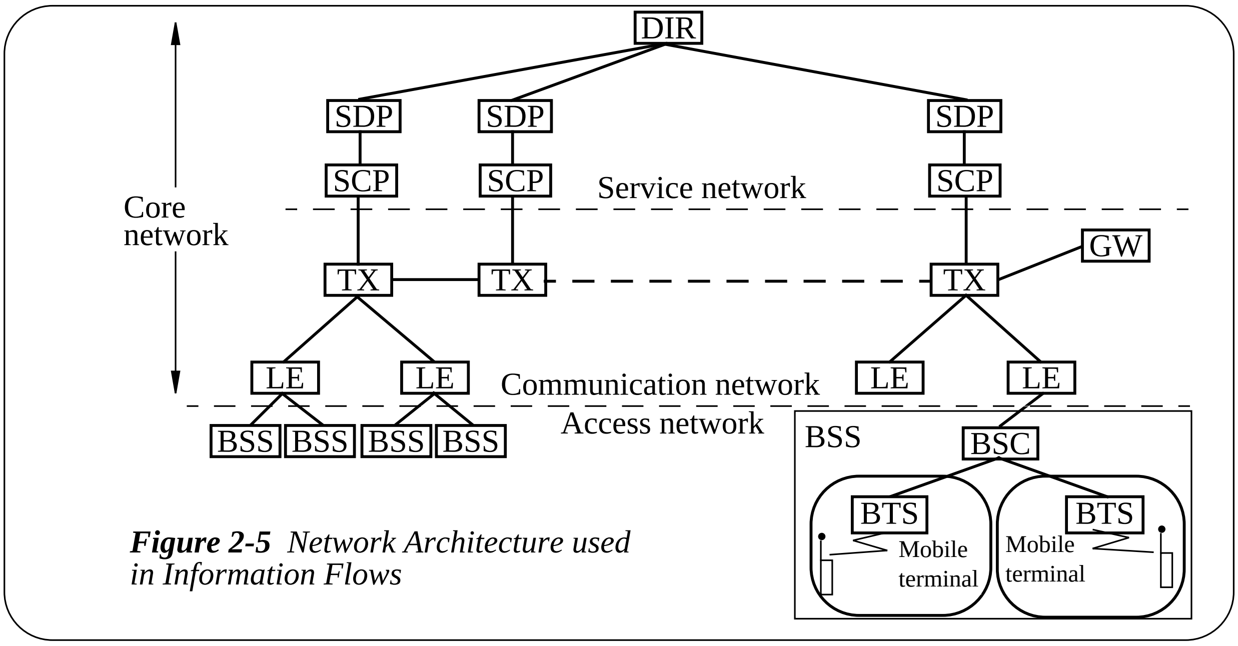


Figure 2-5 *Network Architecture used in Information Flows*

The DIR is a node defined in RACE MONET, it is used to keep track of which database the user data resides in. One of the major reasons for this architecture is to avoid the UMTS number is pointing to a physical network node as this would make changes in the database structure in the network more cumbersome. In this presentation we will not show the actions of the DIR node. The information flows we show, will also work without DIR nodes. The DIR nodes are basically simple table look-up databases, where a user number is mapped to a database address.

All the procedures have more than one possible information flow. For example, a user registration has a different information flow when invoked from a visited network, compared to when it is invoked from the home network. Taking this into account in the calculation of transaction flows in the network would necessitate a detailed mobility model and also that all possible information flows are studied. Such a mobility model was used in [MICL94] and [Ronstrom93]. In this thesis we only show one version of each information flow. This keeps the presentation short and avoids using an unsure mobility model. The information flows chosen should be representative of the most common cases.

All retrieve, update, create and delete operations in the information flows use a unique primary key to access the data, if not otherwise mentioned. This means that retrievals can appear thus:

```
SELECT ATTRIBUTES
FROM TABLE
WHERE PRIMARY_KEY = primary_key;
```

Update, create and delete are performed in a similar way. In some procedures several requests can be part of the same transaction. In this section we do not discuss how charging information is generated, this is discussed in section 2.5 below.

2.2.3 Session Management

Sessions are used to ensure a secure communication. Before a user is allowed to use any network resource, a session must be set-up. The major reason for sessions is to authenticate the user and the terminal. This authentication ensures that the user is allowed to use UMTS services and also makes it more difficult to use stolen UMTS equipment. Another part of the session set-up is to exchange encryption keys, which are used in the air interface to make sure no one can interfere or listen to messages and conversations.

Since terminals in UMTS do not have identifiers, there are also terminal sessions. The terminal receive a temporary identifier during a terminal session. Services that do not relate to a user can be performed (e.g. attach) during a terminal session. If a user is involved in the service, the terminal session must be upgraded to a user session. Sessions can be initiated either by the network (e.g. at a terminating call) or by the user (e.g. at an originating call). We assume a terminal session exists for longer period than a user session, therefore we assume that the normal case is that there exists a terminal session during execution of the UMTS procedures.

A session can be used for several communications. This means that sessions are established before calls can be made. However a session can be established for a longer time than the duration of a call. Therefore, if a user makes several calls in a row, these can all be part of the same session. User sessions are normally released a specified time after service completion.

2.2.3.1 System-Originated User Session Set-up

In Figure 2-6 the information flow of a system-originated user session set-up is shown. In the information flows, we have not shown the flow that would occur if special payment methods were used.

First the database is checked to see if there are any sessions already available (SESS_PAR). Then the database is used to find the user number, given the user identifier (CONV_UI). Then the terminal and the user is authenticated (TERM_AUTH): for terminal authentication an encryption key is needed (RETR_ENC_PAR) from a security database and for user authentication an encryption key and authentication key are needed from the security database too (SEC_M3). Finally the session data is stored in the database (SESS_STORE).

If there already exists a terminal session, then it is not necessary to authenticate the terminal and it is possible to proceed with user authentication immediately. As can be seen, there are five requests to the database, two of which are directed for the security database. Four of these requests are retrievals and one of them is a create request. The retrieval of encryption data for terminal authentication is removed if a terminal session already existed. Also the retrieval of session parameters and retrieval of the UMTS number can be combined into one retrieval request to the database. Therefore there are two retrievals and one create request in this procedure.

2.2.3.2 User-originated User Session Set-up

In Figure 2-7 a user-originated user session set-up is shown. The major difference, apart from who initiated the session, is that it is not necessary to retrieve any encryption key and it is not necessary to convert the identifier to a number. Therefore there are two retrievals and one create in this procedure. If the user invokes a service when he has a terminal session which needs a user session,

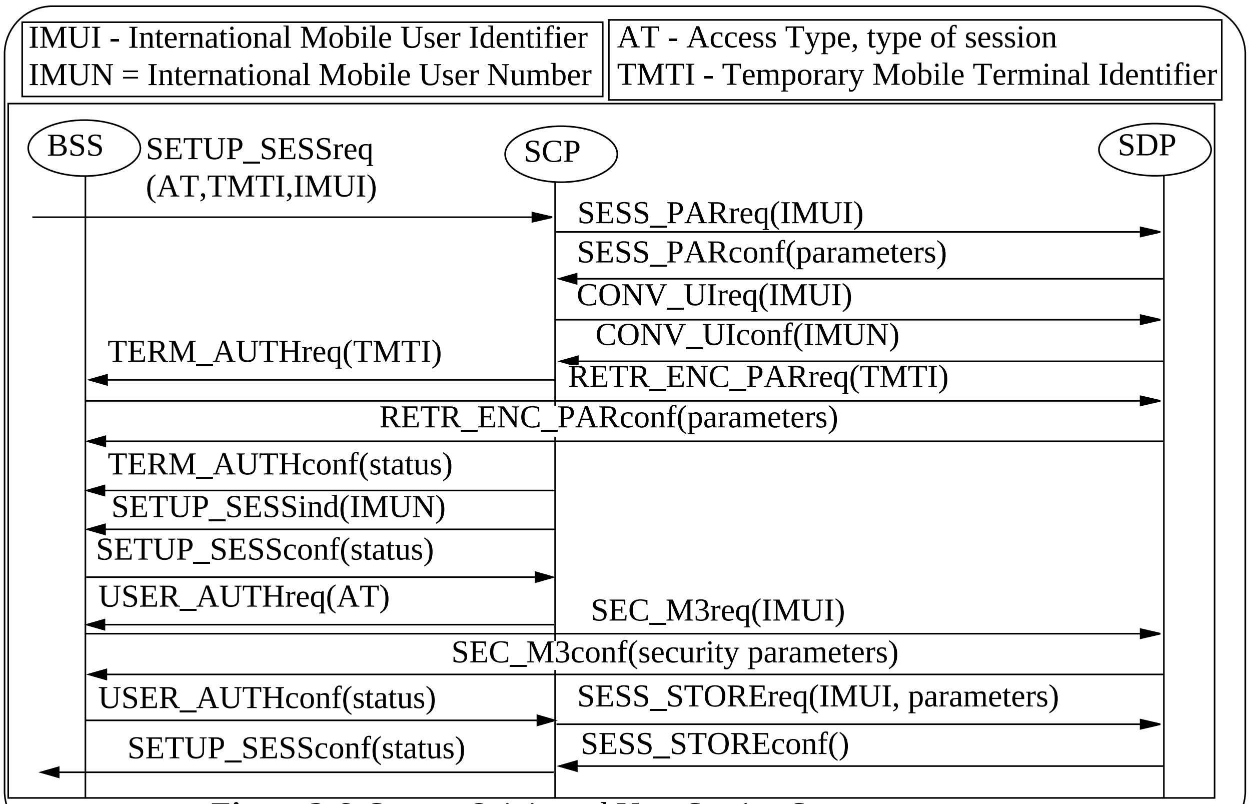

IMUI - International Mobile User Identifier
IMUN = International Mobile User Number
AT - Access Type, type of session
TMTI - Temporary Mobile Terminal Identifier
BSS
SCP
SDP
SETUP_SESSreq (AT,TMTI,IMUI)
SESS_PARreq(IMUI)
SESS_PARconf(parameters)
CONV_UIreq(IMUI)
CONV_UIconf(IMUN)
TERM_AUTHreq(TMTI)
RETR_ENC_PARreq(TMTI)
RETR_ENC_PARconf(parameters)
TERM_AUTHconf(status)
SETUP_SESSind(IMUN)
SETUP_SESSconf(status)
USER_AUTHreq(AT)
SEC_M3req(IMUI)
SEC_M3conf(security parameters)
USER_AUTHconf(status)
SESS_STOREreq(IMUI, parameters)
SETUP_SESSconf(status)
SESS_STOREconf()

Figure 2-6 *System-Originated User Session Set-up*

then an upgrading of the terminal session to a user session is performed. The information flow for this procedure is very similar to a user-originated user session, the requests to the database are the same.

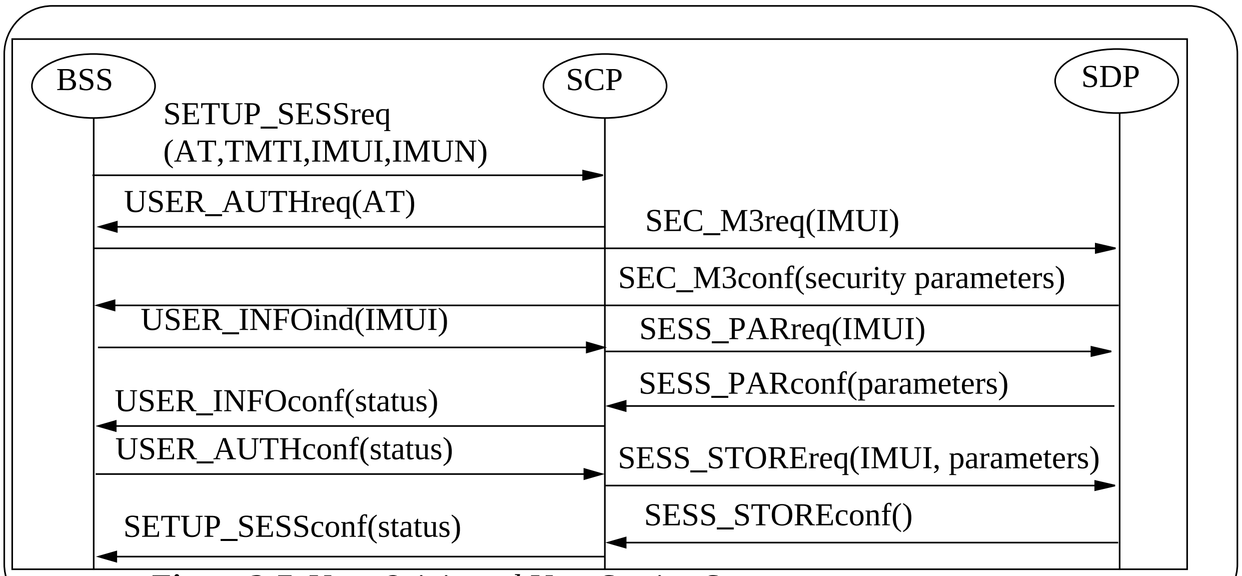
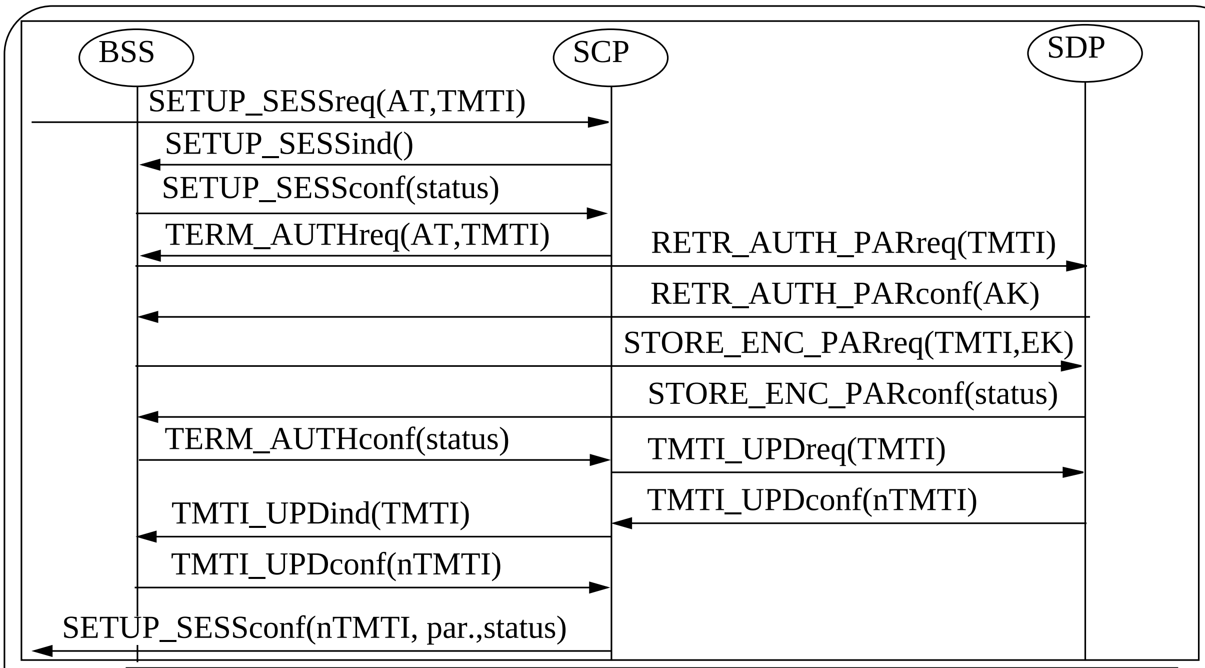


Figure 2-7 *User-Originated User Session Set-up*

2.2.3.3 System-originated Terminal Session Set-up

In Figure 2-8 an information flow of a system-originated terminal session set-up is shown. In this procedure the terminal is authenticated (RETR_AUTH_PAR) and a new encryption key is stored in the security database (STORE_ENC_PAR) and a new TMTI is assigned (TMTI_UPD). This message also creates a new terminal record. The assignment could be performed with a retrieval and an update in a transaction. This comprises two retrievals, two updates and one create request to the database.



EK - Encryption Key, AK = Authentication Key, nTMTI = new TMTI

Figure 2-8 *System-Originated Terminal Session Set-up*

2.2.3.4 User-originated Terminal Session Set-up

In Figure 2-9 a user-originated terminal session set-up is shown: the terminal is authenticated using an encryption key retrieved from the security database (RETR_ENC_PAR) and finally the new TMTI is assigned by the database (TMTI_UPD). This message also creates a new terminal record. Thereby there is one retrieval, one create and one assignment in this message. Thus there are two retrievals, one create and one update to the database in this procedure.

2.2.3.5 Session Release

Finally sessions are released, the information flow for a user session release is shown in Figure 2-9, where the session object is deleted from the database (SESS_REL). A system-originated session release is performed the same way, only the initiator is different. There is one delete operation in this procedure to perform in the database. When releasing a terminal session the terminal number is also deassigned, which involves one update.

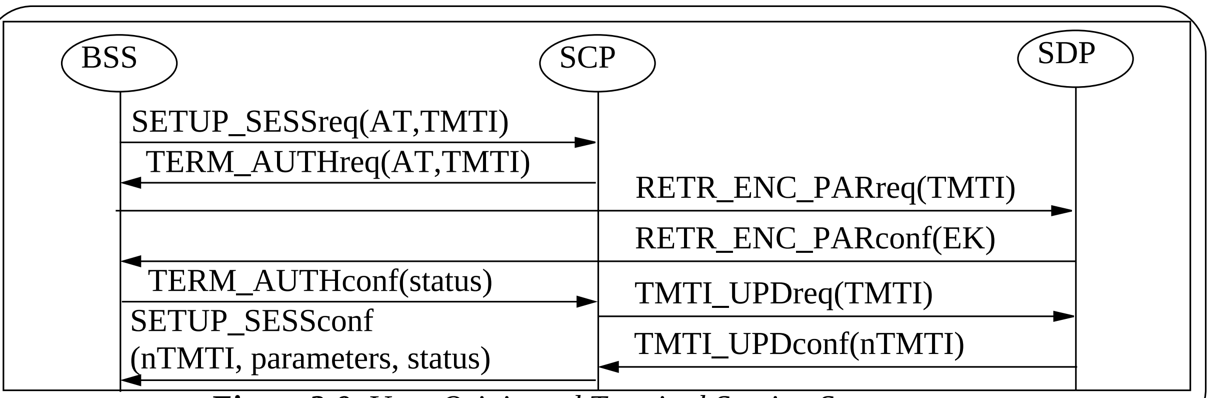


Figure 2-9 *User-Originated Terminal Session Set-up*

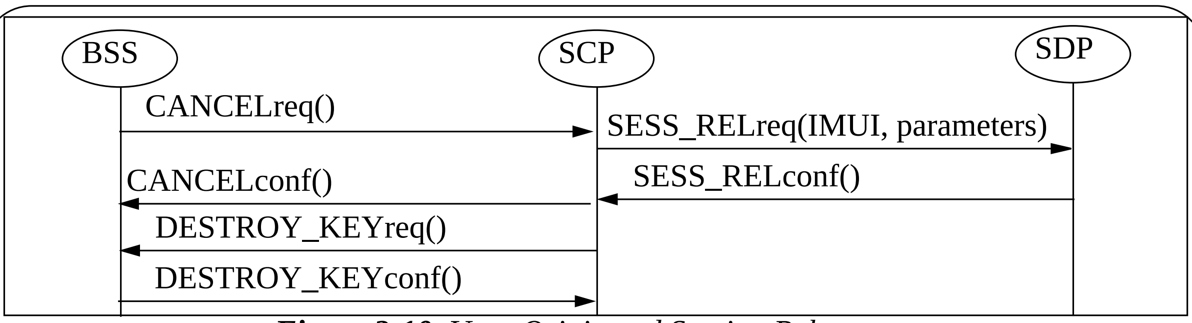


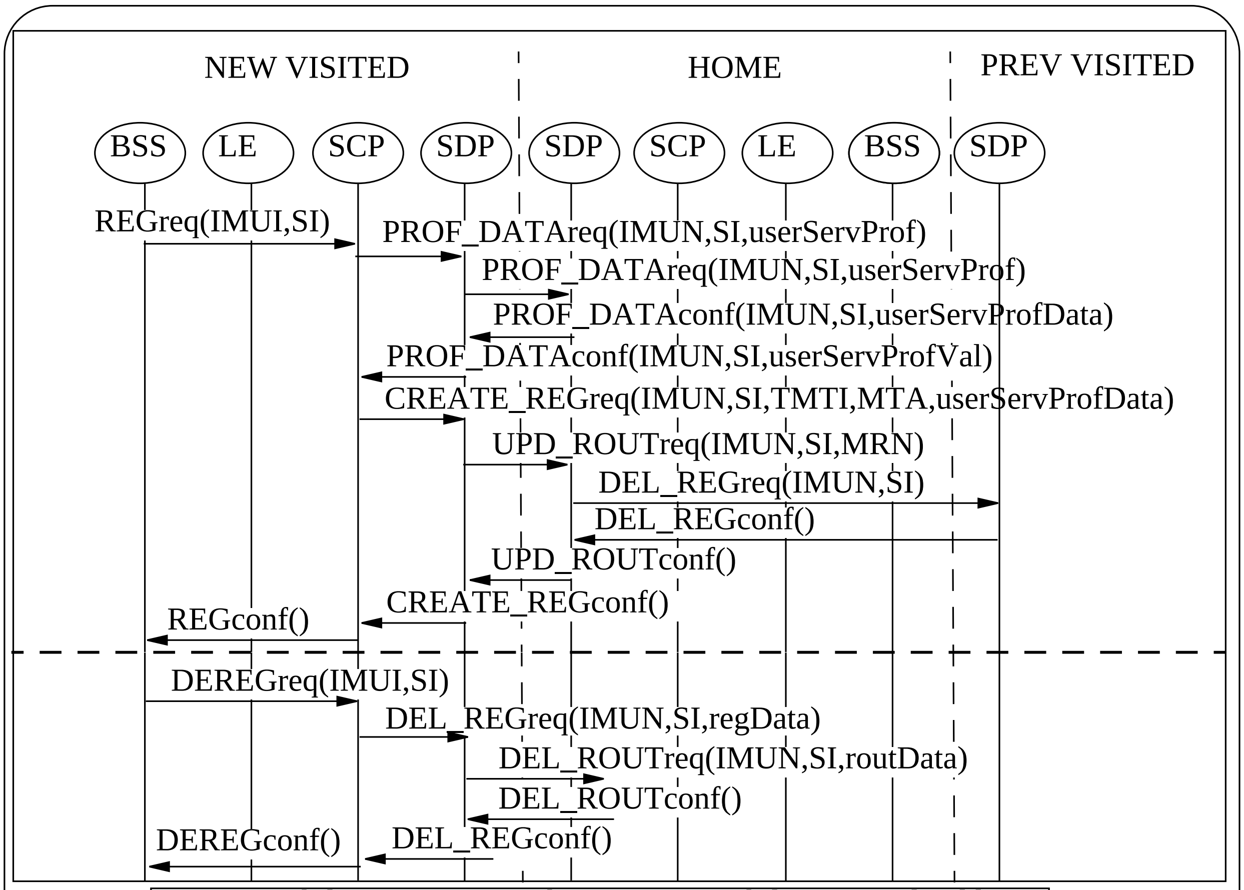
Figure 2-10 *User-Originated Session Release*

2.2.4 User Registration Management

User Registration is performed when a user wants to assign a specific communication service (e.g. voice, fax) to a specific terminal. There could be several users registered on the same terminal. Registration is only necessary to be able to receive calls at the terminal. Outgoing calls are performed as part of a session and registration is therefore not needed for outgoing calls. Registration is performed in a session. There could be more than one registration of a user for different services.

The registration procedure (see Figure 2-11) is one of the more complex procedures in a mobile network. First it is necessary to fetch parts of the service profile of the user to make sure that he is allowed to register for this service (PROF_DATA). If the user is registering in a visited network, then the profile must be fetched from the home network, so there is one retrieval in the visited network and a retrieval in the home network to fetch the requested data. Secondly the new registration must be created. This could involve three steps, creation of the new registration (CREATE_REG), deletion of the old registration (DEL_REG) and creating the routing information in the home network (UPD_ROUT). This procedure has some consistency requirements; these have been studied in [MONET108]. They are not included in the procedure shown here.

In the traffic model we will assume that each registration is new and that there is no deletion of the old registration is necessary. In the information flow we do, however, show the full scenario where the old registration is also deleted. Thus there are two retrievals and two creates in a user registration. The user deregistration is simpler, it deletes the registration (DEL_REG) and deletes the rout-


NEW VISITED
HOME
PREV VISITED
BSS
LE
SCP
SDP
SDP
SCP
LE
BSS
SDP
REGreq(IMUI,SI)
PROF_DATAreq(IMUN,SI,userServProf)
PROF_DATAreq(IMUN,SI,userServProf)
PROF_DATAconf(IMUN,SI,userServProfData)
PROF_DATAconf(IMUN,SI,userServProfVal)
CREATE_REGreq(IMUN,SI,TMTI,MTA,userServProfData)
UPD_ROUTreq(IMUN,SI,MRN)
DEL_REGreq(IMUN,SI)
DEL_REGconf()
UPD_ROUTconf()
CREATE_REGconf()
REGconf()
DEREGreq(IMUI,SI)
DEL_REGreq(IMUN,SI,regData)
DEL_ROUTreq(IMUN,SI,routData)
DEL_ROUTconf()
DEREGconf()
DEL_REGconf()

MRN - Mobile Roaming Number, MTA - Mobile Terminal Address
SI - Service Identifier, SPI - Service Provider Identifier

Figure 2-11 *User Registration /User Deregistration*

ing information in the home network (DEL_ROUT). In Figure 2-11 we have used SDP-SDP messages between domains. These messages will most likely pass through the SCP-SCP interface in the UMTS standard.

2.2.5 Attach/Detach

Attach is used to attach a terminal to the network and detach is used to detach the terminal from the network (e.g. at power-off). When this procedure is executed an update request is sent to the database changing the attach/detach attribute of the terminal profile (UPD_TERM), as seen in Figure 2-12.

2.2.6 Call Handling

Call Handling is used to set-up, release and modify calls. Call modification is similar to call set-up in what database accesses are needed and therefore we only discuss call set-up and call release. Call release is easy, as this does not involve the database at all. In a Call set-up (see Figure 2-13) there are two retrievals in both the originating side and the terminating side, which means four retrievals for one call set-up. The first retrieval is performed to check the user profile if the requested service can be used at this time (PROF_DATA). The second access to the database retrieves the

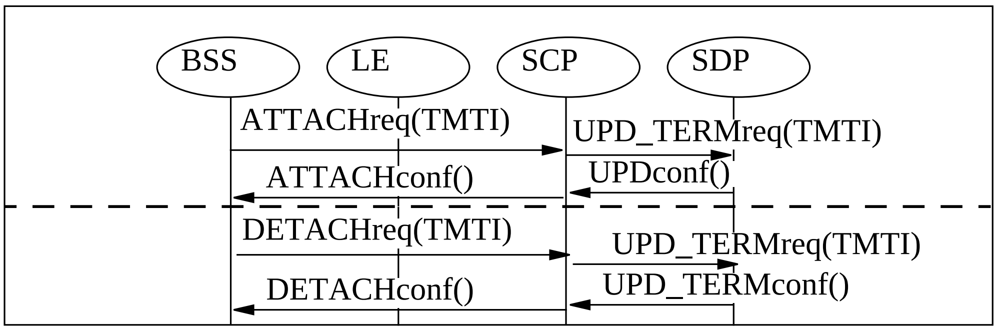
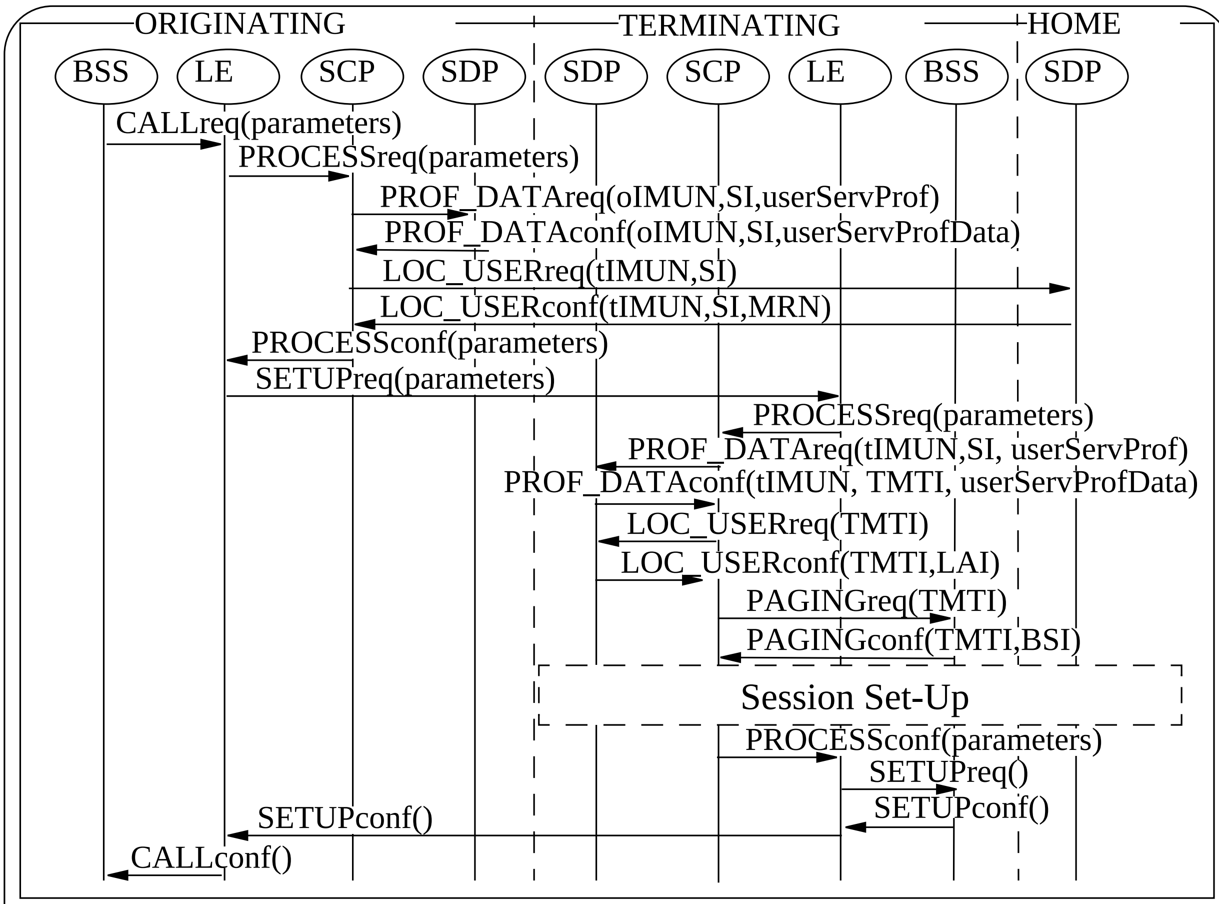


Figure 2-12 *Attach/Detach*

current location of the user (LOC_USER). The originating network needs this information to set-up the call to the right network and the terminating user needs more detailed location information to page the user.



BSI - Base Station Identifier, oIMUI - originating International Mobile User Identifier
oIMUN - originating International Mobile User Number, tIMUI - terminating International Mobile User Identifier
tIMUN - terminating International Mobile User Number, LAI - Location Area Identifier

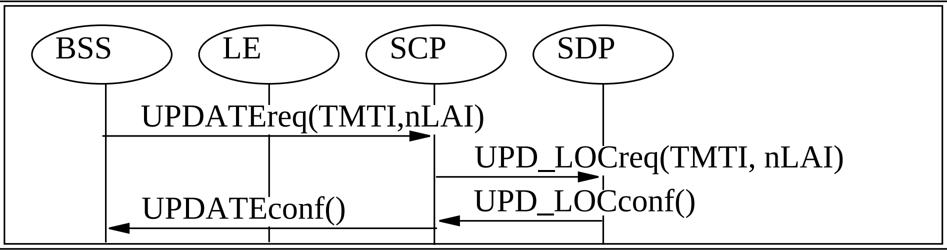
Figure 2-13 *CallSet-Up*

2.2.7 Handover

Handover is performed to change from one radio base station to another; the new base station could be located under a new exchange. It could even be located in a new network. A handover could also be performed internally in a base station between channels in the base station. The database is normally not involved in the handover procedures. Only when a handover is transferring the call to another network is it necessary to access to the security database twice; One to retrieve keys and one to update the keys.

2.2.8 Location Update

Location Updates are performed as the terminal is moving in the network or moves to a new network. By tracking the user's movement with location updates, the user can be called although he is mobile. A Location Update that changes network is called Domain Update in UMTS. This also involves moving the registrations from the old network to the new network. A Location Update updates the Location Area information in the database (UPD_LOC) (see Figure 2-14).



nLAI - new Location Area Identifier, nTMTI - new Temporary Mobile Terminal Identifier

Figure 2-14 *Location Update*

A Domain Update means that a user is moving from one network operator's domain to another. As part of this move, all his registrations in the old network must be moved to the new network. This procedure is similar to a user registration. The registration is, however, retrieved from the old domain and it is necessary to delete the registration in the old domain. Some authentication procedure and assignment of a temporary terminal identifier must also be performed. This is only necessary, however, for the first user on a terminal that performs a Domain Update. We assume this is performed by a terminal session set-up before performing the Domain Update. Finally the location data is updated to reflect the new location. If we assume that there is only one registration of the user on the terminal, there will be one retrieval, two updates, one create and one delete during the Domain Update.

2.2.9 Advice-of-Charge

In Figure 2-15 an Advice-of-Charge information flow is shown, it involves one retrieval of tariff information from the home database (TARIFF) and one retrieval of usage information from the visited database (UM). This retrieval is a scan operation and generates more than one record in the reply and so is slightly more complex than the other retrieval operations of the UMTS database.

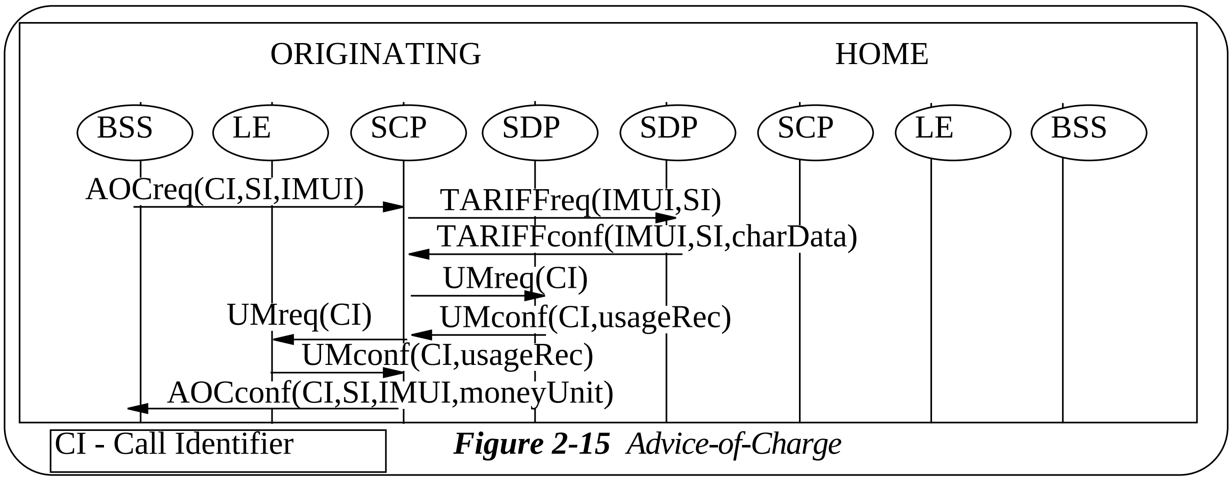


Figure 2-15 *Advice-of-Charge*

2.2.10 Service Profile Management

The user can interrogate and update his service profile with user procedures. Before he gets this access to interrogate and update the database, the network checks that this request is allowed according to the service profile of the user. After this the user can interrogate and update the database according to what is possible according to his service profile. This will then involve one retrieval and a number of user supplied retrievals and updates. Normally there will only be a few accesses by the user when he manages his service profile. Here one access is assumed per service profile management action.

2.2.11 UMTS Characteristics

The most important issue in determining the characteristics of telecom databases used in mobile telecommunication systems is the traffic model. To find the requirements on a specific telecom database, one must also estimate the number of users per database. Current mobile systems have a traffic model which is affected by the high price of mobile communications. In third generation mobile systems the price of communicating with a mobile will most likely be similar to the price of communicating with a fixed terminal today. Therefore calling rates can be estimated by looking at the calling rate in current wireline networks. The calling rate in these networks are 2.8 calls per hour per user (includes both originating and terminating calls) [LATA]. Since the terminals will be mobile and since so many more services are offered through the telecom network, it is estimated that the calling rate will even increase. There will also be call modifications during multimedia calls that will increase the rate of call handling even more. It is estimated to be 4.0 calls and call modifications per hour. We split this into 2.0 originating calls per hour and 2.0 terminating calls per hour per user. This figure is valid during the busy hours of the day.

User session handling must be performed before each UMTS procedure. There could however be instances where one user session is used for several calls and other interactions with the network. Since user sessions are not used in any current network, this figure must be estimated. We estimate it to be 3.0 per hour per user. It is also necessary to have terminal sessions. These sessions can be active for a longer time, and we estimate that there are 1.0 sessions per user per hour.

User registration is also a new procedure and there are no measurements that can be used in estimating the registration rate. It seems probable, however, that users will register a few times a day. Therefore we estimate that there are 0.5 registrations per user during the busy hour and the same number of deregistrations. Attach and detach we estimate to be 0.3 per user during the busy hour.

The number of location updates is dependent on the mobility model. This model depends on many parameters, such as size of location areas, mobility of users and so on. Also the busy hour for location updates does not coincide with the busy hour for calls. Therefore the busy hour figure should not be used, but rather a smaller figure should be used. The rate varies between 0.3 and 2.5 depending on the conditions in [LWB93]. A figure of 2.26 and 2.63 per user per hour is found in a calculation in [MONET109]. We estimate 2.0 per hour per user and of those we estimate that 20% are changing domain (i.e. changing to another operator’s network). This gives 1.6 location updates per hour per user and 0.4 domain updates per hour per user. Finally inter-domain handover is most likely to be very uncommon. We use the figure 0.12 erlang and multiply this by 0.4. It seems likely that domain switches are as likely during calls as otherwise. This gives 0.05 inter-domain handovers

The users will also perform service management. It is estimated to rise compared to today, due to many more services. The rate is estimated at 1.0 per hour per user, equally divided between interrogate, register, erase, activate and deactivate service. Advice-of-Charge is handled in the section on Event Data Services and is not considered in this section.

The number of users per telecom database depends on how many users there are in the network and the policy of the operator. These databases are essential for operation of the network and therefore most operators will require that the system continues to function even if telecom databases are down. This requires network redundancy, that is, another telecom database is always ready to take over if one goes down. So the minimum number of telecom databases in a network is two. This means that theoretically an operator could have 10-20 million users in one telecom database. However even with redundancy, there could be errors that cause the system to fail. Therefore it seems that 1-2 million users is a more appropriate figure. In this way the large operators can also build their systems with more intelligent backup strategies, where several nodes assist in taking over, when a telecom database fails. In our estimates we are therefore using 2 million users per telecom database. From this discussion we reach the figures shown in Table 2-1.

Table 2-1 *UMTS Procedure Rates*

Procedure	Events/user/hour	Total Events / second	Relative Rates (%)
Originating Call Set - Up	2.0	1111	12.0
Terminating Call Set-up	2.0	1111	12.0
Inter-Domain Handover	0.05	28	0.0
Location Update	1.6	889	9.6

Table 2-1 *UMTS Procedure Rates*

Domain Update	0.4	222	2.4
User Registration	0.5	278	3.0
User De-Registration	0.5	278	3.0
Attach	0.3	167	1.8
Detach	0.3	167	1.8
System-Originated User Session Set - Up	1.5	833	9.0
User-Originated User Session Set - Up	1.5	833	9.0
System-Originated Terminal Session Set - Up	0.5	278	3.0
User-Originated Terminal Session Set - Up	0.5	278	3.0
Session Release	4.0	2222	24.0
Interrogate Services	0.2	111	1.2
Register Service	0.2	111	1.2
Erase Service	0.2	111	1.2
Activate Service	0.2	111	1.2
Deactivate Service	0.2	111	1.2
Total	16.65	9250	100

Now by using the figures in Table 2-1 and the information flows from the preceding section we derive the number of retrievals, updates, creates and deletes in the database (see Table 2-2). From these figures it can be seen that modify queries will put a substantial load on the UMTS database, since a modify query is much more complex to execute compared to a read query, which will be seen in forthcoming sections.

Table 2-2 *UMTS Database Accesses*

Procedure	Events/user/hour	Total Events / second	Relative Rates (%)
Retrieval	18.65	10361	53.6
Update	4.95	2750	14.2
Create	5.6	3111	16.1
Delete	5.6	3111	16.1
Total	34.8	19333	100

2.2.12 Conclusion

From the description of the interaction between the SCP (Service Control Point) and the SDP (Service Data Point) it is obvious that there is a rather large amount of interaction between the database and the application for each application message received. The reason is that the database is also used to store session information about the telephone call, and also security information. Thereby the requirements on performance and low delays grow even further compared to the requirements by second generation mobile networks.

It is even possible to visualise a database which contains the actual state of the telephone call. This state variable would need to be updated several times per call. This would create new possibilities in creating services for telecommunications. Active database features could be used to implement most of the telecommunication services.

2.3 News-on-demand

The current WWW-service is envisioned to be similar to how a future multimedia newspaper will look like. It will be possible to read text with images, look at video sequences and have access to large information databases where in-depth explanations can be found for the interested reader. Access to this service could be provided from home computers, TVs, Personal Digital Assistants or some other display terminal new or old. The access could be through fixed terminals or through mobile terminals. Through the News-on-demand service people can have access to the latest news from wherever they are. They will have access to the in-depth stories at the same time, if they need the background to the headline news, or any other news.

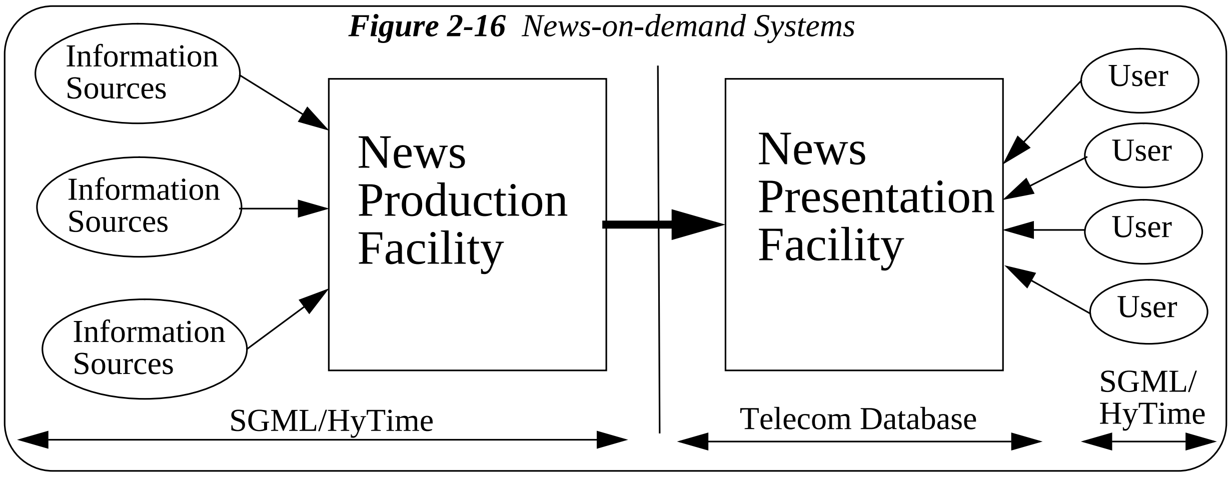
New standards are being developed for multimedia documents with capabilities to contain links to text, images and other documents as well as links to documents with real-time requirements on the presentation of the documents such as audio and video documents. One such standard is HyTime, which is based on SGML (Standard Graphic Markup Language). The documents that use this standards are highly structured. A multimedia document represented in SGML/HyTime is a hierarchical tree of objects. It contains a number of structured objects which consist of other structured objects and can also contain leaf objects. The leaf objects are monomedia objects, such as text, audio, images and video objects.

A News-on-demand system consists of two distinct parts. The first part is the news production facility and the second part is the news presentation facility. In the news production facility the journalists are editing their documents and providing links to other information. Experts in presentation facilities help in developing the multimedia documents so that real-time properties and proper Quality-of-Service are provided. In this part the databases should be flexible, and performance requirements are high and reliability requirements are high. The system could, however, be taken down for service at times and the workload consists more of a few large task rather than many small ones. Therefore the requirements on telecom databases are not applicable in this environment.

The multimedia documents are created in the news presentation facility by a management node that retrieves the document from the news production facility and stores them in the news presentation facility. After this the documents are read-only documents; new versions can be added and indexes can be added and removed from the documents. The main task here is to answer user queries, and the database must be optimised for these queries. This architecture is shown in Figure 2-16.

The most common query is that a user wants to view a multimedia document. This query could of course be such that the whole document is fetched all at once. This means that all video, text, images and audio sequences as well as all the features of the document is retrieved as one very large object. This would be too inflexible and require that information is replicated in many places. Therefore it is foreseen that the query is performed in two steps, where the first step fetches the document description, its synchronisation requirements and all the objects that are part of this document. The second part fetches the parts of the documents, using the document requirements on synchronisation and presentation. This means that many of the objects that are fetched from the

Figure 2-16 *News-on-demand Systems*



database contain an internal structure. The news presentation facility has many similar requirements to the other telecom databases, that is very high performance, very high reliability and also large storage requirements. In [Vittal94], [Vittal95] the conceptual schema of the news-on-demand application has been more extensively studied.

The user of course only specifies the document he wishes to view, and splitting this query into many small queries is performed by the application server. This application server could be a part of the user terminal or part of the network server that handles the user request.

From this we find a simple conceptual schema of the News-on-demand database (Figure 2-17). It contains document objects. These objects contain a set of attributes that describes the document, such as author, date of creation, subject, title, abstract, key words and so forth. It does also contain lists of references to objects connected to the document. The document description itself is also stored as an attribute (basically a Binary Large Object attribute, BLOB). Finally the document object also contains Quality-of-Service parameters. For text-based documents, the document description contains the text object of the document. The database also contains file objects such as image objects, video objects, text objects and audio objects. These objects contain the actual objects in some format, often compressed, such as MPEG, JPEG, GIF, TIFF or any other format. It also contains a set of attributes that describes the object. The descriptive attributes are used when the user issues searches of information. When links are followed, the link specifies the name of the document to fetch.

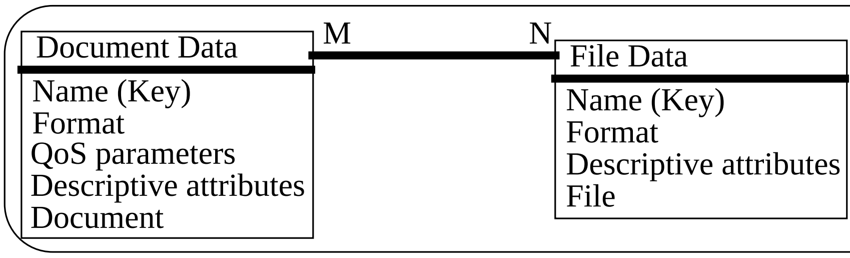
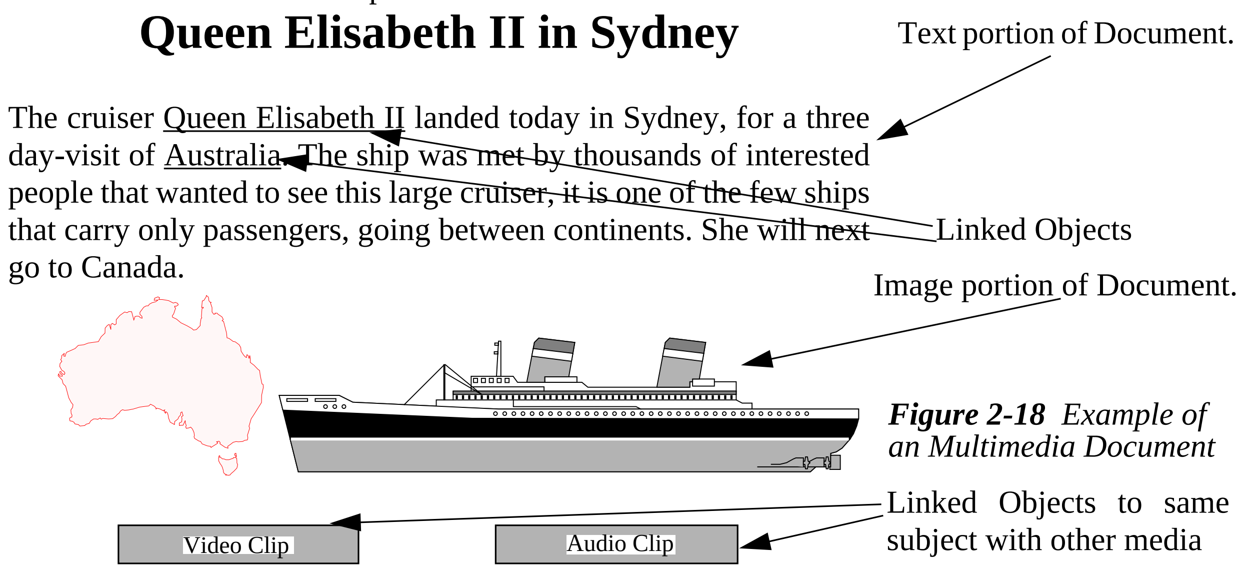


Figure 2-17 *News-on-demand Conceptual Schema in News Presentation Facility*

In Figure 2-18 an example of a multimedia document is shown. The document description would describe the placement of the text object and the image objects, it would contain the text object and references to the image objects. The text would contain the text represented in SGML (or any other standard format); this text would then also contain the links to the documents that describe Queen

Elisabeth II and Australia. The image, showing Australia and the boat, could also contain links to the same documents as the text. The second image also contains links to a video presentation of the same news and an audio presentation of the same news.



Queen Elisabeth II in Sydney

The cruiser Queen Elisabeth II landed today in Sydney, for a three day-visit of Australia. The ship was met by thousands of interested people that wanted to see this large cruiser, it is one of the few ships that carry only passengers, going between continents. She will next go to Canada.

Text portion of Document.
Linked Objects
Image portion of Document.
Linked Objects to same subject with other media
Video Clip
Audio Clip

Figure 2-18 *Example of an Multimedia Document*

In this example, the document description also contains the text object; it also contains the links to the other objects, both the links from the text objects and the image objects. The document could be retrieved by queries that appear thus:

```
SELECT DOCUMENT, FORMAT, QoSParam
FROM DOCUMENT_TABLE
WHERE NAME = document_name;
```

The Document_Format attribute specifies in what format the document description is stored. The client would then process the Document description and issue two queries for the two image objects, these could appear thus:

```
SELECT FILE, FORMAT
FROM FILE_TABLE
WHERE NAME = image_name;
```

2.3.1 News-on-demand Characteristics

Since no trials of a news-on-demand service exist on a large scale, most of the characteristics are arrived at by making estimates based on user behaviour with current technologies.The characteristics of this application are similar to when reading news on paper, watching television news and also similar to reading WWW-pages. News-on-demand is really an interactive use of newspapers, radio news and television news using something similar to WWW as a user interface. Therefore the figures below are based on “guestimates” and are not based on any scientific reports. It is, however, considered valid to perform this anyway since a “guestimate” is better than no estimate at all.

A user is likely to spend fifteen minutes at a time using the news-on-demand service, in the evening a longer time. He will also watch some video clips and sometimes also some audio clips. The video and audio clips are normally thirty seconds to five minutes long. It is likely that many users will scan through news documents to watch for interesting news particular to the user’s interests; then he will use video clips for in-depth information and information where video is the best presentation media.

When reading a newspaper most readers are likely to scan through most of the pages in the newspaper. By doing this they read about twenty to thirty pages in fifteen minutes. It is likely that the documents displayed in a user terminal are somewhat smaller than newspaper pages and so there will be about fifty interactions between the user and the user terminal during fifteen minutes. In addition now the user can also watch video clips of the news. It is likely that video clips will be more heavily used during the evening.

Many users interact with the news-on-demand service in the morning, after work and during the evening. During the evening the interaction could be a little longer, possibly half an hour. Now of course not everybody interacts so heavily, so we estimate that a normal user who has registered for use of the news-on-demand service interacts half an hour per day. This activity is then spread out, 25% in the morning, 25% after work and 50% in the evening. During the morning and after work he is likely to use a mobile device and he is likely to be using the news-on-demand service like reading a newspaper. In the evening the user is more likely to use a fixed terminal at home where he will watch more video clips. He will not be as interactive as during the morning and after work.

This half an hour is normally divided into four sessions. Most of these sessions will be in the “busy hour” in the morning, after work and at the evening. Thus we estimate that 0.8 sessions are started per user per hour in the busy hour.

The size of the documents and their images will vary largely; the document description and text of a document is mostly rather small, around 2 kByte. The images are likely to be between 10-200 kByte. A normal document is estimated at 2 kByte with two figures with a mean size of 50 kByte each. The user terminal could of course make requests for larger chunks than one document at a time. This would decrease the number of requests to the telecom database. It would, however, increase the bandwidth need since the user does not normally reuse any cached information and there would be information prefetched that would not be used, thereby bandwidth is wasted. Whether bandwidth or the possibility to handle user requests is the bottleneck in the future is not known. It is likely that for mobile terminals low bandwidth is more essential. With fixed terminals the bandwidth is likely to be less of an issue.

Two models are introduced (Table 2-3), the mobile model and the fixed model. The mobile model assumes a user who is connected through a mobile device. The user mostly reads documents and is very interactive with the news-on-demand service. The user always interacts with the telecom database in each request to save bandwidth and avoid unnecessary file transfers. The number of video clips is small in this model and the video clips used have a lower quality, to save bandwidth, than those that are used in the fixed model. We assume that video clips are transmitted at 128 kbit/sec in the mobile model. The reader watches two and a half minute of video clips in a fifteen minute session.

The fixed model is where the user is connected through a fixed terminal, most likely connected to his television set. The number of video clips is much higher and there is less interactivity. The video clips are transmitted at 2 Mbit/sec in this model and the user spends twenty minutes out of thirty minutes watching video clips.

Table 2-3 *News-on-demand characteristics*

Busy hour behaviour	Mobile Model	Fixed Model
User sessions	0.8/h	0.8/h
User interactions	20/h	10/h
Telecom Database interactions	60/h	30/h
Non-video bandwidth per user	2 MByte/h	1 MByte/h
Video bandwidth per user	1 MByte/h	60 MByte/h
Video clips	0.5/h	2.0/h

From these models we can derive that the mobile model puts heavy requirements on response times in the telecom database. It also puts high requirements on message processing capability. The need for bandwidth is, however, much less than in the fixed model. The fixed model requirements is similar to the requirements on a video server. There is less interactivity (although higher than in a video server) and bandwidth requirements are twenty times higher than in the mobile model. We derive the figures stated in Table 2-4. It is obviously a very tough requirement to handle the video bandwidth in the fixed model.

Table 2-4 *News-on-Demand Database Characteristics*

Procedure	Events/user/hour, Bandwidth/user/hour	Total Events / second Total Bandwidth/sec
Retrievals Document Table, Fixed Model	10	5556
Retrievals File Table, Fixed Model	20	11111
Bandwidth Requirement, Fixed Model	61 MByte	33.9 GByte/sec
Retrievals Document Table, Mobile Model	20	11111
Retrievals File Table, Mobile Model	40	22222
Bandwidth Requirement, Mobile Model	61 MByte	1.67 GByte/sec

From this discussion it is easy to see that the news reader will have very high requirements on response times. If the response time is more than half a second the user is likely to be bothered. For video clips the user is likely to accept a little longer delay, 1-2 seconds before start-up. Therefore the response time of the telecom database must be less than 100 ms since there are delays in the network and in the user terminal, too.

A conclusion regarding the mobile model is that bandwidth-on-demand will be required. The bandwidth must be up to 2 Mbit/sec to be able to transmit documents of 100 kByte size in half a second. This is what the requirements on UMTS were originally and it should be possible to achieve this bandwidth, at least where the radio cells are small enough (in densely populated areas).

The information in a News-on-demand database can be cached from another information server. This affects the load that the various information servers have to handle, but not the query mix.

It is likely that most of the news read by users is today’s news. Therefore the benchmark focuses on browsing through today’s news. A normal newspaper is from fifty to one hundred pages long, since it will also be combined with television news and radio news. There will also be an hour of video clips and an hour audio clips. It is likely that the newspaper will become bigger and have more resources available than currently and so the size of today’s news-on-demand. It will probably be a bit bigger than normal newspapers and television news.

From this short discussion we “guestimate” a model with the daily news-on-demand consisting of 1,000 articles (documents), 30 video clips and 30 audio clips.

The articles consists of one document part and from zero up to ten images. The size of the document part is between 200-300 bytes up to several kBytes while the mean is 10 kByte. The images vary in size from 1 kB up to several hundred kBytes. The mean is 50 kByte and the mean number of images is 2. The audio and video clips are between thirty seconds and five minutes long and the mean time is two minutes. The video clips are stored both in a form such that both 128 kbit/sec and 2 Mbit/sec transfer rates are supported. The audio clips are stored in a form such that the transfer rate is 64 kbit/sec.

From these figures an interesting conclusion can be drawn. Although the transfer rate from the telecom database is high, the storage need is rather small; the document part consists of 110 MByte, the video part of 58 MByte plus 900 MByte, and the audio part 29 MByte. In total a storage need of 1.1 GByte, which clearly fits in the main memory of the telecom database.

In a real news-on-demand database there should of course also be connections to video archives, document archives and various other information sources. These information sources would of course require a much higher storage volume. These could, however, be a part of another telecom database in a distributed information network.

2.4 Multimedia Email

Most people at work already use electronic mail systems. It helps people stay in contact, even when a person is difficult to reach on the phone. It also provides a way to share information, by sending documents of various types. With the explosion of Internet in recent years this service is also heavily used by private persons. It can solve the communication problem of small organisations, with a limited budget, where it can be very difficult to inform everybody (e.g. about a change in schedule) using current techniques. Using email, a broadcast mail to a mail group, can be used to inform everybody, easily and efficiently. For small organisations and home users, it is necessary that servers are available, which can be subscribed to. It is very expensive to have a server always connected to the Internet that can send and receive emails. This is only affordable for large organisations. Large organisations can also have problems sometimes, e.g. when their servers fail, when a software change is needed and so forth. Therefore, there is definitely a large market for Multimedia email for service providers and network operators. The network operators or service provider then supplies an email server which is reliable and can store large pieces of information. It can then be accessed whenever the user wants to read, create, delete or send emails.

The email service requires high reliability. No user is likely to accept that his emails get lost. It is also crucial that the email service always works, even in a catastrophic situation.

When building up a database of emails it is necessary to have a table of the various emails that exist in the database. In this table the receiver, the sender, the title of the email, the time it was received and an identifier of the email is stored. Also a user-defined category is part of the email object. Each email is an object in this table. The object also contains a list of objects attached to the email and the properties of these attached objects. The attachments could be text files, HyTime documents, video files, the properties of these attached objects are stored as part of the attached object, see Figure 2-19.

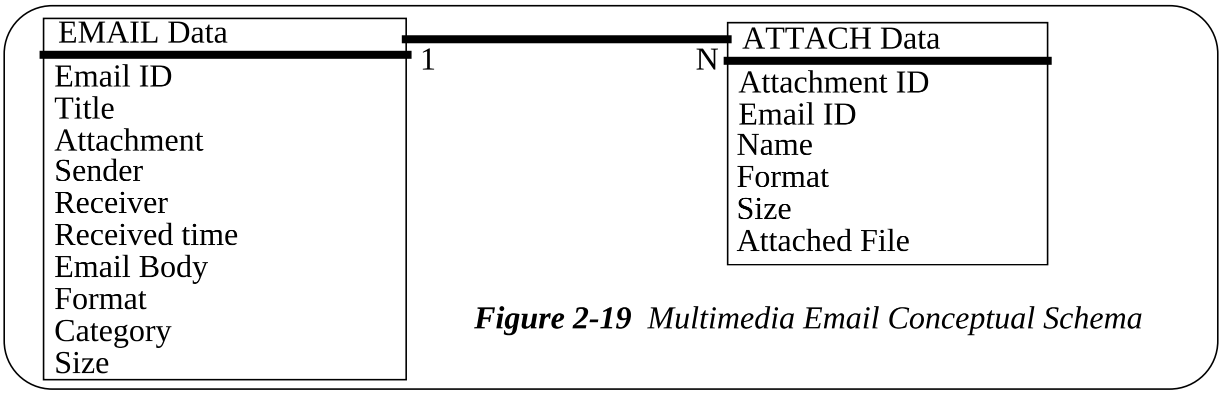


Figure 2-19 *Multimedia Email Conceptual Schema*

The major usage of an email service are:

- Display the email titles in a user-defined email folder
- Display an email and a list of its attachments
- Create a new email
- Delete an email
- Show an attachment (e.g. play a video, display text)
- Send and receive emails

From these usage it is easy to find the query functionality that an email server should be optimised for. The first query that displays the emails of a specified folder is normally used when connecting to the email service, in SQL this query could look like this:

```
SELECT SENDER, TITLE, RECEIVE_TIME, EMAIL_ID
FROM EMAIL_TABLE
WHERE RECEIVER = user FOLDER = folder;
```

The response to this message contains a set of all emails of the specified category that the user has in his mailbox. The second query that is used when reading an email consists of two parts. The first part retrieves the email and checks whether there are any attachments:

```
SELECT EMAIL_BODY, EMAIL_PROPERTY, ATTACHMENTS
FROM EMAIL_TABLE
WHERE EMAIL_ID = email_id;
```

If there are attachments another query retrieves the properties of the attachments:

```
SELECT ATTACH_ID, ATTACH_PROPERTY, ATTACH_NAME
FROM ATTACH_TABLE
WHERE EMAIL_ID = email_id;
```

The response to these queries contains the email body with its properties and a set of attachment properties, their id and name.

The third query is used when sending and receiving emails. In these procedures the email and its attachments are created. This is accomplished by a create transaction that could appear thus:

```
START TRANSACTION;

INSERT INTO EMAIL_TABLE
WHERE EMAIL_BODY = email_body, FORMAT = format,
SENDER = sender, RECEIVER = receiver, TITLE = title, FORMAT = email_format
CATEGORY = category, SIZE = size, ATTACHMENT = attachment;

INSERT INTO ATTACH_TABLE (one for each attachment)
WHERE EMAIL_ID = email_id, FORMAT = attach_format,
NAME = attach_name, ATTACHED_FILE = attach_file;

COMMIT TRANSACTION;
```

To delete an email is easy, one simply deletes the email from the email table and the attachments from the attachment table. This could appear thus:

```
START TRANSACTION;
DELETE FROM EMAIL_TABLE
WHERE EMAIL_ID = email_id;
DELETE FROM ATTACH_TABLE
WHERE EMAIL_ID = email_id;
COMMIT TRANSACTION;
```

When showing an attachment, the attached file is needed; this is fetched with a query that could appear thus:

```
SELECT ATTACH_FILE
FROM ATTACH_TABLE
WHERE ATTACH_ID = attach_id;
```

Finally sending and receiving emails is performed by the email server using some network protocol, such as extended SMTP (Simple Mail Transfer Protocol). During the receiving activity, emails are created. When sending an email, the email and all its attachments are fetched from the database and sent to the receiving site.

The description of the email service is based on personal use of a normal email-tool and the assumption that future email services will be very similar to current email-tools.

2.4.1 Multimedia Email Characteristics

Email is a service that is heavily used today and is likely to be even more heavily used in the future. The major problem is that the use of email is heavily dependent on the user’s maturity in using computers in his work and also on culture in user behaviour. In [Pax94a] there is, however, a study of the development of Internet use. In this report it is found that the usage of email in wide area connections is increasing 30% per year per person; in 1994 there were about three email wide-area connections per day per user and the report also shows that the size of the email is growing. The mean size of emails was around 2 kByte in 1994. If this growth trend continues, we will reach 12 mails per day in 2002 and additionally the report did not include local email connections, thereby increasing email usage even more. From this it is easy to see that it is more or less impossible to predict the future behaviour of users of email. Our presentation is rather based on a model where the users behave as today with ordinary mail, supplemented by the possibility to use email for fast communication when a person is not reachable.

A large part of the current mail system is used to distribute advertisements for various products. Much of the email traffic today is also generated by email reflectors and from email-lists. Thereby it is fairly certain that a major part of the email traffic will be broadcast traffic. From this it is easy to see that normal users will receive many more emails than they send themselves. However email will also be used for person-to-person communication where currently answering machines, voice-mail, short message services (mobiles) and faxes are mostly used. The Multimedia email service is likely to include those services where voice messages, fax messages, text messages and even video messages are all a part of the email service.

There are two usages of emails that are common today: the first is to transfer small text messages, the other is to transfer files as attachments. The size of the text messages is likely to be rather constant; the size and the number of files transferred using Multimedia email is, however, likely to increase for a long time ahead as voice messages, file transfers, video messages and fax messages becomes a more common type of usage of the email service.

The use of email is such that the user will perform email sessions now and then, where he will check what email he has received and send emails. These sessions will occur several times a day mostly; we "guestimate" that in the mean there will be 0.5 email sessions per user per hour in the busy hour. Business users always use communication services more than private persons, in this thesis we do not distinguish between them, we treat the community as one entity.

A person is likely to receive something like twenty emails per day and send about five emails. In the busy hour this would be 2.0 emails received and 0.5 emails sent, that is one email is sent normally during a user session and four emails are received. From this we derive the figures in Table 2-5.

Table 2-5 *Multimedia Email Procedure Rates*

Procedure	Events/user/hour	Total Events / second	Relative Rates (%)
Connect Email session	0.5	278	10
Read an Email	2.0	1111	40
Send an Email	0.5	278	10
Receive an Email	2.0	1111	40
Delete an Email	2.0	1111	40
Total	7.0	3889	100

The reason that more emails are deleted than are sent is that it is possible to broadcast emails; when sending an email, this is also reflected in that there are many more deletions of emails than there are creation of emails.

When connecting an email session a scan of the email table is performed so that the email titles are displayed to the user. When reading an email, the email body is read and a scan of the attachment table is performed if there are any attachments. Sending an email involves creating an email and its attachments, then sending it through a communication channel to the receivers. Receiving an email creates an email and its attachments in the database and deleting emails deletes both the email from the email table, as well as the attachments from the attachment table.

The current size of emails is about 2kByte [Pax94b]. Since this will grow slowly and also voice-mail, fax-mail and other file transfers are likely to increase, we estimate the mean size of email messages to be 20 kByte. The number of email messages that is stored in a user's mailbox is likely to be highly dependent on the user. Most likely the user will pay extra for a larger storage capacity of his mailbox. We "guestimate" that the normal user will have fifty messages stored in his mail-box. From these estimates we derive the figures in Table 2-6. The total is based on the same number of users as in the UMTS database, that is, 2 million users.

Table 2-6 *Multimedia Email Bandwidth and Storage*

2 Million users	Per User	Total
Input Bandwidth	10 kByte/h	5.6 MByte/sec
Output Bandwidth	40 kByte/h	22.2 MByte/sec
Storage Volume	1 MByte	2 TByte

To derive the access rates on the database for the email service, it is also necessary to estimate the number of attachments that there are to an email. We "guestimate" that there are attachments to 40% of the emails and most of the time there is only one but the mean number of attachments we estimate at 0.5 per email. From this we derive the figures in Table 2-7.

Table 2-7 *Multimedia Email Database Accesses*

	Events/user/hour	Total Events/sec	Relative Rates(%)
Retrieve email	2.0	1111	18.1
Retrieve attachments	1.0	556	9.1
Scan email table	0.5	278	4.5
Scan attachment table	0.8	444	7.2
Create email	2.5	1389	22.6
Create attachment	1.25	694	11.3
Delete email	2.0	1111	18.1
Delete attachment	1.0	556	9.1
Total	11.05	6139	100

2.5 Event Data Services

In all applications mentioned so far it is necessary to bill the services. Current billing procedures used by most telecom operators provide a bill at some predefined time. In the future information network it must also be possible to pay when the service is used, in the same way as is done in shops. It should also be possible to tell the system that now I want to pay the bill, in the same way as at restaurants. It must also be possible to pay with electronic cash, to pay with credit cards or any other method of paying. To make this possible, it is necessary to audit what the users are doing in the network. This information must also be immediately available to the billing system, so that on-line billing becomes possible.

This information can also be used to get information on network usage and in many other ways. It is therefore necessary to regulate what information can be legally gathered and what information the users must provide to use the network. It will be used by systems for market research, network planning, operation and maintenance and by regulatory organisations.

The information gathered is defined in event records which are created in the nodes where the event is happening. The event records are then sent to a Charging DataBase that collects all usage records and also transfers them to all parties that need access to them. This database can then both be used to transfer the records, but the records are also stored and therefore database queries can also be issued on the stored records. This particular database does not update the information in the database, information can only be added. At times, however, old event records are deleted and possibly saved in some archive. The Charging DataBase is then another telecom database, so the presentation here will focus on the requirements of the Charging DataBase.

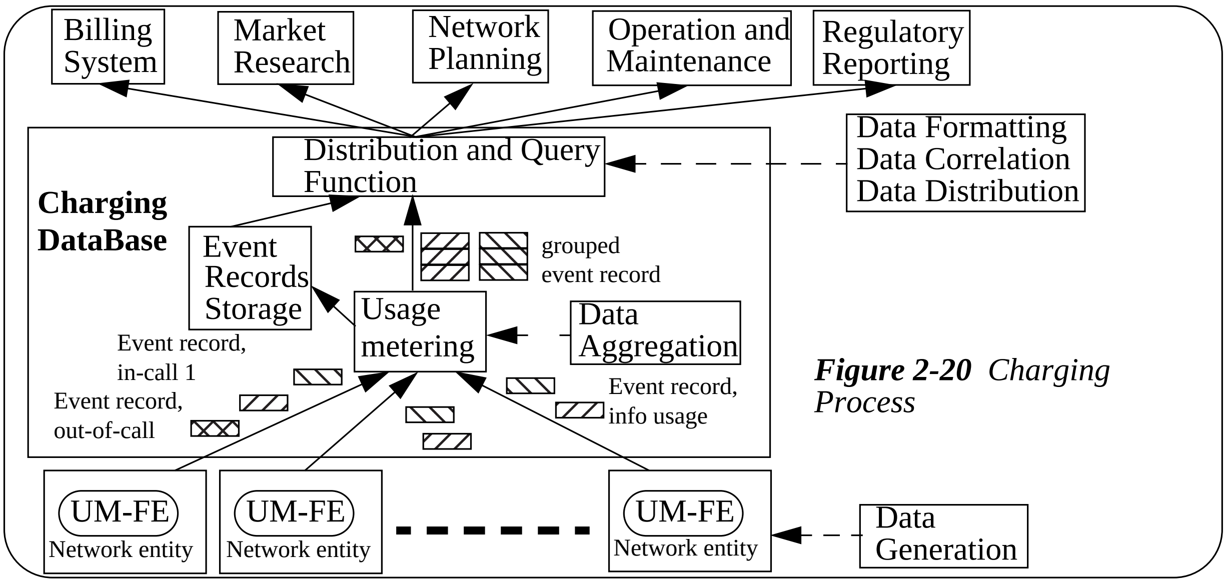


Figure 2-20 *Charging Process*

The flow of information in Figure 2-20 is in accordance with CCITT Recommendation X.742, where the accounting process consists of three processes:

i) A number of usage metering functions, where these functions generate the event records.

ii) A charging process that formats, distributes and collects event records together to form service records and tariffed service records.

iii) A billing process that uses the information provided by the charging process to create the bills.

This architecture was adopted in the RACE MONET project and it seems like a good candidate for the accounting process in third generation mobile systems. A very similar process has also been adopted in DAVIC 1.1 (Digital Audio-Visual Council) where a standard for video-on-demand and other information services has been developed.

In Figure 2-21 a detailed description of the charging process for UMTS as proposed by RACE MONET is shown. The events are collected in the visited network; they are then sorted and formatted before they are transferred to the home operators charging process. At the home operator, the records are associated to user events and stored as service records. Then these records are used together with tariff data to create tariffed service records. These records can then be transferred to the billing process of the home operator.

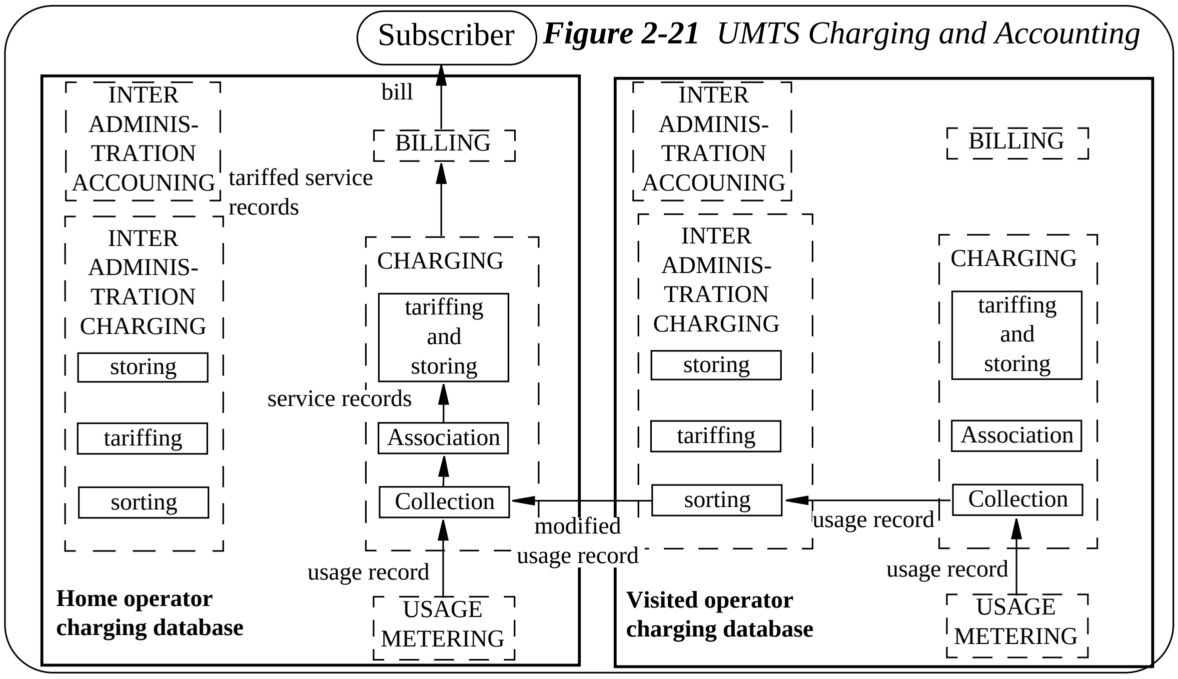


Figure 2-21 *UMTS Charging and Accounting*

2.5.1 UMTS Event Records

In [MONET053] the following events have been identified:

- Mobile originated call record, generated when a call has been terminated at the originating side.
- Mobile terminated call record, generated when a call has terminated at the terminating side.
- Bearer usage record, which specify the bearer resources that have been used by a call.
- Emergency call record.
- Feature usage record, is used for metering UMTS procedures such as registration, deregistration, session management, location and domain updating.
- Supplementary Service record, is used when supplementary services are invoked during a call, such as call waiting.

- Value-added service record, is used when value-added services are invoked during the call, such as a directory service.
- UPT service record, used when a UPT user was involved in a UMTS call.
- Management service record, used when the user invoked a service management function.
- IN service record, used when the call used IN services, such as a freephone call.
- Handover record, used at handover. Not all handovers are metered however, as this would be too much information. Therefore the operator must decide which handovers should be metered.
- Incoming/outgoing gateway record, used when coming into a new network or leaving a network.

The event records should contain a reference to the user and the subscriber of the event. Events that are part of a call or another type of service should have a call identity or service invocation identity so that they can be grouped into service records. There should also be a reference to the network entity that generated the record. A list of possible parameters is given in [MONET053].

2.5.2 News-on-demand Event Records

The events in the News-on-demand application seem rather simple to specify. The news provider would like to know which documents the readers are interested in and therefore each time a document is fetched a usage record is created. Also when performing searches in the database, an event record should also be created. All event records are part of a user service, e.g. reading the newspaper. There is one such service record created per user session.

2.5.3 Email Event Records

Email events that definitely should be recorded are when emails are sent and received. It is also likely that each interaction with the email server also should generate an event record. Exactly which events to record is the decision of the operator. It is likely that creating and sending emails is one service and that connecting to the email server is another service that generates service records based on what has been performed during the email session.

2.5.4 Characteristics of the Event Data Service

We will analyse the flow of event records for the UMTS application, the news-on-demand and the Multimedia email application. The UMTS application generates an event record each time a UMTS procedure is invoked. Also event records are generated that specify the use of network resources, supplementary services, emergency calls, UPT services, IN services and value-added services. Also call release generates an event record that specifies the duration and the stop time of the call. For news-on-demand and Multimedia email we assume that all user accesses to the server generates an event record. The usage models of the previous sections are used in deriving the characteristics of the Charging Database. The extra information we need in the UMTS case is shown in Table 2-8.

Table 2-8 *UMTS Procedure Rates*

Procedure	Events/user/hour	Total Events / second
Network Resource Usage	6.0	3333
UPT Service	0.5	278
Advice-of-Charge	1.0	556
Supplementary/Value added Services	1.0	556

There is a process in the Charging Database that correlates the usage and service records according to various needs. All records pertaining to a call, a UMTS session, an email session or a news-on-demand session are brought together and stored as one service record. These records are then stored in the Charging Database so that external systems can perform queries on the usage information.

There is another process that formats usage records and service records for various needs. These formatted records are then distributed to external systems that have requested to be on the distribution list of these records.

The grouping of event records is triggered by events that specify that a service has been completed. As part of this process, a tariff is also applied on the service record, thereby producing a tariffed service record. This record can then be used for both on-line and off-line billing.

The formatting is performed as records are produced and then the formatted records are distributed. We assume that each event record, each service record and each tariffed service record produced is formatted once and distributed to external systems.

One way to avoid large query transactions each time a user wants to calculate his bill or get an advice-of-charge is to calculate the bill incrementally, this means that it is updated each time a set of service records have been produced. In the following we will assume that this is performed by the application. It could also be performed by having triggers in the database.

There are four types of transactions in a Charging Database. The first type is storage of event records. Some of these transactions trigger another type of transaction. This transaction formats event records and distributes the formatted records, groups event records and updates billing information (and possibly other information requested by the application) for a user. The charging process and the relation between these two transaction types are shown in Figure 2-22.

There is a problem when event records are stored. These are likely to be sent to disk and erased from main memory eventually. Since there could be a long time interval between the first event records and the end of a user session, these event records might have been sent to disk when the user session or call is finished. To avoid this the event records are stored in two tables. The first table represents the original, which is stored on disk eventually. Actually the records are not likely to be read any sooner, so they can be sent to disk as soon as possible. The second table contains similar information and is stored in main memory. This record is deleted as soon as the charging process is finished. This makes sure that the queries from the charging process are always per-

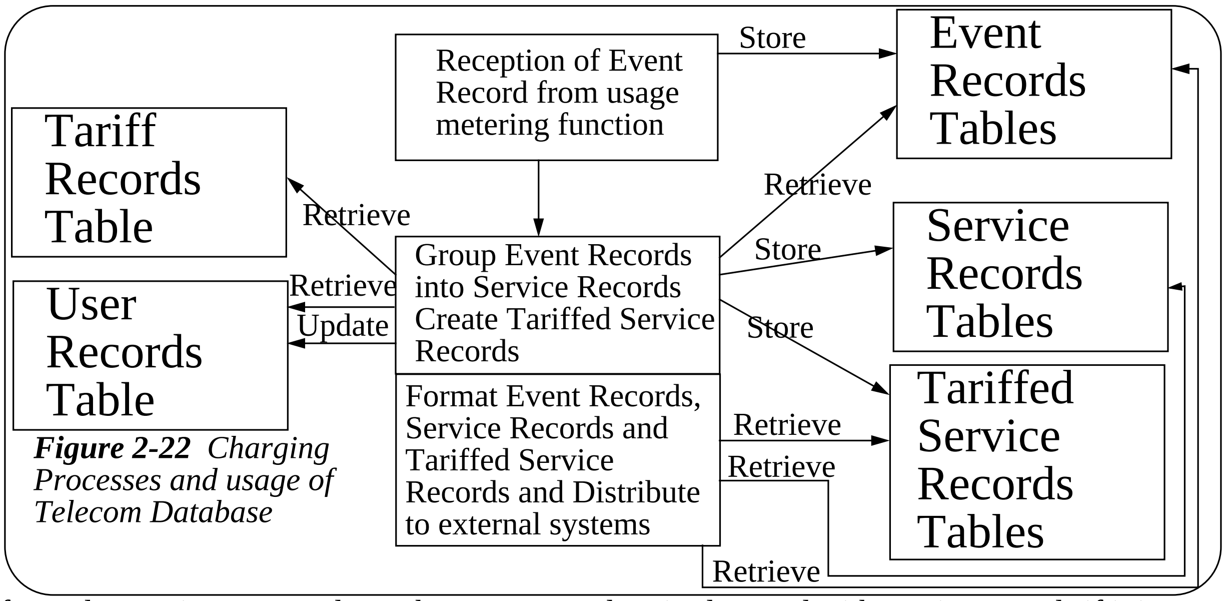

Reception of Event Record from usage metering function
Store
Event Records Tables
Tariff Records Table
Retrieve
Retrieve
Group Event Records into Service Records Create Tariffed Service Records
Store
Service Records Tables
Retrieve
Update
Store
User Records Table
Format Event Records, Service Records and Tariffed Service Records and Distribute to external systems
Retrieve
Retrieve
Tariffed Service Records Tables
Retrieve

Figure 2-22 *Charging Processes and usage of Telecom Database*

formed on main memory data. The same procedure is also used with service records if it is necessary to save them for a later process. In our estimates we do, however, assume that the grouping transaction and the formatting transactions are all part of the same transaction and thereby there is no need to save them in the database. The information can be held in the local variables of the application.

The fourth type of transaction issues queries to calculate the bill and perform advice-of-charge. This transaction is a simple read of the user record since the information is produced as part of the charging process. The fifth type is an uncommon type of transaction that queries the user records, tariff records, event records, service records and tariffed service records. This is used by management systems to perform data mining on the Charging Database for various purposes.

Delete transactions occur, but occur more on a regular basis. Normally records are kept for about three months and then they are deleted. Therefore we will not provide any characteristics on deletions, only on retrievals, updates and creations. All updates are also on the User Records Table, the event records are inserted and sometimes retrieved and finally deleted.

The updates on the User Records Table are normally performed by applying a function on the tuple. If the database supports sending methods, then a method is sent to the database with the proper input. Otherwise a retrieve followed by an update is performed.

To calculate the characteristics of the charging database we need to analyse what happens in the above-mentioned transactions. All events lead to the creation of two records of which one is deleted when the user session or call is ended. Each time a user session, service or call is ended a service record and a tariffed service record are created. Also the event records table in main memory is scanned for all event records belonging to the user session, service or call. Then an update of the user table is performed and a retrieve of tariff information is performed. This gives one retrieval, one scan, one update and two create operations at the end of user sessions, services and calls. If there is only one event record for a service, then no scan operation is needed.

For the UMTS application the following actions triggers the end of session actions: Call Release, Inter-Domain Handover, Location Update, Domain Update, User Registration, User Deregistration, Attach, Detach, Interrogate Service, Register Service, Erase Service, Activate Service, Deactivate Service, UPT Service and Supplementary Services. All except call release only contain one event record.

The derived characteristics of the UMTS, news-on-demand and Multimedia email Charging Database are shown in Table 2-9.

Table 2-9 *Charging Database characteristics*

Application (2 million users)	Retrievals/sec	Scans/sec	Updates/sec	Deletes/sec	Main Memory Creates/ sec	Disk Creates/sec
UMTS	8694	2222	8139	14806	14806	31083
News-on-demand	444	444	444	11111	11111	12100
Multimedia email	556	278	556	3889	3889	5000
Total	9694	2944	9139	29806	29806	48183

2.6 Genealogy Application

A genealogy application uses historical records to trace our ancestors and their families. There is an immense amount of historical records to search. Currently this is done by reading the historical records directly or by reading a microfilmed copy of the record. This means that the genealogist needs to visit an archive or a place where he can read microfilms to do the genealogy. Also it is difficult to trace what other genealogists have found, when searching the records. The forthcoming storage and database technology will revolutionise this tedious research.

There are six groups of objects in this application. The first group comprises the historical records that have been scanned, either directly or from a microfilm. This group also contains refinements of those objects where computers have been used to make the images clearer, to increase the sharpness, brightness and the contrast, possibly also adding colours to make the text more visible. This means that there could be several versions of the scanned objects. It is not desirable that it should be possible to delete these objects, especially not the original scanned objects.

This represents an immense storage volume. Already today there exist millions of microfilms of historical records, each containing about one-thousand images normally of A3-format. One such image needs around 1 MB of storage in compressed form. The image needs to have very high resolution since it represents hand-written text. This text is sometimes very difficult to read, and therefore it is necessary that the parts of the image can be magnified to make it more readable.

Also very many records have not yet been microfilmed. This object class will therefore constantly increase in storage volume, both to store new originals and new copies. Already from the start this will therefore need petabyte, that is thousands of terabytes or millions of gigabytes or billions of megabytes. This will then increase as more and more records are added to this historical database.

The next class of objects are the catalog objects. These objects specify the historical records, and what they represent. This is where the researcher goes when he wants to see what historical records exist in the area of his ancestors.

The third class of objects are transcriptions of the scanned objects. This means that a person or a computer program has read the text and made a note of what was written. There are also many versions of the transcribed objects. Each transcription can exist in many versions; first the original, where one simply tries to read character by character what is written; another version could be where older words have been translated into modern words; yet another version could be where the whole text has been translated into modern language and of course there could be translations to other languages as well. Also there could of course be many transcriptions of the same text. These need not be the same since mistakes could be made, especially when the text is difficult to read.

The fourth class of objects comprises the application objects which represent the information that is actually sought. In genealogy this means information about births, christenings, marriages, deaths and burials, as well as other information that can be used to link persons together into families. There could also be several versions of these objects, representing the findings of different genealogists. Of course the above information could be used for many other applications apart from genealogy. One such application is historical research. Other applications have other requirements on the application objects. In this section we will however focus on the genealogy application.

The fifth class of objects comprises link objects which are used to link all the information together. A transcribed object needs to be linked to the original and possibly also to the version from which the transcription was made. Application objects needs to be linked to all sources (scanned objects as well as transcribed objects and even other application objects).

Now a scanned object normally represents a page (or two) in a historical record. This page consists of a number of information objects. In a book of births there are information objects on the pages that represent births (or christenings). In a book of marriages there are information objects on the pages that represent marriages. In a tax record, the information objects on the pages represent taxes paid (or to be paid). Each of these information objects can now be mapped onto scanned objects. This mapping is represented by the set of scanned objects the information is located in and the area of these scanned objects that the information object consists of. Now this information object can now be linked to various objects. It can be linked to transcriptions of the information object, application objects connected to the information object.

As an example, a birth record is normally a short item, where the name of the child, the birthplace, the name of the father and mother is given. Sometimes there is more information and sometimes there is less information. Now when this birth record is transcribed, a transcription object is created and when somebody finds that this is a relative, he makes an application object, where data about this person is collected and information about his parents, spouses and children can be found. This application object can now be linked to the information object. From this information object, the transcribed objects can be found and from the application object there can also be links to other information objects that is used in collecting the data about this person and his family connections.

The sixth type of object comprises name translation objects. These are necessary to provide translation to standardised names. In historical records it is not uncommon that the same name can be spelled in many ways. An example is Margareta that can be spelled in at least the following ways in Swedish records: Margareta, Margaretha, Marget, Margeth, Maret, Mareth, Mareta, Maretha, Margreta, Margretha, Marit, Marith, Märet, Märeth, Margit, Margith. All these spellings represent the same name and can even be used in different records to represent the same person. The different versions of names can differ in different geographical areas. Also provided in the name translation object could be information on the usage of that name, where it is common, history of the name and so forth. The different objects are shown in Figure 2-23.

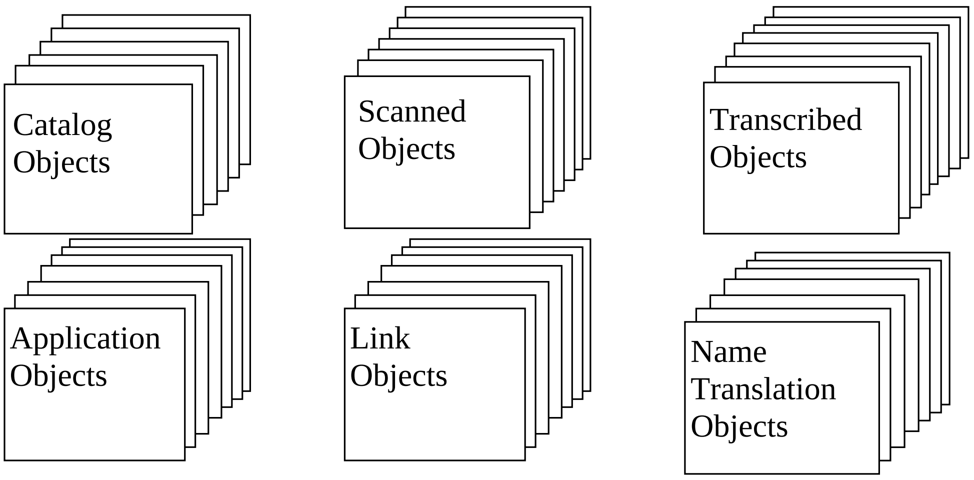


Figure 2-23 *Object Classes*

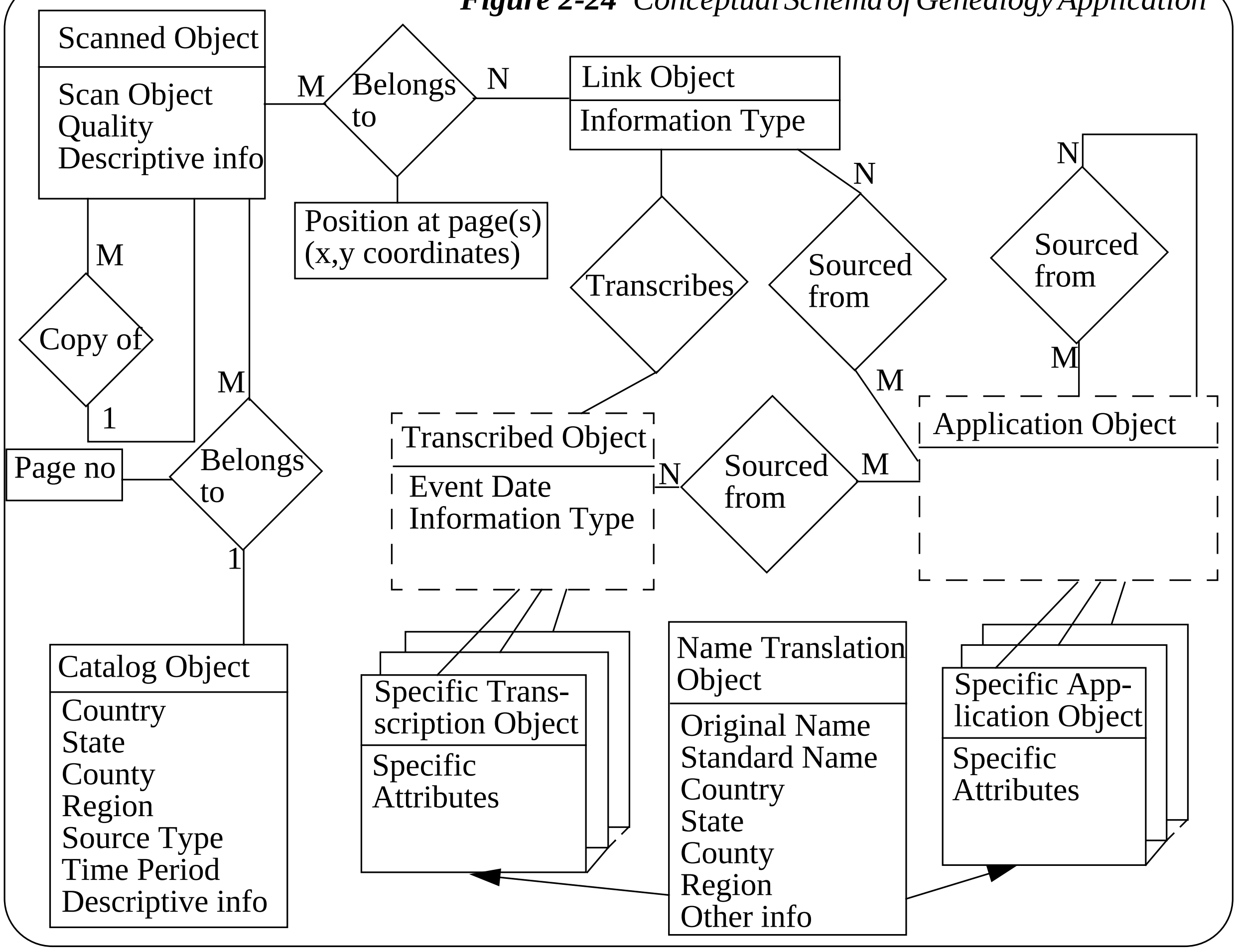
When a genealogists works on these objects, there are many different ways to use the information. The scanned objects are used to derive new information which is fetched one object at a time, the transcribed objects and the application objects are often searched for the sought information.

There are many interesting challenges in this application for telecom databases. It represents an application with a vast need of storage. Current technology cannot satisfy the requirement; possibly around 2005 these requirements can be met at the right price level. It also represents an application where one needs object-oriented technology to represent the information, relational database technology for efficient searches, deductive database technology to assist those who are not specialists in genealogy to also use this technology efficiently. There is also a need for fuzzy queries since names were not standardised. Also solutions in Multidatabase technology are needed to enable queries with similar sources where all sources do not contain the same information and where the same information is represented in different formats. It also represents an interesting application, where millions of people can work on the same application, using a distributed database that is global. It also needs a versioning system; this database will mostly grow, information can possibly be marked invalid, however very seldom entirely deleted.

2.6.1 Conceptual Schema

In Figure 2-24 a description of the conceptual schema of the Genealogy application is given. The scanned objects have relations to the catalog objects, a number of scanned objects represents the information covered by a catalog object. Each scanned object is a page of a catalog object. Copies of the original scanned objects have a relation to their original object. The scanned objects consist of a number of information objects; these are represented by a link object and transcribed objects. A typical information object could be information about a birth, death or marriage. There could also be information on court cases, taxes, letters and many other types of information objects. These information objects are normally short and fit into one page. Sometimes however they span several pages, such as a court case.

Figure 2-24 Conceptual Schema of Genealogy Application



The link object contains information about where on the pages that the information objects can be found as seen in Figure 2-25. The link object specifies which part of the page that is covered by the linked object. This could be specified by the x and y coordinates of the upper left corner and the length on the x-axis and the length of the y-axis.

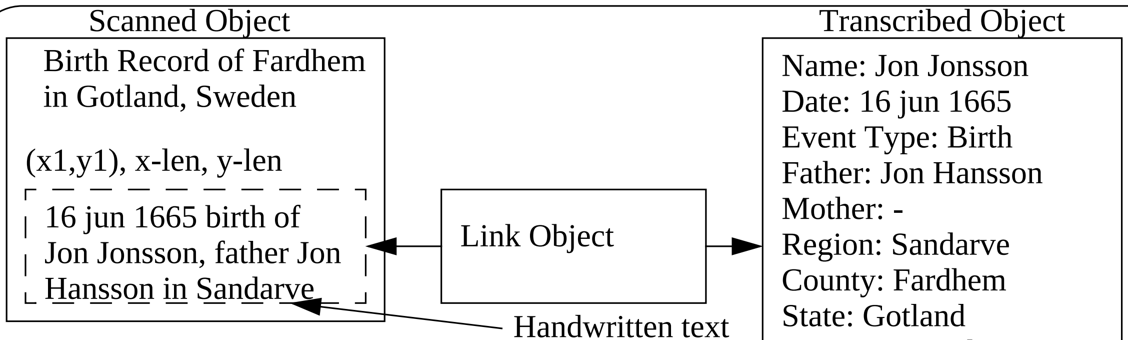


Figure 2-25 *Example of relations between Scanned Objects, Linked Objects and Transcribed Objects*

The information objects can be transcribed into text objects that are machine readable and can be used in search queries. This translation process can also be supported by software that helps in interpreting the handwritten text. This is, however, an application issue not dealt by the database. The transcribed objects and the scanned objects are then used as sources to find the information to store in the application objects. A more detailed description of the relations between the application objects are found in Figure 2-26. The basic application object is an object describing a person. A person has many relations; There are marriage relations, and there are relations to parents and their are relations to children. Other relations such as foster parents, adoptive parents and so forth are also possible. Names in transcribed objects and application objects are often given in the form that they were written in an original document. Many names with similar spellings represent the same name. Therefore there are name translation objects that translate the original names to a standardised name.

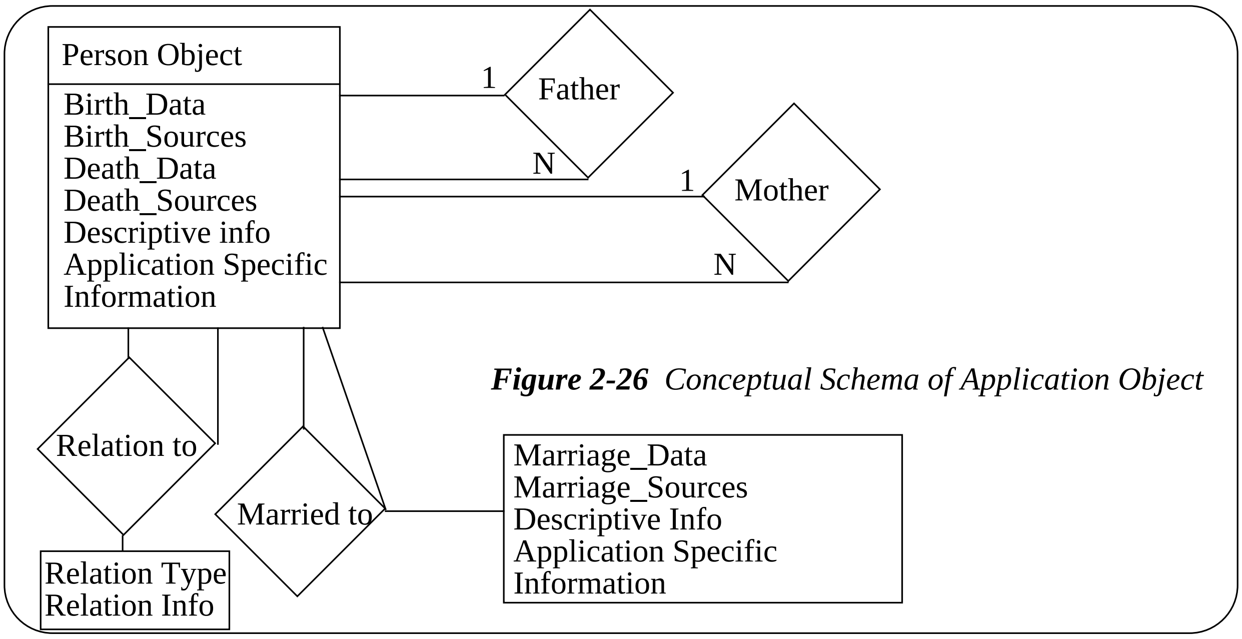


Figure 2-26 *Conceptual Schema of Application Object*

2.6.2 Traffic Model

To analyse the behaviour of this application we need to make guesses on the traffic model. The traffic model is based on user behaviour and how many users that are active with different activities. Most likely very many people will be interested in browsing the data given that the data is accessible to all people. There is, however, a more limited set of people who are updating the data and providing new research results. These people use much more time than the people who only browse the data and thereby there will be a high percentage of the currently active who are updating the data.

2.6.3 Transactions types

In this part we will describe the most common transaction types and provide “guestimates” on how often they will be performed on a world-wide basis.

2.6.3.1 Insert Original Scanned Object

This transaction is performed by scanning the original object, which could be either a microfilm or the original record itself. This activity is performed centrally by a set of scanners (most likely rather expensive) that perform scans during normal working hours. An estimate could be that 500,000 films are scanned each year. This would require a number of years for converting the films and after this the activity would most likely slow down as only original records are scanned. This suggests about 2,000 films per day and about 200 films per busy hour. This means that there will be 200,000/3.600 transactions per second (assuming 1000 pages per film) which means 56 transactions per second. Each transaction inserts one large object.

2.6.3.2 Insert Derived Scanned Object

This activity can be performed by anybody with a powerful computer that can improve the clarity of the original scanned object. It is likely that at least three to five different versions of each scanned object are needed to make sure that the text is as readable as possible. Thereby the rate should be four times as high as the insertion of original scanned objects. This leads to a transaction rate of 222 transactions per second. The transaction is the same as the insert original scanned object.

2.6.3.3 Insert Transcribed Object

Inserting a transcribed object is an activity performed by interested genealogists who normally perform the work in areas of their own research. Transcribing objects in an area helps the genealogical research of this area to take great steps forward. In Kalmar, Sweden, an association of genealogists has been established. One of the main purposes is to transcribe genealogical records from Kalmar. The association has been active for ten years and has now collected more than two and half million records of births, marriages, deaths and "husförhörslängder" (records from a yearly activity where all people were written down with birth data, marriage data, death data and data on ability to read, ability to read scriptures and so forth). Such books were written, covering a period of 5-25 years in most of Sweden from the early 1700s. Other records that are on the list are records describing the heritage of dead people, law books and so forth. Tax records were written each year from 1620s until our days.

The active genealogists who perform the transcription are about 100-200 people. Each of these persons then transcribe about 1,000-3,000 records each year. This represents about 30-100 hours of work per year per person. If a global association existed where the data becomes available to all immediately after transcription, then it is likely that many more would gain an interest in transcribing records. Thereby many thousands of people are likely to gain an interest only in Sweden and millions in the whole world. Thereby it should be possible to transcribe billions of records each year on a world-wide basis. Since people perform the research in their free time, the busy hour is likely to be in the evening. About 10-20 million records per day could be achieved at times and this would create a load in the busy hour of about 4 million records. This gives about 1,000 transactions per second on a world-wide basis.

A transaction that inserts a transcribed object has many more database requests than the insertion of the transcribed object itself. The standard name-table is used to update fields with standard names. These fields are not absolutely necessary since they represent a replication of information. It is likely that reads of the transcription tables are much more common and then it is probably a good idea to store standard names in the transcribed objects. This means about five read accesses to a standard name-table. The links between the scanned object and the transcribed object also need to be inserted. This is either an insert or an update of links depending on the implementation of the links.

2.6.3.4 Check Transcribed Objects

Transcribed objects are checked to determine which of several copies is correct and the correct transcribed object is inserted in another table. This table should have links to the objects that were used in the check. About half as many as the insertions of transcribed objects are likely to appear. This means 500 transactions per second. This transaction then consists of reading the checked objects (normally two) and then inserting the correctly transcribed object.

2.6.3.5 Search Transcribed Objects

Searches of transcribed objects are normally scans of objects that are of a particular type and in a specific region. Normally a few thousand objects are scanned and of those about five to ten objects are returned on average.

This should be a rather common transaction. It is mostly used by people performing research and not only browsing the genealogical database. It is very difficult to guess how many searches are performed. It is likely that at least as many searches as inserts are made and probably even more. We “guestimate” that 2,500 searches per second are performed on a world-wide basis and that on average 10,000 records are scanned. This is in the busy hour.

2.6.3.6 Write Application Object

To insert or update a new application object means in this case to insert/update information about a person in a genealogy or a marriage between two persons already inserted. When an application object is inserted/updated a number of links must be ensured. There should be links to parents, links to source objects. Source objects could be other application objects, transcribed objects and scanned objects. This means that an insert/update of an application object normally also causes the insertion of a number of link objects and also the reading of a number of linked objects.

Understanding the number of insertions is very difficult. Most of this work is performed by researchers but also browsers are likely to create their own copy of the application object where they write down added information that they think is valuable to them.

We “guestimate” the figure to be 3,000 per second on a world-wide basis in the busy hour.

2.6.3.7 Search Application Objects

Searches of applications are likely to be very useful; most likely they are, however, not so common. They are used when no knowledge about a person is known. Then a search could be performed. We estimate that 200 searches per second, on a world-wide basis in the busy hour, are performed. Each search is likely to scan about 50,000 objects.

2.6.3.8 Following a link

The most common operation is to follow some of the links that have been set up by a write transaction. It is likely that browsers in particular will use this technique to find information on their ancestors. Links followed will be both to scanned objects, catalog objects, transcribed objects and application objects.

A guess is that these represent some 20,000 read operations per second, on a world-wide basis in the busy hour. These transaction read one object and we estimate that about 5,000 of these are accesses to scanned objects.

2.7 Reliability and Availability in Telecom Databases

As seen from the presentation of the applications, reliability and availability are very important. In this section we will present the definitions of some terms, some empirical studies of failures and finally a number of approaches to provide good reliability and availability. Much of the material in this section is covered in greater detail in [Gray93] and [Tor95].

2.7.1 Definitions

A common measurement of the reliability of a system is the mean time between failures (MTTF); a common measure of the maintainability of a system is the mean time to repair (MTTR). Now using these two measures we can define the availability as the measure:

$$Availability = \frac{MTTF}{MTTF + MTTR}$$

So availability is the probability that the system is available. Since we require a very high level of availability, it is more common to discuss unavailability. [GS91] defines seven availability classes (Table 2-10).

Table 2-10 *System Availability Classes*

System Type	Unavailability (min/year)	Availability (in percent)	Availability Class
Unmanaged	50,000	90	1
Managed	5,000	99	2
Well-managed	500	99.9	3
Fault-tolerant	50	99.99	4
High availability	5	99.999	5
Very high availability	0.5	99.9999	6
Ultra-availability	0.05	99.99999	7

The availability requirements on telecom databases is between availability class 5 and 6. Some telecom databases will need availability class 6 and some can handle availability class 5. Another requirements is that there should be no catastrophic failures, such as a service not being available for weeks or months.

2.7.2 Empirical Studies

There are several causes of failures. The major contributors to failures are hardware failures, software failures, operator failures, maintenance failures, environmental failures. The main contributor to software failures is errors in the design. Hardware failures can be caused by errors in the design, errors in handling the hardware, environmental causes and hardware can also fail due to its age. Operator and maintenance failures are almost always based on human failures. These could, however, be caused by erroneous operator and maintenance manuals. Humans could also cause failures through strikes, sabotage and so forth. These are classified as environmental faults. Some environmental faults and some hardware faults are caused by disturbances by nature, such as earthquakes, radiation and so forth.

From [Gray93] one can see that the number of failures due to hardware and maintenance is decreasing. This is due to improved hardware and thereby a lesser need of maintenance. Software failures are, however, increasing, the other failure types are rather constant.

From both [Gray93] and [Tor95] it is evident that a good design of the telecom database is not enough. It is also necessary that the maintenance personnel are well trained, all changes should be carefully planned and changes should not be performed if not necessary. There should also be security precautions against sabotage. There should be a stable environment for the hardware through air conditioning and uninterruptable power supply and sites should also be available that can takeover at a major failure of a system (network redundancy). There must also be plans on what to do when disasters occur. Off-line backups should be stored in several safe places and stand-by systems could also be available to take over in case of disasters. In this thesis we will not pursue these issues further, we will mainly focus on what can be performed by a proper design.

To increase availability, the design should focus highly on automatic recovery and maintenance procedures. The recovery procedures should be automatic and not involve any operator interaction. Maintenance procedures should be supported by tools, so that the maintenance procedures are tested properly. These tools should also check for operator errors and should be very simple to use. All maintenance procedures must be performable with the system on-line.

2.7.3 Module Failure Rates

Module failure rates are applicable to hardware and software. Production software normally has between 0.1 to 10 design faults per 1,000 lines of code. Hardware such as disks, boards, connectors and cables normally have an MTTF of at least 20 years if properly designed. The lifetime of these products is, however, shorter and so these figures are based on there being a continous replacement of old hardware. There is also a high failure rate of new hardware; therefore one should run the new hardware in the factory before delivering it to the customer. The failure rates increase if operating temperature is too high; other factors that can increase the failure rate is shocks, vibrations, thermal changes and mechanical stress.

Hardware faults can be classified into soft faults and hard faults. Soft faults are temporary faults that cause an error and then the failure disappears. Examples of these are when a processor by mistake interprets a 0 as a 1 and then after this error it behaves correctly again. These failures do not need repair afterwards. A hard fault does, however, not disappear and the module must be repaired.

Software faults can be classified in two ways. The first classification separates between heisenbugs and bohrbugs. The difference is that bohrbugs show up each time the code is executed. The heisenbugs only show up at specific occasions when special conditions occur. Most faults in production software are heisenbugs, the reason is of course that careful testing removes most of the bohrbugs. The second classification classifies heisenbugs into virulent bugs and benign bugs. Benign bugs are bugs that only occur once and then never come back. Virulent bugs have a tendency to come back, again and again. This classification is based on most software failures in a production system being caused by a few virulent faults. The other faults that cause failures do so only rarely and are called benign faults.

The remaining software faults in production software are extremely hard to remove. Studies even show that after some time of fixing bugs, the mean number of bugs might increase as more bug fixes are introduced. The reason is that when introducing a bug fix, a number of new bugs can easily arise. The conclusion drawn from these facts is that it is not possible with current methods to avoid software errors. It is possible, however, to mask most of the software errors. This can then be accomplished by making sure that a single software failure does not hamper the availability of the system. Since heisenbugs only occur under special conditions, these bugs should not hit all processor nodes simultaneously, as the state of processor nodes is not very synchronised.

2.7.4 Hardware Approaches

In the past hardware was the main cause of failures, therefore many advanced approaches to hardware redundancy exist. The most common variant is that the hardware is doubled or tripled. Tripled hardware has the benefit that there can be a voting scheme, so that two correct hardware modules can identify one faulty module. From what we have found above, software faults are the dominating reason for failures. This means that there redundancy has to be provided by the software. This is solved in a DBMS by replicating the data. Thereby no hardware redundancy is needed since the DBMS puts the data on several independent processor nodes.

We conclude that there is no need for hardware redundancy. What could be needed are, however, failfast modules. When a hardware module does not function correctly, it should immediately report this as a failure. This means that the software can trust the hardware to function correctly.

There is still a need for hardware redundancy, however, for the communication network. This communication network can then be tripled or doubled.

2.7.5 Software Approaches

There are several approaches to availability through software approaches. One approach sometimes used is called n-version programming. This means that a specific function in the system is solved by several design teams independently. At execution the result of these functions is compared. If they agree, everything is ok, otherwise there is a problem. This approach has several problems. First, it is very difficult to detect which version of the function was correct. It is also very difficult to recover the version that failed. Another problem is that most faults arise where the complexity of the function is. This means that the really complex parts might be erroneous in all versions. This is similar to an examination in school; the hard problems are often solved correctly only by a few and sometimes even by none.

Another approach is to use transactions, so that if a fault is detected, the transaction is rolled back and the system continues operating. The reasoning is that the fault was a heisenbug and most likely also a benign bug and therefore the fault is only reported, no major investigation of the fault is performed.

One approach that is commonly used in high availability systems is to have system pairs. This means that all data in the primary system is replicated at a backup system. This means that the backup system is always ready to take over if the primary system fails. This approach can then be used both internally in a system to provide high availability of the system; it can also be performed at a network level to achieve availability from even more fault situations, such as environmental faults, maintenance faults and operator faults.

Another approach is called defensive programming, basically the programmer does not trust any input, he performs integrity checks of his internal data structures. This checking of internal data structures can either be performed while executing the normal functions or it can be executed by specific programs that only execute to check for consistency of the data structures. These programs are called auditors or salvagers.

Most of these approaches can be combined in a system.

2.7.6 Transaction Availability

Availability of Databases actually handles two issues. The first issue is how available the system is for execution of transactions. The second issue is how often committed transactions are lost. We will call the availability of the system to execute transactions Transaction Processing Availability. The measure of this is the probability of the system to abort a transaction due to an internal database failure. The probability of losing committed transactions will be called Transaction Processing Reliability. Clearly it is desirable that Transaction Processing Reliability should be higher than the Transaction Processing Availability.

In our database replication approach we will develop a method that increases Transaction Processing Reliability in a cheap way through the use of a special log server (stand-by node).

2.7.7 Conclusion

It is obvious from the requirements that the telecom database should have replication of data in the system. Since hardware fault tolerance is not enough, this replication must be handled by the software. This replication must also exist between systems, to ensure that catastrophic failures do not stop the system easily.

As much as possible of the maintenance, recovery and operator procedures should be automated to minimise the chances of human failures. This means that recovery procedures should be automatic, and operator and maintenance procedures should be supported by advanced tools. These should be easy to use and well tested. The maintenance organisation must also be carefully managed such that human failures are minimised.

Now since redundancy must be maintained by the software and there are also automatic recovery procedures, there is no need for highly available processor nodes. There could be a need for failfast processor nodes. The most common way to provide this is to have doubled hardware and have a comparator that checks whether they perform equally. If not, the processor node should immediately stop and go into a failed mode.

Even this doubled hardware can be questioned for many systems. If a hardware failure hits the system, it is likely that it will quickly show up in such a way that the software discovers the failure. When compared to the probability of software failures, such as heisenbugs that can cause similar problems, this probability should be small.

Therefore our conclusion is that the telecom database should have replicated data, both internally and on a network level. The communication hardware must be hardware fault tolerant using doubled or tripled hardware. The processor node can have hardware that fails immediately after an error occurs. For our applications this should, however, not be necessary. To improve the MTTF there should also be so-called salvagers executed as part of the software. These should check data structures for inconsistencies and they should also calculate checksums for various data and check that the checksum is correct so that one quickly finds faults caused by software and hardware faults. Processor nodes could also be restarted to remove latent faults in data structures.

In this thesis we do not investigate the fault tolerance of disks. There are many technologies providing fault tolerant disks, such as various RAID-architectures.

2.8 Conclusions on Architecture of Telecom Databases

From the study of various applications of telecom databases we can draw certain conclusions about the requirements on the telecom database. Most of the data studied should be kept in main memory. There are also applications with great needs of larger data sets (email server, genealogy databases, charging database) and these will need disks and might even need some form of data juke-box.

Typical future performance requirements on telecom databases are 10,000 requests per second. Most of these requests are either write a record or read a record. Many applications also need network redundancy to handle earthquakes and other catastrophes. A short delay of read operations is particularly crucial, but also write operations must have a short delay. Typical future delay requirements are 5-15 ms [Ronstrom93].

The impact of the delay requirement is that disk writes can not be used in the transactions. This means that availability requirements must be handled in another way. In this thesis we will study a method where several main memories are used to ensure high availability. The probability of two processor nodes failing at the same time is very small. General power failures can be handled by using battery backups to ensure that log information can be sent to disk before the battery runs out of energy.

Disk storage is needed for the documents in the news-on-demand application, the email bodies and the attachment files, and the set of scanned objects in the genealogy application. These are all examples of BLOBs. It is also necessary to store event records on disk. This is actually the only exception in the applications, of small tuples that need to be stored on disk. If BLOBs are attributes then obviously it must be possible to separate attributes that are stored in main memory and attributes that are stored on disk in a table. Another solution could be that BLOBs are stored in a special BLOB table that is stored on disk.

Main memory data can also be found in the email server, the genealogy application and the news-on-demand application. In these cases they are used to store control structures, indexes and descriptive information about BLOBs.

A requirement in the charging database was the possibility to insert a tuple into main memory and then specifying that it should move to disk at some later time, either by a timer or by specific request. This could also be achieved by an insert into a main memory table, followed by a delete and an insert into a disk table.

Reliability requirements were studied in the previous section. From the charging database and genealogy database we can draw conclusions that many of the requirements on object-oriented databases are very relevant. We need inheritance when defining tables and need to be able to put queries on the supertables. We also need methods as first class attributes. The genealogy database does needs many of the complex data structures provided by the object-oriented model. A pure object-oriented database is, however, not desirable as this would mean that all classes would need to store an object id. Some tables, however, only need two attributes with indexes on both and providing an object id would create an unnecessary and large memory overhead. Therefore a mixture of techniques from relational databases and object-oriented databases is desirable.

Almost all applications have a need of being informed when relevant events take place in the database. This means that a good support of trigger mechanism should be included. To simplify application development there should also be good support for reference constraints that should be automatically maintained after declaration.

Naturally a good support of indexes is needed, especially indexes for various telecom addresses, http-addresses, file names and so on. Both primary indexes and secondary indexes are needed (unique and non-unique).

From another study of an application of a telecom database, a Cache Server for WWW[JOH97], some new requirements were found. This requirement was also found after discussion with an experienced developer of database applications. The problem is that sometimes the application needs to lock tuples for longer time than the transaction. This means that the database must be able to handle locks on tuples which are not involved in transactions. These locks must also be persistent over crashes.

The Cache Server also needs to maintain persistent counters and this is most likely also needed by many of the other applications. It is important that these counters are not handled by the normal concurrency control protocol. This would create a hot-spot which would degrade performance.

The support of join queries is not found to be among the most important aspects. The charging database does, however, require some of this. The charging database has the benefit of not being updated, thereby concurrency control for large queries is not a problem.

It can also be noted that many disk-based applications are create-only applications. Deletes can occur but then usually whole tables or large parts of a table are deleted at a time. This can be used to simplify recovery and transaction handling for these applications.

So the most important features of a parallel data server for telecom applications are the following:

1) Scalability

Telecommunication systems are sometimes very small (e.g. 10-300 users) and sometimes very large (several million users). A database for telecommunication applications must therefore be scalable both upwards and downwards. Scalability downwards is achieved by portability to various platforms. This is not covered in this thesis but is the aim of the distributed real-time run-time system described in [Ronstrom97]. Since the data server described in this thesis can be implemented on a portable run-time system, the data server is as portable as the run-time system is.

2) Security against different kinds of catastrophes

In times of catastrophes such as earthquakes, storms and so forth it is actually even more important that the telecommunication network does not fail. Therefore a database system for telecommunication applications should be secure against different kinds of catastrophes.

3) Very high availability

A telecommunication network must be available always. A lost telephone call can sometimes be a very serious matter. Databases are often centralised and it is even more crucial for them have a very high level of availability.

4) Very high reliability

Transactions which have been committed must not be lost. A transaction could represent an income to the operator and must not be lost. Also users of the telecom network are not likely to accept that the system does not act as they have told it to do in some way.

5) Load Regulation and Overload Control

All telecom networks are built with the assumption that not all subscribers are active at the same time. This means that overload situations can always occur. It is important that databases also have a scheme to handle overload situations. This is not covered in this thesis and is an item for future work.

6) Openness

This will not be discussed in this thesis but is certainly a very important requirement. It is necessary to be able to reuse code, tools, platforms, and so forth developed by others to enable quick development of new services.

3 Telecom Database Benchmarks

To be able to develop a high-performance telecom database, it is necessary to have a set of appropriate benchmarks. With these benchmarks, design decisions can be evaluated and bottlenecks can be found. It also serves as a base for performance models of future designs, both analytical models and simulation models. These new benchmarks do not remove the need for using other benchmarks as well. Therefore benchmarks such as TPC-B, TPC-C, OO7 and other database benchmarks are also valid in the discussion of telecom database benchmarks.

3.1 Formal Benchmark Definitions

If an execution of a telecom database benchmark is to be officially recognised, one should follow some formal rules as set out in this section. If benchmark results are reported that do not conform to these rules, the rules not conformed to should be explicitly stated.

All Telecom Database Benchmarks are executed using the model as seen in Figure 3-1. It is assumed that the communication between the request generator is through a LAN (Local Area Network). A WAN (Wide-Area Network) is also allowed and figures for LAN and WAN can be separately reported. There are no particular requirements on the communication hardware. Typically the communication hardware could be Memory Channel, SCI (Scalable Coherent Interface), Myrinet, ATM or any other communication hardware.

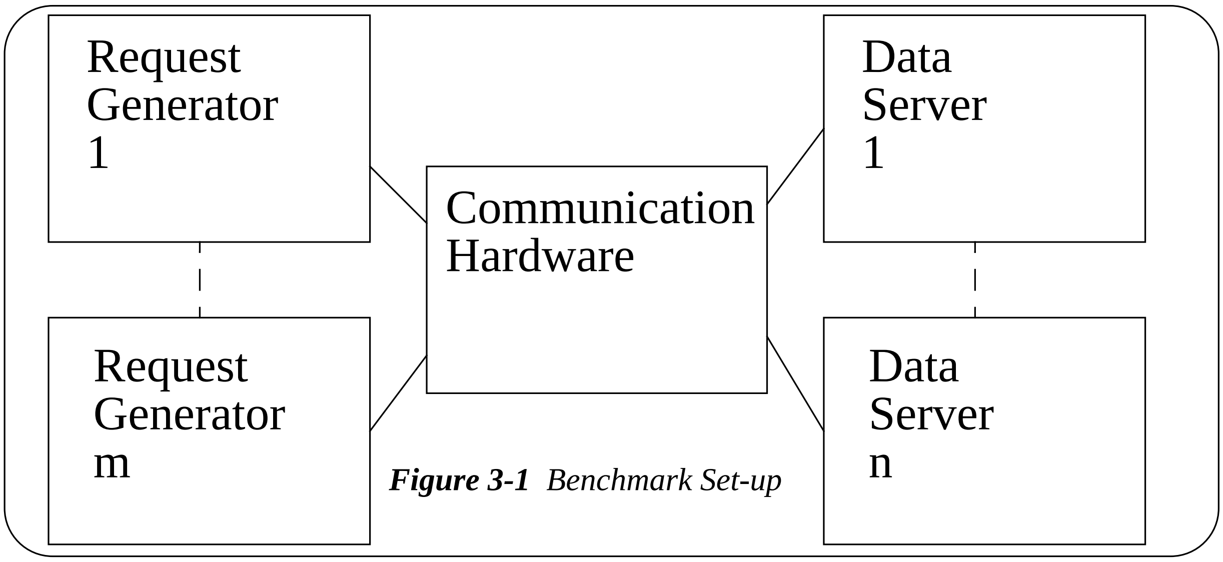


Figure 3-1 *Benchmark Set-up*

The delay between successive requests to the Data Servers should have an exponential probability distribution. This distribution can be achieved by using the following function:

$$delay = -mean \cdot \log(erand48(\ ))$$

Here mean is the mean time between requests that is desired during the test, erand48() is a random function (part of the C library) that delivers a result random number between 0 and 1 and log(x) is the natural logarithm function. After deciding to generate a request, it is necessary to decide what type of request to use. This is also performed by a random function. After applying (int)floor(100*erand48()), where floor is a function in the C library that gives the largest integer which is smaller than the argument, one simply uses this as input to a table of 100 values, each value representing one percent of the requests. Each such value in the table corresponds to a request. If 14% of the requests are to be of a specific kind, there should then be 14 values in this table specifying how to use this function.

It is allowed to pack a number of requests together into a larger packet or use a stored procedure that handles a number of requests instead of a single request at a time. This is, however, only allowed given that one generates requests according to the above and that the measured delay includes the time waiting for a number of messages that are packed together into a larger packet. The delay requirement must still be met, given these premises.

The requirements on the Data Server interface are not specific. It is assumed that some kinds of stored procedures are used in the Data Server. The Request Generator is responsible for the measurement of response delays and the measurement of the performance of the benchmark. For all benchmarks, all parts of Clause 2 of the TPC Benchmarks must be fulfilled [Gray91]. It is acceptable in these tests to use smaller tables than normally used in the TPC Benchmark. These tests verify that the database that is benchmarked satisfies requirements on atomicity of transactions, consistency of transactions, concurrency of requirements and that the system is durable and thereby recoverable.

The partitioning rules of Clause 5.2 in the TPC Benchmarks are also to be used. This rule allows horisontal fragmentation of tables but does not allow vertical fragmentation of tables. Clause 6 of the TPC Benchmarks holds, except Clause 6.3 which is overridden by the response time requirements as defined in the various telecom database benchmark definitions. Clause 6.1 defines the term measurement interval. Clause 6.2 defines the term response time and clause 6.4 defines how to compute the transactions per second of the benchmark. Clause 6.5 and 6.6 gives some additional requirements on reported results. Clause 7 of the TPC Benchmark also holds, and this clause defines steady state and the duration of the tests.

All benchmarks require replication within a system. There is no special requirements on system downtime, however to be a possible telecom database it must be possible to change software, hardware, run-time system, operating system without interruption of service. This replication should be 2-safe, that is both replicas should be updated within the local transaction. If only 1-safe is supported by the system, the achieved performance figures should be multiplied by a factor of 0.9 as a way to show the cost of a less reliable system.

All benchmark results can be reported with two different models of network redundancy. The first is a system without network redundancy, the other two cases also involve network redundancy. This means that all update transactions are executed both on the primary telecom database and also on a backup telecom database. These two telecom databases should be locatable with a geographic distance of at least 500 km between them. This means that the delay of communication between the systems must be the same as if they were placed at a distance of more than 500 km. There are two types of update mechanism possible in this scenario. The first uses 1-safe transactions. This means that the updates are sent to the backup telecom database after the transaction has been committed on the primary telecom database and also after reporting this to the client (in this case the request generator). The second is called 2-safe; in this case both the primary telecom database and the backup telecom database are updated as part of the transaction, except when one of the telecom database nodes has failed (which is not allowed during the benchmark).

A final important issue is that the system under test will contain both the Request Generators and the Data Servers. This makes it possible to use any architecture of the data server. Some databases are always placed in the same process as the application. It is, however, not allowed to take advantage of locality in any way in the Request Generators. All the Request Generators must perform exactly the same algorithm and it is not allowed to ensure that accesses will always be to the Data Server in the same processor node.

3.2 UMTS Benchmark

To develop a benchmark for UMTS it is necessary to define the tables that are most often accessed and the operations performed on these tables. In developing this benchmark we will use the result from the preceding sections.

3.2.1 Data Definition

The UMTS user has data in two places, in his home database and in the database he currently visits. In his home database, see Figure 3-2, the full service profile resides. Also routing data that is used to find where he currently visits and the current registrations of the user.

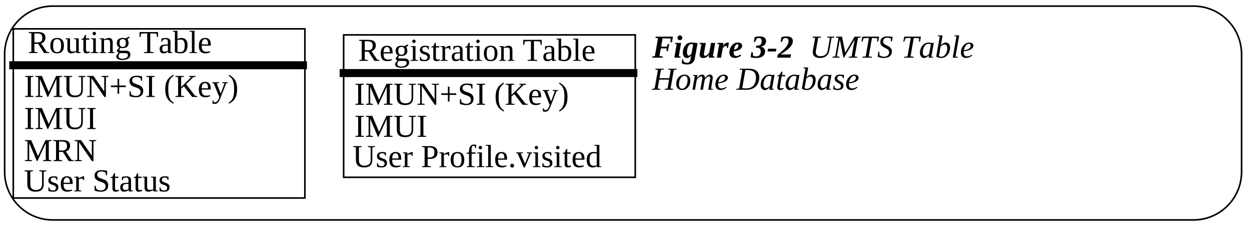


Figure 3-2 *UMTS Table Home Database*

In the visited database, see Figure 3-3, all the current registrations, terminal data and session data are stored. There is also a table that is used for assignment of temporary mobile identifiers. We have redefined the UMTS model somewhat so that there is always a session record, during session set-up and session release this record is updated with session status.

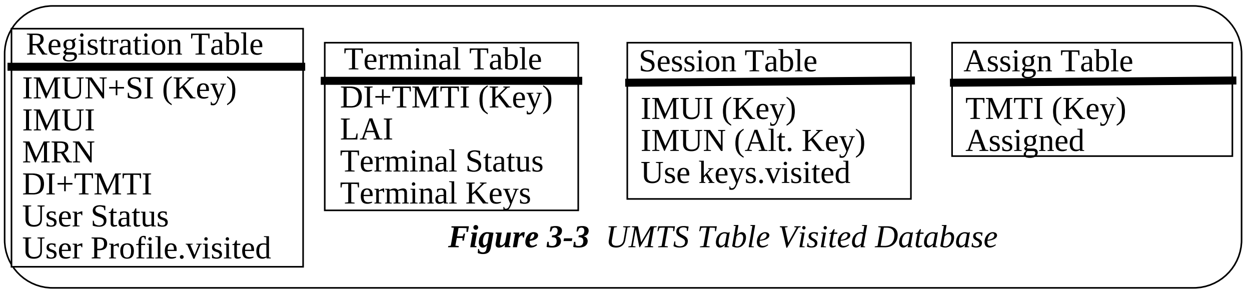


Figure 3-3 *UMTS Table Visited Database*

To derive a proper benchmark it is necessary to make a realistic set of database tables, the size of those tables, the necessary keys on the tables and the access rate of different tables using the various keys. Therefore in Table 3-1 it is shown which tables are used during the various procedures. Now it is should be noted here that the benchmark only involves the normal actions taken during these procedures. If specific supplementary services are invoked, these would most likely use another set of tables that store data used for more complex queries.

Table 3-1 *Table Usage during UMTS procedures*

Procedure	Home Registration Table	Visited Registration Table	Routing Table	Session Table	Terminal Table	Assign Table

Table 3-1 *Table Usage during UMTS procedures*

User originated User Session Set-up				2R, 1U		
System Originated User Session Set-up				1R, 1U	1R	
User Originated Terminal Session Set-up					1R, 1C	1R, 1U
System Originated Terminal Session Set-up					1R, 1U, 1C	1R, 1U
User Session Release				1U		
Terminal Session Release					1D	1U
User Registration	1R	1C	1C	1R		
User Deregistration		1D	1D		1	
Attach					1U	
Detach					1U	
Service Management		1R				
Call Handling		2R	1R		1R	
Location Update					1U	
Domain Update	1R	1C, 1D	1U		1U	

The next step in developing a benchmark is to assess the size of the tables. The size of these tables is dependent on the number of users and terminals. The size estimates are based on educated guesses. However, errors in these should not impact the validity of the UMTS benchmark as much as errors in access rates would.

Another important aspect of the benchmark is the size of the keys. The length of international telephone numbers is up to 15 numbers. A compacted representation of this is possible in eight bytes. Therefore the session table key and the assign table key is eight bytes. The other tables have mixed keys and the size of these is then sixteen bytes.

The visited registration table and the routing table contains one record for each active registration a user has, and we assume the mean of this number is two. The home registration table contains a record for all types of registrations a user can have, and we assume the mean of this number is five. The session table contains one record per user. The terminal table contains one record per terminal in the network and we estimate that there is one terminal per user on average. Thus there is one record per user in the terminal table. Finally the assign table should contain enough terminal identifiers for most situations and therefore we assume it contains ten records per user.

The size of the registration tables should be somewhat bigger than the rest of the tables. We estimate the size of the home registration table to be 500 bytes per record and the visited registration table to be 200 bytes. The other records we estimate to be 100 bytes, except the records in the Assign Table which are estimated to be 20 bytes.

This gives 3,500 bytes per user, which is a reasonable estimate for a UMTS database. It is likely that other information is also stored in the database, such as various services, tariff data and so forth. We do not incorporate this into this benchmark since this information would not affect the validity of the benchmark. With 2 million users in the UMTS database the table definitions in Table 3-2 are derived.

Using Table 2-2 and Table 3-2 we derive the request rates shown in Table 3-3.

Table 3-2 **Table Definitions of UMTS Application**

Table Name	Record Size (including key)	Number of Records in Table	Key Size
Home Registration Table	500 Bytes	10 Million	16 Bytes
Visited Registration Table	200 Bytes	4 Million	16 Bytes
Routing Table	100 Bytes	4 Million	16 Bytes
Terminal Table	100 Bytes	2 Million	16 Bytes
Session Table	100 Bytes	2 Million	8 Bytes
Assign Table	20 Bytes	20 Million	8 Bytes

.

Table 3-3 **Request rates of UMTS Application**

Table	Request Type	Size of Attributes used (exclusive of key)	Percentage of requests
Routing Table	Read	20 Bytes	7%
Routing Table	Update	20 Bytes	1%
Routing Table	Create	50 Bytes	1%
Routing Table	Delete	Not applicable	1%
Home Registration Table	Read	200 Bytes	3%
Visited Registration Table	Read	100 Bytes	14%
Visited Registration Table	Create	150 Bytes	3%
Visited Registration Table	Delete	Not applicable	3%
Session Table	Read	50 Bytes	14%
Session Table	Update	50 Bytes	16%

Table 3-3 **Request rates of UMTS Application**

Table	Request Type	Size of Attributes used (exclusive of key)	Percentage of requests
Terminal Table	Read	50 Byte	14%
Terminal Table	Update	50 Byte	8%
Terminal Table	Create	50 Byte	3%
Terminal Table	Delete	Not applicable	3%
Assign Table	Read	10 Byte	3%
Assign Table	Update	10 Byte	6%

3.2.2 Full-scale Benchmark Definition

In a full-scale UMTS benchmark we use the figures on table sizes derived in the previous section. The number of users are two million users in the database so the total size of the database is 7 GByte of data. On top of this there is also about 1 GByte of data needed for the index data structures. Given replication and overhead of various kinds on top of this the required memory size of a benchmark system is in the order of at least 20-25 GByte.

The Request definition of the UMTS Benchmark is given in Table 3-3 and the Table definitions are given in Table 3-2. The request definition table is based on the results shown in Table 2-1 and Table 2-2. A requirement is that the response time on retrievals from the database must be less than 15 ms and the response time on other queries (update, create and delete) must be below 200 ms. This requirement must be fulfilled by 99% of the queries during the benchmark test.

3.2.3 Simple UMTS Benchmark Definition

When defining a benchmark simplicity is also important. We also define a simple benchmark where there are only two tables. The first table is 100 bytes with an 8-byte key size and the second table is 200 bytes in size and has a 16-byte key size. The first table is a merge of the session and the assign table. The second table is a merge of the rest.

The benchmark is defined by Table 3-4 and Table 3-3

.

Table 3-4 **Table Definitions in Simple UMTS Benchmark**

Table Name	Record Size (including key)	Number of Records in Table	Key Size
Large Table	200 Bytes	1,000,000	16 Bytes
Session Table	100 Bytes	2,000,000	8 Bytes

.

Table 3-5 **Request rates of Simple UMTS Benchmark**

Table	Request Type	Size of Attributes used (exclusive of key)	Percentage of requests
Large Table	Read	100 Bytes	38%
Large Table	Update	50 Bytes	9%
Large Table	Create	100 Bytes	7%
Large Table	Delete	Not applicable	7%
Small Table	Read	50 Bytes	17%
Small Table	Update	50 Bytes	22%

3.3 Table Look-up Benchmark

Number Portability, Name Server and Directory Services are all examples of table lookup applications. The basic functionality is to translate from a number or an address to another number or address. We define here a very simple benchmark: we define two tables, both with two attributes.

3.3.1 Data Definition

The first table has two attributes that are 8 bytes large, the second has two attributes that are 64 bytes (a character array). There will be equally many requests for both tables. Both attributes must be indexed, so that it is possible to translate the number or address in both direction. That is, it must be possible to translate from logical to physical and also from physical to logical, see Figure 3-4.

Address Translation (8 bytes attr.)
Physical Address (key)
Logical Address (key)
Name Translation (64 bytes attr.)
Physical Name (key)
Logical Name (key)

Figure 3-4 *Table Lookup Tables*

3.3.2 Full-scale Benchmark Definition

The benchmark parameters are defined in Table 3-6 and Table 3-6.

Table 3-6 **Request Definition of Table Lookup Benchmark**

Table	Key	Requested Attribute	Percentage of requests
Address Translation	Physical Address	Logical Address	25%
Address Translation	Logical Address	Physical Address	25%
Name Translation	Physical name	Logical name	25%

Table 3-6 **Request Definition of Table Lookup Benchmark**

Table	Key	Requested Attribute	Percentage of requests
Name Translation	Logical name	Physical name	25%

Another requirement is that the response time on retrievals from the database must be less than 10 ms. The 8-byte addresses should be representative of a representation of telephone numbers and the 64-byte names should be representative of addresses on Internet (e.g. URL-addresses).

Table 3-7 **Table Definitions of Table Lookup Benchmark**

Table Name	Record Size (including keys)	Number of Records in Table	Key Size of Primary Key	Key Size of Alternate Key
Address Translation	16 Byte	2 Million	8 Bytes	8 Bytes
Name Translation	128 Byte	2 Million	64 Bytes	64 Bytes

The result will be reported in the number of users that the database can handle according to the given premises and number of requests per second handled.

3.3.3 Small-scale Benchmark Definition

A small scale benchmark for this can also be defined. In this benchmark 100,000 records in each table is sufficient and there is no specific requirements on availability and reliability.

3.4 News-on-demand Benchmark

The news-on-demand benchmark is a benchmark of a service to read the news on electronic media. The basic assumption is that most people read the news in this manner and that their behaviour is similar to the behaviour of how a person reads newspapers and watches the news on TV at present.

3.4.1 Data Definition

There are only two tables in this benchmark, the Document Table and the File Table, see Figure 3-5. Both of these consist of a set of Descriptive attributes, the format definition, a key and the information in a large object. The descriptive attributes, the key, and the format definition in the Document Table are assumed to be 1 kByte in size; in the File Table it is smaller, 400 bytes.

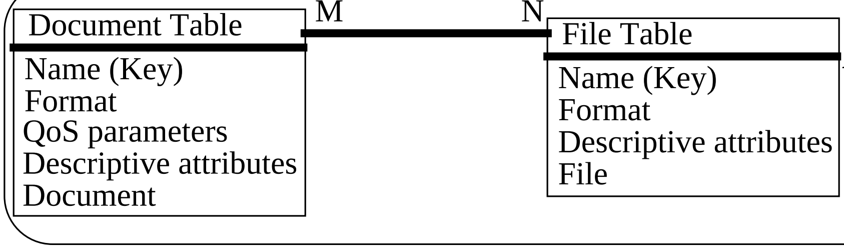


Figure 3-5 *News-on-demand Tables*

The database consists of 1,000 documents in the document table, 300 of size 5 kBytes, 400 of size 10 kBytes and 300 of size 15 kBytes. There are 2,000 files in the file table, 600 of size 25 kBytes, 800 of size 50 kBytes and 600 of size 75 kBytes. There are also 60 video files in the file table, 30 based on the mobile model with 128 kbit/sec and 30 based on the fixed model with 2 Mbit/sec. Of these 30 video objects, 7 are 30 seconds, 18 are 2 minutes and 5 are 5 minutes in length. The contents of the database are shown in Table 3-8.

Table 3-8 **Database Definition of News-on-Demand Benchmark**

Number of Documents	Size	Number of Files	Size	Number of Videos (Mobile Model)	Size	Number of Videos (Fixed Model)	Size
300	5 kB	600	25 kB	7	480 kB	7	7.5 MB
400	10 kB	800	50 kB	18	1.92 MB	18	30 MB
300	15 kB	600	75 kB	5	4.8 MB	5	75 MB

3.4.2 Benchmark Definition

The request rates of the news-on-demand benchmark are shown in Table 3-9 and the database definition in Table 3-8. A retrieval chooses any of the documents, files and videos at random with an equal probability. The delay requirement is that documents and files must be sent within 100 ms. Videos can be retrieved in chunks of any size. The first chunk must, however, arrive within 100 ms and also the chunks must arrive faster than 128 kbit/sec for the Mobile Model and 2 Mbit/sec for the Fixed Model.

Table 3-9 **Request rates in the News-on-demand Benchmark**

Request Type	Mobile Model	Fixed Model
Retrieve Document	32.5%	27%
Retrieve File	65%	53%
Retrieve Video	2.5%	20%

3.5 Email Server Benchmark

The email server benchmark is a benchmark for a future multimedia email service based on the description in the previous section.

3.5.1 Data Definition

There are two tables in the email server benchmark, one contains the emails and the other contains the attachment, see Figure 3-5. The size of the attributes in the email table is 200 bytes plus the size of the email body, the attributes in the attachment table is 100 bytes plus the attached file.

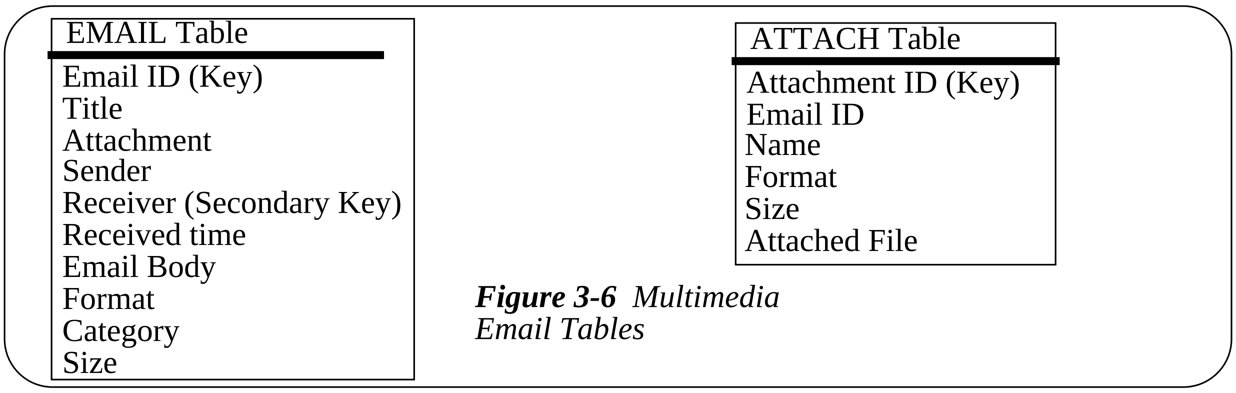


Figure 3-6 *Multimedia Email Tables*

According to [Pax94b], the size of the emails has a distribution that is bimodal, one of the mode for text messages and the other mode for file transfers. The size of the email body is then distributed according to the first mode and the attachment files are distributed according to second mode. The size of the first mode has a mean of 1 kByte that is distributed according to a log-normal distribution. To simplify the benchmark the size of the email body should be 500 bytes with 30% probability, 1 kByte with 55% probability, and 2 kBytes with 15% probability. Also the attachments are distributed according to a log-normal distribution, where the mean of this distribution is 40 kBytes. To simplify the benchmark the attachments are 20 kBytes with probability of 20%, 40 kBytes with probability of 60% and 60 kBytes with probability of 20%. The probability of no attachment is 60%, the probability of one attachment is 30%, and the probability of two attachments is 10%. We assume in the benchmark that there are never more than two attachments. These figures are shown in table format in Table 3-10 and Table 3-11.

Table 3-10 **Definition of an Email Body**

Email Size	Percentage of emails	Number of Attachments
500 Bytes	30%	0
1 kByte	55%	1
2 kBytes	15%	2

Table 3-11 **Definition of an Email Attachment**

% of Attachments	Attachment Size	Percentage of Attachments
60%	20 kBytes	20%

Table 3-11 **Definition of an Email Attachment**

% of Attachments	Attachment Size	Percentage of Attachments
30%	40 kBytes	60%
10%	60 kBytes	20%

3.5.2 Full-scale Benchmark Definition

Before the Email Server Benchmark is executed, the database should be built. This building database process should run 100 million create email transactions. These transactions will create both the email and its attachments according to Table 3-10. There will be a total of 2 million users in the system, giving a total of 1 TByte data in the email database after the building process. The receiver of each of these emails should be chosen at random After the database building the benchmark can start executing.

There are two types of events that should occur in the email benchmark. The first is that a user connects to the email server to check his emails. This event is represented by the control flow given in Figure 3-7. The second event is a create email transaction. Reading and Deleting emails means that the email body, its properties, references to its attachments and also the attachments are read and deleted. Scanning the email table means that the email table is scanned. 100 bytes per email of the user is returned to the application, 20% of the events are connect email sessions and 80% of the events are create email transactions as given in Table 2-1.

Table 3-12 **Request Definition of Email Benchmark**

Procedure	Percentage of events
Connect Email Session	20%
Create Email Transaction	80%
Total	100

The connect email session contains three different types of transactions. The connect email session starts by picking a user, and then the email table is scanned to find the emails of that particular user. The request generator then selects a number of emails for reading, where 8% of the emails will be read. The emails that are picked are those with the most recent timestamp. The read transactions are delayed with an exponential distribution. There will be exponentially distributed time between each read email transaction with 15 seconds as the mean value. After the last read transaction a create email transaction is issued after an exponentially distributed delay with a mean of 15 seconds and finally, after another exponentially distributed delay of 15 seconds, a number of delete transaction are executed. In these transactions 75% of the read emails are deleted and also 2% of the emails with the oldest timestamp are deleted. Thus 10% of the original emails will have been deleted. Rounding should be to the closest integer and upwards if between two integers. The control flow of the session is shown in Figure 3-7.

The figure 75% is supported by investigations of marketing through normal mail. 75% of all marketing papers through normal mail are thrashed immediately.

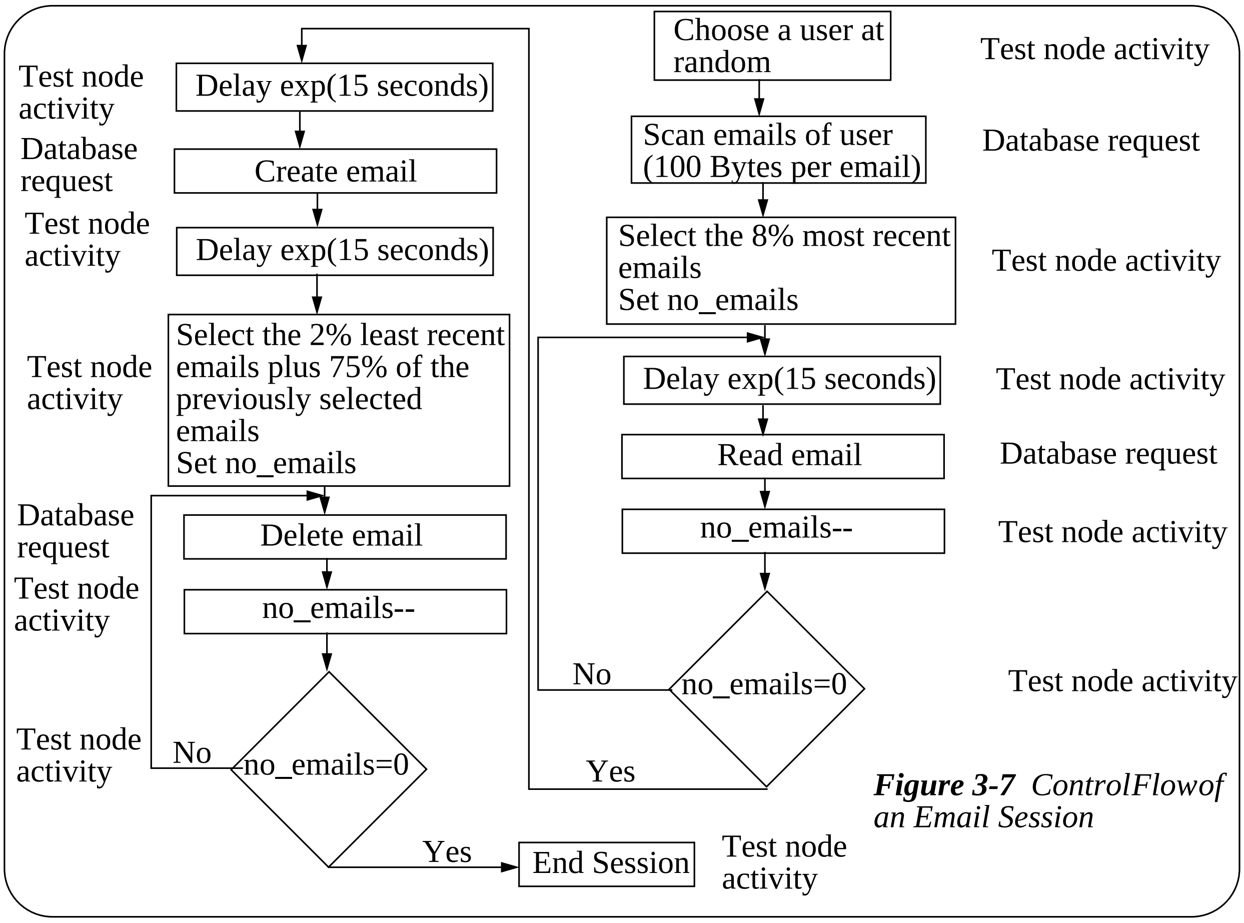


Figure 3-7 *ControlFlowof an Email Session*

The create email transaction is independent of the connect email sessions and creates an email for a random user.

The response time must always be below 200 ms for the read transactions and for the first scan of the email table for at least 99% of the requests. There are no specific delay requirements on the delete and the create transaction.

3.5.3 Simple Email Benchmark Definition

A simple benchmark of this application should decrease the mean size of attachments to 2 kByte and also decrease the number of users to 100,000 users.

3.6 Charging DataBase Benchmark

A common benchmark is defined for charging databases, this includes UMTS, New-on-demand and Multimedia Email characteristics.

3.6.1 Data Definition

The data definition of the charging database is more complex than the other applications. One way would be to simply have one table for each event. This would be sufficient if queries mainly asked for records from one event type. Since most queries request all usage records that belong to a specific subscriber, this would create a vast need of join queries in the system. If these queries are optimised for this particular type of query this might be acceptable. Another way to model the system is where there are two superobjects, the usage object and the tariff object. There is also a user table that contains attributes that are derived from the event records, which is updated as part of the transactions that insert event records. Now there are subclasses of those objects for each event type and each tariff type. The system needs to be optimised to perform queries on the superobjects. The response to these queries should be attributes both from the superobject and from the subobjects.

To simplify benchmark development and definition of the benchmark we will, however, only use four normal tables, the event table, the main memory event table, the tariff table and the user table as shown in Figure 3-8. Functional requirements can be checked by checking the abilities of the databases that are benchmarked.

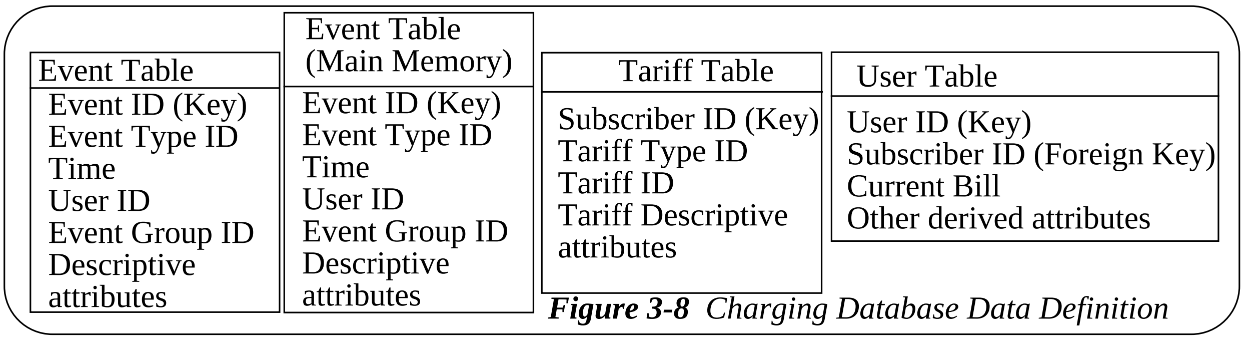


Figure 3-8 *Charging Database Data Definition*

The event table in the benchmark represents the following tables as described by the application description: the event table, the service transaction table and the tariffed service transaction table. This is the table that contains the massive amount of records that represents the disk tables in this benchmark. It is assumed that it contains five billion records of size 200 bytes at the start of the benchmark. Then during the benchmark new records are gathered.

Since an operator normally needs to store records for three months, the actual data storage should be much bigger. This would, however, not change any critical part of the benchmark and therefore we do not require more than five billion records from the start.

The event table in main memory contains the most important attributes of the event table and has a record length of 100 bytes. The attributes in this table are retained between the generation of an event record and the time that the event group is finished. The mean time between creation and deletion is two minutes. This means that the number of records in the table should be three and a half million after which a balance of creation and deletion is reached so that this will be the normal amount of records. The tariff table contains tariff information and there are 100,000 records of 1 kByte in size. Finally the user table contains one record per user of 500 Bytes and it is assumed that we have 2 million users in this benchmark. The tables have simple keys of 8 bytes which normally contain a telephone number.

Table 3-13 **Table Definition of Charging Database Benchmark**

Table	Record Size	Number of Records	Key Size
Event Table	200 Bytes	5,000,000,000	8 Bytes

Table 3-13 **Table Definition of Charging Database Benchmark**

Table	Record Size	Number of Records	Key Size
Event Table (Main Memory	100 Bytes	3,500,000	8 Bytes
Tariff Table	1 kByte	100,000	8 Bytes
User Table	500 Bytes	2,000,000	8 Bytes

3.6.2 Full-scale Benchmark Definition

There are four types of transactions in the system. The first receives event records, creates an event record and an event record in main memory. The second type of transaction is used when a session or call is ended. It scans the event records in main memory, creates three new event records, deletes the retrieved record from the event table in main memory, retrieves data from the tariff table and finally updates the user table. The third transaction is used when only one event records represents a charging event. This is similar to the second type of transaction, except that scan and delete of event records in main memory are not necessary. The fourth type of transaction retrieves data from the user table to calculate a bill.

3.6.2.1 Definition of Transaction Type 1

This transaction contains two inserts into the event table and the event table in main memory. 75% of the transactions are of type 1.

Table 3-14 **Type 1 Transaction (75%)**

Table used in Query	Operation	Size of Data in Query	Query repetitions
Event Table	Insert	200	1
Event Table (Main Memory)	Insert	100	1

3.6.2.2 Definition of Transaction Type 2

The second transaction type is somewhat more complex: it generates three new event records, it also retrieves tariff information and updates the user table. It also scans the event table in main memory for events belonging to the same charging event. The scan retrieves all records in the table with the event group ID equal to a given entity. The type 1 transaction should create the records and the event group ID should be chosen in such a way that 7.5 user records are retrieved on average. Then finally all the retrieved records should be deleted. This transaction type represents 10% of the transactions in the benchmark.

Table 3-15 **Type 2 Transaction (10%)**

Table used in Query	Operation	Size of Data in Query	Query repetitions
Event Table	Insert	200 Bytes	3
Event Table (Main Memory)	Delete	Not applicable	7.5
Tariff Table	Retrieve	250 Bytes	1
User Table	Update	100 Bytes	1
Event Table (Main Memory)	Scan	100 Bytes	1

3.6.2.3 Definition of Transaction Type 3

This benchmark is simpler than type 2 with the same requests except that scan and delete are removed. This transaction type represents 10% of the transactions in the benchmark.

Table 3-16 **Type 3 Transaction (10%)**

Table used in Query	Operation	Size of Data in Query	Query repetitions
Event Table	Insert	200 Bytes	3
Tariff Table	Retrieve	250 Bytes	1
User Table	Update	100 Bytes	1

3.6.2.4 Definition of Transaction Type 4

This represents the simplest transaction. It merely reads the user table and then it is finished. It represents 5% of the transactions in the benchmark.

Table 3-17 **Type 4 Transaction (5%)**

Table used in Query	Operation	Size of Data in Query	Query repetitions
User Table	Retrieve	500 Bytes	1

3.6.3 Small-scale Benchmark Definition

A small-scale benchmark only needs to store 10 GBytes of disk data and 145 MBytes of data in main memory from the start of the benchmark as defined in Table 3-18.

Table 3-18 **Table Definition of Small-Scale Charging Database Benchmark**

Table	Record Size	Number of Records	Key Size
Event Table	200 Bytes	50,000,000	8 Bytes
Event Table (Main Memory	100 Bytes	350,000	8 Bytes
Tariff Table	1 kBytes	10,000	8 Bytes
User Table	500 Bytes	200,000	8 Bytes

3.7 Telecom Database Benchmark

A telecom database benchmark is defined as the logarithmic mean of the number of users that a particular telecom database can handle in the benchmarks defined here together with the following benchmark set; TPC-B, TPC-C, OO7. These benchmarks must, however, also be executed with data replication for high availability.

3.8 Conclusion of Part 2

The telecom database applications have been analysed and it is seen that there are four common sets of basic operations. The first set is simple operations towards a main memory database. This could be to read, write, create or delete objects. The second set is simple operations on disk databases, mostly create operations. The third set of operations is to read, write, create or delete a set of files and possibly send them to a set of destinations. The fourth set comprises complex management queries issued towards the disk database. There is also a set of uncommon operations related to management of the system such as restart, reload, software change, hardware change, conceptual schema changes and other management operations.

This means that the system can basically be divided into four parts. One part handles transactions, recovery, logging and trigger management. Then there are three parts that handle different types of storage. The first handles main memory database objects, the second handles disk database objects and the third handles a transactional file system.

3.9 Further work

The benchmark definitions needs to be refined, and management operations should also be included in the benchmarks. The benchmarks need to be tested and executed to be able to refine them and more studies of telecom applications on a more detailed level regarding data structures is needed. This would give more input to the impact of constraint definitions and indexes on the telecom databases and this can then also be input to refined benchmarks.

III: Architecture of a Parallel Data Server

Part Description

This part describes the basic concepts and architectures of a DBMS used for telecom applications. It provides an understanding of the basic decisions on concepts and architectures of a telecom database. The software architecture of the parallel data server is described in some detail.

The algorithms and protocols developed in this thesis are designed for a hybrid shared nothing architecture. This is important since the failure of one node does not disturb other nodes and thereby a reliable system can be built.

The architecture of the parallel data server is designed for systems where the database is used to store most of the semi-permanent data (data which has a long life and change rather seldomly) and context data(context information saved between communications with the user or other systems). This means that even session data can be stored in the database. This puts heavy requirements on performance and response times of the database. New communication technology SCI (Scalable Coherent Interface) [DOLPH96] and Memory Channel [GILL96] in combination with building the database on top of a real-time virtual machine make this possible [Ronstrom97].

The first step in describing the software architecture is to describe the replication of the system. Actually two levels of replication are introduced with one level internally in a parallel data server and one level between parallel data servers. This is called network redundancy. Secondary indexes and tuple keys are also described in this preparatory section.

In Figure 1 below the replication architecture is shown. Most replicated systems use primary and a backup replica. In this thesis we have introduced a stand-by replica. The stand-by replica does only contain a log. The stand-by replica is involved in the transactions. This means that even if the primary and the backup replicas have failed, no committed transactions have been lost since the stand-by replica contains a log of the transaction. This is very useful in systems with a large number of nodes.

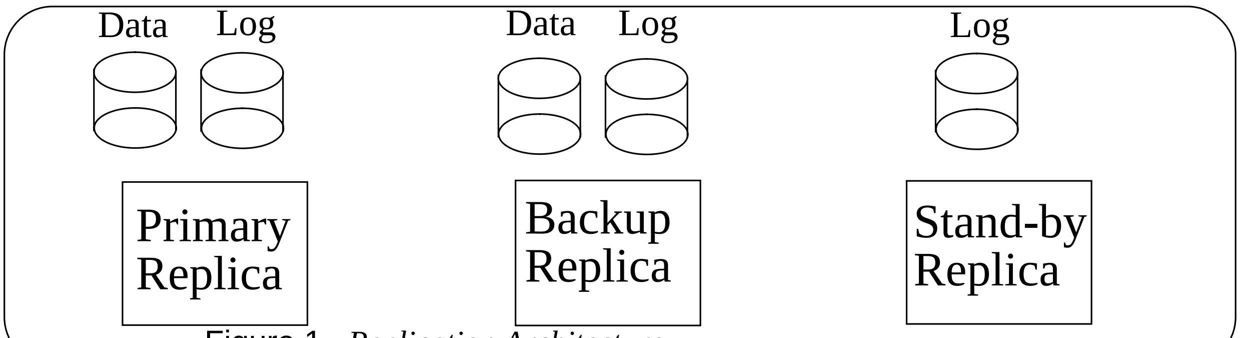


Figure 1 *Replication Architecture*

Designing a distributed database with support of two levels of replication is complicated and requires the use of many different kinds of protocols. The support of stand-by replicas and the use of logging in several main memories instead of flushing the log to disk also creates requirements on adaptions of the protocols and a set of new protocols. Before describing all these protocols we will describe why these protocols are needed and how the protocols are related to each other.

Since we normally have three replicas in a transaction it becomes important to decrease the need of communication. Therefore linear commit is used between the replicas. It would, however, be a serious degradation of response times if linear commit were also used between the fragments involved in the transaction. Thereby a new two-phase commit protocol has been developed which combines a normal two-phase commit protocol with linear commit as shown in Figure 2 below.

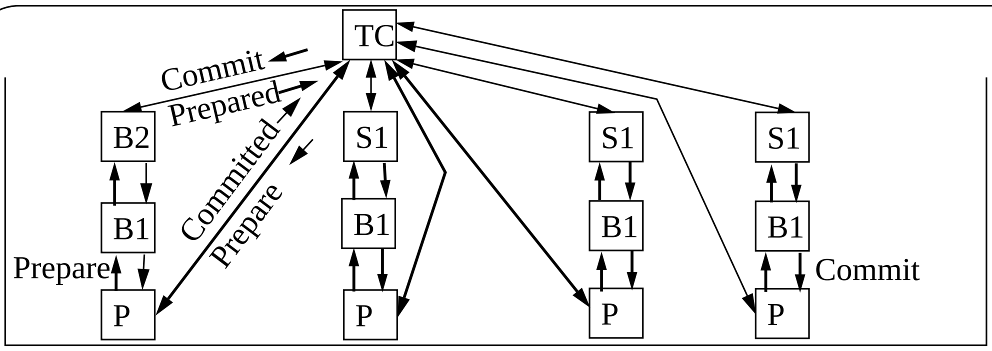


Figure 2 *New Two-phase Commit protocol*

This is described in a chapter which also covers more details of the two-phase commit protocol, read queries and handling of secondary indexes and foreign keys.

The next step is to describe the protocols needed for on-line recovery and reorganisation. By on-line recovery we mean that when a processor node fails, the system tries to achieve the same replication as before the failure through a set of replication protocols. On-line reorganisation includes a set of protocols that change fragmentation and replication after adding or dropping processor nodes. These protocols also enable a better load regulation of the system.

To achieve a non-stop system it is necessary to be able to change the schema without any service interruption. This involves being able to maintain complex relationships between tables even when tables are split and merged. A solution to these requirements is presented. The basic idea is to perform complex schema changes in three steps. A first step that creates the new tables and attributes together with a set of foreign keys, special attributes, and triggers which are needed to perform the schema change. By use of a SAGA-table the complex schema changes are also recoverable and consistent between primary and backup systems.

We will also present the adaptions to the two-phase commit protocols and the necessary new protocols needed to support network redundancy.

All systems can crash and therefore it is also necessary to be able to start a system after a crash. A crash here means that all processor nodes of a system have failed and there is no backup system that can take over. The information for crash recovery must be created during normal operation. This involves logging and checkpointing.

Finally two sections are dedicated to describing two new indexes, the LH^3 and a distributed and compressed B+tree, the Http-tree. It is shown how these indexes play an important part in some of the on-line recovery protocols. The LH^3 is optimised for fast execution (time optimised) and the Http-tree is optimised for low memory requirements (space optimised).

4 Basic Architectural Decisions

This section first gives an overview of some important issues in the design of a parallel DBMS. It also presents the basic architectural decisions that underlie the design of the telecom database. This includes parallel database architectures, how to use a client-server architecture efficiently, how to use the database for session data (context data), the interface types of the data server, transaction concepts, handling of replication, concurrency control, use of logs in the data server, two-phase commit protocols, and how to produce archive copies.

4.1 Parallel Database Architectures

In the literature a number of different architectures for parallel databases are described. These are the shared memory, shared disk and shared nothing, and there are also hybrid ones. A description of these architectures will be given and some conclusions of what best meets the requirements of telecom databases. See [deWitt92] for an overview on parallel database systems.

4.1.1 Shared memory

Shared memory is an architecture where the processors share a common memory and a number of disks as seen in Figure 4-1. Due to the development of processors this solution is quite infeasible without cache memories in the processors. This creates a need of a cache coherence protocol on the interconnection network. This is a common architecture that is supported by all major suppliers of DBMSs. Servers using this architecture are delivered by all major hardware suppliers such as Sun, Digital, IBM and so forth.

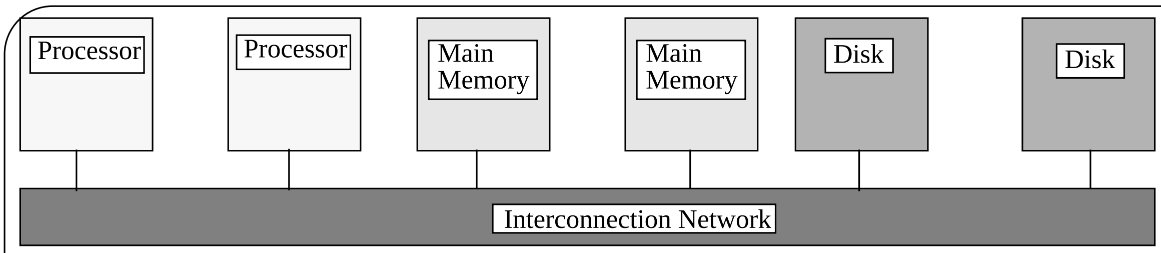


Figure 4-1 *Shared Memory Architecture*

The advantage of a shared memory structure is that it is easy to balance the load. It is also easy to port a DBMS from single-processor implementations since all the data is common as in a single processor. Interquery parallelism is very easy to obtain. Intra-query parallelism needs some new algorithms but is still relatively simple.

Shared memory architectures have several disadvantages. One is that it has a limited extensibility. This is because the processors all use the same memory structures. Therefore it can be seen that some of the semaphores used to serialize access to memory will become bottlenecks when the number of nodes increases above 20-40 [Valduriez93]. Another problem is that to achieve fault-tolerance, memory and disks need to be replicated. However the most difficult problem to overcome, to achieve good reliability, is that one processor node with a software error can corrupt data in the whole system.

The conclusion is that shared memory architectures are an easy way to extend current DBMSs with parallelism to a limited extent. It is, however, not a good solution to systems with very high requirements on fault-tolerance and high performance.

4.1.2 Shared disk

Shared disk is an approach where each processor has its own main memory and this is not accessible from other nodes as seen in Figure 4-2. The disks are however common, so any processor can access any disk.

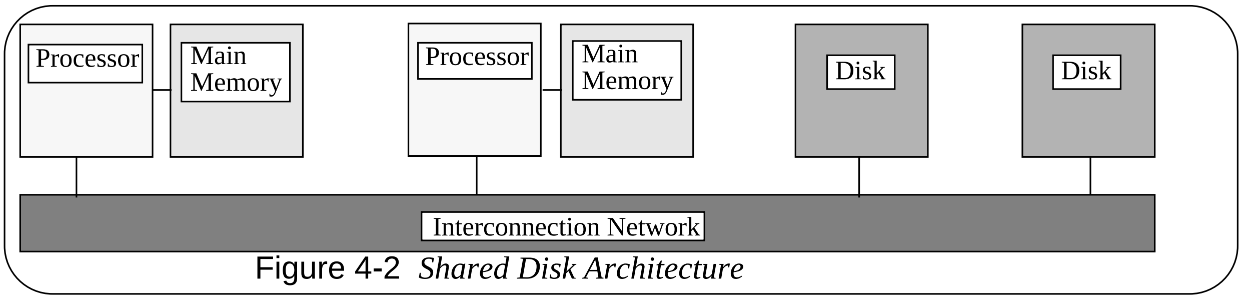


Figure 4-2 *Shared Disk Architecture*

The major difference with shared disk compared to shared memory is that the main memory is private so that a faulty processor does not have the possibility to write in another processor's memory. This solves the major availability problem with the shared memory architecture. It does, however, introduce a complex coherence problem with the disk buffers.

Since there are not any shared memory structures, the shared disk architecture scales better than a shared memory architecture. Its shared disks do however limit the scalability to around 100 processor nodes [Valduriez93]. To achieve good availability it is essential that disks are doubled and possibly even tripled.

The conclusion is that a shared disk architecture can handle the requirements on fault-tolerance and in most cases also the performance requirements. It has an advantage on load balancing but has greater complexity in administering the disk buffers. It is also complicated to achieve software changes where structures on disks are changed.

4.1.3 Shared nothing

Shared nothing means that any processor can access its own disks and its own memory and communication between nodes is only through message passing as seen in Figure 4-3. This means that the system can be viewed as a homogenous distributed database system. So research on distributed database systems can be reused for shared nothing architectures.

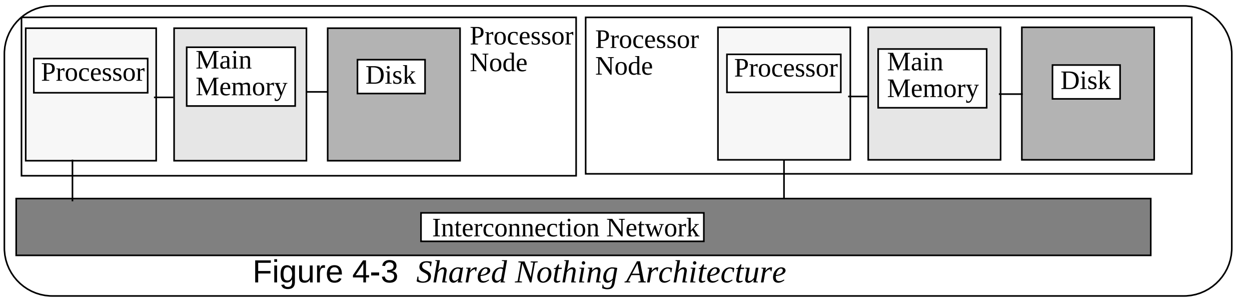


Figure 4-3 *Shared Nothing Architecture*

Since no data structures are shared, there is no data contention that will stop the scalability. However data can only be found where it is replicated and so there will be contention for processor resources that limits the scalability instead. It can, however, scale up to thousands of processors at least by using load balancing and multiple replicas of data [Valduriez93].

Availability is very good since the failure of one processor node should not affect any other processor node apart from erroneous messages which mostly should be found by error checking at reception of messages.

A shared nothing architecture contains many complex functions. The shared nothing architecture provides many possibilities to improve availability. All of these require software solutions. Some of these will be presented later in this paper.

The conclusion is that a shared nothing architecture provides most possibilities to improve availability and scalability. It is not necessarily the easiest to implement, however.

4.1.4 Hybrid Architectures

There are many ways to combine shared memory, shared disk and shared nothing architectures. The most interesting one that will be used in this thesis is to combine shared memory and shared nothing by having a cluster of processor nodes where each processor node has a shared memory architecture. In the computer architecture world this shared memory architecture is called symmetric multiprocessing and is a common method to increase performance of workstations. Since the shared memory architecture is used inside a processor node, there is no problem of availability. Scalability is also solved by the shared nothing architecture. The shared memory structure does, however, make every processor node very powerful and therefore not very many processor nodes are necessary. Therefore scalability is improved compared to a shared nothing architecture. Hardware that is configurated as shown in Figure 4-3 is sometimes called workstation clusters.

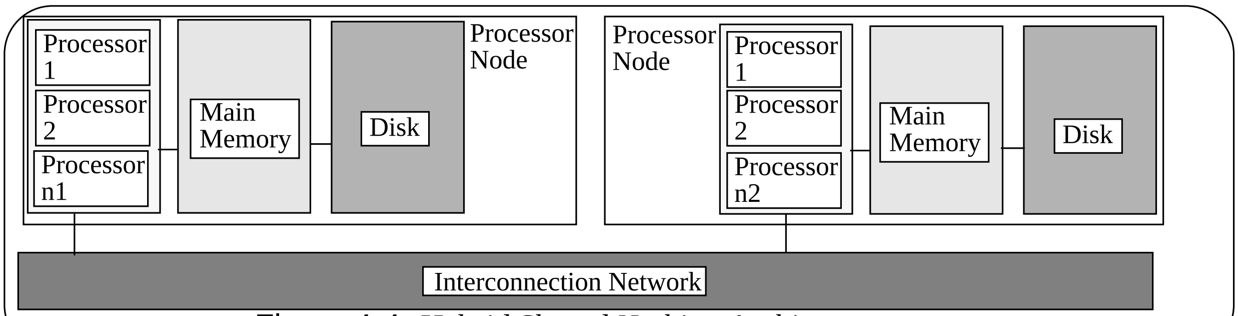


Figure 4-4 *Hybrid Shared Nothing Architecture*

Requirements on high performance, scalability, very high fault tolerance and short response times makes it necessary to communicate very quickly between each processor node. This is accomplished with new communication techniques, such as SCI [Dolph96] and Memory Channel [GILL96].

Many database vendors have a hybrid shared-nothing model to achieve reliability. These are normally systems with a primary node and a backup node. Each of the nodes is a multi-processor node.

The disks could actually be physically shared. However, each processor node must have its own set of files. Thereby the disks are logically separated although there is a physical sharing of disks.

4.1.5 Conclusion

A shared memory architecture can be accomplished by using the same computer architecture as the shared disk and shared nothing architectures by having a cache coherency protocol on the interconnection network. Whether such an architecture is shared memory, shared disk or shared nothing is dependent on the software structure. If software in all processors can access all the data in the machine, then it is a shared memory architecture. If software cannot gain access to all memories but to all disks, then the architecture is shared disks and if memories and disks can be accessed only locally, then it is a shared nothing architecture.

In this thesis the shared nothing model will be used. Each system may, however, be a symmetric multiprocessor machine (shared memory architecture). The reason for choosing this architecture is based on the need of high performance, scalability, and very high fault tolerance in telecommunication systems. To build such a system it is desirable that there are clean interfaces between modules and no other interaction other than through message passing. This also implies that each node has its own operating system and thus a distributed operating system is not desirable.

The reason for having multiple processors on each system is also based on the fact that the replacable unit in a system is a board. The technology makes it possible to build many processors on a single board. Teradata's P-90 (potential successor of DBC/1012) uses as an example four processors on a board with shared memory [Carino92]. Intel's standard high-volume server, SHV, is a more recent example, where four PentiumPro-processors are placed on one board with a common memory. Another example of such a high-performance server is the Ultra Server 2 (a two-processor server).

It would be possible to use the shared disk architecture also. The scalability would not be as good and other solutions of availability would be necessary.

4.2 Client-Server Technology

There are a number of reasons why client-server technology is used. One such reason is to provide specialised services that many applications use. There are many examples of this such as file servers, printer servers, email servers, database servers. The servers provide the clients with the possibility to be mobile, and the clients can be heterogeneous since the communication between clients and servers is performed through messages. Other issues such as security, cost-efficiency and organisational benefits are also important. In this section we will concentrate on the database servers. In [Tor95] the following guidelines for client-server design are proposed:

1. The server interface should be general and independent of the application.
2. The client-server communication protocols should adhere to international standards, both on the transport level and the application level, and support heterogeneous clients.
3. Communications should be insensitive to delays. The data volume sent should be low.
4. As much as possible of the processing should be done on the client side to achieve high server throughput and low latency.
5. It should be easy to use and program

Now point 3 in this analysis is less important due to technical breakthroughs in technology, such as SCI (Scalable Coherent Interface) [DOLPH96] where communication costs between systems have decreased so much that it might even be cheaper than internal communication between processes in an operating system. In designing a telecom database the clients are also servers, i.e. application servers such as email servers, web servers and so forth. Therefore point 4 need not be of great importance in the division between database server and application server.

In Figure 4-5 a few common ways to separate the functionality between the client computer and the computer of the database server is shown.

The A scenario is common in object-oriented DBMSs with long transactions where objects are checked out when needed and checked in when work on them is completed. This scenario could be used to access data in a telecom switch. The telecom switch needs fast access to data and uses a cached copy of the data and updates are propagated to the database server. Other applications could also access the data; these applications use the data through the DBMS. If there are many small update transactions then there will be heavy communication costs in this scenario.

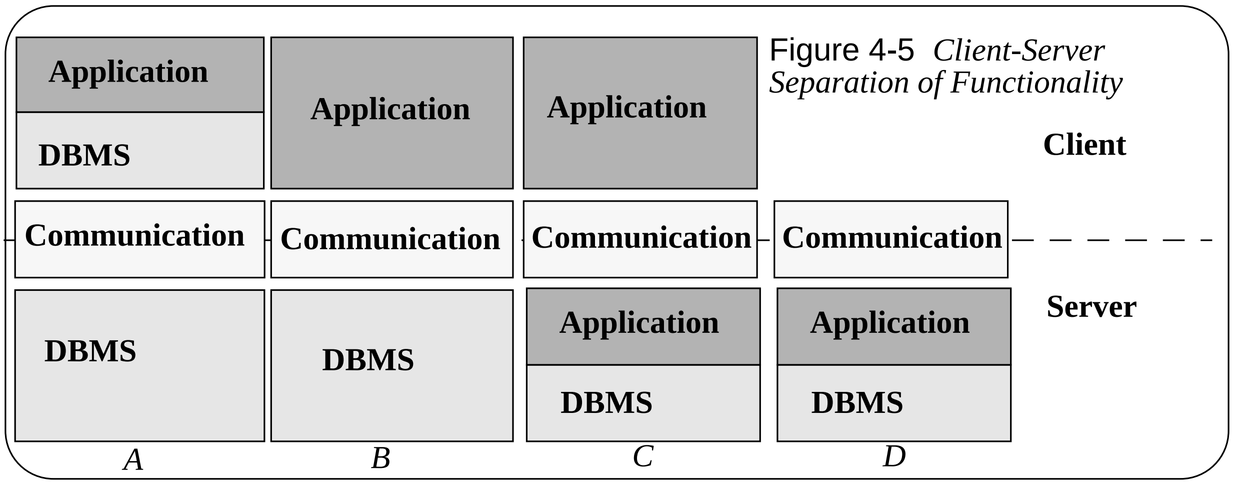


Figure 4-5 *Client-Server Separation of Functionality*

The B scenario is what standards such as ISO SQL and RDA are trying to accomplish. There is a clear separation between the client computers that handle the application and the computers that handle the DBMS.

Scenario C is a solution where one tries to minimise the communication between client and server by performing some processing of the data in the server. In this thesis we will distinguish between stored procedures and methods. Stored procedures can operate on any data in the database through a procedural language. The C interface is useful when stored procedures that perform some algorithms on data are needed. By implementing this language though an interpreter in the database server the clients cannot crash the database server by writing erroneous stored procedures. TPC-B needs this functionality for good performance and also the charging server could make use of such stored procedures.

The D scenario is where there is no separation of client-server, where application and DBMS functionality is not separated. This is mostly used in proprietary systems.

4.2.1 Conclusion

If we analyse these scenarios from a reliability point of view, one can see that the C scenario with support of stored procedures through an interpreter in the database server is the most promising scenario. Most of the database queries will, however, still use the B scenario and perform the work in the application server.

A major benefit is that programming errors in the application have a very small impact on the reliability of the database server. Therefore the reliability of the database server is likely to be very high since most of the code in the database server should be thoroughly tested.

From a capacity point of view these scenarios should have similar performance characteristics. Actually the chosen scenario has some benefits in that the database server has full control over all processor resources, cache memory resources, memory resources and I/O resources. This could be used to gain an advantage in performance, too.

If we analyse the scenarios with soft real-time performance in mind, one can see that scenario D will give the best real-time behaviour. This is possible since all processing of a job is performed in one processor. With priority scheduling, jobs with high priority can be very quickly taken care of. In the other cases there is communication between processors and processes, this means that a job is split into several tasks where each task has to be queued before execution. Since the tasks do not have the same execution time, there will be a greater variance in the in the delay of a job. It will also be more difficult to do priority sheduling since a job consists of many tasks. This problem is alleviated to some extent by the rapid development of processors, communications and memories. The delay of executing a job is rapidly decreasing, thereby making it easier to meet the real-time requirements. Furthermore the telecom applications only have soft real-time requirements. This means that too long delays can be acceptable as long as they do not occur often.

It should be possible to distinguish read-only transactions from update transactions. This makes it possible to optimise these transactions. Scenario B also has the benefit that the transaction coordinator is aware of which records and attributes are updated already when sending the update requests. Thereby execution of triggers and updates of secondary indexes and foreign keys can be performed in parallel with the actual update.

Scenario C is very useful when storing complex data types that are not supported by the DBMS. These data can then be stored as BLOBs where searches in the data are performed by a user-provided function that can execute in the database server.

4.3 Server Architecture

A study of our applications shows that most applications used by the applications need a very simple query interface. There are, however, some management applications that need a more complex query interface. We can therefore divide the database server into two parts which we call the data server and the query server. The data server is where the data is stored; it has a simple interface that can be used by both applications and the query server. The query server contains logic to perform complex queries using the data server interface. The query server can also act as a gateway to other data sources, and there could be several data sources in a system. The query server should be accessible through standardised protocols to enable an open management platform. This layering is similar to the design of most DBMSs where a storage manager handles the storage of data, recovery, concurrency and simple search functionality. The data server corresponds to a distributed storage manager. The query server then contains functionality to handle complex queries such as joins and various other advanced queries.

By this separation of functionality we decrease the complexity of the data server even more and from this we should gain even higher reliability. The layering of the various servers is shown in Figure 4-6. The figure shows that the interface of the data server handles all accesses to the data. The application can gain access to data either directly through the data server or send the query to the query server that then accesses the data server. There can also be network protocols that access data using application protocols, query server protocol or directly access the data server. The management server can use any of the protocols mentioned.

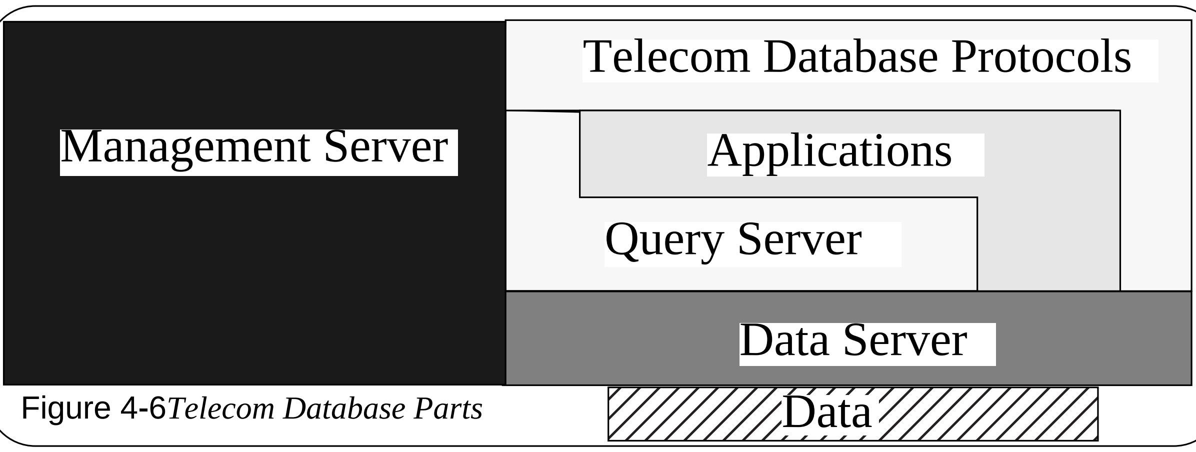


Figure 4-6 *Telecom Database Parts*

The protocols of the telecom database are the protocols that are supported by the telecom database. From Figure 4-6 we can see that these are the application protocols, the query server protocol and the data server protocol.

From this discussion we derive the architecture shown in Figure 4-7, where there are clusters of data servers, query servers, application servers and management servers. The set of application servers is of course very wide, there are servers to support various communication protocols, various communication services and so forth. These various application servers is not an issue in this thesis, where we will mainly concentrate on how to build a parallel data server to support telecom applications.

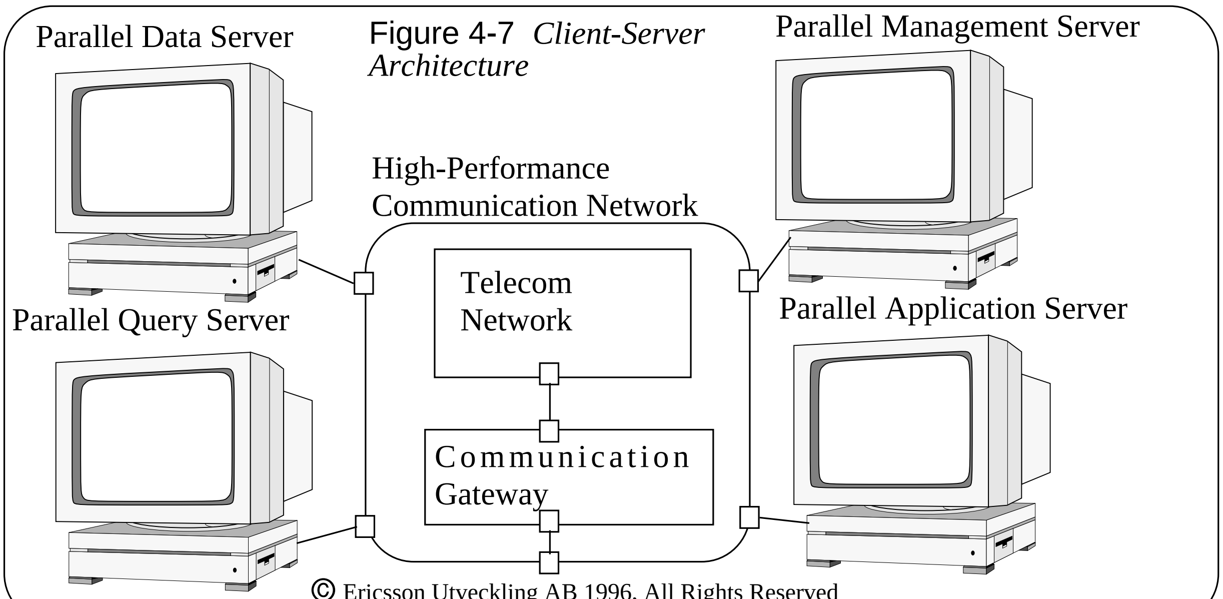

Parallel Data Server
Parallel Management Server
High-Performance Communication Network
Telecom Network
Parallel Query Server
Parallel Application Server
Communication Gateway

Figure 4-7 *Client-Server Architecture*

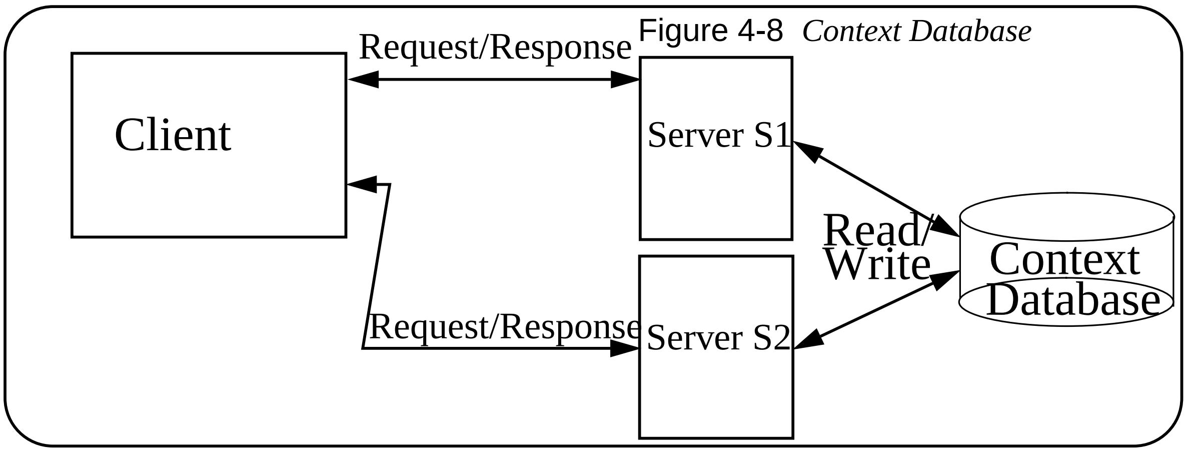
Another major benefit that the architecture in Figure 4-7 provides is that the only server that contains data is the data server. The other servers only contain uncommitted data (transient data) and logic. There are also application and management servers that contain hardware such as communication links; these are not considered in this thesis.

This is accomplished when the application, query and management server uses the data server to store all permanent data. Thus there is not much need to have hardware redundancy in these servers. The data server can also be implemented without using hardware redundancy as was shown in Section 2.7. This means that there is no need for computers with hardware redundancy, which means that one can use ordinary server computers and interconnect them with a reliable communication network. This provides a very good price/performance ratio for telecom databases.

4.4 Session Database

An important feature for load balancing is to be able to always choose the least loaded server to serve the client requests. To achieve this all the servers must have access to the same information. This can be accomplished by storing the session information in the database. If an application has a session with a client, then all session data is stored in the data server when the application waits for the next client action. This means that the next time a client request arrives as part of this session any server can be chosen to serve this request. The data server is then used as a context database as shown in Figure 4-8 below.

Figure 4-8 *Context Database*



Many telecom systems are configured as shown in Figure 4-9 below. If the communication servers do not need to keep any information on which application server that contains the session data, then less reliability is needed in those parts of the system. The communication server can always choose the least loaded application server. This architecture puts heavy requirements on the data server in terms of response times and performance.

Supporting this type of architecture is an essential part of the design of the parallel data server described in this thesis.

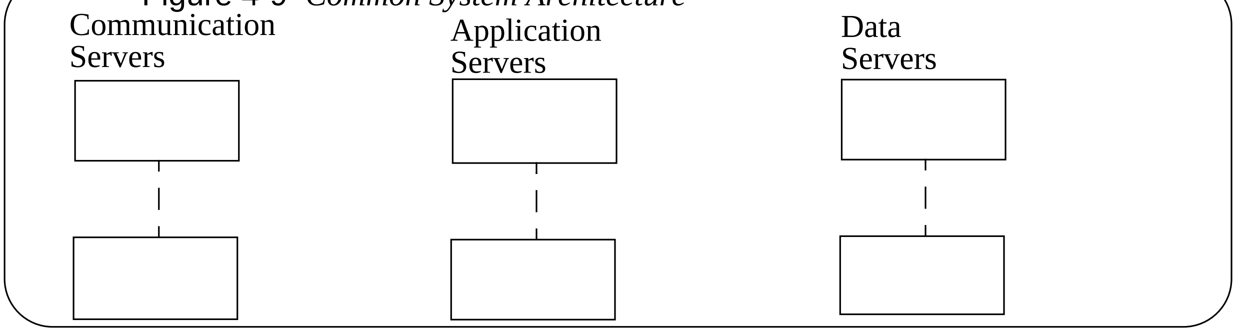


Figure 4-9 *Common System Architecture*

4.5 Network Redundancy

From the discussion in the previous section on reliability it can be deduced that the reliability provided through a parallel data server is not always sufficient. It is also necessary to provide redundancy to be able to handle environmental faults and operational faults. An example is an earthquake that destroys a parallel data server.

To survive this type of fault it is also necessary to provide redundancy on a network level. Thereby the parallel data server must also support network redundancy where reliable data is stored at several telecom databases as seen in Figure 4-10.

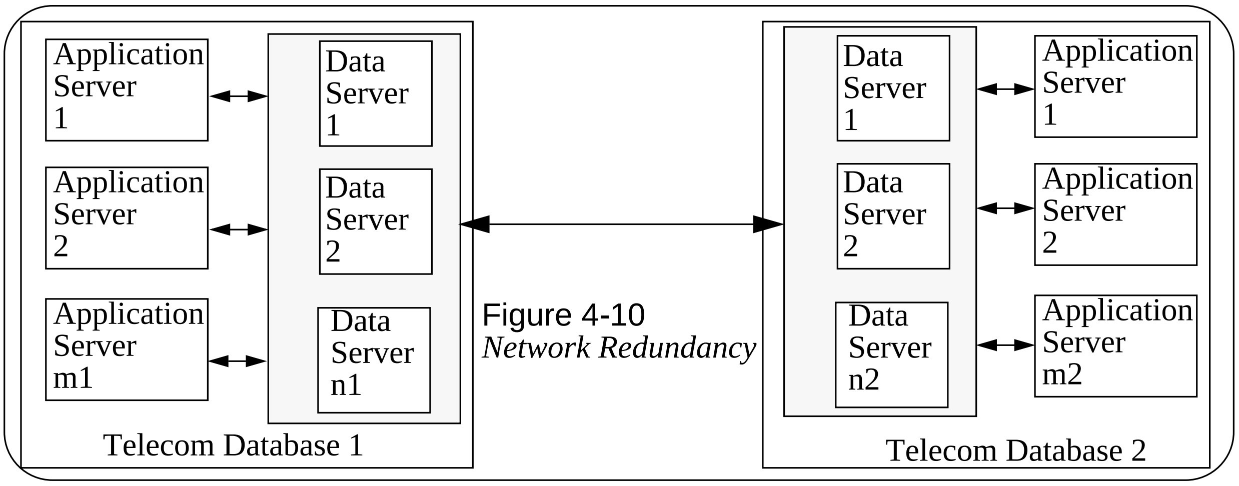


Figure 4-10 *Network Redundancy*

The solution of network redundancy can also be adopted with small changes to act as a replication mechanism that can be used by applications. An example of such an application is mirroring of Web servers.

4.6 Interface Types

There are two dimensions of interfaces. The first dimension is the application interface, which is where the actual database interface is defined. The second dimension is the network interface, which contains a specification of the interface used to send and retrieve the application messages.

The database interface also has two dimensions. The first dimension defines the protocol to use the Data Server. This is a rather simple interface with possibilities to read, insert, update and delete data. It also contains functionality to define new data tables in the database. The second dimension is the interface of the Query Server. In this interface functionality is placed to support complex queries and also to support standard interfaces.

Also the network interface has two dimensions. The first dimension is the interface of the internal network using communication that is optimised for local communication. The second dimension is the interface of the external network for communication in wide-area networks which does support communication through well adopted standards in a local environment.

4.6.1 Interface of the Data Server

The Data Server is the place where the actual data is stored. It supports high levels of reliability and must support applications with high requirements on performance. It must be possible to use this interface for general applications, such as the interface of the Query Server. It must also be possible to use it for applications with a few common queries. Therefore one can either use a generic interface to the data server or use a fast path interface in the data server. The fast path interface is an invocation of a stored procedure that is optimised for a particular application. It is also a generic interface that uses identities instead of names on the tables and attributes. It is possible to also support execution of programs in the data server through the interface to the data server. This is implemented by having an interpreter in the data server.

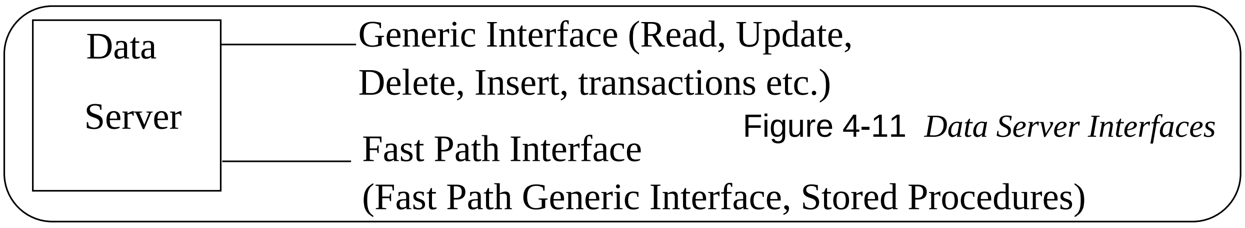

Data Server
Generic Interface (Read, Update, Delete, Insert, transactions etc.)
Fast Path Interface (Fast Path Generic Interface, Stored Procedures)

Figure 4-11 *Data Server Interfaces*

As the Data Server is a distributed architecture, the definition of the application specific interfaces that must be available in all nodes that may use this interface. The stored procedures must also interact with the distributed nature of the DBMS and the distribution of data within the data server.

4.6.2 Interface of the Query Server

The Query Server is where complex queries are executed to serve the management servers and the application servers. The reasons for using the query server can be many. The first is to use the advanced functionalities of the query server, such as support of complex triggers and complex queries. It could also provide an interface that makes the database server look like a relational DBMS, object DBMS, knowledge DBMS or any other type of DBMS. Lastly the purpose could be to provide a standardised interface to the applications, such as SQL, ODBC, OMG or any other standard. These various uses of the Query Server are shown in Figure 4-12.

4.6.3 Interface of the Internal Network

The interface of the internal network is optimised for local communication. To give efficient support for this, the interface uses a distributed shared memory to implement message passing between processor nodes. The advantages of this scheme is that messages sent internally in the telecom database need not go through the operating system. Any type of communication with the op-

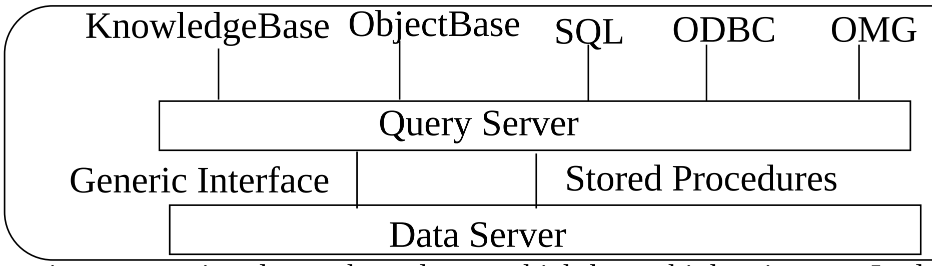
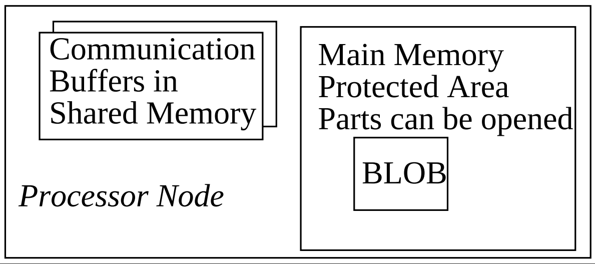


Figure 4-12 *Interface for the Query Server*

erating system involves a kernel trap which has a high price tag. It also often involves one or several context switches to operating system processes. It also often involves copying the data between user space and kernel space and could also involve copying of data within the kernel space. It also almost always involves use of standardised protocols such as TCP/IP. Furthermore, the implementation of these protocols within the operating system is normally performed by calculating a checksum using software. This costs a lot of performance.

When a distributed shared memory is available between the processor nodes, the communication network as such handles the error situations in hardware. The communication is performed in the application process and thereby no context switches or kernel traps are needed. Examples of such interfaces are SCI (Scalable Coherent Interface) [Dolph96] and Memory Channel [GILL96]. The interface of the internal network interface is visualised in Figure 4-13.


Communication Buffers in Shared Memory
Main Memory Protected Area Parts can be opened
BLOB
Processor Node
Figure 4-13 *Internal Network Interface in Processor Node*

Since the interface of the internal network also supports distributed shared memory this can be put to use. It is useful for applications that request very large data structures from the data server to avoid passing this through the message passing interface. This can be performed instead by providing a pointer to the memory space where the object resides. This memory space is normally locked for use of other processor nodes, but can be opened at request for passing large data structures, such as BLOBs (Binary Large OBjects, e.g. a file). These objects can then be transferred with mechanisms that have higher bandwidth than can be achieved using messages.

4.6.4 Interface to External Networks

The telecom database must also provide an interface to the external world. This interface must conform to the international standards on WAN networks (wide-area networks), such as TCP/IP for communication in Internet applications, CCITT no. 7 in communication between telecom applications and other standards such as X.25 and so forth. These protocols are carried by protocols for wide-area communication with support for high bandwidths, such as ATM (Asynchronous Transfer Mode), SDH (Synchronous Digital Hierarchy). There are many issues involved in this that will not be discussed in this thesis.

The interface to the external networks is handled in a communication gateway as shown in Figure 4-7. This gateway function could either exist in all or some of the processor nodes, and it could also be implemented as a separate function with specialised hardware.

4.6.5 Interface of the Application Servers

The interface of the Application Servers is the most common way to access the telecom database. This interface represents functions used by the application such as get html-file, update location of mobile terminal, insert event record and so forth. This interface is implemented on top of external network interfaces.

Common application protocols are HTTP (Hypertext Transfer Protocol) using TCP/IP, SMTP using TCP/IP, MAP (Mobile Application Part) using CCITT no.7, INAP (IN Application Part) using CCITT no.7 and so forth. Examples of protocol stacks of these application protocols are shown in Figure 4-14.

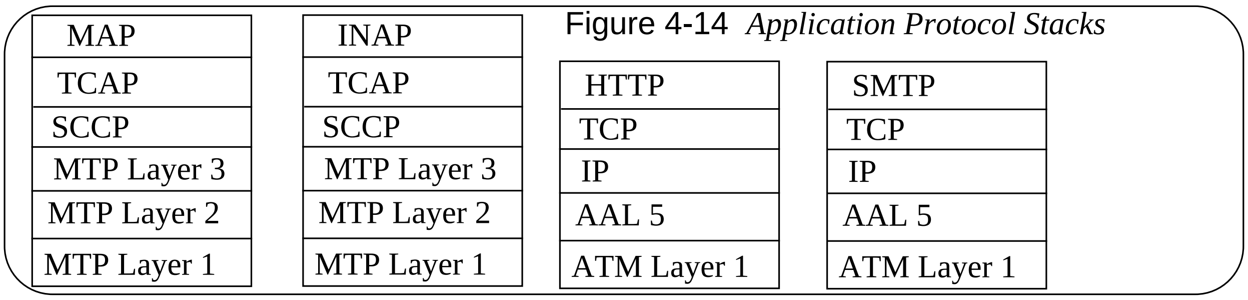


Figure 4-14 *Application Protocol Stacks*

MTP (Message Transfer Protocol) is the lower layers of the CCITT no. 7 protocols. SCCP is an add-on to MTP to provide logical network addresses. MTP uses physical network addresses. TCAP is a transaction oriented add-on to SCCP that provides transaction protocol services to the application protocols. AAL 5 is an adaption layer on top of ATM that is used for transporting data packets on top of ATM. IP (Internet Protocol) is an internetworking protocol used on Internet. TCP (Transmission Control Protocol) is a transaction protocol service on top of IP. HTTP is the protocol used by the World Wide Web and SMTP is the protocol used for emails. Instead of TCP, UDP (User Datagram Protocol) can sometimes be used. UDP does not provide secure delivery and should mainly be used for small datagrams where TCP is seen as overkill.

4.6.6 Interface of the Management Servers

Management applications of the telecom database are also accessed through interfaces, where one common interface used in telecom applications is the Q.3 interface. The Q.3 interface has the possibility to read, update, insert and delete data objects, and events can also be reported. There is no transaction protocol in the Q.3 interface.

4.7 Transaction Concepts

The transaction concept is a very powerful concept that is heavily used in many applications. Many extensions to the basic transaction concept are possible. The applications that we do consider are, however, not of this type and therefore this presentation only presents some basic transaction types.

4.7.1 Flat Transactions

The most used type of transaction is sometimes called a flat transaction. A flat transaction is an atomic unit, either all of the processing is performed or none of it. A flat transaction should have the ACID properties (ACID = Atomic, Consistent, Isolated and Durable).

The atomic property says that either all actions of the transaction are performed or none. The consistent property makes sure that after a roll-back of a transaction, all data structures are kept consistent. The consistent property also assumes that all transactional modifications by an application are consistent with the properties of the database. The isolation property gives the transaction program the view that it is the only program that accesses the database. The most common isolation property is serialisability. A set of serialisable transactions executed in parallel has the same behaviour as the same transactions would have had if executed in some serial order. This means that it is not possible to see any intermediate states within a transaction from another transaction. The durable property means that when a transaction is committed, the actions of the transaction are durable, even in the case of system failures. Losing data from committed transactions is a very serious failure. It can make the whole database inconsistent. Therefore the normal action, if this happens, is to restart the system from an archive copy.

4.7.2 Flat Transaction with Savepoints

For most transactions in our applications with flat transactions are sufficient. Some management transactions could, however, use savepoints which are internal to a transaction and cannot be seen by other transactions. The idea is that the transaction can be rolled back not only to the start of a transaction, but also to a savepoint. This can be advantageous for large transactions and complex transactions where one needs to roll back to some previous state of the transaction. A good example of such an application is a travel booking system. One could also have persistent savepoints, such that a system can be restarted with a live transaction, but this becomes rather complex to support.

4.7.3 Distributed Transactions

Since in our assumptions a telecom database is a distributed system, it is necessary to update more than one processor node, even in a flat transaction. This means that we need distributed transactions. The most common way to proceed is to use the two-phase-commit protocol. This protocol is part of several transaction standards; a description of this protocol can be found in [BHG87].

It is necessary to both use optimised versions of the two-phase-commit protocol and also versions of the protocol with higher reliability against blocking. The major reliability problem with the two-phase-commit protocol is when the coordinator fails. This can leave the transaction in an unknown state, which means that all resources of the transaction must be kept until the coordinator has restarted. This is clearly not desired, and therefore we shall in all situations have means to choose a new coordinator in case the coordinator fails. This coordinator must then retrieve sufficient knowledge of the transaction to decide whether the transactions should commit or abort. Also it is necessary to use an optimised version of the two-phase commit protocol to decrease the load on the system at the cost of a slightly higher delay.

This transaction protocol is an important contribution of this thesis which is described in chapter 6. It will also be shown how to integrate the two-phase-commit protocol with network redundancy, where copies of the data also reside on another telecom database.

4.7.4 Replica Handling in Distributed Transactions

The two-phase-commit protocol was developed for coordination of transactions, consisting of updates to data residing at more than one processor node. To update replicas of the same data as part of the transaction also requires a method of maintaining the copies. One common way to handle the copies is the read-one-write-all-available (ROWAA) method. This means that if a read is to be performed, only one of the copies is used, and at update all available copies are updated. Other methods that are used are based on reading and writing quorums of copies, which is used in systems of nodes, where each node has low availability. In [HELAL96] most of these methods are presented.

There is also a distinction as to whether the copies are updated inside the transaction or as soon as possible afterwards. Updates inside the transaction are called 2-safe and as soon as possible is called 1-safe.

4.7.5 Conclusion

The conclusion from this section is that normal flat transactions will be sufficient for the applications. Since we are building a parallel data server it is necessary to use distributed transactions. The transaction protocol is an important issue that needs further study in this thesis to achieve a balance between quick responses and low communication overhead. Since the processor nodes seldom fail, the ROWAA method is used for updating the replicas of the data.

4.8 Theory of Concurrency Control

Concurrency Control is needed to isolate transaction programs from each other, those that execute in parallel. [BHG87], [PAPA86] and [Gray93] contain deeper studies of Concurrency Control. In databases this means that several users will be able to access the database simultaneously and still view their respective execution as the only one currently active in the database. Two important laws of concurrency control are important to follow for all implementation [Gray93]:

1) Concurrent Execution of Application Programs should not cause them to malfunction.

2) Concurrent Execution of Application Programs should not have lower throughput or much higher response times than serial execution of Application Programs.

Many resources have been spent researching various ways to achieve concurrency in databases. The methods developed can basically be divided in two dimensions. The first dimension is that there are locking methods and there are timestamp methods. Locking methods works by locking the records at some point in the transaction so that other transactions do not have the possibility to access the records until the records are unlocked. Timestamp methods ensure a serialisable execution of a transaction by ensuring that updates to records always follow an increasing timestamp. The other dimension is whether the method is pessimistic or optimistic. Pessimistic methods mainly achieve serialisable execution by waiting for the resources before accessing the records. Optimistic methods access the records directly and validate that the execution was correct at the end of the transaction. If the execution was incorrect, the transaction is aborted. Pessimistic methods can also abort in some cases, mainly due to deadlock situations.

Although so much research has been performed, almost all implementations use locking methods that are pessimistic. The few exceptions to this rule mainly use optimistic locking methods. The most common method is to use strict two-phase locking as will be described below. One of the reason for this is that the concurrency control method highly affects the other algorithms in a DBMS. Locking methods are often easier to integrate with logging, recovery and distributed transactions. Pessimistic methods also make development of distributed replication and recovery algorithms more straightforward.

If the applications discussed in a previous part of this thesis use locking on record level, no problems are foreseen with hot spots. Possible hot spots that could occur are of course if counters are used. If these are used heavily in an application, then alternative concurrency control methods have to be considered for these data [BERN93]. Performance of pessimistic locking is no problem since the locks are integrated with the index structure. The extra cost of locking is thereby almost negligible.

In this thesis we will therefore concentrate on pessimistic locking methods and their behaviour in replicated databases.

There are two ways to allocate locks, The first method is to allocate all locks at the start of the transaction, which is called static allocation of locks. This can be useful in systems where it is known early on which records are included in a transaction. Since in most systems this is not the case, the more common method is dynamic allocation of locks, in which case a record is locked when it is known that it is going to be used in a transaction. In pessimistic locking methods, this means that the record is locked immediately before the record is used the first time in the transaction. The following presentation will assume that dynamic allocation of locks is performed.

4.8.1 Transaction Dependencies

The actions performed in a DBMS are actually few. First there are database actions READ, UPDATE, DELETE, and INSERT. These can be categorised as READ and WRITE actions. One could also support INCREMENT and DECREMENT as database actions, which would then be a third category DELTA [BERN93]. There are also actions of the transactions BEGIN, PREPARE, COMMIT and ROLLBACK. In Figure 4-15 the various depencies between transactions are shown, where delta operations behave as writes in relation to reads and writes.

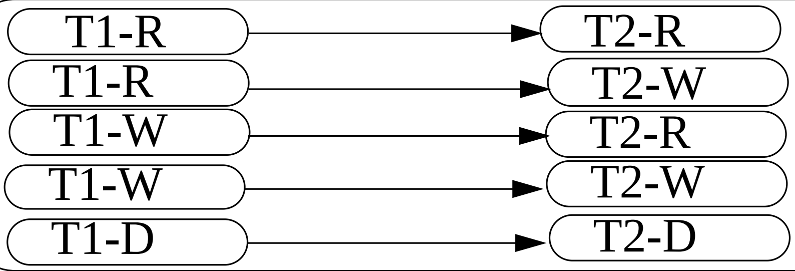


Figure 4-15 *Transaction Dependencies*

4.8.1.1 The Three Bad Dependencies

There are three dependencies that one must avoid to have an isolated system. The first is Lost Update, which occurs when a transaction reads an object and later uses the read to update the same object. If another transaction had the possibility to write the object between the read and write, then this write will be lost since it is overwritten by the first transaction. Hence the write by the second transaction is lost.

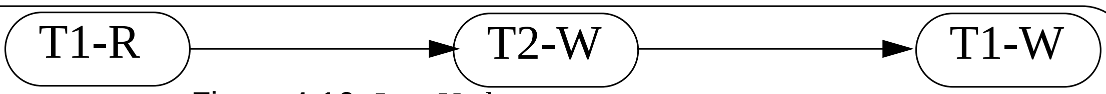


Figure 4-16 *Lost Update*

The second bad dependency comes when a first transaction writes an object and later writes it again (e.g at Rollback), if a second transaction reads the first write, it reads something that was never committed and thereby it reads a dirty object.

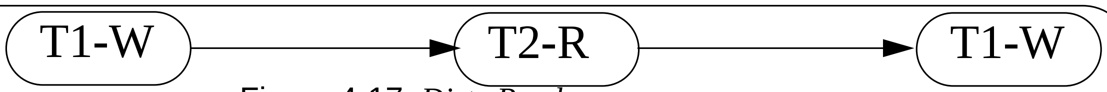


Figure 4-17 *Dirty Read*

The third bad dependency occurs when a transaction reads the same object twice and another transaction writes the object between those reads. This is called unrepeatable reads, the two reads of the first transaction do not deliver the same result.

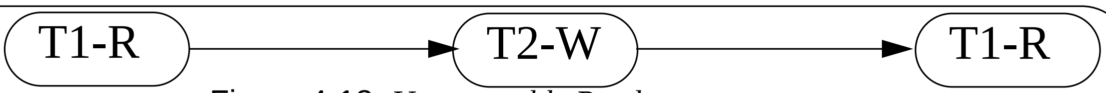


Figure 4-18 *Unrepeatable Read*

Now it can be shown that if these three dependencies are avoided, then the execution of transactions is isolated.

4.8.2 Locking Theory

If transactions are to execute isolated, they must follow some rules. The objective of these rules is to ensure that an execution of a number of transactions in parallel can be serialised. As we already saw in the previous section, there are three bad dependencies that need to be avoided to accomplish this. Since we opted for pessimistic locking, this means that a lock must be held before any actions can be taken on database objects. There are Shared Locks (for READ), Exclusive Lock (for WRITE) and delta locks (for DELTA) [Hvasshovd92]. In Table 4-1 the compatibility matrix of these locks is shown.

Table 4-1*Lock Compatibility Matrix*

Compatibility	**Shared**	**Exclusive**	**Delta**
Shared	**Compatible**	**Conflict**	**Conflict**
Exclusive	**Conflict**	**Conflict**	**Conflict**
Delta	**Conflict**	**Conflict**	**Compatible**

Now we proceed with some definitions:

Definition 1: A transaction is ***well formed*** if all its read, write, delta and unlock actions are covered by locks and if each lock action is eventually followed by a corresponding unlock action.

Definition 2: A transaction is ***two-phase*** if all lock actions precede the unlock actions. This means that the transaction has two phases, a growing phase when the locks are acquired and a shrinking phase when locks are released.

Now an important locking theorem can be proved [Gray93][BHG87]:

Locking theorem: If all transactions are well formed and two-phase, then all transactions will be isolated from each other, and the history can be serialised.

The proof of this theorem is based on showing that the dependency graph of the transactions does not contain any cycles if transactions are well formed and two-phase. If there is a cycle in the dependency graph, then a transaction is executed both before and after another transaction, hence the history is not serialisable.

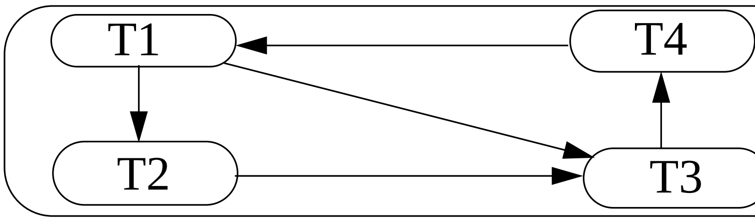


Figure 4-19 *Dependency Graph with Cycle*

An interesting extension of the strict two-phase locking scheme in telecom databases is to use multiple versions of the data. These old versions would then only be used by transactions that read data. These transactions could either be large query transactions or simple read transactions. By reading the old version of the data, they ensure that there are no concurrency problems. A read transaction will always find data to read. This will be further elaborated in the next section.

4.8.3 Degrees of Isolation

One problem with strict two-phase locking is that an application can read a large part of the database thereby making these parts of the database inaccessible for write transactions. This is not desirable in databases used by many users. Also many applications do not need full serialisation - lower degrees of isolation are sufficient. Therefore four levels of isolation have been defined, degrees 0, 1, 2 and 3.

Degree 0: A degree 0 transaction does not overwrite another transaction’s dirty data, if the other transactions are of degree 1 and higher.

Degree 1: These transactions do not have any lost updates.

Degree 2: These transactions do not have any lost updates and no dirty reads.

Degree 3: This is full isolation as provided by strict two-phase locking.

This leads to the following locking protocols:

A degree 0-transaction only locks data that it writes and releases this write lock as soon as it is finished with the write. A degree 1-transaction also only locks data before writing. It does, however, keep the lock until commit time. A degree 2-transaction sets write locks properly, and it also acquires read locks before reading. It does, however, release the read locks immediately after reading the data. Finally a degree 3-transaction does not release the read locks until the transaction is committed.

This means that degree 2-transactions provide all the benefits except repeatable reads, and this is seldom necessary in large query transactions and definitely not in small transactions. Therefore this transaction degree is of interest. Degree 1-transactions can read uncommitted data and this should be avoided. Degree 2-transactions can get even higher concurrency by reading the old version of the data if the data is write locked.

If a degree 2-transaction reads an object and later updates this object, then it should not release the read lock directly after reading. If the read lock is released immediately, then a lost update can occur and this means that the transaction can only be of degree 0.

4.8.4 Other Isolation Concepts

An issue is whether to lock at tuple level or to lock at page-level. In most of the applications the update activity is performed either on new objects or by users updating their own data. Thereby tuple locking will decrease the contention for locks. Using page locking would increase the contention considerably with respect to the frequency of the update activity in some applications. Another issue is that locking pages do not fit well with the distribution of data, there would be many complications due to locking of pages. Therefore locking is performed on tuples.

Another issue is whether to support delta locks where add and subtract are supported by the locking protocol. There are, however, not very many uses of add and subtract in the applications we have investigated and therefore it has been decided to avoid the complexities that delta locks introduce.

Statistical attributes is a case where delta locks can be useful. It is, however, also possible to solve the problem with statistical attributes by other means.

4.8.5 Conclusion

The conclusion is to use ordinary pessimistic lock methods, strict two-phase locking. Reading data could, however, be performed with some kind of time-stamp if needed to increase concurrency of reads. It is also desirable in some telecom applications to avoid any unnecessary waiting. Then reading an old consistent version of data can be acceptable if it means that no waiting for locks is introduced.

4.9 Log handling

The reason for using logs is the need for atomic and durable transactions (there could be other reasons as well, which are not discussed in this thesis). If a transaction is updating several data objects atomically, then simply overwriting the old data is not enough. There are several methods to achieve durable and atomic transactions. Almost all methods are based on the use of a log. There are, however, methods that perform the updates so that all updates can be performed as one atomic write operation. This is called shadowing, which has performance problems [Gray81] and logs can also be useful in system crashes, when it is needed to recover an old version of the database.

The use of log records involves two parts, the ***REDO*** part and the ***UNDO*** part. The REDO part is used at recovery if the update on the data object was not yet reflected on durable media (e.g. disks). The UNDO part is used if the data object was written and later a decision to abort the transaction was taken.

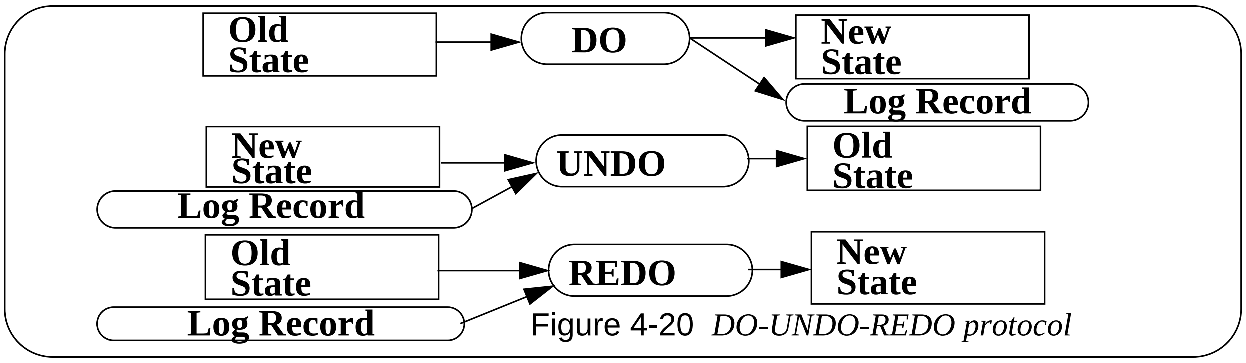


Figure 4-20 *DO-UNDO-REDO protocol*

There are four methods to use the REDO and UNDO part, the UNDO/REDO algorithm, the No UNDO/REDO algorithm, the UNDO/No REDO algorithm and the No UNDO/No REDO algorithm (shadowing) [Gray93].

The UNDO/REDO algorithm is the most flexible algorithm in which pages can be written to durable storage at any time, both uncommitted data and stale data can be found in the data pages. The WAL (Write-Ahead Log) rule must be obeyed, however. It states that before a dirty page is written to disk, all log records that belong to the dirty writes must be flushed to disk. The log can then be used to recover an up-to-date version of the data objects.

The UNDO/No REDO algorithm requires that all updates are reflected on the durable data pages before commit. Thereby all pages that contain updates by the transaction must be flushed to durable media before commit.

The No UNDO/REDO algorithm requires that the data page in the durable media never contains any uncommitted data. Thereby dirty pages must never be flushed to disk.

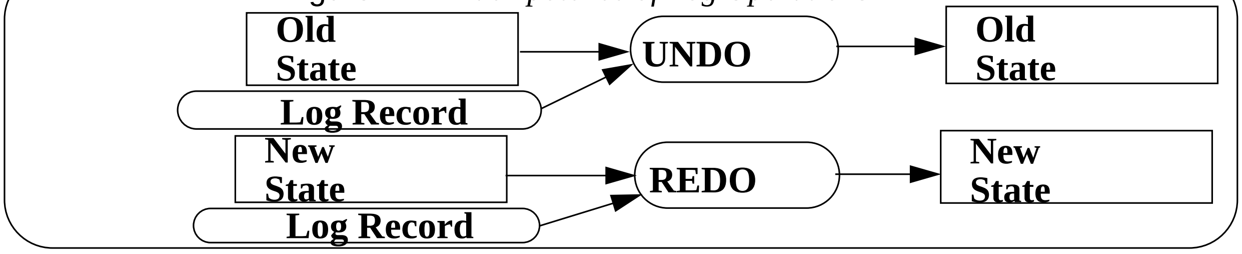
The No UNDO/No REDO algorithm requires that the pages on disk only contain committed updates and these must also be up-to-date. This means that dirty pages are not flushed to disk until committed. When a transaction commits the pages must be written as one atomic transaction. Postgres [Sullivan92] uses this scheme by having versions of the tuples in the pages. This means that both the last committed value and new uncommitted values are found on the disk pages. Another technique used is the shadow technique used by System R [Gray81]. [BHG87] describes these variants of logging in more detail.

In disk databases the UNDO/REDO algorithm is most commonly used. In main memory databases it is common to use the No UNDO/REDO ([DALI94], [DBN94], [DBS91]) since in main memory one can easily have a working record during the transaction and then at commit simply install this as the up-to-date version. The REDO part is, however, still needed since main memory is not a durable medium and also to enable crash recovery where an old version of the database needs to be installed. The No UNDO/REDO algorithm has the benefit that the log records become smaller which improves the performance of the database since logging is a substantial part of the load on the database.

In most implementation of logs a LSN (log sequence number) is used. This number is basically a primary key to the log record. With the UNDO/REDO algorithm there must be references to the next log record to UNDO when a transaction is aborted. This reference must be part of each log record. In No UNDO/REDO it is necessary at least to know where to start reading the log at recovery, so there needs to be an order of the log records and it must be possible to find points in the log. These points could however be start of a log file rather than LSN if these are not used. A log which only contains REDO information is always started from the recovery start point and read until end-of-log or any other marker in the log is found.

If delta locks are used, the LSN can be used to give idempotent log records, the log record is then applied only if Tuple(LSN) < LogRecord(LSN). If semantically rich locking schemes are not used, the read, update, insert and delete operations are idempotent in themselves.

Figure 4-21 *Idempotence of Log Operations*



A log is basically an append-only file that grows all the time. Since files cannot grow eternally the implementation is mostly done by using a set of files, when one file is full, the log moves to the next file. The LSN usually consists of a concatenation of file number and byte address in the file that the log record is placed in.

When using the UNDO/REDO algorithm in combination with LSNs, it is necessary to also introduce CLR (Compensation Log Records) [Hvasshovd96] to be able to get an always increasing LSN number on the records (or pages); the CLR is used to log UNDO of an update. CLRs are not needed in the No UNDO/REDO algorithm since UNDO by using log records is never performed.

Also to avoid being forced to REDO the log since the beginning of the log there must be checkpoints where the system knows that all transactions before the checkpoint have been installed in the data pages. This means that at a restart, it is only necessary to perform the REDO of the log from the checkpoint, while all other log records have already been installed in the data pages.

There are three ways to achieve checkpoints; transaction consistent checkpoints, action consistent checkpoints and fuzzy checkpoints. Transaction consistent checkpoints means that the checkpoint on disk contains only committed updates. Action consistent checkpoints means that data on disk contains uncommitted updates but all database operations are completed. This means that the data on disk is consistent even though there still remain uncommitted updates. Fuzzy checkpoints have no guarantees on consistency and also contain uncommitted updates.

Normal implementation of both transaction consistent and action consistent requires that all updates to the data pages are postponed for the duration of the checkpoint, but this is clearly undesirable in a highly available system. Therefore fuzzy checkpoints are used.

However, action consistent checkpoints and transaction consistent checkpoints can be achieved by using a combination of fuzzy checkpoints and logging of physical page updates. This is performed in INFORMIX-OnLine Dynamic Server [Informix94] and in Dali [DALI97].

Fuzzy checkpoints are created by first writing a checkpoint log record, then writing one data page at a time to disk; this means that when all data pages have been written to disk, all transactions preceding the checkpoint log record have been flushed to disk. If a crash occurs during the checkpointing, the system has to start from the last checkpoint log record of a completed checkpoint.

4.9.1 Log Information

The REDO and UNDO information specifies an update in the database, and this information can be of different types [Gray93].

One type is called physical logging, which means that the updates of the data structures internally in the DBMS are logged. This basically means that updates of pages are logged. This means that one logical update operation could result in many updates of the pages and thereby many log records. This is a simple method that requires a large amount of log information.

Another type is logical logging. In this scenario only the logical update is logged and no information on updates of the pages are logged. This creates problems with atomicity, as a database update contains a number of updates on pages rather than a single update. It is very difficult to create an atomic update of all the pages with good performance.

The third type is a mixture of the two previous which is called physiological logging. In this scenario the log information is logical within pages but physical to pages. Thereby the log records describe the logical operations on the pages that were updated.

4.9.2 Log Handling in Distributed Systems

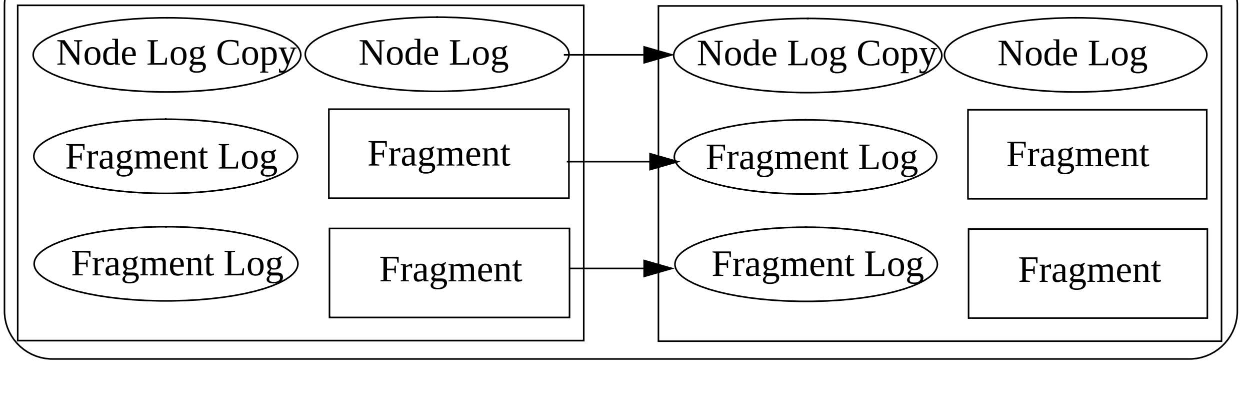
In a Distributed Database several new problems and new possibilities occur that needs to be handled. One method to decrease the load on the disks using the UNDO/REDO algorithm is to use nWAL (neighbour WAL) [Hvasshovd96]. This means that by updating the log in two main memories, the log is as safe as it would have been on disk. This means that writing to disk is only necessary when the log pages are full.

When using No UNDO/REDO the normal two-phase commit protocol is sufficient to ensure reliable data storage. If the data is replicated it will then be updated in several main memories to ensure that the update is not lost.

Another issue that comes up is the requirement on automatic recovery at node failures. This boils down to being able to move and copy fragments between processor nodes. This means that log records created in one processor node must also be able to recover another processor node. This means that the log records must be Location and Replication Independent [Hvasshovd96].

To accomplish this, the log records must be logical and this involves problems. The problem is that using logical log records the checkpointing process must be action consistent and this was undesirable. To overcome this problem, the log is divided into local logs which are physiological and the fragment logs which are logical. The local log contains a log of updates to pages, which are local to the node. The fragment log contains the updates on the database records. Thereby the local log can be used to create the action consistent state needed to execute the logical fragment log. This is shown in Figure 4-22.

Figure 4-22 *Location and Replication Independent Log Records*



There is yet one more problem to handle: fragments can be split and they can be joined, and the fragment logs must handle this, too. This can be solved by putting a split log record in a fragment log when the fragment and similarly a join log record is put in the fragment log when the fragment is joined.

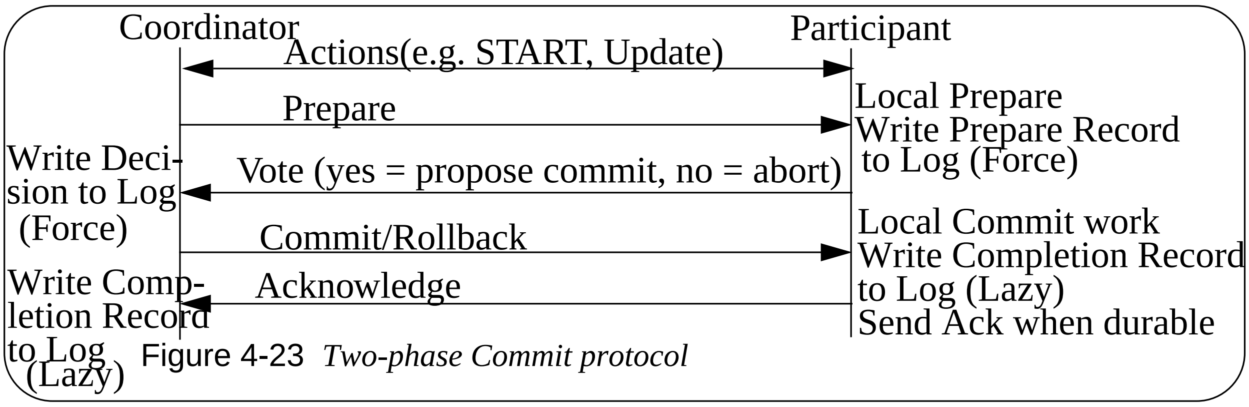
4.9.3 Conclusion

The conclusion is that it is necessary to use Location and Replication Independent Log Records of all updates to the database. The internal updates on pages are logged in a local log.

Previous implementations of this concept have used local logs in combination with an UNDO/RE-DO log [Informix94][Dali97][Tor96]. We will, however, use No UNDO/REDO through the use of new ideas on tuple storage. By using No UNDO/REDO without semantically rich locking schemes, the updates are also idempotent and thereby updates can be performed several times on a record without risk of inconsistency. This simplifies the recovery algorithms.

4.10 Two-Phase Commit

A well-known method to achieve distributed transactions is the two-phase commit. The two-phase commit consists of three phases, of which two are concerned with committing the transaction. In the first phase, the updates of the transaction are performed. The second phase asks all participants to prepare their part of the transaction for commit and reply with yes if it was possible. This yes is also a promise to commit the transaction if told to do so. If all participants voted yes, the third phase commits the transaction otherwise the transaction is aborted. The protocol actions is shown in Figure 4-23.


Coordinator
Participant
Actions(e.g. START, Update)
Prepare
Local Prepare
Write Prepare Record to Log (Force)
Write Decision to Log (Force)
Vote (yes = propose commit, no = abort)
Commit/Rollback
Local Commit work
Write Completion Record to Log (Lazy)
Send Ack when durable
Acknowledge
Write Completion Record to Log (Lazy)
Figure 4-23 *Two-phase Commit protocol*

The participants in the transaction can actually have a number of sub-participants that are controlled by them without the knowledge of the coordinator. The coordinator and participants can be organised in a hierarchical scheme as seen in Figure 4-24.

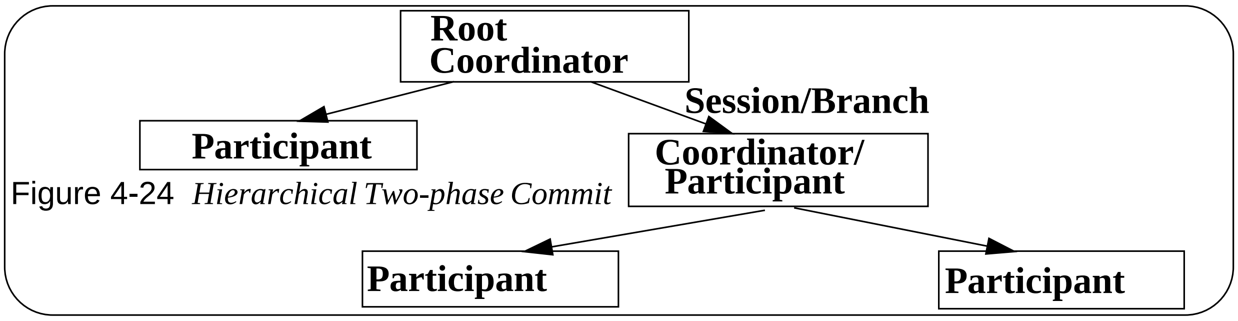


Figure 4-24 *Hierarchical Two-phase Commit*

Now we will proceed to show some optimisations and extensions of the two-phase commit protocol that are useful in a reliable telecom database.

4.10.1 Transfer-of-Commit

The first issue is the possibility to transfer the commit decision. This is very useful in situations where the client is unreliable and the server is reliable. Clearly it is not desirable that a user with his home PC should be able to be commit coordinator using a telecom database. This would clearly produce undesirable reliability characteristics of the telecom database.

4.10.2 Read-Only Commit Optimisation

When transactions are read-only, no prepare phase is needed. Only the commit phase is needed to release the locks and also read-only participants in read/write transactions can use this optimisation.

4.10.3 Linear Commit Optimisation

Performing the full two-phase commit protocol contains many messages. An optimisation which decreases the number of messages involved in a two-phase commit is the linear commit [BHG87]. This gives a higher delay to decrease the communication costs of the transaction. Even with the optimised communication schemes used in this thesis, it is still desirable to decrease the communication cost. The linear commit optimisation performs a transfer-commit to the next participant by sending the prepare message all the way until the final participant receives the prepare message. He notes that he is the last on the chain, commits the transaction and sends the commit decision back to the other participants (see Figure 4-25). There is also a complete phase in this protocol, which could be piggy-backed on other messages in the telecom database.

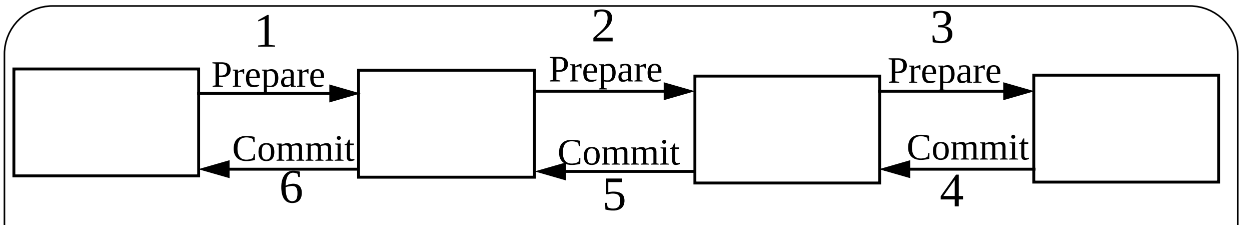


Figure 4-25 *Linear Commit Optimisation*

4.10.4 Non-blocking Commit Coordinator

The final extension of the two-phase commit protocol that we need is to avoid the reliance on one commit coordinator. This could lead to all participants being blocked until the coordinator can be restarted and the locks must therefore be kept for a long time which decreases the system availability. To avoid this, a backup coordinator is always used in all transactions as shown in Figure 4-26.

In this scheme, no decision is taken until both the primary and the backup coordinator has recorded the decision. If the primary coordinator fails, then the backup coordinator can take over immediately since no commit decision is taken without informing the backup coordinator. If both coordinators fail, we do, however, get into a blocked situation again since no coordinator knows the status of the transaction.

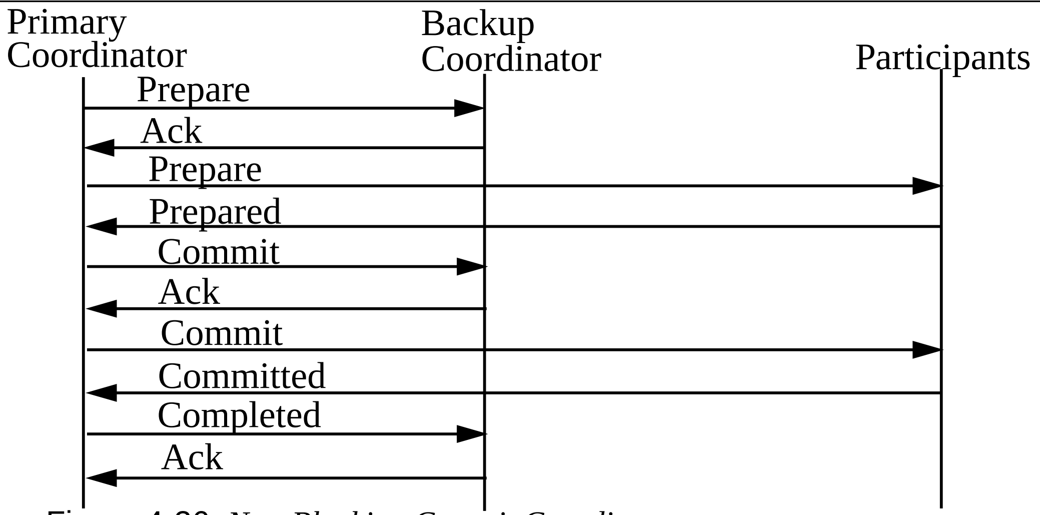


Figure 4-26 *Non-Blocking Commit Coordinator*

4.10.5 Conclusion

Since our design relies on high availability the transaction should not be blocked easily. In the normal two-phase commit protocol a transaction is blocked when the transaction coordinator fails. This is clearly not acceptable. One approach would be to use a non-blocking commit coordinator. This is the approach taken in ClustRa [ClustRa95]. This would create a performance overhead that is not desirable. We choose another method where a system coordinator becomes the new transaction coordinator after failure of the transaction coordinator as described in [CHAMB92]. The new transaction coordinator creates the status of the transaction by asking all processor nodes to supply information on all transactions where the transaction coordinator has failed. If any node knows of a commit decision, then the transaction is committed. If any participant has not been prepared, then the transaction is aborted. If all transactions have prepared but no participant has committed, then the status of the transaction is unknown. If the data server is coordinating the transaction, then the transaction can be aborted in this situation except in rare cases. When the data server is a participant in a transaction distributed over several systems, then the transaction cannot be aborted in this unknown state.

Using a traditional two-phase commit protocol has the benefit of a low delay. Using a linear commit has the benefit of a smaller communication overhead with a larger delay. Our replication structure normally involves three replicas of each data written, and there could also be several fragments involved in each transaction. TPC-B is a typical example of such a transaction [Gray91]. Thereby using linear commit would sometimes increase the delay too much. Using a normal two-phase commit protocol would be advantageous from a delay point of view but increases the number of messages from 24 to 48 in the TPC-B transaction using three fragment replicas. Our solution is a mixture of the normal two-phase commit protocol and the linear commit protocol. Internally between the replicas in a fragment, linear commit is used. Between the fragments we do, however, use a normal two-phase commit protocol. This gives an extra delay but decreases the number of messages from 48 to 32 in the TPC-B transaction. In simple transactions this involves only one fragment and the commit protocol degenerates into the linear commit protocol. In this case some optimisations can be performed to further decrease the number of messages.

4.11 Archive Copy Production

To create an archive copy, it is necessary to be able to recreate a consistent state of a distributed database. To perform this it is necessary to have a point in time from where it can be decided for each transaction whether it happened before or after this time. We will call this time a global checkpoint. This checkpoint will be transaction consistent.

One way to do this is to simply broadcast a stop message to cease all commit processing and then insert a checkpoint in all logs and then the commit processing can continue as usual. The new global checkpoint can be written to disk in a lazy manner to assure that the system only stops commit processing a very short time. Then an archive copy that is transaction consistent can be created by reading the database from a checkpoint preceding the global checkpoint to restore and apply all log records with a smaller global checkpoint than the global checkpoint to restore.

More sophisticated methods that have a smaller impact on the delay of transactions can be invented [DBN94]. In systems with up to about fifty processor nodes the delay of creation of a global checkpoint can be done within less than one millisecond and this should be sufficient since it does not seriously impact real-time behaviour. Such a figure assumes that a real-time run-time system is used with the use of priorities.

5 System Architecture of the Parallel Data Server

In this section of the thesis the system architecture will be described. This encompasses description of replication structures of the data, the various protocols needed and their relationships. Handling of keys, schema, fragmentation and secondary indexes.

5.1 Replication Structure

In this section we will describe the replication structure. There is replication on two levels where the first is internal in the system and the second is between systems. We will first show the replication method internal in the system.

In our models we will assume that mean time to failure (MTTF) of a processor node is 4 months due to software failures and that other failures decrease the MTTF to 3 months. The repair time is one hour (MTTR). The MTTF is based on a report from Japan discussed in [Gray93] and the MTTR is based on a structure where recovery is done by a copying process from the primary node to the new backup node.

5.1.1 Local replication structure

The ideas of this section have been developed in discussions with Shahid Malik and more details on error handling is found in [MALIK97].

As in most distributed databases the replication objects are fragments of a table. A fragment contains a subset of the table. The fragmentation function is further described in Section 5.3.3. In this section we presume that a fragmentation has been performed and that it is possible to deduce where the replicas of a fragment reside and of which type they are.

The most common method of replicating information is to use a ***primary node*** and a hot stand-by node. We will call the hot stand-by node a ***backup node*** in this article. This provides very reliable operation. The probability of both the primary and the backup node failing at the same time is very small.

If both the primary and the backup node fails for any fragment, then committed transactions could be lost if a write to disk of the log is not involved in the commit process. Then the consistency of the whole system is jeopardized causing a shutdown of the system. Thereby the system fails in this case if two nodes fail at the same time. Assuming the above MTTF and MTTR we calculate the probability of a shutdown of the system in an hour to be $(n^2-n)/2200^2$ (2200 = number of hours in 3 months), where n is the number of processor nodes in the system. If a shutdown causes the system to be unavailable for two hours then a shutdown of the system can be acceptable at most once per 20 years of system operation. This means that if n is greater than six the reliability requirements can no longer be met.

To avoid the problems with reliability requirements a scenario could be to write all transactions to disk at commit time when only a primary is alive. This will however lead to that those transactions will suffer a long delay and also the performance will be impacted at failures.

This means that using the primary and backup strategy, it is not always possible to fulfil both the delay requirements and the reliability requirements. It is likely that the reliability requirements are more important and therefore the probability of a transaction not meeting the delay requirements becomes $n/2200*(2/n) = 1/1100$. The factor $2/n$ comes from the fact that not all transactions are affected by a node crash. Only those transactions which try to access a fragment which had either the primary or the backup on the failed node. If transactions consist of several operations then the probability of missing the delay requirements increases even more.

To write to disk during the commit process also creates a performance overhead. The overhead of writing a disk block is similar to the execution of database requests in processor nodes. This overhead, too, only comes at a time when a node has failed, and in this case the primary node already has an overhead to assist in creating the new backup node.

One solution to this problem is to use two backup nodes. This creates an increase in the price of the system which is not always acceptable. All the memory needs to be triplicated in this case.

Our solution to this problem is to introduce a third type of replica, the ***stand-by replica***, we will call the owner of this replica the ***stand-by node***. The stand-by node participates in all transactions, but only logs the transactions. A similar idea was used in [RRDF93]. The difference was that log records were sent to the stand-by node after committing the transaction. This type of replication is referred to as 1-safe. In our case the stand-by node is involved in the two-phase commit protocol, and this type of replication is referred to as 2-safe.

The stand-by node only stores the log records of the transactions. These log records are sent to disk as soon as possible without causing a performance overhead. Thereby only a small log buffer is needed in main memory. Also the processing in the stand-by node is very small and therefore it does not cause any large performance overhead. This conclusion relies on the assumption that communication between processor nodes is cheap. This has not always been the case. TCP/IP messages on Ethernet has a cost in the same order as the execution of the database requests. New technology, such as SCI [DOLPH96] and MemoryChannel [GILL96], has made it possible to send messages by using a shared memory between processor nodes. This decreases the load of communication by a factor of 10 and sometimes even down to a factor of 1000. Thereby communication costs are not such a severe performance problem any more.

The replication structure used is shown in the figure below.

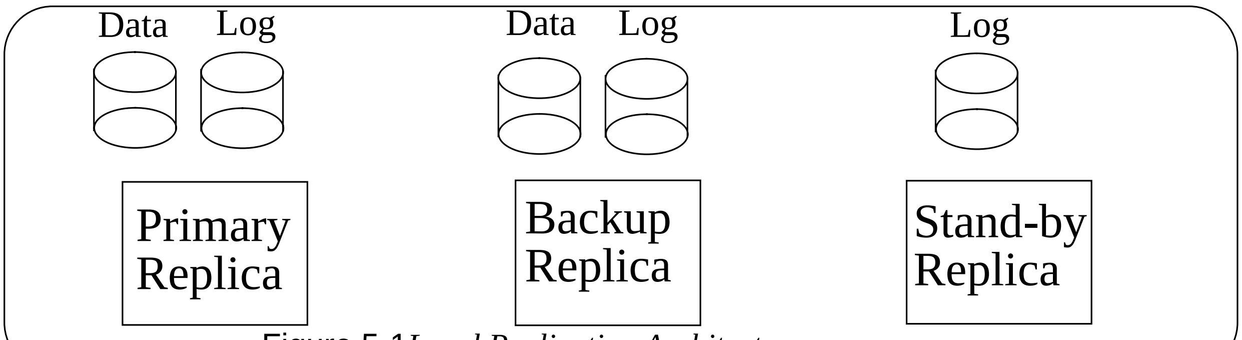


Figure 5-1*Local Replication Architecture*

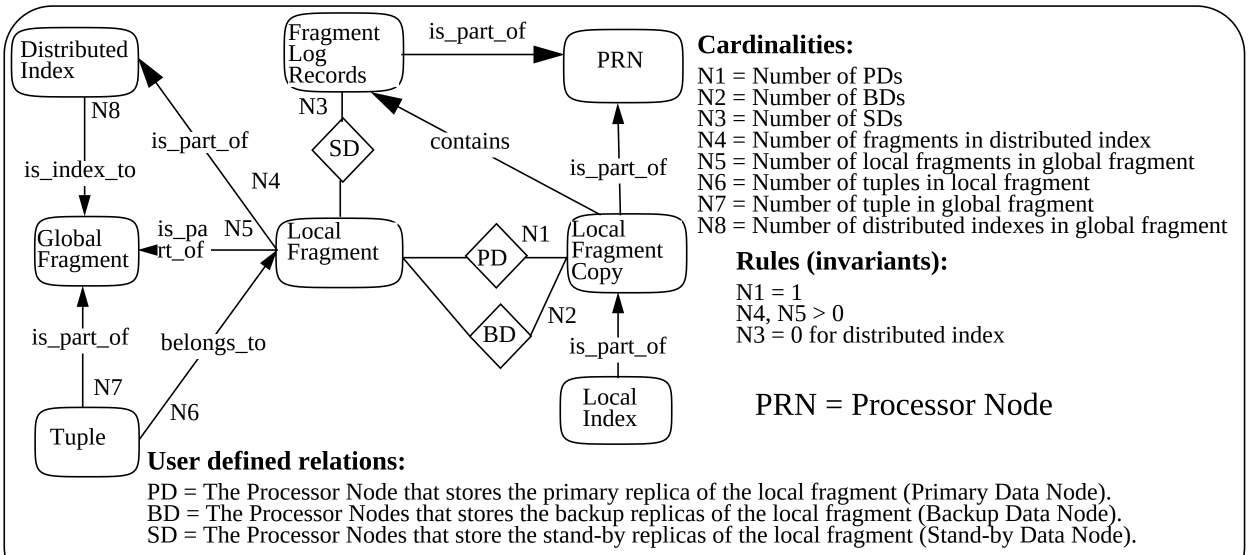
The benefit of the stand-by node is that even when the primary and all backup nodes have crashed, the stand-by node ensures that the system is still available. Some fragments in the data server have become unavailable but the system is still available. The stand-by node can then be used to restart a new primary node. The stand-by node needs access to an archive copy in addition to its log records to restart a new primary node. To restart primary and backup nodes is a costly process that involves creating an up-to-date replica of the data. This creation process could take up to an hour if the size of the database is large. To start a stand-by node is rather easy and can be performed in a few seconds.

Another option is also to use a primary node together with a stand-by node. This provides a good reliability although the availability is rather low.

The replica in the primary node will be called ***primary replica***. All updates start in the primary replica, and the write locks are first granted there. When the primary replica fails, a new primary replica must be assigned to take over to be able to continue processing transactions. The primary replica is involved in all read operations that are part of updating transactions. As will be shown in 8.2.1.1, within a system all replicas are involved in read operations of updating transactions when there is also replication on the global level.

The replicas in the backup nodes are called ***backup replicas***. The backup replica is always up-to-date and is part of all update transactions where the fragment is involved. This replica, too, contains locks on the data. The reason is that the backup replica can be used for special read operations. It can handle read with commit, where only one tuple is read with degree 3. It can also handle read-only transactions of degree 2. This means that all reads on the backup replica only hold the lock on a tuple while reading it, and when the tuple has been read the lock is released. Thereby deadlock situations are never introduced due to reading the backup replica. It does also simplify handling of failures of backup nodes. The backup node does not contain any unique data, such as read locks, and therefore failure of the backup node is easy to handle. Being able to perform simple reads at the backup node introduces a possibility for load regulation as we have seen that most telecom applications contain many simple read transactions.

In Figure 5-2 a conceptual model of the distribution of fragments within a telecom database is shown. All replicas, the primary replica, the backup replicas and the stand-by replicas within a telecom database are 2-safe, that is they all participate in the two-phase commit protocol.



Cardinalities:

N1 = Number of PDs
N2 = Number of BDs
N3 = Number of SDs
N4 = Number of fragments in distributed index
N5 = Number of local fragments in global fragment
N6 = Number of tuples in local fragment
N7 = Number of tuple in global fragment
N8 = Number of distributed indexes in global fragment

Rules (invariants):

N1 = 1
N4, N5 > 0
N3 = 0 for distributed index

PRN = Processor Node

User defined relations:

PD = The Processor Node that stores the primary replica of the local fragment (Primary Data Node).
BD = The Processor Nodes that stores the backup replicas of the local fragment (Backup Data Node).
SD = The Processor Nodes that store the stand-by replicas of the local fragment (Stand-by Data Node).

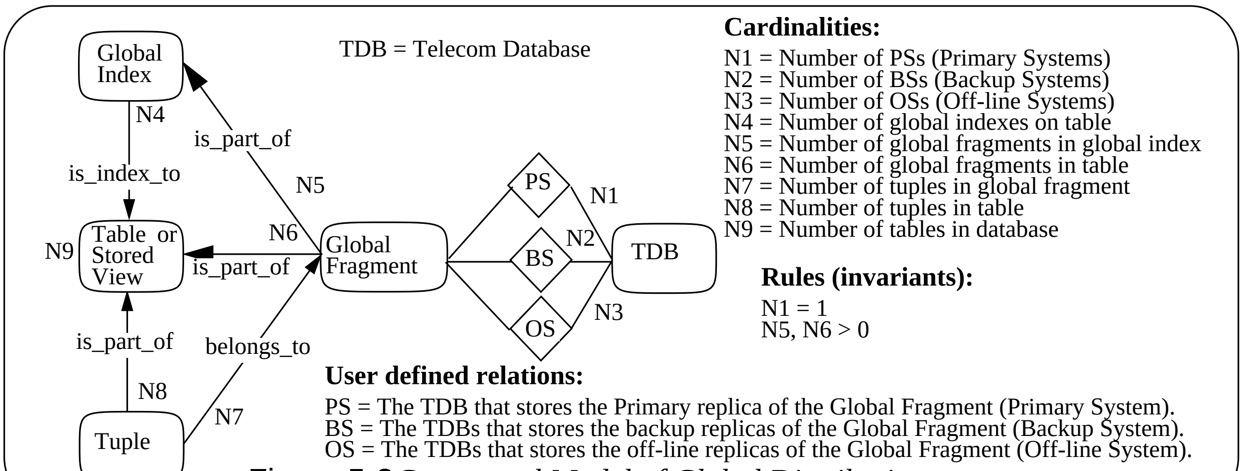
Figure 5-2 *Conceptual Model of Local Distribution*

A local fragment could be either a primary replica, a backup replica or a stand-by replica. A local fragment itself is part of either a global fragment or a distributed secondary index. As seen in Figure 5-3 a global fragment is part of a table or a stored view or a globally distributed index. From this we deduce that the architecture should use two fragmentation functions. The first fragmentation function fragments the table between the telecom databases and the next fragmentation function fragments the global fragment within a telecom database. Since a local fragment is a part of a global fragment, the global fragmentation function must ensure that tuples in a local fragment are not part of more than one global fragment. In this thesis the local and global fragmentation are the same, which means that there is a one-to-one mapping between global and local fragments. This is not a requirement but a simplification used in this thesis which should not be of any great impact as long as the global replication protocol is not standardised. If two telecom databases developed by different vendors are to cooperate, it is likely that they would not use the same fragmentation scheme.

5.1.2 Global replication structure

Between telecom databases we are mainly concerned in providing safety against catastrophic situations, although the methods we develop can also be used in globally distributed databases. To support catastrophic situations with one extra copy is usually sufficient. This copy should be possible to start as soon as possible and we do not use stand-by replicas in this case. The primary replica on system level is stored in the ***primary system***.

This backup replica on system level is stored in the ***backup system***, and it can use a combination of 1-safe and 2-safe transactions. The decision is based on which data is involved in the transaction. The decision as to which data is 1-safe and 2-safe is decided by the application developer. The decision is on an attribute level, where updates involving attributes that must be 2-safe means that the whole transaction must be 2-safe. If no such attributes are involved the transaction can execute as 1-safe. Some applications might also need an off-line replica stored in an ***off-line system*** that receives all transactions as 1-safe. They could also receive the transactions in a delayed manner. Therefore we will discuss three types of global replicas, the primary replica, the backup replica, and the off-line replica.



Global Index
TDB = Telecom Database
N4
is_part_of
is_index_to
N5
PS
N1
N9
Table or Stored View
N6
is_part_of
Global Fragment
N2
BS
TDB
OS
N3
is_part_of
belongs_to
N8
N7
Tuple
Cardinalities:
N1 = Number of PSs (Primary Systems)
N2 = Number of BSs (Backup Systems)
N3 = Number of OSs (Off-line Systems)
N4 = Number of global indexes on table
N5 = Number of global fragments in global index
N6 = Number of global fragments in table
N7 = Number of tuples in global fragment
N8 = Number of tuples in table
N9 = Number of tables in database
Rules (invariants):
N1 = 1
N5, N6 > 0
User defined relations:
PS = The TDB that stores the Primary replica of the Global Fragment (Primary System).
BS = The TDBs that stores the backup replicas of the Global Fragment (Backup System).
OS = The TDBs that stores the off-line replicas of the Global Fragment (Off-line System).

Figure 5-3*Conceptual Model of Global Distribution*

5.1.3 Local Fragment states

To better understand the reliability and availability features of our model the state diagram of a local fragment is shown in Figure 5-4. The state that must be avoided is the unrecoverable state. If this state is reached then committed transactions might be lost and thereby the consistency of the whole database is unsure. Thus the system must be shut down and restarted from a checkpoint that is transaction consistent. The restart procedure will be described later in the thesis. All other states are recoverable and no shutdown of the system is necessary.

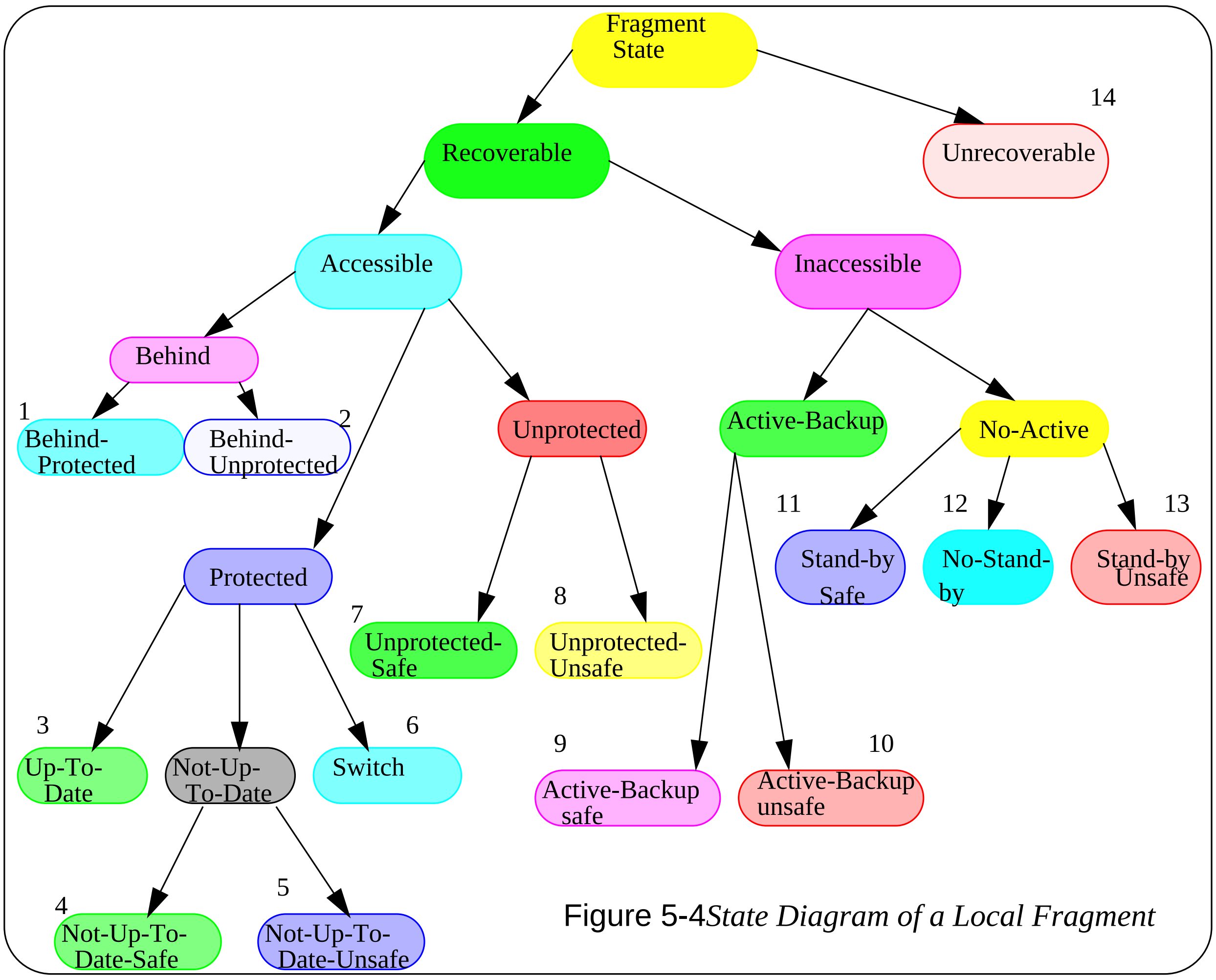


Figure 5-4*State Diagram of a Local Fragment*

The fragment can either be available for user queries (accessible) or not. If there is a primary replica the fragment is in the accessible state, otherwise it is in the inaccessible state. Every time a node fails, the state of the fragment goes to one of the unsafe states. Since node failures often are correlated, all nodes should immediately force the log to disk at failure situations and also force the log to disk at commit for a certain period of time after the node failure. When one node has completed the force of the log to disk, the state reaches one of the safe states. If there is only a primary replica and no other replicas, the fragment is in the unprotected state, otherwise it is in the protected state. In the unprotected state the log is always forced to disk at commit since there is only one node that has a copy. When being in one of the accessible and safe states the recovery procedure can start and the state goes to Behind, which means that a recovery process is ongoing.

If there is no primary replica but there is a backup replica, the fragment is in the Active Backup state. Then when reaching a safe state, the backup replica can be promoted to primary replica and the recovery process can start.

If neither a primary replica nor a backup replica is available, then if a stand-by replica is available the fragment reaches the No-Active state. When it has reached the safe state here the recovery process can start by using a protocol to promote a stand-by replica to a primary replica. This protocol can only be used at a node which contains an archive copy of the fragment saved on disk. If the stand-by node does not contain this, it waits until some processor node that contains an archive copy is restarted. Then the restarted node can use the same protocol to promote the archive copy to primary replica by using the archive copy and the log that was saved in the stand-by replica.

In the unusual case that the stand-by node also fails when in the Stand-by Safe state, then the state becomes No Stand-by. In this state there is no primary replica, no backup replica and no stand-by replica. However, information is available on disk to start up a new primary replica without losing any committed transactions. Thereby the system can continue operating although this fragment is unavailable.

In Figure 5-5 an example of a local distribution is shown. An important issue in the replication structure is how to handle failures. Our approach is that as soon as failures occur, the system itself should reconfigure itself. If node 3 fails in Figure 5-5 the fragments F1, F2 and F3 are left with one less backup replica, thereby reaching the Not-Up-to-Date-Unsafe state. An I-am-alive protocol which will be described in section 7.1.1 finds out that node 3 has failed and the system coordinator distributes this information. Node 1 is the primary configurator of F1 and will decide how the new distribution of replicas will be. Assume that it decides to create a new backup replica at node 5, then a new backup replica is created at node 5 using the copy fragment protocol described in section 7.1.4. If node 1 also fails before the backup replica at node 5 has been created this recovery process is lost and the fragment reaches the state Active-Backup unsafe. The backup configurator of F1 is promoted to primary configurator and decides how the new distribution of replicas will be. It does promote its own replica to primary replica.

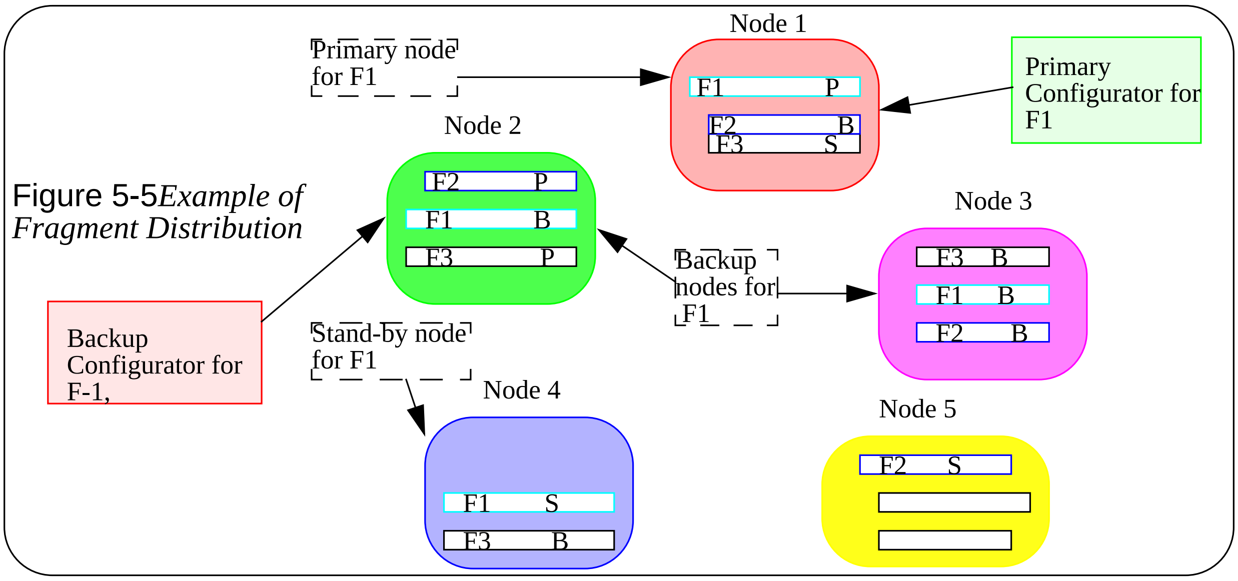


Figure 5-5*Example of Fragment Distribution*

If there is replication also at the global level, then the state machine of the local fragments must be connected to the state machine of the global fragment they belong to.

When the local fragment state goes to Unrecoverable, then the system is no longer usable. If this state is reached and there is a backup system, then the backup system must take over since the primary system is going to be shut down and restarted. If the backup system goes to Unrecoverable, then the backup system must be restarted.

All other Inaccessible states in the primary system, Active Backup and No-Active states, do not affect the state of the global fragment, since these are only temporary malfunctionings that can be solved by aborting the transactions.

The state No Stand-by is not allowed at the backup system. When there is not even a stand-by node in the backup system, the backup system cannot participate in the transactions committed at the primary system. Thereby the backup system becomes Unrecoverable if it goes to the state No Stand-by.

5.1.4 Related Work

Most of the replicated databases uses a primary and hot stand-by architecture. The updates are first performed at the primary node and then either 1-safe or 2-safe or a mixture of 1-safe and 2-safe is used to update the hot stand-by node. The hot stand-by node is then ready to take over processing when the primary node fails. Examples of such systems are Tandem Remote Duplicate Database Facility [Tandem91], Sybase Replication Server [Sybase95], CA-Open Ingres/Replicator [Ingres93], Informix-OnLine Dynamic Server [Informix94] and ClustRa [ClustRa95]. Most systems that support shared nothing also support some kind of fragmentation. Shared disk architectures usually only support replication of a site and thereby only support a two-node system. Each node could then of course be a powerful symmetric multiprocessing node (e.g. Sun Enterprise server).

Using a stand-by server for storage of log records can be found in IBM Remote Site Recovery [RSR95] and Remote Recovery Data Facility [RRDF93]. The stand-by server gets the log records in a 1-safe fashion. The idea is to raise fault tolerance in already installed IBM systems.

5.1.5 Conclusions

There are no products or research prototypes known to the writer of the thesis, where a system using primary/hot stand-by architecture has been combined with a 2-safe stand-by server to save the log records. The stand-by server is a very cheap way of achieving higher reliability of the system. It does not involve any great processing overhead and the main memory overhead is neglible. These benefits are gained although a 2-safe stand-by server is used due to the cheap cluster communication used by the run-time system and also optimisations in the two-phase commit protocol as described in section 6.1.

The replication structure described in this thesis avoids system failures as long as not all replicas fail within a very short time (about 100 ms). This can be supported by other database management systems, but only at the cost of forcing the log to disk at each commit. In this system no disk operations are needed during write transactions.

Since the normal read/write operations do not involve disk operations, the traffic to the disks is not so serious. Also since communication is cheap the communication overhead is small. This means that a very reliable system can be built using small and cheap modules with standard hardware. If a high performance system is needed, the nodes can instead be multiprocessor nodes with high-performance processors. Those multiprocessor nodes should mainly be one-board multiprocessor nodes since these nodes gives the best price/performance ratio. Even single processor systems can be used. In this case the disk is used for all commits and thereby some reliability and performance is sacrificed. A minimum reliable system is using two nodes and a minimum system with very high reliability should be using a minimum of four nodes.

5.2 Reconfiguration and Recovery Protocols and the Relation Between Them

The protocols in the parallel data server have complex relationships. We will start the description of these relationships by describing what types of information that is needed in a parallel database and how we choose to replicate this information. Then all protocols described later are simply various methods to change and recover this information.

A parallel database differs from a centralised database in that it must be able to continue operation even in the presence of node failures. All the protocols in a parallel database must have a well defined action in the presence of simple and complex scenarios of node failures. This is the most serious complication in building a parallel database which is very reliable.

Other actions that reliable parallel databases must handle concurrently with user transactions are: adding new nodes, dropping nodes and updating schema information. They must also be able to react on load changes and change the distribution of data as a means to regulate the load. Other means of regulating the load must also be supported. On top of this the underlying run-time system must be able to handle software changes, fault tracing and memory management and still provide real-time services to the applications.

5.2.1 Information in a Parallel Database

First a little thought on the basic idea of a database. The idea is to store *application information* in a separate entity that handles the data so that the application need not worry about data structures, safe storage, recovery and complex transaction handling. It also provides advanced search facilities to the data. However the basic idea is to insert, update, delete and retrieve application information using transactions.

5.2.1.1 Information Structure in a Parallel Database

To be able to find the application information after storing it, the information must be structured. The application information is structured in tuples. These tuples are organised into tables. The tables are defined by a set of attributes. To find the data one must usually also have an index on the tuples. The simple concept of tuples, tables, indexes and attributes is the basis of databases.

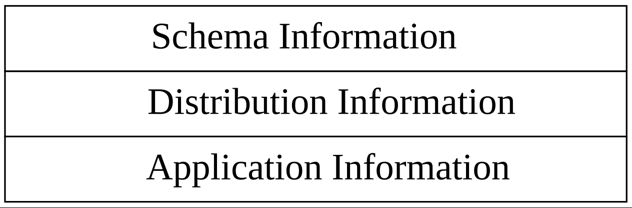
The database must contain a definition of the tables and their attributes and indexes. Since this information can also change, it must be treated with care too. We refer to this information as *schema information*. Modern databases have included a lot of application support in addition to the simple concept of tables, indexes and attributes. Tables can have relationships to each other in their definition through inheritance and attributes can be complex. Also the database can provide alternative views on data; these can even be materialised and kept up-to-date with the original information.

Referential constraints and other constraints can be maintained. Finally triggers that react on database operations are often supported by database management systems. The reaction could either perform another database operation, but it could also be to inform applications on database events.

All this complex set of additional information about the database is maintained by the database management system. This additional information is also a part of the schema information. The schema information must also be updated through transactions so that changes of the schema information can occur concurrently with user transactions.

In a parallel database the tables can be fragmented to enable storage in more than one node. These fragments can also be replicated. To maintain this information we also need information about the nodes in our system. This information is updated through the use of special transactions. We call this *distribution information* since it represents the knowledge of the distribution in the system.

Figure 5-6*Information in a Parallel Database*



All these three types of information must be handled by the system and be recovered in failure situations. Figure 5-6 gives a hint of the order of usage of the information. The schema information must always be accessed first in user transactions. The distribution information is needed to find the data and must be checked before accessing the application information.

5.2.2 Replication of Information

The replication of information is another important part of the architecture of a parallel database. All the above three types of information are replicated in some manner. Schema information is needed in all nodes that handle application requests. Schema information is very often read, actually in every transaction. It is updated only rarely. Thereby we decide that schema information is fully replicated in all nodes of the parallel database. Distribution information is also read in every transaction and is updated seldom compared to the number of times that it is read. Therefore we decide that also distribution information is replicated in all processor nodes of the system. The replication of application information and index information was covered in section 5.1 and 5.4.

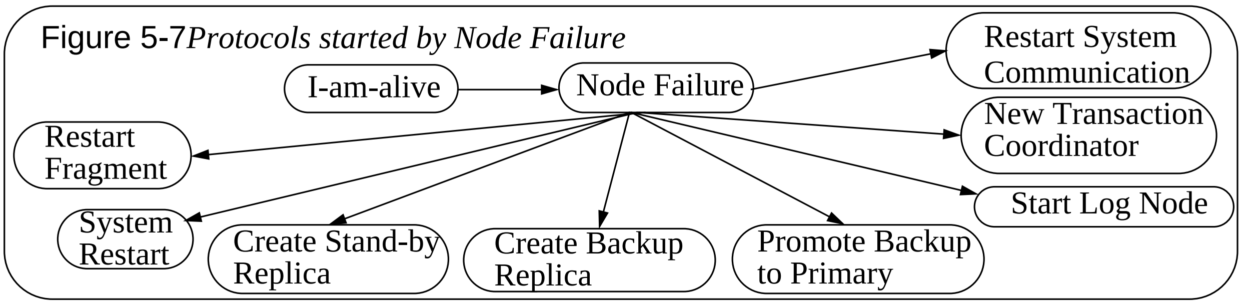
Replication of application information is also performed between systems. Distribution information is unique for each system and is therefore not replicated between systems. Schema information contains different parts. Table and attribute information must be the same in the replicated systems, too. Other schema information is, however, not necessarily replicated, and it can be unique in a system. The existence of a secondary index in the primary system does not necessitate a secondary index in the backup system and vice versa. The schema information that controls the execution of the transaction is the schema information in the primary system. The backup system only needs to know the changes on table and attribute information to act properly. The backup system must however replicate most of the schema information in the primary system to be able to take over as the primary system.

5.2.3 Handling of Node Failures

The recovery protocols include a protocol to find out which nodes are alive. Earlier studies of distributed databases have concluded that node failures must also be serialisable in the transactions [BHG87], therefore a state transition protocol is needed to ensure that node failures are serialisable.

A node failure does affect the reliability of a system. It causes a set of fragment replicas to be unavailable. The idea of the data server is to automatically start a copy process of those fragments that have lost replicas. All protocols in the data server must have methods to handle node failures, since they can always occur and must always be handled properly. As mentioned above the failure of a node can be made into a serialisable event, which makes it a lot easier to handle them in a proper way. In Figure 5-7 all actions that can be started by a node failure is shown. Node failures are discovered by the I-am-alive protocol.

Figure 5-7*Protocols started by Node Failure*



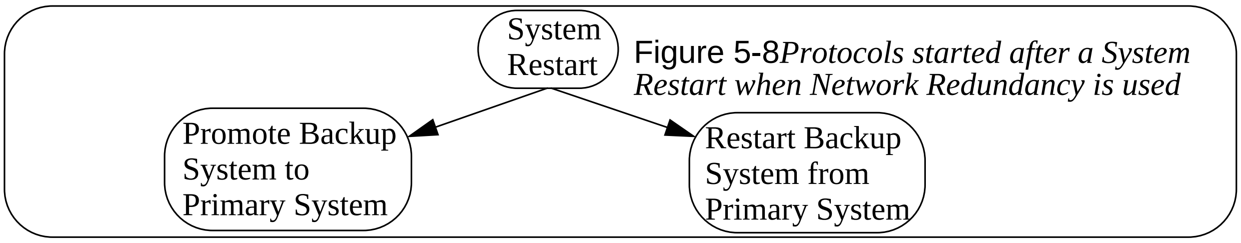
Promotion of backup to primary is used when the primary replica of a fragment fails. Creation of backup replica is used when a backup replica fails and creation of a stand-by replica is used when a stand-by replica fails. Restarting a fragment is used when all primary and backup replicas have failed, and it uses a stand-by replica and a local checkpoint to restart the fragment. System restart is needed when committed transactions have been lost, which happens when the state Unrecoverable is reached as described in section 5.1.3. The failure of a node may cause a number of transactions to lose their transaction coordinator. There is a protocol to start up a new transaction coordinator for these transactions by retrieving transaction information from the participants that are still alive. The failure of a node could cause the failure of a replica of a local log. This log is replicated on another node that could fail. If this happens, a new log node must be started.

If the system is involved in replicating between systems, then a node failure could cause a log channel between the systems to be lost. Then the system communication must be restarted.

The decision to start these protocols is distributed on three logical entities. Fragments are recovered by the owner of the primary replica or the replica that will be promoted to a primary replica. One node in the system is assigned as coordinator, which becomes the new transaction coordinator of those transactions that have lost their transaction coordinator. This node also checks if a system restart is needed. Finally the new log node is started by the owner of the log.

5.2.4 Handling System Failures

If a system restart is needed, the system restarts from an old but consistent global checkpoint. If the system is a backup system, it is dropped as a backup system and restarted by using information from the primary system together with its own information. If the system is a primary system the backup system should take over as primary system and the crashed system should be restarted as a backup system.


System Restart
Promote Backup System to Primary System
Restart Backup System from Primary System
Figure 5-8*Protocols started after a System Restart when Network Redundancy is used*

5.2.5 Handling of Add Node

A new node can be added as a result of a node restart or by the operator placing a new node in the system. The major issue here is to start the new node. To start a node it must first create the distribution information and schema information since this information is fully replicated in all nodes of the system. A node is not fully started until this information has been created. If an operator has placed a new node in the system, then some fragments should eventually be split to use the new node in the most efficient manner. This split is delayed until a proper time to avoid overload situations. A new node helps other nodes when fragment replicas are moved to this node. This move is performed by first creating new replicas in the added node and then dropping replicas in other nodes.

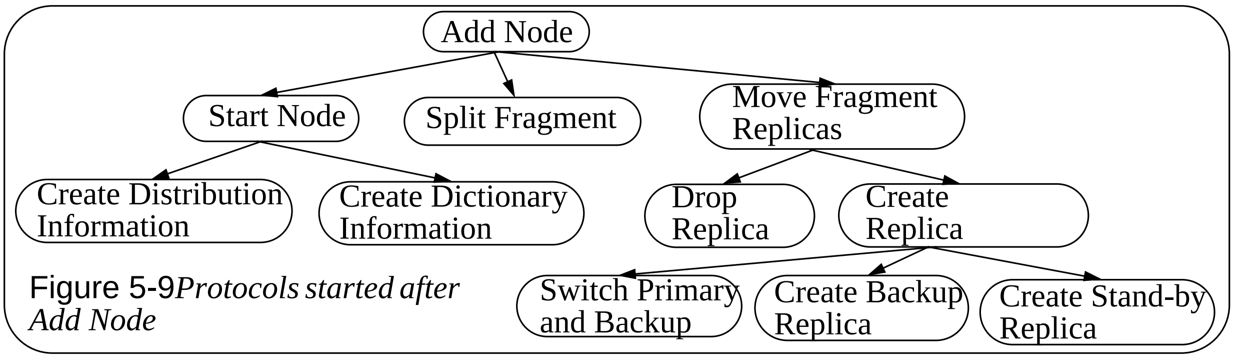


Figure 5-9*Protocols started after Add Node*

5.2.6 Handling of Drop Node

Drop Node is similar to a node failure. Drop Node is, however, an operator initiated event and is therefore a controlled node failure. This means that before the node is allowed to stop, the system must ensure that all fragments are replicated, according to the table definition, in the remaining nodes of the system. This is performed by moving the replicas to other nodes. Also Drop node means that there are fewer nodes in the system and thereby some fragments should be joined. It is not necessary to have a fragmentation that is much greater than the number of nodes in the system. Join fragment is performed at a time when suitable to avoid overload situations. To be able to join fragments is not an essential requirement. Too many fragments in the system only create some memory overhead.

5.2.7 Handling of Time-out Events

All activities to save information for crash recovery are performed at certain time intervals. These activities are therefore started by a time-out event. The I-am-alive protocol is also started by a time-out. There are two basic activities performed to prepare for crash recovery. The first creates global checkpoints. This checkpoint is transaction consistent and represents a point from which the system can be restarted. All transactions are tagged with a global checkpoint identity. Thereby one

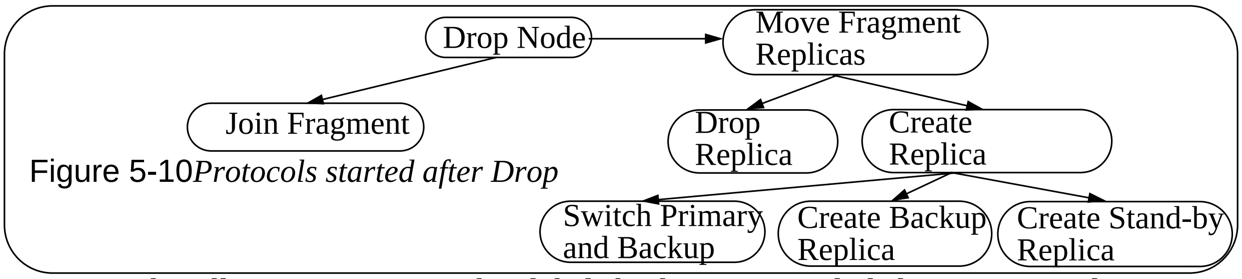


Figure 5-10*Protocols started after Drop*

can ensure that all transactions up to the global checkpoint are included in a restart and no transaction after the global checkpoint is reflected in the restarted database. The actual information used in a crash recovery is found in local checkpoints performed at each node with certain time intervals together with the information saved in the fragment logs.

After a transaction is completed, a check is performed to see whether the global system checkpoint can be increased. When a transaction is completed all log records have been saved on disk. When all transactions belonging to a global checkpoint have completed, this global checkpoint can be recovered at restart. The global system checkpoint is the newest global checkpoint from which the system can be restarted.

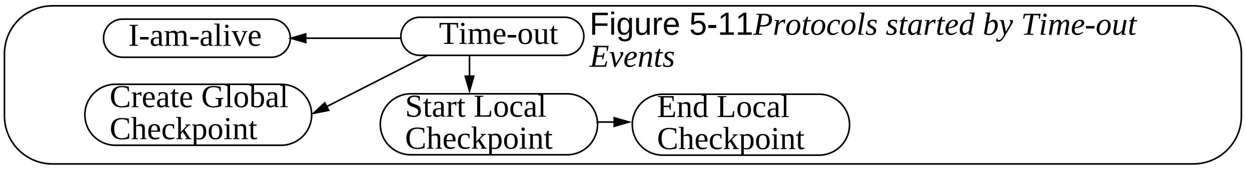


Figure 5-11*Protocols started by Time-out Events*

5.2.8 Handling of Overload Events

Another event that can cause the start of recovery protocols are overload situations. There are a set of load regulations that can be performed in the parallel database. To protect a highly loaded node one can ensure that the overloaded node does not become transaction coordinator. Another step is to ensure that simple read transactions are not performed at that node if not necessary. Another step that can be taken is to switch roles between primary and backup replicas. Primary replicas do more processing since they handle all read operations of update transactions and the primary node also executes methods. Therefore moving primary replicas from an overloaded node is a step that decreases the load on the node. If none of the previous steps has been able to prevent an overload situation, a final step is to abort some transactions that need to access data in overloaded nodes.

If a node is often overloaded, then long-term actions can be taken that moves fragments to other nodes. This action is preferably performed when a low load situation has been reached (if it occurs). If no low load situation occurs, the long-term actions have to be performed to ensure that more traffic can be handled after the change.

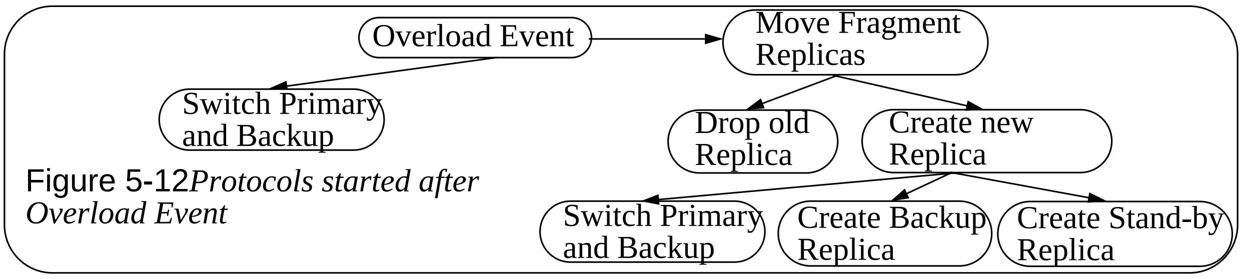


Figure 5-12*Protocols started after Overload Event*

5.2.9 Handling of Schema Changes

Schema changes can require a number of special protocols. The set of possible schema changes is actually very large and in this thesis we do not try to cover all possible schema changes. We will study adding and dropping tables, attributes, indexes and foreign keys. Complex changes of attributes in a table will also be studied. Finally splitting and merging tables will be studied. These schema changes should cover a large part of the common ones.

When two levels of replication are used, the schema changes must also be performed in the backup system in a proper way. This will also be studied for the above set of schema changes.

5.2.10 Other Protocols

A protocol that is normally not used but could be used for specific purposes is to switch between the primary system and a backup system. This protocol can be started explicitly if the operator wishes to replace a telecom database with a newly installed one. This new telecom database could even be developed by a different vendor and the switch protocol is then used to switch between the old and the new system. The switch could also be used in situations where it is necessary to change the load situation on a long-term basis.

5.3 Schema Definition

A number of facts about the schema definitions need to be specified. These concern primary and secondary keys to tables, fragmentation of tables, methods as first class attributes, stored procedures, views, constraints and triggers.

5.3.1 Tuple Keys

In relational databases a primary key is used to get access to a record. A primary key should be unique and is normally related in some way to the application information. In object databases normally an object identifier is used as primary key to a class. The object identifier is generated by the database management system when an object is created. Our investigation of applications in chapter 2 showed that both these types of primary keys are needed. We will call these two key types tuple keys in this thesis. We will call object identifiers tuple identifiers since we have used the word tuple to define the records or objects of the database.

When accessing a tuple in the database the tuple key is used. Without the tuple key the tuple cannot be found in any other way then by scanning the table. If the fragmentation is not based on the tuple key, then it is also necessary to have the attributes on which the fragmentation is based available as part of the query for direct access to the tuple. Fragmentation on the tuple key is performed by using the hash-based LH^3 structure as shown in chapter 11. Fragmentation on other attributes also uses a hash function. The attributes used by the hash function are called fragmentation attributes.

Without this attribute the query has to search in all possible processor nodes for the tuple. The reason that accessing the record goes through the tuple key is that the tuples can move. Within a processor node it can move between pages, but can also move between processor nodes and even between telecom databases. Therefore it is not possible to use any type of key that is associated with physical storage information. Also the log records needs to contain the tuple key.

At the processor node there must then be a local index that maps the tuple key to a physical key. This index could be dependent on the fragmentation function as in LH*LH [KARL96]. Since all accesses to the tuple must go through this index, it is natural to also place the locks on the tuple in the index.

If the query does not specify the tuple key or the fragmentation attributes, it would be beneficial to have a secondary index. The secondary index then maps attributes that are part of the query to the tuple key and the fragmentation attributes that can be used for direct access of the tuple.

We will have two types of tuple identifiers. One is only unique within a system and is therefore shorter. It is eight bytes. Two bytes are used as the table identifier of the tuple and the other forty-eight bits contain a unique number normally assigned in a sequential manner.

The other tuple identifier is globally unique. It is sixteen bytes long. Of these sixteen bytes, thirty-two bits are used for a table identity. Sixty-four bits contain a sequential number assigned by the system that assigned the object. The other thirty-two bits consist of a country code and a network code. These thirty-two bits are used to deduce who was responsible for creating the object. When indexes are built for those tuple identifiers, it is likely that only the sequential number is needed in the elements of the index.

5.3.2 Schema entities

When defining tables a number of attributes and their types are specified. Stored procedure represent programs that can access any tuple in any table of the database. They are executed in the transaction coordinator by use of a compiled program or by an interpreter. Stored procedures decrease the communication between the application and the database. In the TPC-B benchmark the full transaction can be a stored procedure. Then only one request and one response is needed between the application and the database.

5.3.3 Fragmentation of Tables

To enable a scalable system, the tables should be fragmented. Various types of fragmentation can be used, horisontal fragmentation and vertical fragmentation. Vertical fragmentation is sometimes used in distributed databases that are built out of a number of autonomous databases. In the parallel data server the fragmentation is used to enable parallel execution of database queries on a table. Thereby it is more natural to use horisontal fragmentation. Vertical fragmentation could be used if there are attributes that can become hot-spots. Vertical fragmentation can also be used for security reasons. This can be handled by the application by defining two tables and defining the original table as a view on these two tables. Even more than two tables could be used. Since we use tuple locking, there is more concurrency using this vertical fragmentation.

With horisontal fragmentation there are several ways to derive the fragmentation. As mentioned above two methods are used. The normal case in the parallel data server will be to use the tuple key in the LH^3 structure to derive the fragmentation. In this case the fragmentation is performed automatically by the DBMS as declared by the user.

In some cases the fragmentation is based on fragmentation attributes which is not the tuple key. For example the receiver of emails is the natural fragmentation attribute in an email table. In this way all emails of a particular user can be clustered together and easily found when looking for all of them. More advanced methods to fragment tuples could be used but are not discussed in this thesis.

The fragmentation of a table can either be FULL, meaning that there will be at least one and usually several fragments per node. The fragmentation can also be MEDIUM with a percentage. In this case only a percentage of the nodes are used to store fragments of the table. Both those fragmentation methods will then use LH^3 structure to derive the fragmentation.

If distribution transparency is not desired, the fragmentation can be declared as SINGLE. This means that the table is not fragmented at all. If SINGLE is specified, one could also specify the desired nodes in which the table should be stored. The system will try to use these nodes if possible although other nodes can be used if node failures and overload situations cause things to change.

A final possibility is to specify the number of fragments using the LH^3 structure. In this case, too, it should be possible to specify the desired nodes of the fragments.

5.4 Distributed Indexes

There are two types of distributed indexes needed, primary indexes and secondary indexes. The primary index is used to access the tuples and the secondary indexes translate from a secondary key to a tuple key. Primary indexes are always on the tuple key. In our case we assume that LH^3 is always used. Sometimes only the local part of LH^3 is used if fragmentation is not based on the tuple key.

The normal procedure in databases is that secondary indexes and the tables are placed on the same processor nodes. In a distributed database there are some problems in using this method if the secondary key does not contain the fragmentation attributes. It is then only possible to find the processor node where the index resides using a broadcast to all processor nodes. Therefore secondary indexes are handled in a similar way to how tables are handled, where the fragmentation attributes are always the secondary key attributes.

Secondary indexes are stored as a special type of table with its own fragmentation that is independent of the table. The only exception is secondary indexes on the fragmentation key. In this case the database strives to collocate the fragments of the secondary index and the fragments of the table. This means that when the entry in the secondary index has been found, the tuples are also found in the same node.

5.4.1 Secondary Indexes

Three types of secondary indexes are possible:

- The first is a unique index which maps the attributes of the unique key to the tuple key. It is a one-to-one mapping between the value of the unique key and the value of the tuple key.
- The second type is an index that given the tuple key provides the fragmentation attributes. This is a normal unique secondary index except that it maps from tuple key

to the sought attribute instead of the opposite.

- The third type is a non-unique index where the attributes of the index maps to a set of tuple keys.

A secondary index is basically treated as a normal table, also regarding fragmentation. The index fragments do not contain any stand-by replicas. It is more natural to recreate the index after failures in this case. Even backup replicas of secondary indexes are not always needed. Most indexes are local to a parallel data server, although it should be possible to have them global as well.

In some databases secondary indexes can be used to lock ranges of keys and also to provide scanning of tables with consistency degree 3. In our case to be able to lock to broadly has a negative impact on real-time delays in the database. Also none of the applications has any real need to lock ranges of keys, and therefore locks are only possible to hold on specific key values. It is possible to lock a non-existent tuple, but it is not possible to lock a range of key values. Thereby there must be a specific lock record with the key value of each locked tuple. The locks are gained using the tuple key always and are therefore located in the same processor node as the tuple itself.

There are three index operations that we are interested in, Fetch, Insert and Delete. Fetch Next is also possible, but only Fetch Next of degree 2 is supported so there is no assurance that there is no other value in the range inserted during the scan. Fetch reads the index to find the tuple key. Insert is performed after retrieving a lock on the tuple in the primary replica and so is the delete operation. When insert in the secondary index is performed the secondary index tuple is locked until the transaction is committed and also deleted secondary index tuples are locked until the transaction is committed. Thereby readers will find all possible key values including uncommitted such. When the lock is released the tuple may have been deleted and this must be checked by the read operation.

When a transaction reads a secondary index tuple it does so with a short-term lock. After reading it, the lock is released. This is done to avoid deadlocks if another transaction needs to update the index and holds a lock on the tuple. When the reader uses the tuple key to find a tuple, the tuple may have changed since reading the secondary index and the tuple might no longer fulfil the search condition. Thereby the tuple must be checked after gaining a lock on the tuple, to ensure that the secondary index attributes are still the same. If they are not the same, the transaction releases the lock on the tuple and retries the action on the secondary index or fetches the next.

Updates of secondary indexes are further described in section 6.4.1.

5.4.1.1 Solving Deadlock Problems in Secondary Indexes

There is a problem with deadlocks as mentioned earlier when using secondary indexes. The reason for this is that there are locks on both tuples and on the indexes. However readers of the index start by acquiring a read lock on an entry in the secondary index and then try to acquire a lock on the referred tuple. Those transactions that update the secondary index do so by first acquiring a lock on the tuple and then acquiring write locks on the secondary index. Obviously a deadlock can occur between readers and writers of the secondary index. These deadlocks can also become common and should therefore be avoided rather than discovered by some time-out function.

The problem as mentioned was that the order of locking differs. The secondary index is only a copy of the tuples in the table and it is therefore natural to decide that the owner of the lock on the tuple should have higher priority than the owner of the lock on the secondary index. Therefore no change is necessary in the behaviour of write transactions, the write transactions knows that if a lock is held on the secondary index, it will be released shortly and will not participate in a deadlock with the tuple lock.

The reader can now perform this back-out function in two ways. The first is to lock the data and then try to lock the tuple. If the read lock on the tuple is acquired, then continue as normal, and if a write lock is held by another transaction, then put the lock request on queue and at the same time release the lock on the secondary index. When the lock is released and acquired by the reader it is necessary to check that the tuple still fulfils the search condition; the tuple might have changed due to the actions of the write transaction.

The second possibility is to acquire a short-term lock on the secondary index and after reading the secondary index releasing the lock and trying to acquire a read lock on the tuple. The same argument as in the first method is also valid here; the tuple must be checked that the search condition is still true when the lock on the tuple has been acquired. If the search condition is no longer true, a new search of the secondary index is performed.

The second method is chosen since it releases the lock on the secondary index immediately. Thus it is not necessary to release this lock at commit of the transaction and only tuple locks need to be released as part of the commit.

5.4.2 Related Work

Distributed indexes have been used in many distributed databases (e.g. NonStopSQL and DB2 [Mohan92]). The method of avoiding deadlocks is new to the knowledge of the author of the thesis. The method of accessing tuples only through a primary key is used in NonStopSQL where both tables and indexes can be partitioned.

6 User Transaction Protocols

An important part of the recovery handling is the commit phase protocol. This must be efficient to achieve high performance since it is a part of all write transactions. It must also be designed to avoid blocking in almost all situations (it can be proved that totally non-blocking commit protocols are not even possible to design [Skeen83]). At the same time it must also be designed to handle real-time requirements.

There are two variants of the commit phase protocols that must be handled separately.

- The first variant is when the application transfer the commit decision to the data server. This should be the normal situation since the data server is likely to be more reliable than the application.

- The second variant is when the application coordinates the commit phase. The reason could be that several systems are involved in the transaction. In this case the application sends a prepare message to the data server and later a commit or abort message. Since a positive response to a prepare message is a promise to commit if so decided, the data server must ensure that it can commit even in the presence of node failures.

We will start by describing the commit phase as started by a commit message from the application that transfers the commit decision to the data server. Then later we will show the adaptions of this protocol to handle a prepare message from the application. We will also show how read queries are handled.

The transaction protocol developed in this chapter is one important result presented in this thesis. It is a combination of a normal two-phase commit and a linear commit protocol. It has benefits for performance and delay of transactions. As will be shown it also simplifies the design of recovery protocols where no disk writes are used. In chapter 8 we will also show how it can be very nicely integrated with a second level of replication (network redundancy).

6.1 Normal Commit Phase handling

In Figure 6-1 the normal messages sent in the commit phase in one fragment is shown. This is the basic component of the two-phase commit protocol used in this thesis. The major reason for sending the commit messages in the opposite order is to ensure that the primary data node receives the message last. Thereby the primary data node is prepared to send the log records to the backup system since it knows that all the backup and stand-by nodes have heard the commit decision. The primary data node can also release the locks already in the commit phase for the same reason.

The primary always locks the tuple first and basically the backup nodes would not need to lock any tuples. However, to enable a fast take-over and to enable simple reads of the backup, the backup nodes will also lock the data. The reason for starting the prepare messages in the primary data node is that prepare and update are often put together. The update message will go to the primary data node since this node receives the lock requests first. The complete message can basically go in any order; the reason for sending them in a linear fashion is to decrease the number of messages.

The complete messages are normally piggy-backed on other messages, such as start, prepare and commit. The only extra delay incurred because of this is that backup and stand-by nodes have to wait longer for the lock release message. This means that simple read messages can be delayed somewhat longer in some cases. The simple read transactions often read the latest committed ver-

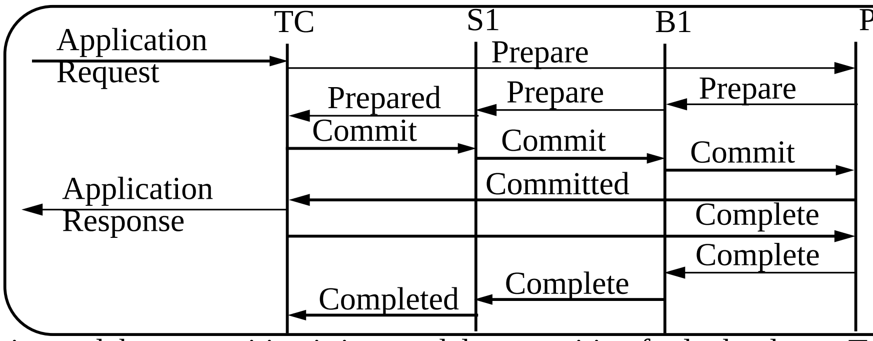


Figure 6-1*Commit Phase*

sion and then no waiting is incurred due to waiting for lock release. To send the complete messages as special messages would incur a substantial performance overhead. To speed up lock release processing for the other simple read transactions is not enough reason to send special complete messages.

If the backup nodes receive a new operation on an object which is locked, committed but not completed, then the backup can safely release the lock of the committed tuple since the primary obviously has released its lock. Thereby no extra waiting due to uncompleted transactions is necessary.

6.1.1 TPC-B transaction

TPC-B is an example of a transaction that involves several tables and thereby several fragments [Gray91]. There are four fragments involved in each TPC-B transaction. These transactions are handled by using the commit phase protocol in parallel in each of the fragments. This means that we have a normal two-phase commit protocol between the fragments and that we have a linear commit optimisation inside a fragment between the replicas as seen in Figure 6-2.

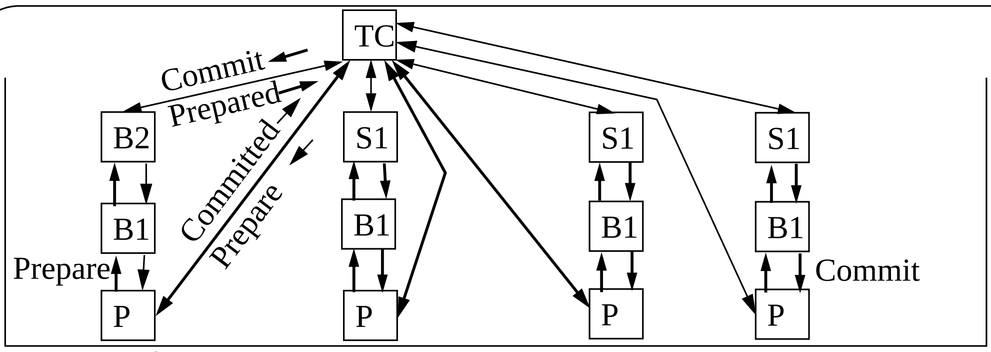


Figure 6-2*Commit Phase Protocol in TPC-B transaction*

The use of linear commit within a fragment substantially decreases the number of messages. The above described mechanism gives 2*f*(c+1) messages in the prepare and commit phase, f is number of fragments and c is number of replicas per fragment. This gives 32 messages for a TPC-B transaction with three replicas per fragment. Using a normal two-phase commit protocol would give 4*f*c messages, 48 messages. Using a linear commit would give 2*f*c messages, 24 messages.

If decreasing the number of messages is the only measure of interest, then obviously a linear commit optimisation is preferable. The linear commit does, however, give a delay of 2*f*c (=24 here) messages sent serially. The normal two-phase commit gives a delay of the maximum of four messages with f*c (=12 here) such executing in parallel. Our variant leads to a delay of the maximum of 2*(c+1) (=8 here) messages with f (=4 here) such executing in parallel. Clearly our variant achieves a balanced effort between short delay and high throughput.

6.1.2 Simple Write Optimisations

In a simple write transaction there are optimisations that can be performed so that the distribution cost of the transaction is diminished. When optimising these simple write transactions a normal linear commit is basically achieved. A simple write transaction refers to transactions that only write one tuple. It could also refer to a number of writes to the same fragment. From the applications we studied it was observed that this is a common transaction type and therefore important to optimise. These optimisations are performed without changing the basic ideas in the commit phase protocol. It is a matter of moving the transaction coordinator role.

As can be seen in Figure 6-1 there are 2*(c+1) messages in the prepare and commit phase. If we enable the transaction coordinator to migrate, it is possible to decrease the number of messages. The first step is that the communication between the last node in the prepare chain and the transaction coordinator can be removed. This means that the message to the last node says “prepare and commit”. Thereby two messages can be removed. The message flow of this is shown in Figure 6-3.

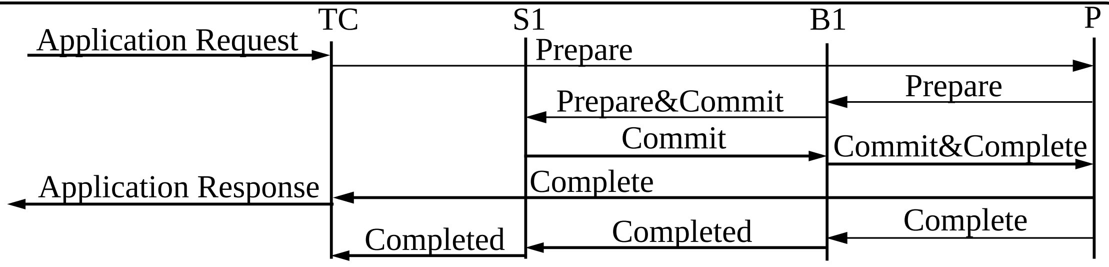


Figure 6-3*Optimised Commit Phase*

This produces 2*c messages in the prepare and commit phase. After this optimisation there are no more optimisations to perform in the two-phase commit protocol. However, locality of transaction coordinator can be used to decrease the number of messages even more. If the transaction coordinator is placed in the same node as the primary replica, then two more messages are removed from the message chain. This leaves 2*(c-1) messages in the prepare and commit phase as seen in Figure 6-4.

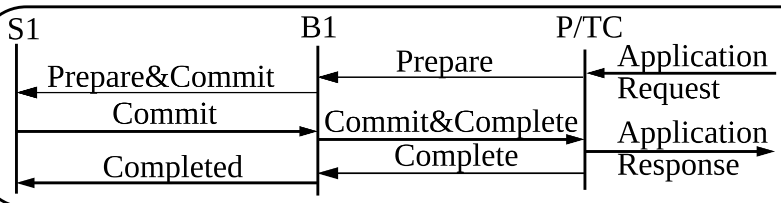


Figure 6-4*Optimised Commit Phase with locality of Primary Replica and Transaction Coordinator*

In our system architecture we have opted for separation of application nodes and database nodes, which is an architecture that relies on an efficient communication mechanism. This means that in all message flows described above there is also one request message from the application and one response message to the application. If this efficient communication mechanism is not available, then the system would most likely need to collocate the application and the database and also it is likely that only two replicas would be used. In this scenario a very optimised commit phase is achieved where two messages are all that is needed in the prepare and commit phase as seen in Figure 6-5.

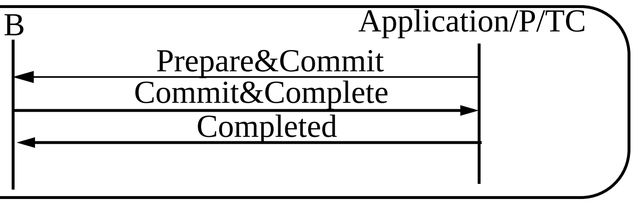


Figure 6-5*Optimised Commit Phase with locality of Primary Replica, Transaction Coordinator and Application*

This optimisation method could also be used with several fragments, but the savings would not be substantial since the savings can only be applied for one fragment. The other fragments have the same cost as before. Also when transactions use several fragments distributed on many processors, there is more information in the transaction coordinator. Thus it is not so easy to move the transaction coordinator. In the simple write scenario each replica contains all information of all writes and thereby the role of the transaction coordinator can easily be taken by each of the replica.

6.2 Handling Node Failures in Commit Phase

The first action of a node failure is to ensure that all nodes know that the node has failed. Then the recovery processing is started so that the desired replication is achieved again as soon as possible. These actions have been discussed previously and are discussed in section 7.1.4 on protocols to copy fragments. The second action is how to handle the active transactions when a node fails. This will be studied in this section.

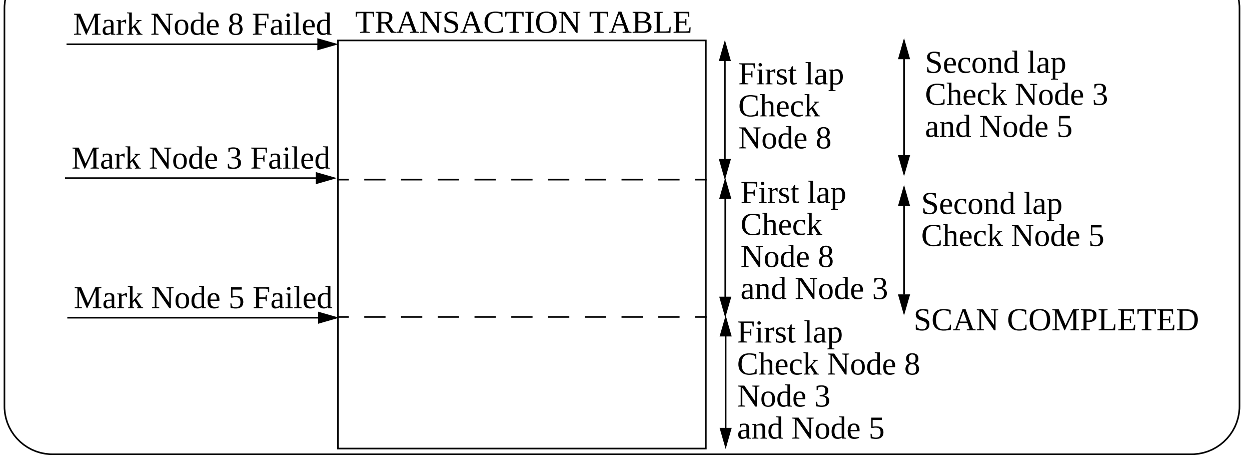
We will study two cases of node failures in transactions. The first is where a participant in the transaction has failed and the second where the transaction coordinator has failed.

6.2.1 Failure of a Data Node

The major problem in this case is that the failure of data nodes breaks the linear sending of the prepare, commit and complete phases. Thereby the message will never come back to the transaction coordinator if no action is taken. It is assumed that the full responsibility of the failure situations is on the transaction coordinator if the data nodes fail. The data nodes do not take any actions on time-out or similarly. This simplifies failure processing since there is only one source of actions on failures.

When a node failure is reported, the transaction coordinator must check all active transactions to ensure that they are not stopped due to node failures. This is performed by scanning the active transactions. If a transaction is waiting for other nodes, these nodes are checked to detect if any failure has occurred with them. If failures have occurred, the action is to restart the phase that the transaction has already started. The messages will still be sent in the same order with the failed node excluded. Thereby there is no risk that messages arrive at the nodes in any other order than the normal linear order. If a transaction receives a message that it has already received, it simply ignores the message and passes it to the next node. To find those transactions that need to restart phases, the transaction table is scanned. However multiple node failures could occur and also new node failures could occur during the scan process. This is handled as shown in Figure 6-6 where node failures are marked as to where the scan process was currently when the node failed.

Figure 6-6*Scan Transaction Table in Transaction Coordinator*



All failures of data nodes are handled in this fashion except failure of the primary data node in the prepare phase. In this case it is already obvious that the transaction will be aborted and this phase is started instead of a new prepare phase.

When the optimisations for simple writes are used, the same algorithm is used by the original transaction controller. This means that a longer chain of messages will occur. Failures should, however, not be so common.

6.2.2 Failure of the Transaction Coordinator

When the transaction coordinator fails, the transaction will be stopped until a new transaction coordinator has taken over this role and finalised the transaction. The new transaction coordinator will be the system coordinator. When a node receives a report of a node failure, it starts two scans. It scans the transaction table as described above. It also starts a scan of all transactions which the node is participant in. During the scan of the transaction participant table, each transaction is checked to see if the transaction coordinator has failed. If the transaction coordinator of this transaction has failed, the node sends a report on the transaction to the new transaction coordinator. This report contains transaction status, fragment identities involved, which role this node had in each of the fragments, and if the transaction is a simple write transaction. The scan process uses the same marker method as described in Figure 7-5. When the scan process is completed, the node sends a scan complete message to the new transaction coordinator. When the new transaction coordinator has received all transaction information and has received scan complete from all nodes alive, then it has all the necessary information to make decisions on the transactions.

The decision will be to abort a transaction if no node reports that it has heard a commit decision. As described below there is one exception to this rule when there is a whole fragment is lost. In this case the transactions are blocked if all nodes have reached the prepared state. If any node has heard a commit decision, it decides to commit the transaction. The abort and commit message and also the complete messages are sent in the usual linear fashion within a fragment. If the new transaction coordinator also fails before it has finished its work, another coordinator is automatically chosen and this new coordinator performs exactly the same actions. In this way this method continues until all nodes have failed or until the system finds itself in an inconsistent state and decides to perform a shutdown of the system.

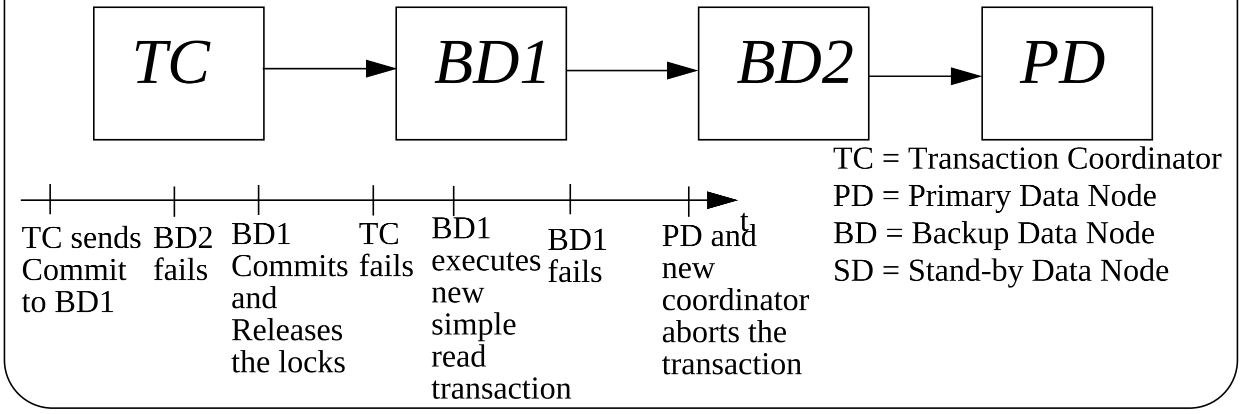
Using optimisations of simple writes, a failure of the original transaction coordinator does also use this functionality.

6.2.3 Special Error Situations

The actions taken by a processor node on behalf of a transaction can survive a node crash only if the information is written to disk. Therefore if all nodes fail that contain copies of a fragment, without writing committed information to disk, the fragment is inconsistent and the system must be shut down and restarted from a global checkpoint by using local checkpoint information, local logs, and fragment logs. The two-phase commit protocol, however, forces the log to disk when a processor node fails and some time after this failure. Also when there is only one replica of a fragment, the log must be forced to disk before committing the transaction. Thereby there could be situations when all nodes that contain copies of a fragment fail and the system is still operational.

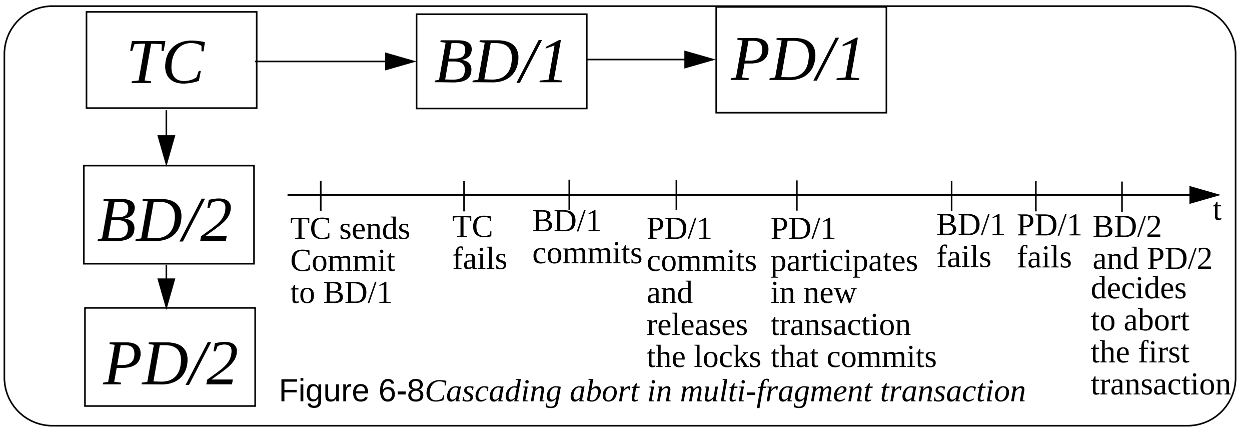
One approach in the situation where all participants are prepared is to simply abort the transaction and leave it at that. This is called presumed abort [Mohan83]. There are problems with this approach in our database if locks on backup replicas are released in the commit phase. Let us assume that a backup node has heard the commit decision, releases the locks and proceeds with new transactions. The backup node can process simple read transactions. Now assume we get the situation in Figure 6-7, where we can obviously get an inconsistency due to releasing the locks on the backup data nodes too early. By ensuring that the primary node releases locks before the backup nodes and also ensuring that all nodes with replicas of the fragment have got the commit message before the locks are released, the problems with inconsistencies due to this problem are avoided.

Figure 6-7*Aborted results displayed causing inconsistencies*


TC
BD1
BD2
PD
TC = Transaction Coordinator
PD = Primary Data Node
BD = Backup Data Node
SD = Stand-by Data Node
t
TC sends Commit to BD1
BD2 fails
BD1 Commits and Releases the locks
TC fails
BD1 executes new simple read transaction
BD1 fails
PD and new coordinator aborts the transaction

The solution is that the commit message does not mean that the backup and stand-by nodes release the locks. They release the locks when the complete message arrives instead or when the primary node requests a new transaction on locked data. If this happens, the primary node must have released its lock and thereby the locks can also be released in the backup and stand-by nodes.

There is still, however, one more problem with the method of aborting if all alive nodes are prepared and none aborted or committed. This problem arises when all replicas of a fragment crashes after releasing locks and the transaction is still aborted. In addition the system did not crash. An example of events that lead to such a situation is shown in Figure 6-8. The problem is that all replicas of a fragment are lost where the transaction was committed and even logged on disk. Then a new transaction coordinator decides to abort the transaction. Thereby we conclude that if a whole fragment is lost, the transaction should not be allowed to abort in this unsure situation. In this particular situation the transaction must be blocked.



TC
BD/1
PD/1
BD/2
PD/2
t
TC sends Commit to BD/1
TC fails
BD/1 commits
PD/1 commits and releases the locks
PD/1 participates in new transaction that commits
BD/1 fails
PD/1 fails
BD/2 and PD/2 decides to abort the first transaction

Figure 6-8 *Cascading abort in multi-fragment transaction*

There is still a problem with this scenario. How does the new transaction coordinator know that there is a whole fragment missing from the transaction? To be able to get this information the new transaction coordinator must know about all involved fragments, and this would cause an increase in the message sizes sent during the transaction. This increase is not really justified for such a small benefit. Therefore it is better to block all transactions that are unsure if there is a fragment that is totally lost after a node crash. One special transaction, the simple write transactions, can however be unblocked with a small effort. In these transactions the fragment id is sent anyway as part of the messages and there is only one bit needed that specifies that this is a simple write transaction. Thus those transactions are never blocked. If the whole fragment is lost, then obviously no node will have heard about the transaction since all nodes are lost.

6.3 Handling Prepare-to-Commit Requests from Application

When the application sends a prepare request to the data server it creates a number of problems. The first problem is that the failure of a transaction coordinator is not a reason to restart a system. This would be necessary if the algorithm described above is used. The problem is that when all participants have prepared, the transaction must not be aborted or committed until the application sends its decision to the data server. Thus the above algorithm must be changed so that when all fragments involved in the transaction are prepared and none committed, then the transaction must wait until the application has requested abort or commit. This algorithm is used by the new transaction coordinator after failure of the transaction coordinator.

If all fragments are available after the failure of the transaction coordinator, the new transaction coordinator can check that all involved fragments and their replicas have reported on transaction state. If any fragment is unavailable, this fragment has saved a transaction state on disk before failing since the system is still operational. If any fragment replica has received commit, then the new transaction coordinator will commit the transaction. If any fragment has not yet been prepared, the transaction can safely be aborted and this can be reported to the application. If all fragments are prepared, the new transaction coordinator waits for the decision of the application. When the abort or commit arrives, it is executed on all fragments. This is performed even in the situation when a fragment is not available. It is assumed that if the application decides to commit, then it must have heard prepare from the old transaction coordinator before its failure. Reporting of prepared-to-commit to an application can only be performed if the new transaction coordinator knows that all involved fragments have reported the status of the transaction.

Another problem is that failures of the primary replica in this state are not allowed to crash the system. This means that all nodes must have complete information on which tuples that are locked by this transaction, even the stand-by replica. This means that all read requests in these transactions must be performed at all nodes (actually only the read lock needs to be set at backup and stand-by replicas). Otherwise the read locks would be lost when the primary replica fails and other transactions could cause serialisability errors. Also the stand-by node must set locks on all update operations performed.

There is yet another problem, this relates to the schema information. The locks on schema information are only set in the transaction coordinator. In the previous algorithm these locks were always released before the commit message was sent to the other nodes. Thereby failure of the transaction coordinator did not cause any problems with concurrency on schema information. Actually the problem only relates to hard schema changes since these need not be serialised with user transactions. A soft schema change allows both versions to execute concurrently and presents therefore no problem in this case.

A failure of the transaction coordinator causes all locks on the schema information to be lost for this transaction. Now in this situation no new locks will be requested of the transaction. Also schema information is not updated and therefore an abort does not necessitate a rollback of schema information. Thereby the two-phase locking rule is actually adhered to even in the failure case.

What could remain a problem is if anyone is removing tables, indexes or attributes involved in the query before the commit has been performed on this transaction. Removing views, constraints, referential constraints, triggers, stored procedures and so forth should not cause any problems since they have been used already by the transaction when it has reached the prepare phase. Thereby removal or additions of such schema information can be done and it is serialised with this transaction. Also adding of new tables, indexes and attributes causes no problem since they could not have been involved in the transaction.

The solution to this problem is that hard schema changes that need to remove tables and attributes must wait until all prepared transactions in the failed transaction coordinator have committed or aborted.

6.4 Referential Integrity and Secondary Indexes

There are various constraints that can be used. One type of constraint is that attribute values must be within a specified domain. This constraint can be checked by the transaction coordinator at reception of an update or insert message. Another constraint is the unique constraint, which can be ensured by having a unique index on the attributes that should be unique; if the values of these attributes are already in the index, the transaction must be aborted.

Foreign keys are used to implement referential constraints. If a tuple is inserted that contains a foreign key, it must be ensured that the referred tuple exists. This is accomplished by reading the referred tuple and gaining a read lock on the tuple to ensure that it does not disappear while executing the transaction. If a tuple is deleted and there are references to the tuple, there might be some action to be performed on the referencing tuples. The action is specified in the declaration of the table. Some possible declarations are Cascade on Delete, Set NULL on Delete, Set Default value on Delete and No Action.

The deleting transaction should then gain exclusive locks on all tuples referring to the deleted tuple. To do this there must be a secondary index on each foreign key; these indexes can then be used to find all tuples that refer to the tuple to be deleted.

We use a strategy where the inserting transaction will try to acquire a lock on the referred tuple before acquiring locks on the index of the foreign key. This means that a deleting transaction can safely get references to all tuples that refer to the tuple. A deadlock could occur if a tuple updates the foreign key to refer to a tuple that is deleted. This situation can only be solved by normal deadlock handling where one of the transactions is aborted. The delete transaction should first acquire a write lock on the tuple to be deleted before searching the secondary indexes on the foreign keys.

Referential constraints can cause situations where a foreign key refers to more than one table. These foreign keys normally use a tuple identifier as foreign key. To find this tuple there must be a table identifier in the foreign key. Therefore tuple identifiers contains a table identifier.

There are a number of possible ways to handle indexes, constraints, and referential constraints together with the replication in our data server. One possibility would be to let the primary replica perform all checks on attribute constraints, referential constraints, updates of indexes and so forth. This would mean that the primary replica would become sub-coordinator of a set of operations within the transaction. This is the solution that produces the minimum number of messages. It does, however, increase the size of the messages since it is necessary to transfer the schema information. Of course schema information is also available in the node of the primary replica. To access schema information in this node causes problems with concurrency control of schema information. Also it would be necessary to handle failures of the primary replica with special protocols. This would complicate the recovery protocols.

Another option would be to send the operations to the primary replica. When the primary replica discovers a need for updating an index, performing constraint checks and so forth, then it sends this information to the transaction coordinator. This increases the number of messages. It does, however, not require special handling of failures of the primary replica. It does not solve the problems of concurrency control of schema information.

A third solution is to let the transaction coordinator control all activities of advanced database facilities. It does so by sending requests to the primary replica to get information on whether index updates are needed, whether referential constraint checks are needed and so forth. In this manner the transaction coordinator is in full control of all activities in the transaction and can start the activities in parallel. Since only the transaction coordinator needs to check the schema information, we do not have any difficulties with concurrency in this solution. The only drawback in this solution is that more messages might be sent than in the other solutions. Since our data server uses an efficient communication scheme, we decide that this is not a serious problem. The centralised design of the advanced database facilities simplifies the design and we can use rather simple algorithms to perform complex tasks.

We will show the chosen method with some examples.

6.4.1 Update of Secondary Index

When executing an update, insert or delete operation the schema information is checked to see if the operation influences any secondary indexes. To be able to perform the index updates, the attributes of the secondary index must be read. It is also necessary to set a write lock on the tuple at the same time. This is necessary to avoid deadlock situations as described in a previous section. The update of the tuple and the reading of the tuple is requested in the same message. After reading the index attributes, the index updates (could be both insert and delete) can be performed in parallel. In Figure 6-9 the flow of messages is shown until the first update messages. The messages following are not shown, they are similar to the TPC-B transaction shown above.

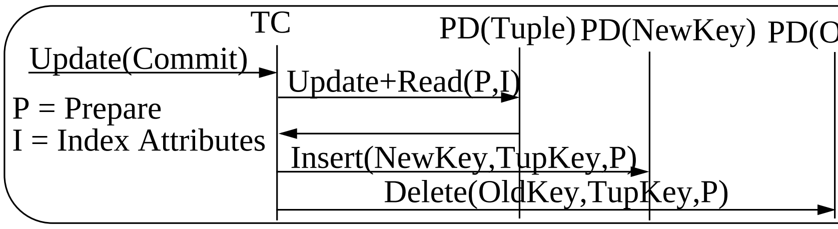


Figure 6-9*Example of an Index Update*

6.4.2 Referential Constraint

The example we show here is how to delete a tuple with a number of references to it. The attributes that refer to the tuple being deleted should in this example be set to NULL.

The first step is to delete the tuple. This ensures that the tuple is locked and that no transaction will be able to create new references to the tuple. The tuple is locked with a delete lock to indicate that the operation will try to delete the tuple. A dummy read is used in the same message to create a quick acknowledge of the lock operation. Then the indexes of the foreign keys are scanned to find all tuples that referred to this tuple.

To ensure that no deadlocks are introduced, a transaction that is inserting a tuple that contains a foreign key will start by trying to get a lock on the referred tuple before inserting anything into the index of the foreign key. If a transaction updates the foreign key attribute of an existing tuple, the transaction will also first try to lock the referred tuple. If it finds the tuple locked for deletion it knows that a deadlock will occur by using schema information. Thereby it will abort its transaction to ensure that no deadlock occurs.

This means that when a lock on the tuple has been acquired, there is no risk of running into deadlocks when requesting all entries in the index of the foreign key. When the referring tuple keys have been returned, the SET NULL operations and the deletion of all entries in the index of the foreign key can be performed in parallel.

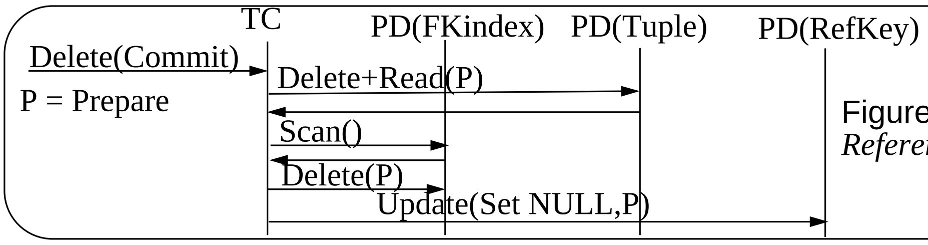


Figure 6-10*Example of a Referential Constraint*

6.4.3 Related Work

Handling of referential integrity and secondary indexes is standard information found in most text-books on databases. The new information in this thesis is how to integrate constraint handling in a parallel database. This is often referred to as a complex problem. The solution in this thesis relies on the fact that communication is cheap. This avoids the need of complex solutions to avoid communication. [ÖZSU91] treats both centralised and distributed handling of constraints.

6.5 Read Queries

Before describing the recovery protocols we will describe the protocols to handle read queries. Most read queries are simple read queries in our benchmarks with only one tuple read in the transaction. These transactions are handled with some extra optimisations. Also read-only transactions can be somewhat simplified. Also shown is the use of secondary indexes in read queries. Read transactions that are part of updating transactions read the primary replica. In some situations these read queries also need to set locks in all other replicas.

6.5.1 Simple Read Transactions

Transactions that only involve a simple read operation using the tuple key are very simple. If the operations only involve 2-safe data, then either the Primary System or any of the Backup Systems can be accessed, otherwise the Primary System is accessed. In the chosen system either the primary copy or the backup copy is accessed, otherwise the primary copy is accessed. Using 1-safe data in the Backup System could only be done at the risk of reading inconsistent data since there are no guarantees on the consistency of 1-safe data until a recovery procedure has propagated all updates of transactions to the Backup System. Read queries are not logged but they use locking and therefore the only part of the two-phase commit protocol needed is the commit phase in which the locks are released. Simple read queries performs this immediately after reading the tuple. This means that the coordinator can transfer the commit responsibility to the node where the data is read. Thereby this node can send the data directly to the requesting application as seen in Figure 6-11. The worst thing that can happen is that the message is lost due to some failure. This would be discovered by the application from a time-out. At time-out the message can then be resent.

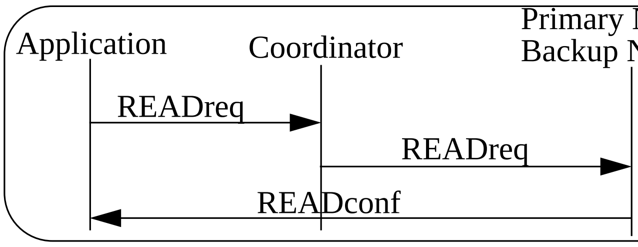


Figure 6-11*Messages in Simple Read Query*

If the simple read query goes through a distributed index, the query is a little bit more complex. In this case the index must be a unique index, otherwise it would be a scan transaction.

In Figure 8-12 the normal case of reading a tuple through a secondary index is shown. In this case the tuple is found at the node where it was supposed to be and the unique attributes of the tuple were still correct when reading the tuple.

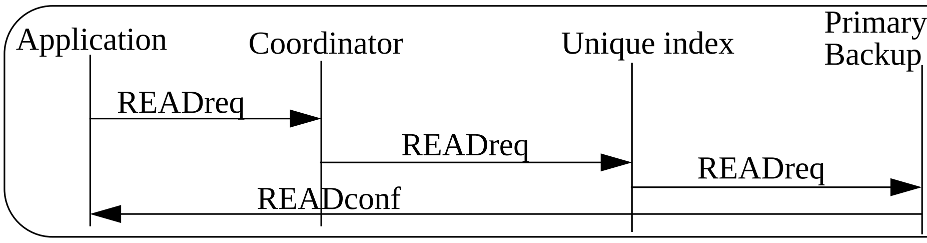


Figure 6-12*Messages in Simple Read Query with unique index, normal case*

The possibility exists, however, that the tuple is deleted when coming to the primary node or the unique attribute has changed. In this case the READreq is simply sent once more to the unique index and retried. If the unique index says not found, then a not found message is sent to the application. All these messages transfer the control of the message to the receiver so that error cases have to be handled by a time-out in the application node, followed by a resend.

6.5.2 Read-Only Transactions

This covers transactions that consist of several read requests either through a secondary index or through the tuple key and also covers scan queries. The major difference between a set of read queries compared to the simple read query is that the coordinator does not transfer the control. Hence the coordinator is contacted from the Data Node to ensure that the coordinator can release the locks at commit time. At commit time the locks are all released by using a linear commit optimisation and without a prepare phase since no updates are involved. If the coordinator fails, the system coordinator will take over as coordinator of this transaction as described above.

6.6 Related Work

Many reports have been written on various commit phase protocols. The most important protocol is the two-phase commit protocol, variants of this protocol include linear commit protocol [Gray79], non-blocking two-phase commit protocol [Gray93], presumed abort [Mohan83], presumed commit [Mohan83]. A number of these variants were described in section 4.10.

6.7 Conclusion

In this chapter a new version of the two-phase commit protocol has been developed. It uses a linear commit protocol between the replicas of a fragment, and it uses a normal distributed two-phase commit protocols between different fragments involved in a transaction. It also uses the presumed abort protocol with certain restrictions on total failures of fragments and on releasing of locks in backup and stand-by nodes. Finally a new transaction coordinator is automatically assigned when the transaction coordinator fails as in [CHAMB92]. The change of direction used in a linear commit protocol is used in our case to ensure that the primary replica receives the commit decision last. Then the primary replica knows that the transaction is safely committed and can release the locks. Later we will also show how this can be used to provide network redundancy as well.

The use of a linear commit protocol within a fragment not only provides better performance, but it also simplifies many of the recovery protocols since the number of possible states to handle is reduced.

This protocol tries to achieve a high reliability of the two-phase commit protocol, soft real-time and high performance. All of these are conflicting requirements but the mixture of variants of the two-phase commit protocols achieves a balanced solution to all these requirements in a system where communication is cheap and where messages are executed by the run-time system with soft real-time.

7 On-line Recovery and Reorganisation Protocols

In this section we will describe the protocols that handle recovery after node failures where there still remains operational primary or backup replicas. Another important item is how to reorganise the fragments when the database grows. To support these protocols a number of basic protocols are needed.

7.1 Basic Protocols

The basic protocols need to find the failed nodes and to handle the transaction that changes the distribution information. All on-line recovery protocols use a copy fragment protocol to send the fragments to the new replicas.

7.1.1 I-am-alive Protocol

The reason for having an I-am-alive protocol is to ascertain that nodes with errors are quickly found and that the rest of the nodes reach a consensus on which nodes are up and running. The idea is that no node can enter this trusted set of nodes without confirming to two other nodes in this trusted set that it is up and running. To be kept in the trusted set, each node must within a certain time interval report to its neighbours that it is alive.

If these two neighbour nodes agree that the node is not up, they can start a Node Failure transaction. If two nodes agree that a new node is ready to enter the trusted set, they can start an Add Node transaction. An Add Node and Drop Node transaction can also be started by the operator.

The reason for informing two nodes is to raise the level of security in making the right decision. Erroneous decisions could otherwise be based on temporary faults on nodes, links or even software faults in the error detection. There is of course still some possibility of error but it should be smaller.

The idea of this protocol is to arrange the nodes in a doubly linked list. The idea is that each nodes send I-am-alive to both its neighbours in the linked list. If an error occurs, this is then discovered by the neighbours. This is shown in Figure 7-1.

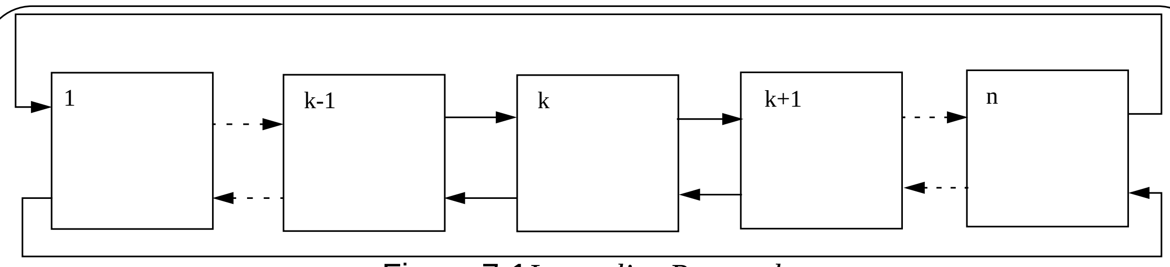


Figure 7-1*I-am-alive Protocol*

When a node does not receive any I-am-alive message within a predefined number of time intervals, the node sends a NodeFailReport to the coordinator node (the first node in the linked list). If the failed node is the coordinator, it chooses the next node in the linked list as the new coordinator and sends the NodeFailReport to this node instead. This message also acts as an indication to the receiver that it has been appointed as the new coordinator. When the coordinator receives the NodeFailReport from both neighbours of a failed node, it initiates the Node Fail Protocol.

The node that discovers the failed node keeps this information until a new active decision has been activated where the failed node has been dropped. In this way subsequent failures of coordinators and so forth will not lead to incorrect active sets being activated. If a coordinator proposes a new set of active nodes where the failed node is still active, the node will report this to the coordinator. The coordinator must then either remove the failed node or the node reporting the failed node. If a node reports failures that other nodes do not discover, then the node reporting the failure is dropped from the set of active nodes by the coordinator.

The I-am-alive protocol should be executed at a higher priority level than user messages to ensure a bounded delay time of discovering failed nodes.

7.1.2 State Transition Protocol

The state transition protocol is used to change the state of the set of active nodes, to change the set of copies of a fragment. They all use the same type of protocol as shown in Figure 7-2.

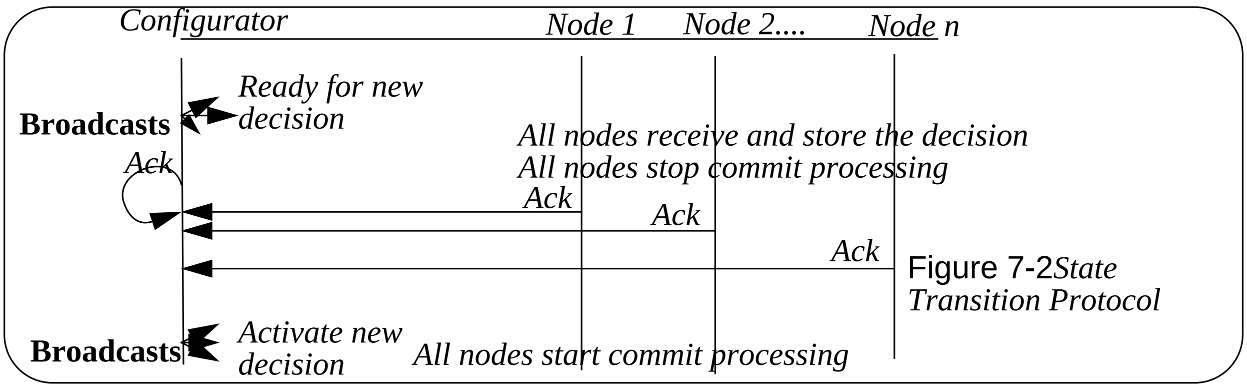


Figure 7-2*State Transition Protocol*

The idea of this protocol is that when the Ready for new decision message is received at the nodes in the system, then commit processing is not processed until the Activate new decision message is received. This means that the state change is inserted at a serialisable time since all commits before the Ready for new decision message have occurred before the state change and all commits after this message have occurred after the state change. This means that all transactions should have executed as if the state at commit time was the state all the time. If nodes fail during the transaction processing, then if none was a primary replica, the transaction can be committed without the failed node and if new nodes are added to a fragment during transaction processing, these nodes must be inserted into the transaction before the transaction can be committed.

The reason that failures of backup nodes do not cause any need for aborting the transaction is the transaction protocol. Since the primary is always accessed first, the method uses primary copy locking. Thereby no information is lost when the backup fails. Only simple read transactions could be lost at the backup node. These simple read transactions are restarted at time-out if no reply has arrived.

All state transition protocols should execute on a higher priority than user messages to ensure a bounded delay of the state transition protocols.

A number of state transitions do not need to stop commit processing after the prepare message. An example is when a fragment has been copied to its new replica. It is not necessary to prepare this since the replica is already involved in all transactions. It is only necessary to inform all nodes so that they agree on that this node is now up-to-date and can be used as primary in failure situations.

7.1.3 Node Fail Protocol

The Node Fail Protocol executes as a state transition protocol, where the state it updates is the set of nodes currently active. The protocol is initiated by the system coordinator after receiving information from two nodes that a node has failed according to the I-am-alive protocol. The system coordinator also takes over the role of transaction coordinator for those transactions where the failed node was transaction coordinator. Each node that receives the fail message then replies with the state of each transaction it was involved in where the failed node was transaction coordinator.

The node fail protocol should execute on a higher priority than user messages to ensure that the node fail protocol has a bounded delay time. This means that some user messages that is sent by the failed node may be in the message buffers when the node fail protocol is executed. The easiest solution to this problem is that the run-time system removes all those messages when receiving Ready for new decision and then ensuring that messages arriving from the failed node are not accepted until the node is restarted again. I-am-alive messages from the failed node should, however, be accepted at all times to ensure that it can be decided when the failed node has recovered.

Since it is possible with network partitions we ensure that this can never happen by keeping within certain rules of when a node fail protocol can be executed. If a majority of the nodes in the previous active set of active nodes remains in the set of active nodes in the new decision, then the new decision can be activated by the state transition protocol.

If half of the nodes or a majority of the nodes have failed according to the system coordinator, then the node fail protocol cannot be executed. If this happens, all nodes in the set of nodes still active are told to stop commit processing. The decision on what to do with the set of nodes must then be taken by an operator. Until the operator inserts the proper command, the commit processing is suspended. Another solution could be to shut down the system and then restart it again.

7.1.4 Copy Fragment Protocol

A new backup replica is created by the primary node after executing a state transition protocol where the system is informed of the new copy. The basic idea of the copy fragment protocol is rather simple. The sending node sends the fragment, tuple by tuple until the last is sent. When the last tuple is sent, a new state transition protocol is executed to inform the system of that the new copy now is up-to-date. The receiving node receives all write transactions. If the affected tuple has arrived already, it applies the write transaction to the tuple. If the tuple has not arrived it ignores the write but participates in all phases of the two-phase commit protocol. This is shown in Figure 7-3. Insert operations are always processed in the new copy.

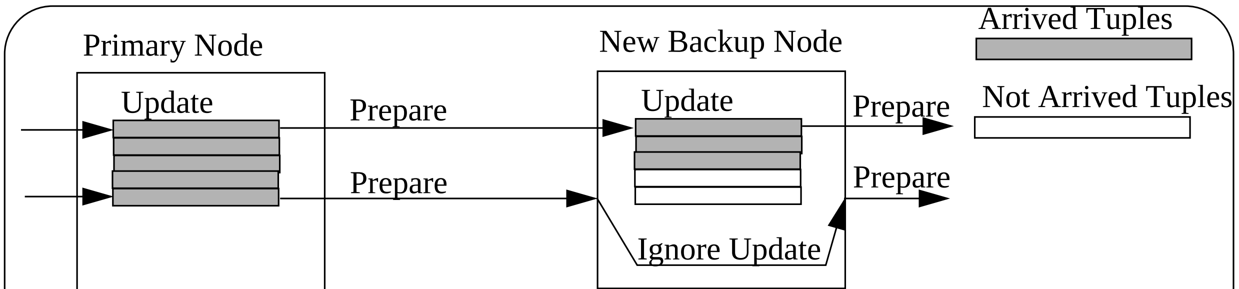


Figure 7-3*Handling of Write Transactions in New Backup Node*

The complexity of the protocol is shown in Figure 7-4. The basic problem is that transactions proceed to execute in parallel with the copy fragment protocol, thereby four algorithms are needed to accomplish the protocol: one algorithm that handles the copy process in the primary node, one algorithm that handles the transactions in the primary node, one algorithm that handles the receive process, and finally one algorithm that handles the transactions in the new backup node. Since the copy fragment protocol is highly integrated with the distributed data structures, its description will be given in Chapters 11 and 12 where these distributed data structures are presented.

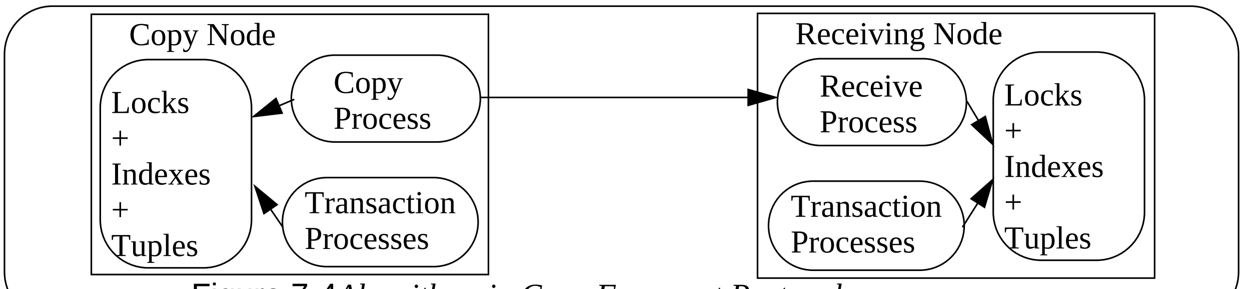


Figure 7-4*Algorithms in Copy Fragment Protocol*

There are two steps in declaring the new replica up-to-date. The first step is reported when the copying process is finished and the new replica contains all tuples. Before the new replica can be declared as recoverable, it must also perform a local checkpoint. This means that if the node fails after declaring itself up-to-date but not yet recoverable, it is not useful in a total failure situation. The reason is that it is only after the first local checkpoint that the replica is stably stored on disk.

7.1.5 Create Global Checkpoint Protocol

The idea of a ***global checkpoint*** is to create a checkpoint that is transaction consistent. All transactions and the log records belonging to those transactions are marked with an identity of the global checkpoint they belong to. This can be used in restart situations to recover from total failures. It is also useful in starting stand-by nodes. A stand-by node is started when it has received all log records belonging to a global checkpoint and all thereafter and is also participating in current transactions. The global checkpoints are also used in restarting fragments that have failed. Global checkpoints are not necessary in the copy fragment protocol since this protocol is integrated with normal transaction processing.

A global checkpoint is generated by simply executing a state transition protocol where the coordinator assigns the new global checkpoint identity. More complex protocols could be developed but this simple protocol is seen as enough in this context. The bounded delay time achieved by executing at a higher priority should keep the delay time down to 0.5-1.0 ms which should be acceptable for most applications. This short delay is dependent on using a database system executing on a real-time run-time system.

When the prepare message is received, all commit processing in the transaction coordinator is stopped and a global checkpoint record is inserted in all fragment logs. Also the global checkpoint ID is updated with the new value. When the activate message arrives, commit processing can start again and all subsequent commits use the new global checkpoint ID. All log records belonging to this global checkpoint ID will be after the log record describing the global checkpoint.

Only commit processing in the transaction coordinator is stalled. Operations executed before reaching the commit point in the transaction coordinator can continue and also operations that have passed the commit point in the transaction coordinator.

Also schema information needs to be a part of this transaction consistent checkpoint. It will be shown in Chapter 9 that it is needed to read two version numbers, one that describes the current schema version and one that describes the last completed schema version.

7.2 Replica Handling Protocols

The replica handling protocols are needed to make new copies when a copy has failed. They are also needed to change the state of replicas. Backup and primary can switch roles and backup replicas can be promoted to primary replicas. After failure of all primary and backup replicas a stand-by replica can be recovered and promoted to primary replica. The following protocols are performed by the primary node and the nodes that will be promoted to primary node of the fragment. The idea of these protocols is to ensure that fragments are again replicated as desired after a node failure.

For each fragment of which the processor node is primary node, the following action is taken:
1) Decide which nodes should receive a backup replica
2) Execute a state transition protocol with the new decision
3) Execute the copy fragment protocol
4) Mark the new decisions as active (drop old decisions)

In parallel with this a check of stand-by replicas is performed by the same node:

1) Decide which nodes should receive a stand-by replica
2) Execute a create stand-by replica protocol

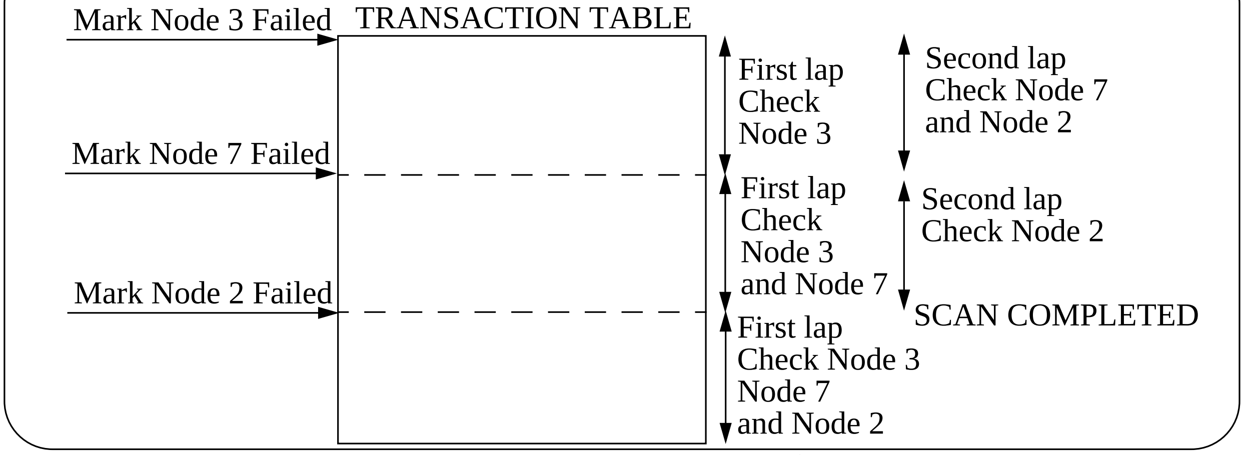
When the primary node has failed, the following actions are taken by the node to become primary node:
1) Execute a state transition protocol that promotes a backup node to primary node and marks the new decision as active (drop old decisions, even if other backup copies were starting, since the primary node failed that kept the information of the copy fragment protocol), the actions in this state transition protocol are further described below.
2) Execute the actions described above as primary node.

To find what actions a specific node should perform after a node failure, the nodes must scan their fragment distribution table. Since several failures can occur during this scan process, it is necessary to put a marker when starting the scan process after a failure, so that the end of the scan process can be found. When a failure occurs during a scan process, the scan continues from where it was and a marker is put at this spot in the fragment distribution table.

7.2.1 Promote Backup Replica to Primary Replica

This transaction is executed when the primary replica fails. All transactions that have not been prepared yet must be aborted since the primary replica failed. For those transactions where the commit has been transferred to the data server, prepared but not committed transactions can also be aborted. This is performed in the normal commit phase of the transaction when it will check whether

Figure 7-5*Scan Fragment Distribution Table*



the primary node is still alive. Thereby the request to promote from backup replica to primary replica can be immediately acknowledged and the promotion to primary replica is completed as soon as the Activate new decision message is received.

This simple logic is possible since the only possible difference between the states of the primary and the backup replica is the state of active transactions and also the set of locks that are acquired in the primary replica and the backup replica. If all active transactions are aborted, then all these transient states are set to aborted and all locks are released and the primary and backup replica do not differ at all.

If the global replication protocol is used, then all reads (excepts simple read transactions) set locks on all nodes. No important information is lost when the primary replica fails. Thus it is not necessary to abort transactions that have not yet committed.

7.2.2 Create Backup replica

There are no specific actions to take when creating a backup replica; it is sufficient to execute the copy fragment protocol.

7.2.3 Create a Stand-by Replica

When nodes fail, the system starts to force the log to disk at commit. Also in systems with only one replica of a fragment the log records are forced at commit in this fragment. To avoid having to force the log to disk at commit, the system can create a new stand-by replica to raise the reliability level again so that the log need not be forced to disk any more.

A stand-by replica is valid from a certain global checkpoint. To ensure that it is clearly defined which log records are produced before the creation of the new stand-by and which were created after, a new global checkpoint is started simultaneously with the create stand-by protocol. The new stand-by replica is then immediately valid from this global checkpoint. The primary can then send older log records so that the new stand-by replica can be used in restart situations. To do this it must have all log records from the global system checkpoint.

If the new global checkpoint number is *n+1*, then all log records belonging to global checkpoint *n* and earlier must be shipped before the new stand-by replica is up-to-date for use at total recovery. When the primary replica discovers that all these log records have been sent, it informs all the nodes that the new stand-by replica is now useful from the global system checkpoint.

7.2.4 Switch Primary Replica and Backup Replica

In this case both the primary node and the backup nodes are active. After receiving this decision, no new operations to the fragment are started and no new phases of the two-phase commit protocol are initiated. All active operations must execute until they are finished and likewise must the transaction phases do. When all operations on the fragment have halted, the primary replica and the backup replica must be set in the same lock state. Lock information must be transferred so that the new primary node contain the same locks as the old primary node. When this is finished, the new state can be installed and the Ready for new decision can be acknowledged and as soon as the Activate new decision is received the switch has been performed and all operations can continue. Simple read operations can still proceed during the switch process. These have the same behaviour independently of whether they are read at the primary or at a backup node.

7.3 Start Node Protocol

When a node is started or restarted after a failure, some dictionary and distribution information is likely to be invalid. Before the node can participate in user transactions, it must therefore create dictionary and distribution information which is valid. There are two cases when starting a node. Either it is the first node started after failure of all nodes or other nodes have started already.

If several nodes start after a complete failure then one should be selected to start first. Then this node is used to start the other nodes in parallel. This is slightly slower but simplifies the start node protocol and such a recovery after complete failure should be very uncommon.

7.3.1 Create Dictionary Information

The dictionary contains a set of system-defined tables, these tables are seldom updated. The simplest path to create the information is then to assign another node to copy its data to after locking all dictionary information. When starting the first node the dictionary information is recovered during the recovery process from disk files.

More complex procedures could be designed to allow concurrency of dictionary updates during the start of a new node. These are not so difficult to design but seem an unnecessary complexity burden. A simple idea to use would then be to the keep a log of all updates during the copy process. After completion of copy process the log is also executed and then a lock on dictionary information is set and any remaining log records are executed on the new node and after this the locks on dictionary information are released.

If the node assigned to copy the information fails during the copy process, then a new node is assigned to copy the information from scratch. Also here it is possible to do better if a more complex start node protocol is used.

7.3.2 Create Distribution Information

When starting the first node the distribution information is very simple, as there is one alive node and no fragment is assigned to this node.

When new nodes are added to an already existing system the only change of distribution information is that there is a new node added. In this case it is not possible to lock all distribution information during the copy process. Doing so would mean that node failures were not allowed and this cannot be certain. Also repairing after node failures is more important than starting new nodes using a simple protocol.

What we do is then a variant of the copy fragment protocol that was shown in section 7.1.4. We start by performing a state transition protocol to inform all other nodes of a new node that is to be started. After this transaction the new node will be involved in the I-am-alive protocol and will be a part of all updates to the distribution information.

The next step is to start copying the information. First node information, then table information and finally fragment information. To avoid complex problems this protocol is executed at the same time as the create dictionary information. This means that no dictionary updates are possible and thereby no tables are created, changed or dropped during the copy process. Thus we do not need to consider updates of table information.

Updates of node information could of course occur at any time, but the size of node information is such that we assume this copy as an atomic event. The node information is copied in one chunk. If updates to this information occur before the chunk, the update is acknowledged and ignored at the new node. If it occurs after copying the chunk the update is performed as usual. Fragment information is shipped one fragment at a time. Split Fragment and Join Fragment transactions are not allowed during the copy process. Thereby the fragments remain the same during the copy process and only updates to fragment information is necessary to consider.

When updates to fragments are performed, the new node performs the update if the fragment has arrived; otherwise it ignores the update and only acknowledges the update. It will receive the update when the information about the fragment is shipped. When a prepare transition message arrives the fragment record is locked so that it cannot be shipped. The lock is released when the activate transition message arrives.

7.4 Join/Split Fragment

Join and split fragment are actions that can be handled by the copy fragment protocol in conjunction with algorithms local to a node. The split/join algorithms are dependent on the use of distributed linear hashing and a distributed B+tree. The splitting/joining of a fragment can be a process that consumes much resources. If a fragment splits, the index must be split, the fragment file must be split and the tuples must be shipped to another processor node. Disk fragments can contain many GBytes of data and should therefore not be moved very often. Main memory fragments contain up to a number of hundred MBytes of data and can be moved more often, but even they should be moved with caution.

One reason to split a fragment is to divide the fragment into more processor nodes. Now a telecom database only adds nodes as a management activity, and thereby splits occur as a result of adding more nodes to the system. If nodes are removed from the system, joins could occur to decrease the number of fragments in the system.

Another reason to split and join fragments is to perform load balancing. Most of the time load balancing can be handled by moving fragment replicas and by switching roles between primary and backup replicas. Also the transaction coordinator role can be removed from loaded nodes. These are however mostly short term solutions to the load problem. Joining and splitting a fragment can be performed as a more long term solution to a load balancing problem.

The conclusion is thus that join/split fragmentation is an activity that can be performed when the system is not highly loaded. The join/split activity can almost always be postponed until a time when the system is moderately loaded or even during low load. The join/split fragmentation must be performed on-line since telecom databases must always be on-line.

7.4.1 Split Algorithm

The split algorithm is a rather long process, it consists of a number of steps, most of them involving the use of the previously described on-line recovery protocols. A number of algorithms local to nodes are also used.

The idea used here is that when a fragment is created, it is always prepared for a split operation. Thereby a split operation can reuse work done by on-line recovery. Thereby if a fragment replica has been created since the last split, it is prepared for the split operation. If there are fragment replicas that have not been prepared for the split, then these replicas must be moved before the split can take place.

The split algorithm consists of the following steps:

1) If the primary replica is not prepared, a backup replica is created. When finished, the roles of a backup replica and the primary replica are switched. After this switch the old primary replica can be dropped. If there are backup nodes not yet prepared, then perform the same algorithm here, i.e. create a new backup replica and when finished drop the old backup replica. Stand-by replicas do not need to be split before the actual split is performed.

2) The software should be prepared to handle requests concurrently that use the old fragmentation and new requests that use the new fragmentation. When all replicas have been prepared for the split, then the distribution information in all nodes is updated to reflect the split. Until local checkpoints have been produced for the split fragments, the distribution information should keep information that the split was performed previously and is not stably saved on disk yet.

7.4.2 Join Algorithm

The join algorithm is similar to the split algorithm. One of the merging fragments is assigned to be the moving fragment. This fragment is moved so that it is collocated with the other fragment in each replica. The fragments are locally merged already during the copy process. Finally the join decision is broadcasted to all other nodes using a state transition protocol. It consists of the following steps:

1) Create a new backup replica of the moving fragment. This replica is collocated with the primary replica of the remaining fragment.

2) Switch so that the new backup replica of the moving fragment becomes primary replica of the moving fragment.

3) Delete the old primary replica of the moving fragment.

4) Create a new backup replica of the moving fragment. It is collocated with a backup replica of the remaining fragment.

5) Delete an old backup replica of the moving fragment. If there are more backup replicas then perform step 4) and 5) again until all backup replicas have been moved.

6) Create new stand-by replicas of all stand-by replicas in the moving fragment. These are collocated with the stand-by replicas of the remaining fragment.

7) The last step is to inform all nodes in the system of the join of these fragments. The software must be prepared to handle requests both from before the join and after the join concurrently.

7.4.3 Split/Join Coordinator

Since the split/join process is a long task, obviously there are many possible error situations during the process. The system coordinator must handle the starting and ending of each of the steps described above. The state of the split/join process is reported to all nodes as part of the state transition protocols executed during the process. The system coordinator also handles the load situation and is therefore capable of assigning the proper node to receive the moving fragment even in situations where nodes fail.

7.4.4 Node Failures during Split/Join Process

If an error occurs during the split/join process, the process stops. It is not restarted until all the involved fragments have fully recovered from the failures that occurred.

This means that when executing some of the steps above, the steps could have been performed already. If so, they are simply acknowledged immediately as finished and the next step can be started. Thereby the activities during the disrupted split/join process can be reused as much as possible.

7.5 Related Work

The I-am-alive protocol is an old concept and is realised in [ClustRa95] and also in many other products. It is sometimes called a heartbeat function. There is even an operating system interface to I-am-alive protocols when using Digital workstations using Memory Channel. The I-am-alive protocol in this thesis has added an extra ring in the I-am-alive protocol. This is to make sure that a failing node does not start deleting its neighbour from the alive set.

In [Skeen85] a state transition protocol is used to update a copy status table which contained information on the distribution of copies and the status of the copies. It also contains a description of a node fail protocol and some protocols describing how to handle node failures. [CHAMB92] describes state transition protocols, scanning of fragment distribution tables, and handling of node failures. In an appendix to [CHAMB92] a proof is also given that the method of dynamically distributing data produces a correct behaviour. The method in [CHAMB92] performs a system restart when all replicas of a fragment fail and does only support access through a primary key. This thesis contains new information in Chapter 11 and 12 on how to perform the copy fragment protocol in conjunction with a derivative of linear hashing and a compressed B+tree.

[CHAMB92] uses an on-line recovery of fragments where the fragment is scanned and the transactions are updating all fragments. It does, however, lock the full fragment and then transfers it. Our approach is to transfer one tuple at a time while using the same general idea. The general idea to scan the process while user transactions update both the new and the old fragment is used in many of our algorithms. It can be applied for schema changes and also to perform a scan of table and receive a result which is consistent without locking the entire table.

The concept of splitting and joining fragments has been discussed in [Hvasshovd96]. No details are given on how the split and join are performed. Also this split and join process is used has to handle node failures rather than handling growing and shrinking database systems. Other work considered how to perform on-line split and merge of fragments where the fragmentation algorithm is more static [Tandem96][Sybase96]. No details on the Sybase method were given. The split is performed without actually moving the data. This is possible if the data is clustered on the fragmentation attribute. In our case this is not possible since the fragmentation is performed by a hashing algorithm. It is possible to prepare for one or two splits but eventually a split where data is moved is necessary. The Tandem approach used the log, the new fragment was created by a scan process and a number of scans of the log. Then an outage was required during the actual change of fragmentation. Our approach avoids this by handling requests before the split and requests after the split concurrently. We have a soft split fragment process. Thereby the outage in our approach is almost neglible. There is also much work on splitting fragments using LH*-variants. One such report is [KARL96]. These reports do, however, not consider transactions. Another report [Litwin96] considers a replication scheme but does not consider transactions and the replication is also static and cannot change in case of failures.

Not many research reports, if any, discuss protocols to create schema and distribution information in starting nodes. They are a necessary part of the design of a parallel database but are normally not reported in research articles.

8 Protocols to Implement Global Replication

This section describes the second level of replication protocols and how they can be used together with the local replication protocols to ensure a higher level of reliability. The idea is that replicas of the telecom database should exist on other telecom databases. The telecom database is then replicated as a whole. It is not possible to have different replication on different fragments on this level. The telecom database must be replicated as a whole. The replicas at the backup system themselves may, however, and even should be fragmented. This chapter also makes a detailed account of failure situations and how they are handled.

8.1 Protocol between Telecom Databases

The transaction is first executed at the primary system and then later executed on the backup system, so the backup system must contain all data that is within the transaction. The basis of this protocol is the protocol developed in [KING91] that has been extended with more details on how to interface the global replication protocol with the local replication protocols used in this thesis.

The fragmentation at the backup system can be used so that fragments of the global replica can actually be placed on different telecom databases. Also the primary systems can be viewed as a part of a globally distributed database. Another view of the primary system and the backup system is shown in Figure 8-1. The ***kick-off nodes*** send the log records of a certain fragment to the backup system and these log records are received by a kick-off node in the backup system. Using the terminology in [KING91] the kick-off nodes are the host nodes. In [KING91] there is also a set of store nodes, and these correspond to the data nodes in Figure 8-1. They contain the actual data of each fragment.

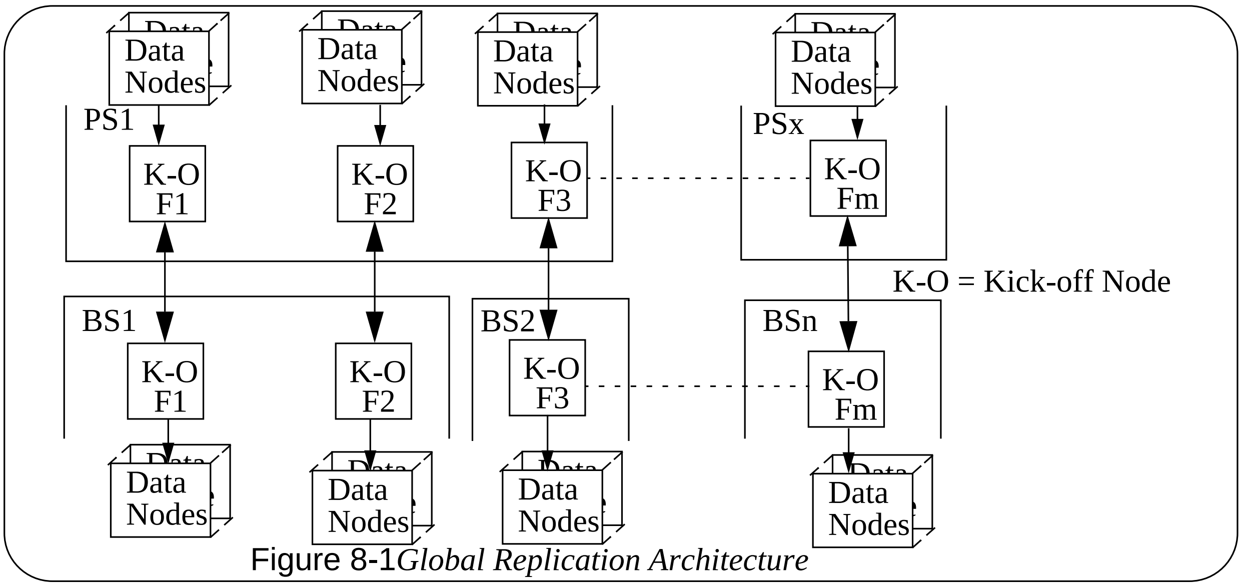


Figure 8-1*Global Replication Architecture*

The basic idea in [KING91] is that each transaction uses a serialisability protocol (e.g. two-phase locking) to access the data in the primary system. Each transaction is executed until completed in the primary system before contacting the backup system. When a commit decision has been reached, the primary system assigns a ticket to the transaction in each of the global fragments. Then the messages are propagated to the backup system where the tickets are used to retrieve locks in the same serialisability order. The data resources in the primary system are held either until the ticket has been assigned or until the backup system has completed its transaction and acknowledged the transaction. The first case becomes 1-safe and the second case 2-safe. These protocols can be chosen per transaction.

The idea is that the application describes which attributes of a table that need to be updated in a 2-safe manner. If any such attribute is updated in the transaction, the transaction should be 2-safe.

All operations in the primary system are assigned a ***ticket number*** and a log record is generated that is later sent to the backup system. Even read operations belonging to transactions that perform updates must be part of these log records, the reason for this is shown in [KING91] and a brief explanation is also given in Section 8.2.1.1. Pure read transactions, however, do not need to be sent to the backup system since they do not change the state of the database.

The transaction in the backup system must be handled by a two-phase commit protocol since it is a distributed transaction. This is necessary to synchronise the updates on different fragments in the backup system. As shown in Figure 8-1 several backup systems can be involved in the transaction.

The fragmentation at global replication level and local replication level may be different. If they are different, the kick-off nodes in the primary system must take care of assigning the tickets used by each transaction. The kick-off nodes in the backup system must take care of the locking protocol in the backup system. In our model the fragmentation at the primary system and backup system will be the same. The fragmentation is decided by the primary system. This means that fragment splits and fragment joins must also be handled in the network interface.

If global replication protocols are ever standardised, it is likely that local fragmentation at the primary system and the backup system differ. The protocols that we develop in this thesis, however, only specify the fragmentation algorithm in the global replication protocol. Other systems could use these protocols and implement another fragmentation algorithm locally. We will, however, describe how it is implemented when they are the same since this implementation becomes more efficient.

8.2 Adaption of Transaction Protocols for Global Replication

In our model as shown in the previous section, the data nodes are a set of nodes that store primary, backup and stand-by replicas of the fragment. In Figure 8-5 it is shown how the global replication architecture and local replication architecture interact. The idea is that the local replicas are ordered as in the local replication protocol. The primary replica acts as the kick-off node in the primary system. The assignment of tickets is also performed by the primary replica. This is necessary since all nodes must have information on the ticket number if the primary replica fails. The kick-off node in the backup system is normally the primary node, but in some rare cases it could be another replica type. This is possible since the backup system is not used for reads thereby if a fragment only has a stand-by replica, and the fragment can continue to participate in transactions of the primary system. The backup system should however strive to have a primary replica of each fragment so that take-over processing is fast.

As mentioned above the message to the backup system is sent after the commit decision. This means that the backup system must commit. It is not allowed to vote no and abort the transaction. This is certainly true in the 1-safe case and is used also in the 2-safe case to enable flexible use of

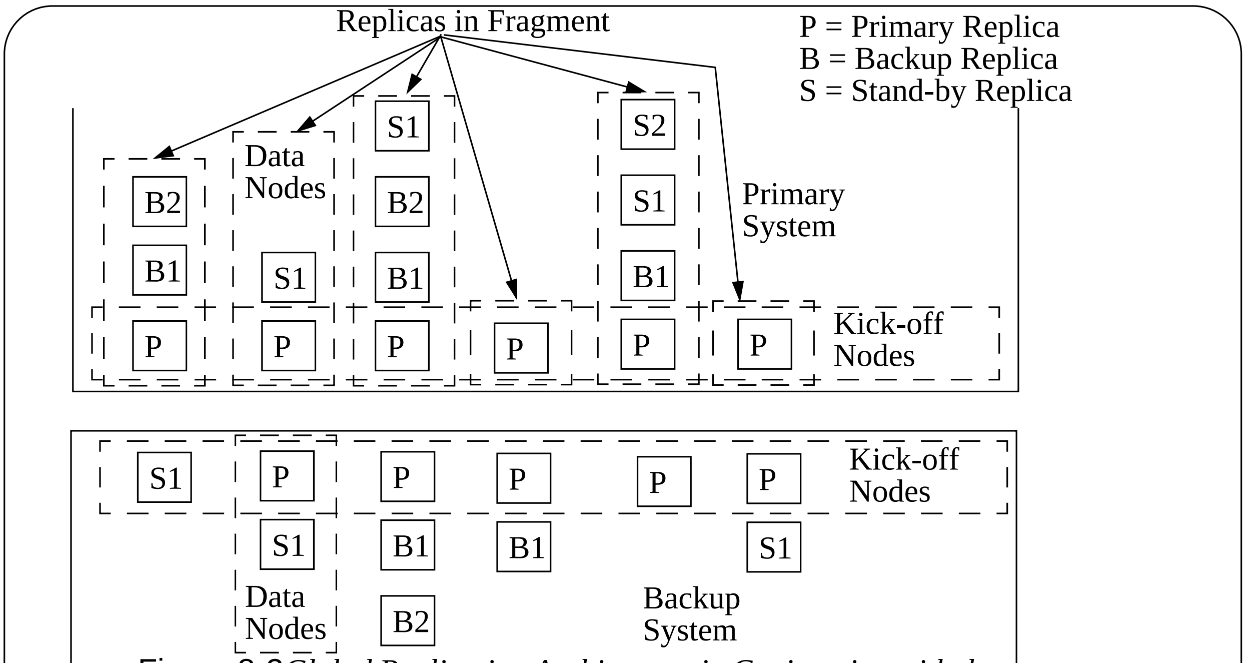


Figure 8-2*Global Replication Architecture in Conjunction with the Local Replication Architecture*

1-safe and 2-safe. To make sure that the commit is safely stored, the commit decision is first sent to the stand-by and backup replicas. The primary replica is the last node to receive the commit message. This means that the locks on the tuples are held until the message is sent to the backup system.

8.2.1 Commit Phase in Primary System

The commit phase is an extension of the commit phase described in section 6.1. The idea is very simple. When the commit decision reaches the primary replica, then it gathers the log records for this fragment in the transaction and sends them off to the backup system. If 1-safe is used, the commit phase continues as soon as the commit message have reached the primary replica as seen in Figure 8-3. The commit phase is concluded without waiting for the backup system. The completion phase does, however, not finish until the backup system has acknowledged the execution of the transaction.

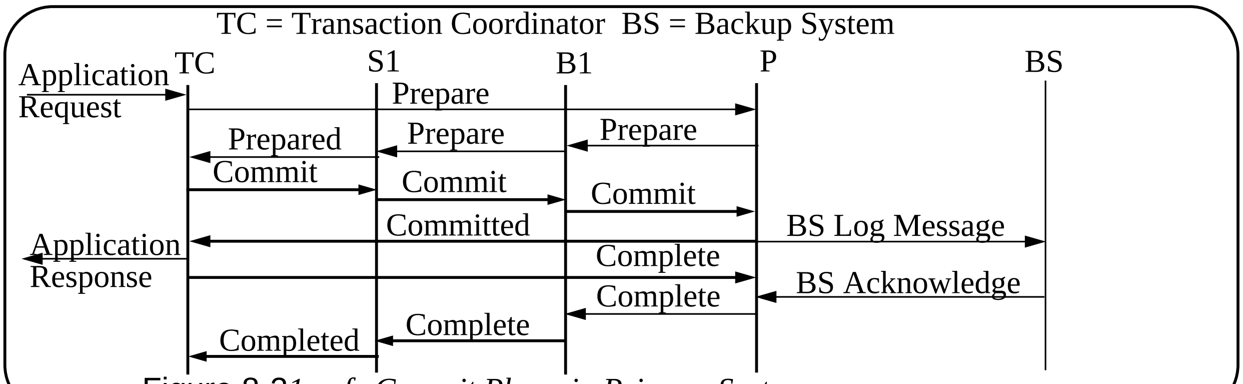


Figure 8-3*1-safe Commit Phase in Primary System*

If 2-safe processing of the transaction is used, the commit phase also waits for the backup system as seen in Figure 8-4. In Figure 8-5 the messages in the backup system are also shown.

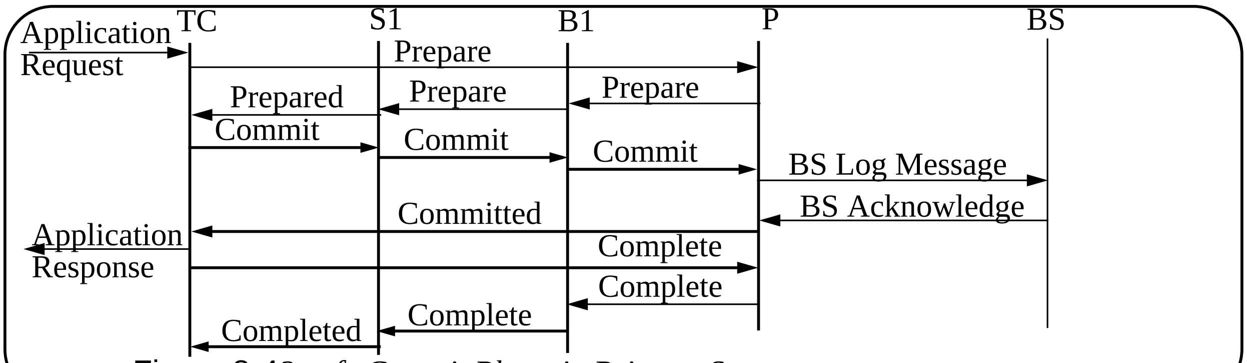


Figure 8-4*2-safe Commit Phase in Primary System*

This is the simple adaption needed of the two-phase commit protocol described in the previous section. All the optimisations are still usable and with multiple fragments involved in the transaction, the protocol described here is used in all fragments. Each fragment has its own log channel to the backup system.

8.2.1.1 Considerations for Lock Handling

As mentioned in [KING91] the read operations of write transactions must also be logged. The reason is that the protocol aims to restart a consistent backup system even if there are lost transactions. If a particular transaction is lost that performed some updates based on reading another tuple, then no transaction that updated the read tuple must survive the crash. If such a transaction did survive the crash, the consistency of the backup system is not ensured when restarted.

Only read operations of write transactions performed as 1-safe need be logged. 2-safe transactions are certain to survive the crash at the backup system if they were completed in the primary system.

This means that read operations need to be executed in all replicas. The actual reading of the object is only needed in the primary replica, but all the replicas must keep track of which objects have been read. If the primary replica fails and the log message has not yet been sent to the backup system, then the backup and stand-by replica must be able to send the log message of the transaction to the backup system. This is needed since the primary system decides to commit and then sends the message to the backup system. Failure of the primary replica in this situation does not mean that the transaction can abort, therefore the backup and stand-by must be able to take over all processing and using global replication as in [KING91], which involves keeping track of read operations.

The backup and stand-by replica must also keep locks on all tuples to ensure that serialisability is not endangered. If the primary replica fails, the locks on tuples must still be kept. Therefore the backup and stand-by replica also keep all locks on all tuples.

8.2.1.2 Ticket Assignment

Since the primary replica is the last node to receive the commit decision, the ticket assignment must be performed in the prepare phase. Otherwise a separate ticket sending phase would be needed before the backup system can be contacted. This means that the primary replica assigns a ticket number already in the prepare phase, before committing the transaction. This could introduce bubbles in the ticket numbering if transactions were aborted and no report of this reaches the backup system. Therefore the aborted ticket numbers are reported to the backup system.

8.2.2 Transaction Handling in Backup System

The transaction handling in the backup system is a normal two-phase commit protocol as described in [KING91]. The two-phase commit phase protocol as described in the previous section is adjusted to handle also transactions in the backup system. When receiving the message from the primary system, the backup system understands this as a number of operations and a prepare message. It starts a combined update and prepare phase going from the primary to the last replica in the chain. Finally the message is sent to the transaction coordinator. The transaction coordinator then sends the commit message to the last replica in the chain and the commit decision is then propagated to the primary replica and then the acknowledgement is sent from the backup system to the primary system.

8.2.2.1 Assigning Transaction Coordinator

An issue that was left open in [KING91] was how to assign the transaction coordinator in the backup system. Our solution is to use the primary replica in the chain for simple writes to enable the optimisations as described in the previous section.

With transactions that involve the use of several fragments another algorithm is used. The primary system decides which node should be transaction coordinator. It does however only assign it logically. It assigns a logical node id (e.g. 2 bytes) at random or in a round-robin fashion. This number is translated into a real node id in the backup system.

The translation table is handled as a fully replicated data structure within the backup system. The logical node id and the global checkpoint identity is used to find the transaction coordinator. Thereby the backup system can change the mapping to perform local load regulation without involving the primary system. By involving the global checkpoint identity the change can be performed in the system without the need of a complex algorithm. All fragments know the global checkpoint identity of the transaction and will thereby all choose the same transaction coordinator. If this coordinator fails, it is handled as if the transaction coordinator had failed as described previously.

8.2.3 Conclusions

In Figure 8-5 the message sent in the commit phase in the primary system and in the backup system is shown with a transaction where three fragments are involved.

These ideas can also be used to build global databases. It is however an important restriction that the backup system must contain all data contained in the primary system. The primary system might also be distributed as in Figure 8-1. The delay aspect is also important to consider if building global databases. It is likely that it is necessary to place the primary as close as possible to those who are changing the data. In some applications it might even be very difficult to use this scheme.

Telecommunication applications are very often connected to some part of the world. Subscriber data should be placed in the same part of the world in which the subscriber lives. Genealogical data for Sweden is mostly used by swedish genealogists. Thereby this architecture can most often be used to create global databases for a newspaper, genealogy, subscriber data and so forth.

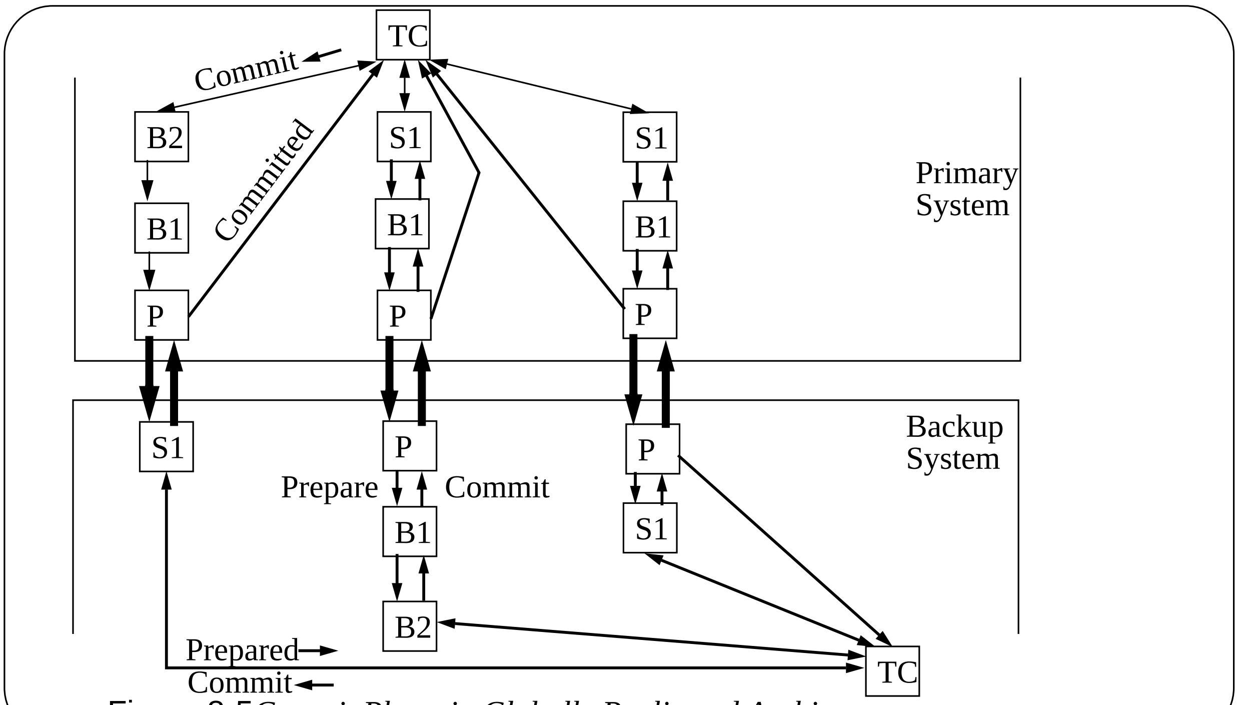


Figure 8-5*Commit Phase in Globally Replicated Architecture*

8.3 Handling Processor Node Failures Using Global Replication

Global replication introduces a new complexity in handling the connection between the primary system and the backup system. This connection is set-up between the primary replicas in the primary and backup system. Failures of these primary replicas must be handled to assure correct behaviour.

The backup system handles transaction in a different way compared to the primary system. The primary system has the ability to abort transactions when failures occur. This is not allowed in the backup system. Failure to commit a transaction in the backup system leads to failure of the backup system, not only the transaction. Therefore transactions must be ensured commitment in the backup system. This means that handling node failures in the commit phase is a little bit different in the backup system compared to the primary system.

8.3.1 Handling Node Failures in Backup System

If all nodes containing a replica of a fragment fails in the backup system, the backup system fails to participate in transactions. Therefore the backup system must be declared failed in such a case. This means that some of the problems with node failures (see Section 6.2.3) in the backup system disappear. In the primary system it was necessary to delay releasing locks in backup and stand-by nodes until the primary node committed. The reason was the possibility of reading backup data with simple reads and the possibility of total failure of a fragment. Neither of those restrictions apply in the backup system. Therefore locks can be released already in the commit phase in the backup and stand-by nodes. In the actual design this feature is probably not used. The same solution as in the primary system is most likely used since the higher concurrency cannot be used anyway.

To lower the possibility of failure of backup systems, a new stand-by node is started if a replica of a fragment fails. The creation of stand-by copies is somewhat different in the backup system. This is due to the global checkpoints being taken in the primary system. The stand-by node is then first started from the lowest global checkpoint which has not yet been received in the backup system.

Complete failures of all fragment replicas are not handled in the backup system. The backup system must be restarted anyway in the case of a total fragment failure. Therefore it is not necessary to flush the log to disk in error situations.

8.3.2 Handling Failure of Transaction Coordinator in Backup System

The handling of a transaction coordinator that failed is similar to the handling in the primary system. The coordinator still takes over and performs the same actions. The difference is how to handle the committing and aborting of transactions. In the primary system it is allowed to commit transactions. This is not allowed in the backup system. Aborting a transaction in the backup system can only be performed when the primary system has failed. If the backup system aborts a transaction known to be committed in the primary system, then the backup system is declared failed. Thereby in uncertain situations, the backup system must wait until the required messages arrive from the primary system.

8.3.3 Failure of Primary Replica in Primary System

When the primary replica fails, the new primary replica must take over the communication with the primary node in the backup system. To be started, the new primary replica must also find a ticket number that is certain to be bigger than what can have been sent to the backup system and which is not already known to have been assigned by the failed primary replica. The new primary replica knows of some transactions that have been sent to the backup system although not yet completed. These are taken care of by the normal handling of failures in the primary system. The new primary replica also knows of transactions that have not yet been prepared in the new replica. These will definitely be aborted due to failure of primary replica. Therefore it does not matter whether the failed primary replica actually had assigned already a ticket number to this transaction. The transaction can be safely discarded from the list of possible owners of ticket numbers.

The problematic messages are those that have passed the new primary replica to be committed (these passed when the new primary replica still was a backup replica) and where there has been no message stating that the backup system has received the message. These transactions are in an unknown state. There are several possible options on what could have happened.

a) The message might not have been sent at all.

b) The message can have been sent, but not acknowledged.

c) The message can have been sent and even been acknowledged.

To find out about these unknown transactions, the new primary replica asks the primary node in the backup system about its state on the ticket numbers. It asks particularly for all ticket numbers higher or equal to the minimum of the unknown ticket numbers.

During the time the primary system waits for the reply, the primary system has started to process requests again. The new primary replica then sends the following information to the primary node in the backup system (in priority order):

1) New committed transactions

2) Aborted ticket numbers, from highest to lowest ticket number

3) Old committed transactions from highest to lowest ticket number

The idea is to use the time waiting for the backup system to respond and still to avoid as much as possible resending messages already received by the backup system. If 2) and 3) are not performed, then there is no possibility for duplicate messages in the backup system which could be beneficial as a simplification of the design.

When the response is received from the backup system, the states of all ticket numbers are known. The new primary replica sends the information about the tickets in the following priority order:

1) Aborted transactions from highest to lowest ticket number (that was unknown in the backup system)

2) Old committed transactions from lowest to highest ticket number

3) New committed transactions as they arrive

8.3.4 Failure of Primary Replica in Backup System

When a new primary node in the backup system has been assigned, it sends a take-over message to the primary system. This message actually contains the same information as the response by the primary node in the backup system after failure of primary replica in primary system. The handling of the take-over message is the same as handling of the response given previously.

8.3.5 Failure of Primary Replica in Primary and Backup System

This case is handled in the same way as the handling of failures of the primary replica in the backup system.

8.4 Global Recovery Protocols

The recovery protocols needed are protocols to restart a system as the backup system when a primary system is alive. A protocol is also needed to promote the backup system to the primary system when the primary system has failed and also a protocol to switch the roles of the primary system and the backup system. To ensure that there is one view on which systems are alive and which are not, there should also be an I-am-alive protocol even at network level.

8.4.1 Restart Backup System from Primary System

The idea of this protocol is rather simple. The idea is to scan all the fragments and send one tuple at a time to the backup system. Each fragment is, however, performed independently of others and thereby there is a parallel activity in all fragments. The restart is finished when all fragments at the backup system are declared as up-to-date. A fragment is declared as up-to-date when all scan transactions of the entire fragment have been acknowledged (see [KING91]). The scan process is performed by using the same algorithm as the copy fragment protocol. The major difference here is that there is no receiver in the system. The read lock can thereby be released as soon as the message has been sent to the backup system. The details of the behaviour of this protocol are shown in the section describing LH^3. All tuples read and sent must be tagged with a ticket also to ensure that concurrent transactions can be performed.

If the sending system fails before the restart is completed, then the restart was unsuccessful and there are no procedures to continue. If no other system contains a backup copy, the database has crashed and must be restarted from an archive copy.

8.4.2 Promote Backup System to Primary System

The process used for the backup system to take over after failure of primary system follows [KING91]. Since locks of write transactions are serialisable to the order in the primary system, those transactions that cannot be committed must abort. When all active transactions have committed or aborted, the backup system is ready to take over as primary system.

8.4.3 Switch Primary System and Backup System

A controlled switch between primary and backup system can occur when it is necessary to bring down the primary system for some reason. It could be that the primary system is to be replaced, a major software replacement is to be performed or something else. The switching between primary and backup replicas within a telecom database is performed by stopping all operations and then proceeding with the transactions with the new constellation. This is possible since it is not necessary to move the transaction coordinator. When switching systems the coordinator of the transactions must be moved to the new system. This seems plausible. It doe, however, seem overkill to create a very complex take-over protocol to perform this type of switch-over 1 second faster than otherwise. This should be a very uncommon action and is therefore allowed to be more costly when it is executed.

The conclusion is, therefore, that a switch is performed by taking the following steps:

1) Prepare Backup System such that all fragments have a primary replica
2) Start a Global State Transition Protocol that switches the roles of the primary and backup systems, when the prepare message arrives no new transactions are accepted
3) Wait until all active transactions have been executed on both primary and backup system
4) Set Backup System as Primary System and Primary System as Backup System
5) Start accepting new transactions at the new Primary System
6) Activate the new global state such that transactions can be started again

This simplifies the design of the switch since there are no active transactions when the switch is performed. Thereby it is easy to create a distinct time when the switch was performed. It is not necessary to send any states of the primary system to the backup system. It is likely that this type of switch-over will take a few seconds, which should be acceptable if it only happens rarely.

This protocol can be used to replace a system also when the system is not using global replication. Such a replacement starts by starting the new system as backup system first, then the switch is performed and finally the old primary system is taken out of action. To avoid a lengthy switch due to long transactions it should be ensured that no long transactions are occurring during the switch. If by mistake there are transactions that are running a long time, these transactions should be aborted during the switch, after some time-out.

8.5 Related Work

The major sources on providing network redundancy or global replication to secure against crashes due to earthquakes, for example, is [KING91][POLY94]. The ideas in [KING91] were very appropriate in this architecture and has therefore been used without any major rework. The next chapter of this thesis will describe how to handle simple and complex schema changes also in relation to a system using global replication.

The main new result in this chapter is the integration of network redundancy into our new two-phase commit protocol where no writes to disk are performed. Also how to handle node failures that are involved in the communication between the primary system and the backup system have been discussed.

9 Crash Recovery Protocols

If a fragment replica fails and there is still other active fragment replicas alive (i.e. primary or back-up replicas) then it is possible to recover fragments using the on-line recovery protocols. However if all fragment replicas fail, then it is necessary to restart the fragment using information saved on permanent storage (e.g. disks).

If all fragments fail, a system shutdown might also occur. A system restart is handled by performing some initial load of schema information and then restarting the fragments independently of each other. As soon as a fragment is recovered, the system can start accept transactions using this fragment.

To be able to perform a restart of the system or a fragment we need to store information during normal transaction processing that makes this possible. The information needed will be described and we will also describe how to use this information to restart a fragment in different failure cases.

One item needed to restart a fragment is the fragment log. In previous chapters the fragment log was referred to as the log. Now that we go into more detail about crash recovery, we will also find the need of a local log. Thereby we use the name fragment log to differentiate between those two logs. This log saves all operations on the fragment, as well as information on transaction outcome. This log is logical since it is necessary that it is location and replication independent. This independence makes it possible to use a log produced at one node to restart another node. This is a basic requirement to be able to use the idea with stand-by nodes as described earlier. Thereby we can restart a node by using a local checkpoint and a fragment log which resides on another node.

Since the log saves only the logical information of the operations, these log records can only be used in a database with indexes, tuple information, and transaction information which is consistent and available. This means that before the fragment logs can be applied we must produce an action consistent database. This means that the database reflects a state where no action is active; transactions can, however, be alive.

This requirement can be fulfilled by stopping the database while creating an action consistent checkpoint. This means that the real-time requirements of the telecom database would not be met. Hence this solution is not used. Instead a fuzzy checkpoint method is used. Fuzzy checkpoints do, however, not produce action consistent checkpoints. Hence we need to add another log that is used together with the fuzzy checkpoint to produce an action consistent checkpoint. This log makes it possible to put the data structures in an action consistent state, and thus this is not a log of logical database operations. It is a log of actions on the internal data structures of the database. The database is organised in pages and the log saves the actions on these pages. To avoid the need of logging management activities we use a form of physiological logging [Gray93] where we specify in a logical manner actions on a specific database page.

The production of the fuzzy checkpoint and the log is performed locally in a node and we will call the log activity ***local logging***. The files saved on disk are normally to be used by the same node; if nodes can access a common disk subsystem, these files could also be used by another node to restart the fragment. The telecom database still uses shared nothing so while operating on database pages these are always internal to the processor node. At recovery these files could, however, be taken over by another node if the hardware of the system supports this.

This section will start by describing how restart information can be found. To restart it is necessary to restore schema information, fragment information and restore fragments.

9.1 System Restart

A system restart is performed on our method by first assigning the global checkpoint to restore. Then schema information is restored to find which tables should be recovered. Using distribution information to find all fragments of the tables to restore and recover information for each fragment. Then each fragment is recovered and the system is prepared to start when distribution information have been built for the restarted system.

9.1.1 Find Global Checkpoint to Restart from

The first step in a system restart is to decide from which point in time the restart is performed. In our case this means deciding which ***global checkpoint to restore***. If the failure occurred such that the system restart was ordered by alive nodes, these alive nodes know which is the latest global checkpoint that can be restored. If the system restart happens after a total failure where all nodes failed, it could be difficult to decide which is the latest global checkpoint that can be restored since nodes can have failed in any manner. It is difficult to find this information in a restart situation if not all nodes are available. We assume that there are two options to solve this. Either the operator makes the decision based on error reports that the operator received during the total failure. Otherwise the latest global checkpoint among the nodes to restart is found and used as the global checkpoint to restore. [Skeen85] contains a description of a solution to the problem of restarting from a proper checkpoint.

9.1.2 Restore Schema Information

The first step in the recovery process is to find the tables and indexes that were used at the global checkpoint to be restored by crash recovery. This includes all tables and indexes defined in the schema version that was current during the global checkpoint to restore. This version must be checkpointed when creating a global checkpoint. The older versions of the schema are not needed since at crash recovery there is obviously no active transaction that uses older schema versions. Thereby only the current schema version needs to be restored. Since all nodes have access to this schema information, all nodes can recover this information in parallel.

This information is found in files describing a checkpoint of schema information and files containing a log of all schema changes performed after the checkpoint as seen in the figure below.

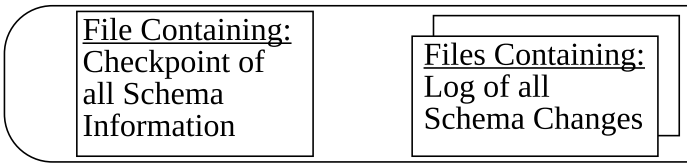

File Containing:
Checkpoint of all Schema Information
Files Containing:
Log of all Schema Changes

Figure 9-1*Files needed to restore Schema Information*

There are two exceptions that needs special care. One situation is when a table or index is part of the global checkpoint to restore but there are no local checkpoints to start the restoration from. In this case the table (or index) is created but no local checkpoint was produced before the crash. Hence an empty table (or index) is created instead of using information in a local checkpoint. Then the log records of the fragments are executed on this empty table (or index).

Another case is when a table has local checkpoints but is not part of the global checkpoint to restore. In this case the table was dropped before the global checkpoint. This table can then be ignored in the crash recovery.

The ***Saga Table***, described in the next chapter, is used by complex schema changes. It contains information on schema information that should be removed at crash recovery since the complex schema change did not finish before the crash. Thereby a number of tables, indexes, foreign keys and attributes could be dropped before crash recovery starts. If needed one could define entries in the Saga Table that defines how to continue the processing of a complex schema change. In this case these entries are performed after crash recovery and before the system is started again. In this case also the final schema change needed must be specified.

9.1.2.1 Schema Checkpoint

To be able to restart the system the schema information must be checkpointed. This ***schema checkpoint*** can easily be synchronised with the global checkpoints. The schema checkpoint should contain the schema information on the current version at the time of the global checkpoint. This can simply be written to disk as a background process after the global checkpoint.

It is likely that it is more efficient to do a complete write of the schema information only occasionally and then only write log records describing each transaction that changes the schema. The checkpoint of the schema information should then contain a reference to a file with complete schema information and a reference to the log file(s). Then at restart, information that was dropped before the last completed version can be dropped.

Schema checkpoints should be performed by all nodes to enable all nodes to do a system restart. If the system uses a physically shared disk subsystem, then this could be avoided by those nodes using the same version of the database software.

9.1.3 Restore Fragment Information

After restoring the schema information it is known which tables and indexes should be part of the crash recovery. Then the next step is to find the fragmentation of each table and to find the information needed to restart the system. Each node should have access to fragment information needed by crash recovery. However only one node that acts as coordinator of the crash recovery will use this information to restart the system. This coordinator decides which nodes will restore which fragments. The actual method to restore a fragment will be described in section 9.2.

The node to restart the system must have access to a file that contains descriptions of all fragments. For each fragment the following items must be stored:
1) Table identifier
2) Fragment identifier
3) Nodes of replicas and which global checkpoint they have completed
4) Local checkpoints and which global checkpoint that were completed before the local checkpoint
5) For each local checkpoint information about the backup log node is needed (see section 9.3.2)

The information in this file is created by checkpointing the distribution information.

The node that restarts the system will then choose for each fragment a node to restore a primary replica of each fragment. It will inform this node about which local checkpoint to use and which global checkpoint to restore. This node must contain the following information:

1) Files of the local checkpoint
2) Files of the fragment logs from the global checkpoint before the local checkpoint up to the global checkpoint to restore
3) Files of the local UNDO log

The description below states what information items are needed to be able to restart a system given the restored schema information. This information is typically found in a set of files with predefined names. These files then contain references to other files that contain the actual information needed for the system restart, such as the database pages, fragment logs and local UNDO logs. We will also provide a less formal textual description of this information below.

```
SET OF Table Descriptions {
   Table Reference,
   Table Name,
   Table Properties,
   SET OF Fragment Descriptors {
      Fragment Reference,
      Fragment Properties,
      Primary Node {
         Node Reference,
         Completed Global Checkpoint,
         SET OF Local Checkpoints {
            Local Checkpoint Reference,
            Properties of Local Checkpoint,
            Global Checkpoint completed before Local checkpoint
            IF This Node = Primary Node THEN
               SET OF Description of File for the Local Checkpoint
               SET OF Description of File for the local UNDO log
              Node Reference of Backup Log Node,
            ENDIF
         },
         IF This Node = Primary Node THEN
            SET OF Description of File for the Fragment Logs
         ENDIF
      },
      SET OF Backup Nodes {
         Node Reference,
         Completed Global Checkpoint,
         SET OF Local Checkpoints {
            Local Checkpoint Reference
            Properties of Local Checkpoint,
            Global Checkpoint completed before Local checkpoint,
            IF This Node = the Backup Node THEN
               SET OF Description of File for the Local Checkpoint,
               SET OF Description of File for the local UNDO log,
               Node Reference of Backup Log Node,
            ENDIF
```

```
        },
         IF This Node = Primary Node THEN
              SET OF Description of File for the Fragment Log,
         ENDIF
      }
      SET OF Stand-by Nodes {
         Completed Global Checkpoint,
         IF This Node = the Stand-by Node THEN
            SET OF Description of File for the Fragment Log,
         ENDIF
      },
      SET OF Backup Logs {
         IF This Node = Backup Log Node THEN
            Reference of Local Checkpoint,
            Primary Log Node Reference,
            SET OF Description of File for the local UNDO log
         ENDIF
      }
   }
}
```

If the node does not contain enough fragment logs and/or local UNDO logs, the node controlling the restart will inform the primary node of which nodes contain this information.

First of all the fragment information must specify how each table is fragmented. Then for each fragment there are three entities that need to be found. The first are the local checkpoints of a fragment. How many such are kept per processor node is defined by the database administrator, one or two could be a normal figure. It is also desirable to have access to physiological UNDO-log records. These log records are used to restore database pages to a consistent state. They can be saved in more than one main memory to avoid forcing the log to disk at each page write. Thereby in some crash situations it is necessary to have access to the UNDO-log in the processor node of backup local log. Finally we need access to the fragment log.

To find the local checkpoints of a fragment the table identity and the fragment identity is provided. Then a set of local checkpoints are found with the information about the processor node that created the local checkpoint, the identity of the local checkpoint, and the last global checkpoint finished before the local checkpoint. Through this information all files of the local checkpoint can be retrieved.

To find the UNDO-log files of a fragment the table identity and the fragment identity is provided. Then a set of UNDO-log files are found with information about the identity of the local checkpoint, the processor node that created the local checkpoint and the processor node that saved the UNDO-log file. This information can be used to find the files.

To find the files of the fragment log the table identity and the fragment identity is provided. Then a set of files with fragment logs are found with information that specifies the processor node that created the file, the starting global checkpoint of the file and last global checkpoint of the file. All log records belonging to a global checkpoint of a fragment replica are saved in the same file.

9.1.4 Use of Schema Information during Crash Recovery

All logical log records are marked with a version number of the schema. The only schema information that is needed during recovery is information on tables, attributes and indexes. However, the recovery is performed per fragment and a fragment is part of either a table or a secondary index. Thereby it is not possible for log entries to use a table not defined. If the table was dropped, then its fragment would not be restarted as mentioned above. Thereby only schema information regarding attributes need to be handled.

The attributes are specified in the log records by an attribute identity. In the schema information it is known which attributes are used at the global checkpoint to restore. It is also known at which schema version these attributes were created. If an attribute identity is found in a log record that needs to update an attribute that has not yet been created, then obviously the attribute used in the log record will be dropped before the global checkpoint to restore. Thereby these attributes are ignored in the log records.

From this analysis we find that the only attributes needed to define in the storage are those that are defined in the current schema version of the global checkpoint to restore.

In a local checkpoint we know which schema versions were active when it was created. We can thus deduce which attributes of the local checkpoint should be dropped and also which attributes need to be added since the attributes were added after the local checkpoint. As we will show later, there can be several schema versions concurrently available. The local checkpoint specifies the last completed schema version at the time of the local checkpoint and the current schema version at the time of the local checkpoint. Those attribute that were created after the current schema version of the local checkpoint need to be added. Those attributes with the same attribute identity as these new attributes have obviously been dropped and the attributes with attribute identities that are not among the currently defined attributes have also been dropped since the local checkpoint.

9.1.5 Archive Copies

When schema information, fragment information and the information to restore fragments are written to an off-line media such as tapes, an archive copy is produced. The files in an archive copy can be used in exactly the same structure as described above. Reference to a processor node is, however, only treated as a reference to a directory in the archive copy. The archive copy, too, only needs to contain one copy of each information entity.

A system restart from an archive copy is similar to a restart from information on disks. The only difference is that the information is read from an off-line media and then copied to the various nodes that should receive copies and then the restart can be performed as a normal system restart.

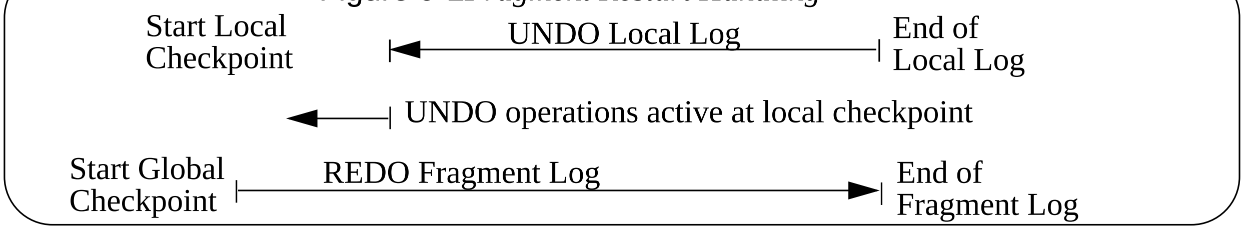
9.2 Restarting a Fragment

Restarting a fragment is a part of a system restart. It is also used when all replicas have failed but some replica survived at least long enough to flush the fragment log to disk. Hence the system is still operational even though the fragment is inaccessible. In this situation the fragment can be restarted without losing any committed data.

It can also be used by a stand-by node to become a backup node. This could be useful if the fragment is very large and disk-based. The stand-by node must have access to the data pages of a backup node and the UNDO log of same node. It must also be able to use this data, i.e. it has to support the same version of the data structures and the log structure as the backup node.

In Figure 9-2 the restart handling of a fragment is shown. The simple idea is basically to start with a physiological UNDO to get an action consistent checkpoint, to UNDO all operations active at the checkpoint, and then to apply the logical REDO log for the fragment. As mentioned previously the local checkpoint is action consistent. The local checkpoint also contains the set of locked tuples at the checkpoint. To simplify the recovery we will UNDO those operations before executing the fragment log.

Figure 9-2*Fragment Restart Handling*



We must, however, take care of the independence of local and global checkpoints. It must be ensured that the global checkpoint is older than the local checkpoint.

The restart of a fragment after a total failure of the fragment is performed by the following algorithm:

1) Install transaction information from local checkpoint

2) Install local checkpoint which is action consistent

It consists of the following steps:

- Execute the Local UNDO-log from the end to the start checkpoint marker. If a page accessed in the execution of the local log is not in the main memory, read it from disk.
- If it is a main memory based fragment, read all unaccessed pages from disk into main memory.

3) UNDO all active transactions before starting the execution of the REDO log. In our case this is performed by aid of the list of all tuples involved in transactions. Thereby no locks are set when starting the execution of the REDO log.

4) Execute the REDO fragment log from the last global checkpoint before the production of the local checkpoint. The log must be executed until the end of the log if a fragment is restarted and until the global checkpoint to be restored if the system is restarted.

The recovery of different fragments is independent of each of the others. As soon as a fragment has been recovered, it is possible to start processing requests that only access recovered fragments.

9.2.1 Handling the Logical Fragment REDO Log

The fragment logs are not synchronised with each other. The fragment logs produced on the primary, backup and stand-by need not even be equally ordered. They are, however, equivalent in terms of serialisability. This means that the fragment log does not contain a marker exactly where the local checkpoint was produced. This would be possible in the fragment log generated at the same processor node as the local checkpoint is produced. This is not enough, as it must be possible to use the fragment log of the stand-by replica to restart a fragment. Hence it becomes necessary to start processing of the fragment log at some point in the log that is sure to be before the production of the local checkpoint.

This point is placed at a global checkpoint. Each global checkpoint has a well defined point in the fragment log and it is possible to find out which global checkpoints were the last produced before the production of the local checkpoint.

To certify that this scheme actually works we must also check that executing the log records produced before the production of the local checkpoint does not harm the consistency of the system. To do this we need to analyse all possible events that can occur. The log record could have been produced before or after the local checkpoint and the tuple existed or not when executing the log record. We need to perform this event analysis for updates, inserts and deletes.

1) Insert log record, tuple exists:
Can only happen before local checkpoint => Ignore log record. Insert could only set old values that will later be overwritten by later log records or it could set the value which was set at the time of the local checkpoint.

2) Insert log record, tuple does not exist:
This could happen before or after local checkpoint. If the insert is performed before the local checkpoint, then the tuple will be deleted by a log record executed before the local checkpoint, and the insert can safely be performed. There is no possibility to distinguish between this case and the case where the log record is after the local checkpoint.

3) Delete log record, tuple does not exist:
Can only happen before local checkpoint => Ignore log record, this has the same behaviour as a deletion of the tuple.

4) Delete log record, tuple exists:
If it is before the checkpoint, the deletion will obviously remove a tuple that should be in the database. There is, however, no way to know this when executing the log record since there is no knowledge of whether it executes before or after local checkpoint. Hence the deletion must be performed. The tuple will certainly be inserted again by a later log record before the local checkpoint. Otherwise the tuple would not have been in the local checkpoint.

5) Update log record, tuple exists:
There is no way to know whether it was executed before or after the local checkpoint => Perform update. If the update sets an older value than in the local checkpoint, the new value will certainly be set by a later log record.

6) Update log record, tuple do not exist:
Can only happen before local checkpoint => Ignore log record. This must be followed by a delete log record before the local checkpoint, hence the log record can safely be ignored.

The conclusion is that the fragment log records must be executed as normal log records. The only special actions needed are when strange situations occur, i.e. deletions of already deleted tuples, inserts of already inserted tuples, and updates of non-existing tuples. These strange events can only occur due to executing log records before the local checkpoint. These log records can simply be ignored since it is known that they execute before the local checkpoint was produced. The other log records are executed normally and as shown above their action will still produce a consistent database.

9.2.2 Creating Backup Replica from Stand-by Replica by Restarting Fragment

The scheme above can be used but special consideration is needed when upgrading from stand-by node to backup node. The reason for this is that transactions are ongoing when this upgrading is performed. First the stand-by replica ensures that it has executed all committed transactions in the fragment log.

The next step is to start executing the ongoing operations. An abort, commit or a new operation on the same tuple in this transaction could arrive during execution of the operation. Then this event must be queued up until the execution is completed. It must actually wait until all operations of this transaction are completed.

When execution of all transactions has started, new transactions are also queued up until all previous transactions are completed. When the previous transactions have been finished the queued operations can be started. The last step is to inform all nodes of this state change.

9.3 Production of a Local Checkpoint

To produce a local checkpoint it is necessary to produce a snapshot (action consistent) of the database objects (see Figure 9-3). As mentioned above this is accomplished by producing a fuzzy checkpoint in conjunction with physiological logging during the checkpoint production. The physiological log can either be an UNDO log or a REDO log. Using a REDO log means that the checkpointed pages must not be overwritten between the checkpoints. This means that a directory is needed that specifies where pages can be found. Using a main memory database, it is possible to change this by having two copies of the data.

Using an UNDO log means that we can still use in-place updating. It has been verified in many products, prototypes and research articles that in-place updating is the superior method [Gray81]. The reason is that otherwise the directory of pages must be written each time a page is written, causing a serious degradation of performance.

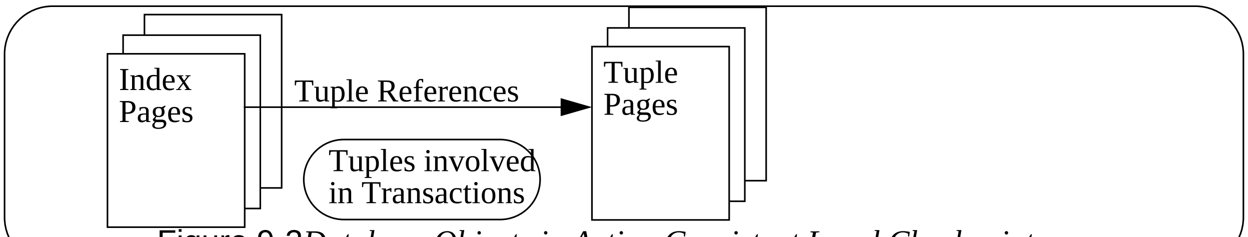


Figure 9-3*Database Objects in Action Consistent Local Checkpoint*

Thus the solution is to use an UNDO log. The consequence of this is that it is necessary to use WAL (Write-Ahead Log). This means that before a page is written to disk, the UNDO log records must be stored safely.

The production of local checkpoints are performed per fragment, which means that only a portion of the pages in the processor node is needed. All pages that contains data of the fragment must participate in the local checkpoint production. Even pages where information on other fragments is stored as well. The following algorithm is used to produce the local checkpoint:

1) All operations on the fragment are stopped. Those that are active execute until they are completed. No new operations are allowed to start. When all active operations are completed a start checkpoint is written in the local UNDO-log. This includes information on tuples locked by active transactions and the directory of the LH^3 index structure. When information has been saved the operations can start processing again. Also production of log records are started.

Those operations which are hanging on a lock and have not yet started should not be started before the local checkpoint. This ensures that the complete fragment will not suffer from a deadlock due to a deadlock situation involving a tuple in the fragment.

It is possible to allow read operations to continue during the stop. If this is allowed it is necessary to ensure that the set of locks covered by the checkpoint is consistent.

2) Start fuzzy checkpoint process. This means that one page at a time is locked. The UNDO log records are flushed and then the page is written to disk. When the page has been written, the page is unlocked. Main-memory based data that have several versions of the pages on disk do not need to flush the log records before writing the pages to disk. For them it is enough that they are flushed before the checkpoint is finished.

Before the page is written to disk and after the start of the local checkpoint, all activities on the page are logged in the local log. The information in the UNDO log records is dependent on the structures stored in the pages. After writing the log records, the local logging is stopped if the page is main-memory based (it will be described later why).

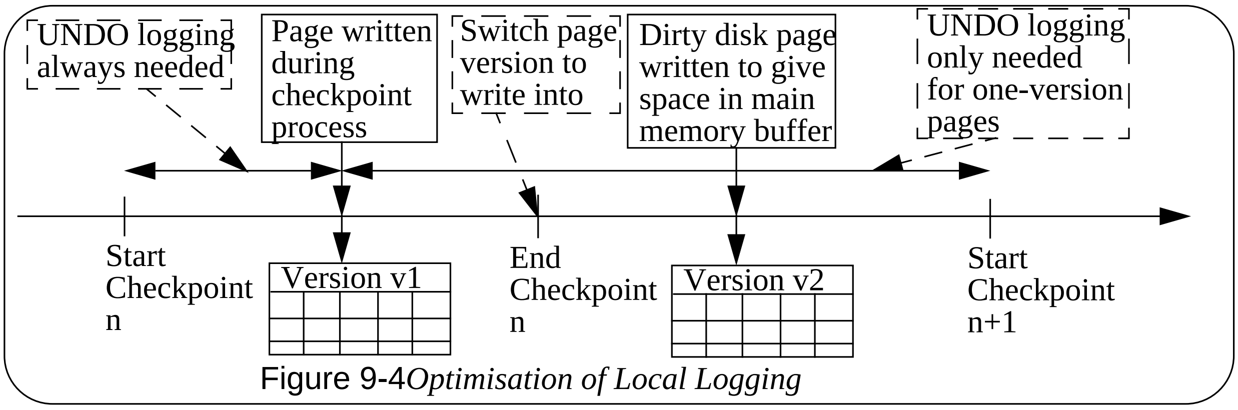
3) When all pages have been written to disk the local checkpoint is completed. An end checkpoint log record is written. The UNDO-log files are sent to disk and all other files involved in the local checkpoint. When it is certain that all information are stored on disk, the other nodes are informed of the completion. This is a part of the distribution information.

9.3.1 Local Logging Method

Disk-based pages are normally stored on disk and only when accessed for reading and writing are they kept in a main memory buffer. This means that the pages in the main memory buffer are allowed to be written to disk at any time, not only at checkpoints. Since in-place updating is used, the checkpointed pages can thereby be overwritten after a checkpoint. This means that UNDO logging must be performed all the time, not only in the checkpointing process.

Main memory based pages are only written to disk at checkpoints. They are always stored in main memory pages.

If there are two versions of the main memory pages on disk, the page writes of a checkpoint do not destroy the previous checkpoint. Thereby it is possible avoid a lot of UNDO logging. The UNDO logging is only needed between the start of the checkpoint process and the actual writing of the dirty page. Pages that are clean at the start of the checkpoint process are preferably taken care of immediately since these do not need to be written to disk in the checkpoint. When the checkpoint is finished, the pages are written to the other version of the database. This is shown in Figure 9-4.


UNDO logging always needed
Page written during checkpoint process
Switch page version to write into
Dirty disk page written to give space in main memory buffer
UNDO logging only needed for one-version pages
Start Checkpoint n
Version v1
End Checkpoint n
Version v2
Start Checkpoint n+1
Figure 9-4*Optimisation of Local Logging*

Main memory databases can have two (or even three) versions on disk without any great overhead in cost or performance. Therefore the above given method can be used.

9.3.2 nWAL Method

As described in a previous section of the thesis the system can be optimised by using nWAL. The idea of WAL is to avoid crash situations where a page has been written to disk but there is no information on how to UNDO changes on the page. This is important in the case where only one version of the pages exists on disk. If there are several versions of the pages on disk, then this is not needed. Then log records can be written to disk asynchronously to the page writes. Disk based data normally only have one version on disk and hence WAL is needed for the UNDO log records in our method of crash recovery.

nWAL means that instead of forcing the log to disk, the log is sent to a backup node and saved in main memory at that node. We call this backup node the Backup Log Node, the owner of the data we call the Primary Log Node. When the log is saved in two nodes, the log is defined as stored safely. Obviously writing the log records to disk would produce a lot of flushes of disk pages, the actual information that needs to be forced to disk is most of the time very small. Thereby using another node as safe storage would decrease the amount of information to flush and also speeds up the process since neighbour nodes are closer than a local disk (using modern communication techniques).

We therefore opt for the nWAL method of the local log of disk-based data. The fragment log is handled by the two-phase commit protocol in the system. Thereby we need to ensure that the local log is safely stored in both our own node and in the backup log node before writing main memory pages to disk. This means that we need to also design algorithms how to handle the failures of the primary log node and the backup log node.

We need to handle primary log node failures, and backup log node failures. The handling of these failures must also be publicly available to all nodes so that the consistency of the system is known to still exist.

9.3.3 Primary Log Node Failure

When the node containing the data fails, the backup log must ensure that the log is saved on disk. This is started immediately when the report on the primary node failure is received. When the log has been saved on disk, this is reported to all nodes. Thereby the primary log node can be restarted even if the backup log node fails as seen in Figure 9-5.

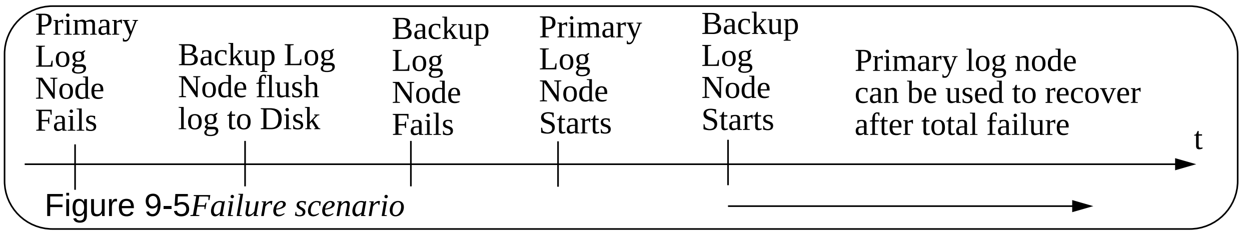

Primary Log Node Fails
Backup Log Node flush log to Disk
Backup Log Node Fails
Primary Log Node Starts
Backup Log Node Starts
Primary log node can be used to recover after total failure
t

Figure 9-5*Failure scenario*

9.3.4 Backup Log Node Failure

If the backup log node fails, the primary log node must use WAL (i.e. write the log to its local disk before the page is written to disk, instead of writing the log to the backup log node) until a new backup log node has been started. If checkpoint processing is currently ongoing, this processing is suspended as soon as the active page writes have been concluded. The primary log node also sends its log to disk as soon as possible. When this is finished and WAL processing has started, the primary log node reports this to all nodes.

A new backup log node is now started and as soon as this has been accomplished, the WAL processing can be concluded and nWAL processing starts again as normally. Checkpoint processing can also proceed again after starting the new backup log node.

9.3.5 Global System Checkpoint

An important part of preparation for recovery processing is to know what global checkpoint to restart the system from at a total failure. This global checkpoint must have been completed in the system to be restarted. Therefore the system must keep track of the last global checkpoint that is finished. A global checkpoint is finished when all transactions belonging to this and previous checkpoints have been completed and all log records of those transactions have been sent to disk. When this condition has been fulfilled the system can be certain to be able to restart the system from this checkpoint even in cases of failure of all nodes.

The system thereby keeps track of the maximum global checkpoint that is finished. When the coordinator decides that a new such maximum global checkpoint exists, the information is provided to all processor nodes and must be stored safely on disks. This global checkpoint is called the ***global system checkpoint***.

When a system is restarted, one has to find this maximum global checkpoint and start the system from this or an earlier global checkpoint.

Each processor should maintain the oldest transaction that has not yet been fully completed. Each time a transaction completes, a check is performed to see if any global checkpoints are fully completed in this processor node.

9.4 Related Work

The method we have described is related to the method described in section 4.9.2. Our contributions are mainly to provide all the necessary details to show crash recovery works in a situation with the combination of fuzzy checkpoints, location and replication independent logs, and local logs. The use of a local UNDO log is new and we also describe its use in some detail. We also show a solution of how global checkpoints can be unsynchronised in various nodes and still the logs can be location and replication independent. We also show how schema information is used during crash recovery.

The Informix-OnLine Dynamic Server uses also a similar scheme where an UNDO log is used to record all changes to database pages and a logical REDO/UNDO log is produced describing the operations on the database. At recovery the first pass uses the checkpoints and the physical UNDO log to create an action consistent database. Then a REDO pass followed by an UNDO pass is performed using the logical REDO/UNDO log [Hvasshovd96]. Our method uses the nWAL method instead and ideas presented in chapter 13 enable us to avoid storing UNDO information in the fragment log.

[Sullivan92] reports on a scheme used by Postgres where instead of using a log the database stores all versions of the tuples. It assumes the existence of persistent RAM memory to avoid frequent disk I/Os.

Our approach furthermore uses a local checkpoint and a global checkpoint process. These processes are not synchronised. No approach reported previously has used such a checkpointing to the knowledge of the author. It is reported in this thesis how log records before a local checkpoint do not destroy the consistency of the database in the crash recovery. Certain precautions are needed and these are reported here.

10 Schema Changes

This section will discuss how schema change operations can be performed without any major interference with user operations. A great variety of schema change operations are possible. We will describe some of them here.

In the literature there is a distinction between schema evolution and change propagation [PET95]. Schema evolution defines which schema changes are allowed and change propagation handles the propagation of those changes to the existing database information. This chapter deals with change propagation. Most of the research literature has focused on the problem of schema evolution

Our schema contains tables, indexes and attributes. Changes to these affect the local data storage and must therefore be prepared before the actual schema change transaction can be performed. The attributes can be methods which can be read-only or both readable and updatable. If the method is readable and updatable, then a separate method for reading and another for updating is needed. A method can only operate on attributes in the tuple which the method is a part of.

The schema also contains attribute constraints. Removing an attribute constraint is a pure update to the schema. Adding an attribute constraint means that all tuples in the table need to be checked and changed in accordance with the attribute constraints. This is not covered in this thesis. Table constraints in real-time database can be very difficult to handle. If the sum of all salaries of employees is not allowed to increase above a limit, the sum must be kept in a tuple (otherwise it has to be updated for every update in the table). This tuple would be a hot-spot which would endanger the availability of the data in the table. Adding table constraints is not covered in this thesis. Schema updates of triggers is not covered in this thesis. However, triggers are used as part of the solution to implement the schema changes. Adding a simple trigger and removing a simple trigger can be done by simply updating the schema information. However, complex trigger changes cause more problems. Neither views is covered in this thesis. Referential constraints do require a set of changes to the schema information that will be described.

Those tables that use a tuple identifier as tuple key are affected by the structure of the tuple identifier. In our case we include a table identifier in the tuple identifier. This makes it easy to follow a link to a new object. If it did not include the table identifier, then we would need a secondary index on tuple identifiers to find the table identifier of the tuple. The inclusion of a table identifier affects schema changes that change table structures, such as split and merge of tables.

We will also show how schema changes are propagated to a backup system when global replication is used.

10.1 Concepts of Schema Changes

In this section some concepts of schema changes will be introduced.

10.1.1 Soft/Hard Schema Changes

There are two possible ways to update the schema. The first is a ***soft schema change***. In this case the old transactions are executed using the old schema and new transactions are started using the new schema. The old and the new transactions can execute concurrently. In this case the concurrency problem using schema information is smaller.

The other option is ***hard schema change***. In this case the transactions on the old schema is executed until all have finished. During this time no transactions that are affected by the schema change can start. Then the schema is changed and new transactions are started using the new schema.

Most schema changes can be performed using the soft method. There are some changes that are incompatible with each other where the hard method must be used. As an example assume that the old schema contains one attribute with hours worked and another attribute with payment per hour. If the new schema replaces this information with an attribute of the total payment, this attribute is the product of the previous attributes. Obviously updates of the total payment cannot be converted into updates on the total time and payment per hour. Hence a hard schema change is needed.

Almost all changes in the schema information can be specified as adding or dropping schema entities. Tables, attributes, attribute methods, attribute constraints, table constraints, secondary indexes, foreign keys, views and stored procedures can all be added and dropped. Most changes can be specified as a number of adds and drops. These changes can obviously use soft schema change. Adding something does not conflict with earlier transactions that were not allowed to use it. Dropping something does not cause conflicts either. Earlier transactions can safely use the entities dropped as long as they still exist, new transactions will not be bothered by this. This means that changes to schemas are not serialised. Transactions that use old schema information can be committed after transactions that use new schema information. As shown this does not cause any problems, and only hard schema changes need serialisation.

Some changes cause logical conflicts, as shown above. The problem here is when changes are specified that use a function to map from the previous entities. If the inverse of this function does not exist, transactions before and after the change cannot execute concurrently. A definite border must exist between transactions before and after the change. This is implemented by hard schema changes. All transactions using the changed entities must be finished before the schema change can be performed. This problem is the same problem that exists for updates through views.

10.1.2 Simple/Complex Schema Change

Another dimension of schema changes are whether they are ***simple*** or ***complex***. A ***simple schema change*** can be executed as one transaction. Most simple schema changes only involves schema information. In this section we will always refer to simple schema changes as schema changes which only involve changes to schema information. ***Complex schema changes*** are long-running transactions. To execute a complex schema change as one transaction would block update operations for a long time. Most telecom applications require constant availability of data for updates. A typical example is HLR in mobile systems, these nodes keep track of mobile phones. Thus they need to be updated very often when the mobile phones move around. Hence we need to implement complex schema changes as a set of small transactions that together form a long-running transactions. A similar approach to on-line reorganisation was used in [Salzberg92].

Complex schema changes are a good example of a long-running transaction. We will use the concept of ***SAGAs*** [Garcia87] to control those long-running transactions. Complex schema changes fit very well into the SAGA model. The idea is to run the long-running transaction as a set of transaction. To rollback the long-running transaction a set of UNDO-transactions are performed. All schema changes are either of the type ADD or DROP something. This provides an UNDO-transaction for all schema changes.

The complex schema changes always incorporates adding a new schema entity which depends on the data in the tables of the old schema. To UNDO such a long-running transaction the new schema entity is dropped again. This could sometimes involve yet another long-running UNDO-transaction.

An important part of our solution to the problem with schema change is to use triggers along with the many small transactions. [Dayal90] describes in general how triggers and transactions can be combined to form long-running transactions. Our contribution in this thesis is to apply this to the problem of complex schema changes.

10.1.3 Simple/Complex Conversion Function

Yet another dimension of the problem with schema changes is whether the conversion functions are ***simple*** or ***complex***. This definition was introduced in [Ferrandina95]. A ***simple conversion function*** only uses local information in the tuple being accessed to create the converted data. ***Complex conversion function*** can also use information from other tuples to create the converted data.

Our method of performing complex schema changes allows both simple and complex conversion functions.

10.1.4 Summary

There are at least three dimensions of schema changes, soft/hard schema changes, simple/complex schema changes, simple/complex conversion functions. Simple schema changes which only involves changes to schema information do not use any conversion function at all and they are always soft schema changes.

Complex schema changes always involve changes to a number of tuples. These changes require a conversion function. The conversion function is, in our case, implemented both by transactions that scan the tables and by the triggers that ensure that the new schema entities contain the correct data. These conversion functions can be both simple and complex.

Whether a schema change is soft or hard depends on the conversion functions. If the conversion functions are invertible, then the schema change is soft. Otherwise it is hard.

10.2 Handling of Schema Changes

In this section we will describe some important items needed in performing schema changes.

10.2.1 Schema Version

Since concurrent transactions can use different schema versions the system must be able to handle more than one schema version. Each transaction uses a specific schema version and each schema change creates a new schema version. This version number must be tagged on all log records of the system and also provides a natural path to ensure that schema changes also occur on the backup system in the same order.

The version number is handled by the system coordinator. Only this node can increment the version number. If this node fails, a new system coordinator is assigned. The new coordinator must first check which the maximum known version number is and ensure that all nodes are consistent with each other. Then the new coordinator can start executing transactions to change the schema.

Using version numbers in this manner creates a bottleneck. This bottleneck should not cause any problems. Still tens or hundreds of schema changes can be performed per second. It is likely that even in the busiest period, the number of schema changes per minute should not exceed ten. Hence this bottleneck is not serious at all.

Actually there can only be one active schema change per table at the time. When a schema change starts changing a table definition, no other transaction can start changing the table definition concurrently. This is true also during the execution of a complex schema change (see Section 10.4). This means that a logical lock must be acquired on all tables involved in the schema change during the whole execution of the schema change.

10.2.2 Reading Schema Information by User Transactions

The user transactions read schema information in several stages and new versions can be created during this process. The user transaction should, however, continue to use the same schema version in all stages. The user transaction sets locks on all used schema entities. This ensures that hard schema changes are not started until all transactions that use the changed entities are finished.

If the transaction tries to access a schema entity that has been locked by a hard schema change, then the transaction will not be able to read all schema information using the same schema version in all stages. Thereby the transaction should be aborted. If desired and if possible the transaction can be restarted immediately afterwards.

Soft schema changes are used from the moment that the locks are released and the version counter has been incremented. The new schema are only used by new transactions. Old transactions will continue to use old schema versions.

10.2.3 Execution of Simple Schema Change

A simple schema change is performed similar to a two-phase commit. The first phase is a prepare phase where all actions in the schema change is distributed to all nodes. All software entities are prepared to handle new tables, attributes and indexes in this phase. This involves preparing all initial fragments of new tables and indexes. It involves allocating storage for all fragments and initialising the storage. New attributes are prepared, these are handled as dynamic attributes that do not exist if NULL. Hence no storage allocation is needed for new attributes until an attribute value is set on the new attribute in the tuple. Attributes that are methods need to have their software loaded and linked into the database software in all nodes.

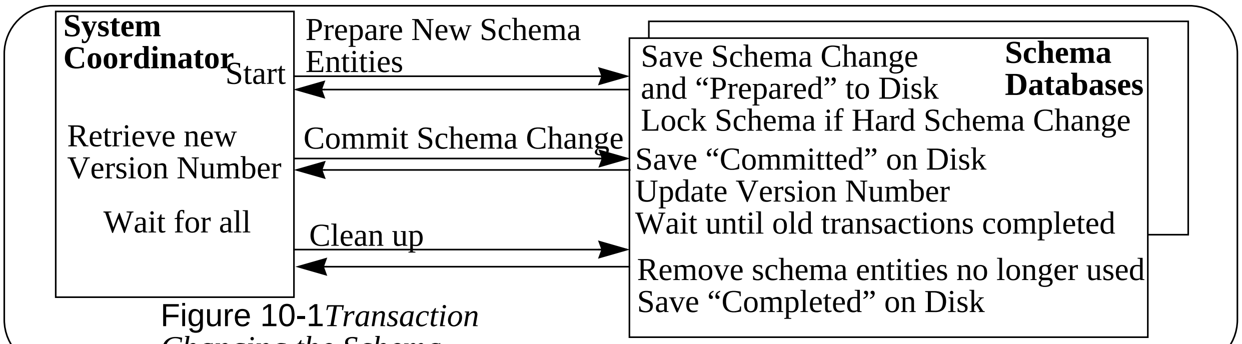

System Coordinator
Start
Retrieve new Version Number
Wait for all
Prepare New Schema Entities
Commit Schema Change
Clean up
Schema Databases
Save Schema Change and “Prepared” to Disk
Lock Schema if Hard Schema Change
Save “Committed” on Disk
Update Version Number
Wait until old transactions completed
Remove schema entities no longer used
Save “Completed” on Disk

Figure 10-1*Transaction Changing the Schema*

The first phase is also a prepare phase in a two-phase commit scheme, where locks on all entities of the schema to change are set. If the change is soft, then those locks are immediately acquired. If the schema change is hard, then the locks are not acquired until all transactions have released their locks on the schema information. When the locks have been acquired in all nodes then the schema information is updated. The update of the schema information is also logged to disk before sending the acknowledgement of the prepare message. Thereby it is easy to discover which schema version is the current one at restart. After acquiring the locks, logging to disk and performing the updates an acknowledge is sent to the system coordinator.

When the locks have been acquired, no transaction can start if it is a hard schema change. Also all transactions using the old schema must have been completed.

If a soft schema change is performed, then transactions can continue to execute using the old or the new schema information. Old and new transactions can execute concurrently.

The second phase is the commit phase. It starts when all nodes have acknowledged that locks are acquired and schema information have been updated. The first step of this phase is to acquire a version number of the schema change in the system coordinator. When this has been acquired, the commit together with the version number is sent to all nodes.

When the log message that describes the commit of the schema change has been flushed to disk in the nodes, the locks can be released and operations can start again. The version number is updated in the nodes before releasing the locks. Each node ensures that the schema versions are installed in the order prescribed by the version number.

The commit is not acknowledged until all transactions using old schema numbers are completed. This ensures that the system coordinator knows that the old schema is no longer used when it receives the commit from all nodes.

The third phase is a clean up phase, similar to the complete phase in two-phase commit; it starts when all nodes have acknowledged the commit phase. A clean up message is sent to all nodes. Tables, attributes and indexes that are no longer used are removed from all software entities. When all nodes have acknowledged this phase, then the transaction to change the schema is completed.

10.3 Simple Schema Changes

We will start by describing the simple schema changes that add and drop tables, attributes, foreign keys, and drop indexes. Without these operations it is difficult to maintain a database system.

10.3.1 Table Operations

Adding a table can be easily performed by preparing the fragments of the table and then performing the schema change. Old transactions will not know about the table and new transactions will and so there is no concurrency problem. The same reasoning holds for adding a view. Of course some attributes of the table should also be defined in the schema change.

Dropping a table is slightly more complex. This change can be performed in a soft manner. Of course the updates of the old transactions will be dropped when the table is dropped. New transactions will not be able to access the table any more. The same reasoning holds for dropping views.

10.3.2 Attribute Operations

Adding an attribute is also a simple operation that can be performed with a soft schema change. It could be slightly more complex to prepare it, however. If the attribute can be NULL, then it can easily be added. For this to be true, it is necessary to prepare all tuples with a number of unused NULL bits that can be used by new attributes on the tuple. These attributes will always specify NULL until the attribute is defined. Similar treatment could be used to add an attribute with default value as a simple schema change.

If it should be set to a value which is a function of old attributes, then a complex schema change is needed to add an attribute to all tuples of all fragments of the table. The schema change is not performed until all preparations are completed. Adding a method as a new attribute can also be performed easily by preparing all nodes to handle the method. We only consider methods that operate on one tuple here.

Dropping an attribute can be performed with a soft schema change. No preparations are necessary. The dropped attributes remain in the tuples until the tuples are reorganised. This can be performed either as a local process or as a part of a copy fragment protocol. The copy fragment protocol only transfers those attributes that are still used.

10.3.3 Index Operations

Dropping an index is a simple schema change that can be performed as a soft schema change. Often dropping an index means that a number of stored query execution plans must also be recompiled. This can be performed within the same schema change transaction. Doing it within the schema change ensures that real-time capabilities for user transactions are kept since the recompiling is done without any user waiting. Transactions to change the schema will of course take longer time. This is not a time-critical operation if user transactions are allowed during the schema change.

10.3.4 Foreign Key Operations

Adding a foreign key is made by adding attributes in the foreign table, an index on those attributes and adding the foreign key. These additions should be performed as one transaction to change the schema information. Then normal user transactions can perform any scan operations that are needed to make use of the foreign key. If the foreign keys needs to be set than the method to perform complex schema changes should be used.

The foreign key is represented by a set of attributes in the foreign table, and a special trigger on the referred table. Also an index on those attributes is normally kept to ensure that consistency checks and actions can be performed efficiently. When a foreign key is dropped this entails dropping the special trigger, the index and the attributes of the foreign key in one transaction that changes the schema.

Changing a foreign key means that the operation on deletion of the referred tuple is changed, which is performed as a soft schema change.

10.4 Complex Schema Changes

The next set of schema changes are complex and require that the change is performed as a set of schema changes and a set of user transactions. These are needed by most databases at times and are often performed by using user transactions written especially for the particular change needed.

Here we describe a general approach that can handle very complex schema changes as changes of attributes, adding a secondary index, splitting a table, merging a table and adding foreign keys. The split/merge can be performed both horisontally on tuples and vertically on attributes. Also complex combinations of schema changes can be used together. This thesis does not investigate any side effects these combinations can cause.

10.4.1 General Approach

All complex schema changes below use the same five phases.

The first phase is a schema change that adds a number of attributes, tables, foreign keys, triggers and indexes. It also adds a number of triggers and foreign keys needed to ensure that the new attributes, tables and indexes will be up-to-date when the second phase is finished. It does also add a number of triggers and foreign keys to ensure that a soft schema change is possible.

The second phase performs a scan processes on a number of tables. When this phase is finished both the new and the old tables and attributes are consistent. Thereby transactions on the new schema can start as soon as this phase is completed.

The third phase makes it possible to execute transactions on both old and new schema if the change is soft. Hard schema changes can only use test transactions which do not use schema parts that are not compatible. This phase contains a possible security check on the new schema. If the new transactions behave well, then the schema change will be committed. If there are many errors with the new schema, then the schema change will be aborted. This security check can be performed by only executing test transactions or by only executing a small percentage of the transactions using the new schema. After executing the test transactions to completion, a decision is taken whether to abort or commit the schema change.

A schema change is a management activity handled by some database administrator. The database administrator should also be responsible in handling these tests of the new schema. In telecommunications there is usually some personnel responsible for operation and maintenance who has this role.

If the decision is to abort the schema change, then all new tables, attributes, foreign keys and triggers are removed and the old schema is kept. No more transactions on the new schema are allowed. If the decision is to commit the schema change, then all transactions will start to use the new schema. The currently executing old transactions are executed until they are finished. Transactions started from here use the new schema.

When the old transactions are completed, then the last phase removes the triggers needed by the scan processes, removes any attributes used by the change process and also removes attributes, tables, indexes and foreign keys from the old schema that are not part of the new schema. The control flow of a complex schema change is shown in the figure below.

Phase 3 is different for hard schema changes. It waits until old transactions are completed. No transactions are started during this wait. After this the schema is changed to use the new schema and then transactions can be started. These transactions can only use the new schema. There is no possibility to restore the old schema other than through a system restart.

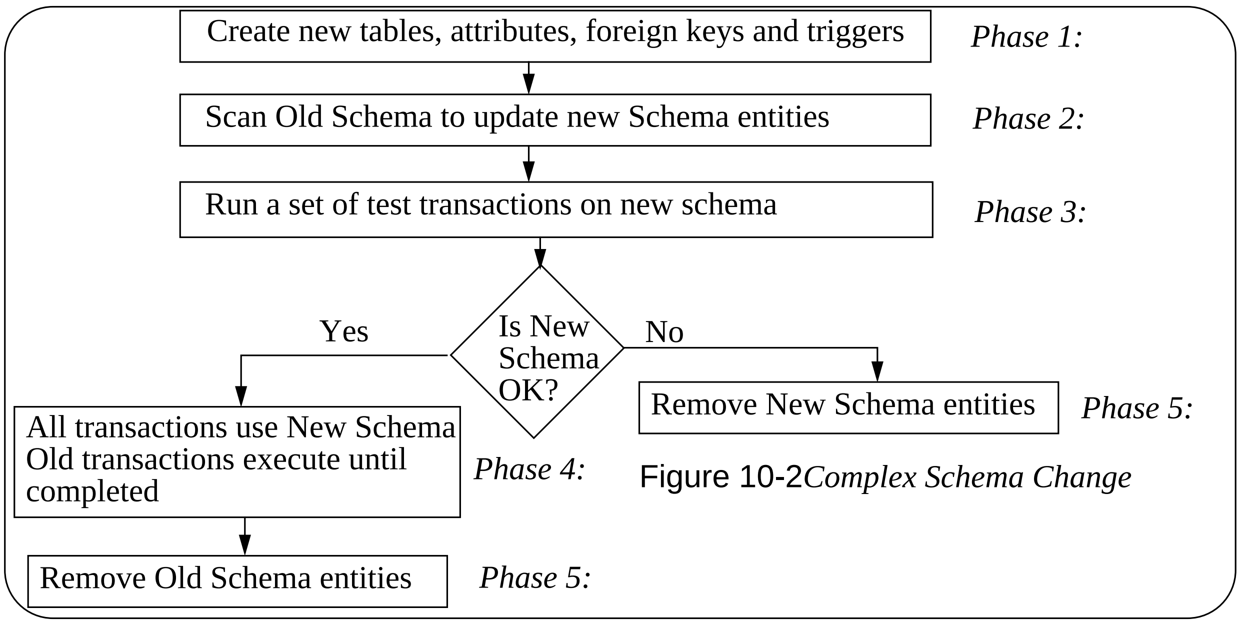

Create new tables, attributes, foreign keys and triggers
Phase 1:
Scan Old Schema to update new Schema entities
Phase 2:
Run a set of test transactions on new schema
Phase 3:
Is New Schema OK?
Yes
No
All transactions use New Schema Old transactions execute until completed
Phase 4:
Remove New Schema entities
Phase 5:
Remove Old Schema entities
Phase 5:

Figure 10-2*Complex Schema Change*

The complex schema changes are examples of long-running transactions. Only when all of those transactions have been performed is the complex schema change actually finished. To be able to abort a complex schema change the system must maintain UNDO-information. This is necessary if the decision is to abort the schema change and is also necessary if the primary system fails during the change. Then the complex schema change should be aborted. Sometimes work is also needed after completion of the schema change in a clean up phase. REDO-information is needed to ensure that this work is actually performed even in cases of failures. The REDO/UNDO-information must be maintained across transactions and must therefore be stored in a reliable manner. We use a special ***Saga Table*** to store the REDO/UNDO-information. The Saga Table is also replicated in the backup system if global replication is used. We will show the use of the Saga Table in the schema changes below.

Normally UNDO entries are inserted in the Saga Table during phase one, REDO entries are inserted in phase four and the entries are deleted in phase five.

Since changing the schema also affects the applications on top of the database, the schema changes must to some extent be synchronised with software changes. One natural path to take then is to perform the software change in parallel with the schema change. Then new transactions using the new schema are generated by the new software. The decision to continue the schema change is then affected by the quality of the new software.

Another option is to ensure that the schema change is backwards compatible. This means that all transactions using the old schema can still continue. In this case the old transactions will operate on a view of the database schema instead of directly on the tables of the database.

Using soft schema changes must be combined with a check that avoids circular triggers. Otherwise the update on the original table will trigger an update on the new table. This update will in its turn trigger another update on the original table. This must be avoided by keeping track of who originated the update.

10.4.2 Change Attributes

In this case it depends on type of change whether the change is soft or hard. We will here consider only changes that occur within one table. There are two situations to consider:

i) Adding a new attribute that is derived from old attributes.

ii) Changing attributes by adding some new attributes and at the same time dropping old attributes.

We will show how to perform a change of attributes and adding a derived attribute can be solved in a similar way.

Phase 1: The first step is to add the new attributes and also to add a method that operates on the old attributes to update the new attributes. A trigger is installed that on update of the old attributes also performs an update of the new attributes through the added method. A soft schema change is possible if the inverse of the method to derive the new attributes exists. If this method exists then it is added together with a trigger that updates the old attributes through the inverse method when new attributes are changed.

This is performed as one transaction that changes the schema information. In this transaction the Saga Table is also updated with the UNDO-information.

Phase 2: Scan all tuples and update new attributes through the added methods. During these operations the triggers are active. Each update of a tuple by the scan process is treated as a normal user transaction.

Phase 3: After scanning all tuples in all fragments of the table, the new attributes will exist in all tuples of the table. Here the method has two directions dependent on if the change is hard or soft. If an inverse method exists, then old and new transactions can execute on the same tuple in any order without causing problems, hence the schema change can be soft. In this case the old attributes could even be kept if the methods and their triggers are kept. If no inverse method exists, then the schema change must be hard.

The conclusion of this complex schema change is performed as described in the previous section.

10.4.3 Add Index

Adding a secondary index on an already existing table uses only three phases since the index is new and not used by any old transactions. Thereby as soon as the scan process is completed the schema can be updated. Creating a secondary index on an empty table is compressed into one phase.

Phase 1: This phase informs all nodes of the new secondary index through a schema change. In this phase the secondary index has the state specifying that it is in the building phase. In this state the secondary index cannot be read.

A special attribute is added to the table that describes whether a tuple has updated the secondary index yet. If the attribute is defined, the tuple has updated the secondary index and, if not defined, it has not. A trigger on the table is specified. When a transaction updates the tuple, the secondary index should be updated if the special attribute is defined. The trigger should be specified such that inserts always update the secondary index and define this attribute. An entry is inserted into the Saga Table that specifies the removal of the special attribute, removal of the trigger and removal of the index in case of aborting the transaction to add an index.

Phase 2: This phase performs the actual creation. This process is actually not started until all old transactions have completed to ensure that no transaction update the index attribute during the scan process without the trigger being fired. In this process all tuples are scanned and an entry in the secondary index is created if the special attribute was not defined.

Phase 3: When the scan processes have been completed, the schema is updated to reflect that the secondary index is now also readable. This schema change is also soft. The entry in the Saga Table is removed and so is the special attribute that was defined in phase one. A normal secondary index has a special trigger on updates of the attributes of the secondary index.

Error handling is simple, if any node fails during the scan process it simply restarts the scan. The special attribute will specify whether the change on the secondary index was performed. If less recovery is needed and instead a faster creation process, then the special attribute could be specified as volatile (meaning that it is not involved in any recovery operations, opposite to durable). This means that any failures would necessitate a restart of the creation of the secondary index.

There is no problem in this algorithm if the index is over several tables. Then all tables are scanned and the same failure handling can be used.

10.4.4 Horisontal Split Table

There are two types of splits possible, the first splits a table horisontally and separates the tuples into a number of tables dependent on some condition. The second splits the table such that each tuple is split, the table is vertically split. We will study both the operations to split a table.

The attributes of the new tables could be the same as before the split or combined with other changes that add or drop attributes. Each tuple is sent to one and only one of the new tables. The split is complicated by foreign keys and secondary indexes. Also updates during the scan process could change the tuple to belong to another of the new tables. This must be treated in the preparation phases by triggers. The split is performed in the normal five phases.

Phase 1: In this phase the new tables are created but they are not usable by normal user transactions. Secondary indexes on the new tables are added. If there are foreign keys in foreign tables that refer to the original table, these foreign keys must be updated to reflect the split. This is performed by adding a new foreign key on the foreign tables which should have the same characteristics as the old foreign key. The original table is also equipped with a foreign key that refers to the tuple that has been created in the new tables. This foreign key specifies that the tuple in the new table shall be deleted when the tuple in the old table is deleted. In Figure 10-3 the new schema entities and their usage is shown. At the start of the split process all new attributes are NULL, the new secondary indexes are empty and the new tables are empty. Also the index of the new foreign key is empty since all foreign keys are equal to NULL at the start.

In this phase also four types of triggers are introduced. These triggers ensure that the foreign keys, secondary indexes and the new tables are consistent after the scan process.

The first trigger is executed when foreign keys in the foreign tables are updated. When a foreign key is updated, the referred tuple is checked to also be able to update the new foreign key. If the new foreign key on the original table contains a NULL value, then the scan process has not reached this tuple and no special action is needed. Otherwise the value of the foreign key in the original table is copied to the new foreign key in the foreign table.

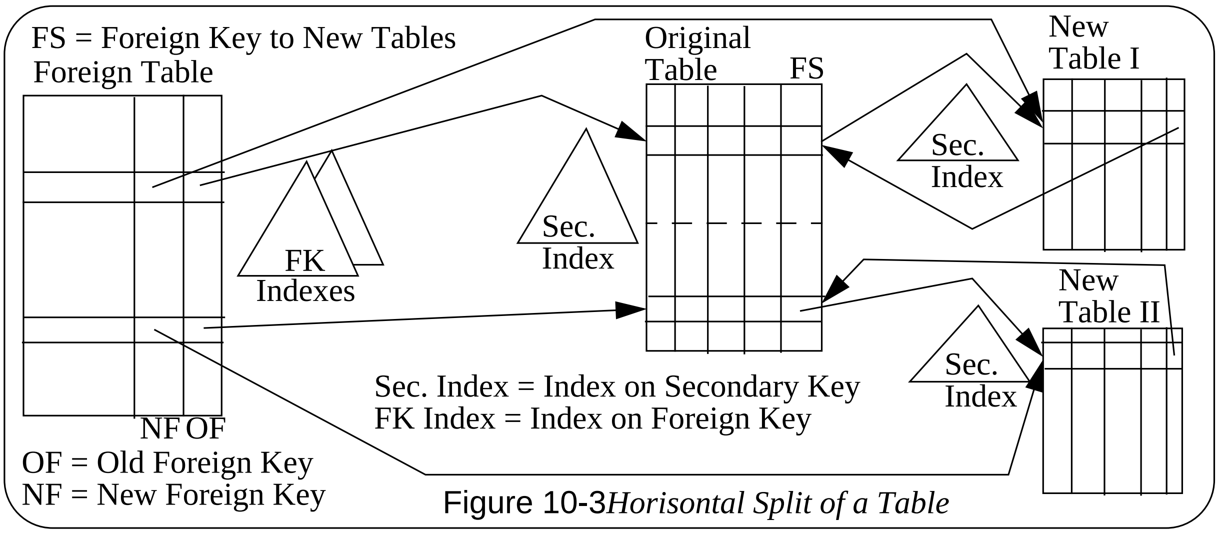


Figure 10-3*Horisontal Split of a Table*

The second trigger is executed when the new foreign key in the original table is updated. This update means that all new foreign keys in the foreign tables should also be updated. The tuple key of the original tuple is used to find those foreign keys that needs to be updated. This trigger is invoked by the scan process, by inserts and also updates that changes the new table of a tuple.

The third trigger ensures that updates to the table are also performed on the new tables. This means that when updating the original table, the update is also sent to the affected new table. The transaction coordinator must sometimes first read the original table to deduce which of the new tables are affected by the write operation. These reads should set a write lock to ensure proper concurrency when parallel updates are started on secondary indexes, new tables, foreign keys and so forth. The updates of the new table also updates the secondary indexes of the new table. The trigger should be specified such that inserts also inserts into the new table and updates the foreign key of the original table.

In some cases the update actually affects two new tables when an update of a tuple moves the tuple to another new table. An example could be that the table is split on the salary attribute. Persons with salary above 100,000 will be in table New2 and those below 100,000 will be in table New1. If a person receives a salray increase from 90,000 to 110,00 his record will be moved from New1 to New2.

In this case the update leads to an update of the original table, a delete in one new table, and an insert in another new table as shown in Figure 10-4 (only primary node is shown in the figure, the replication is not important in this problem). Also the new foreign key in the original table needs to be updated to reflect that the tuple has moved to another new table.

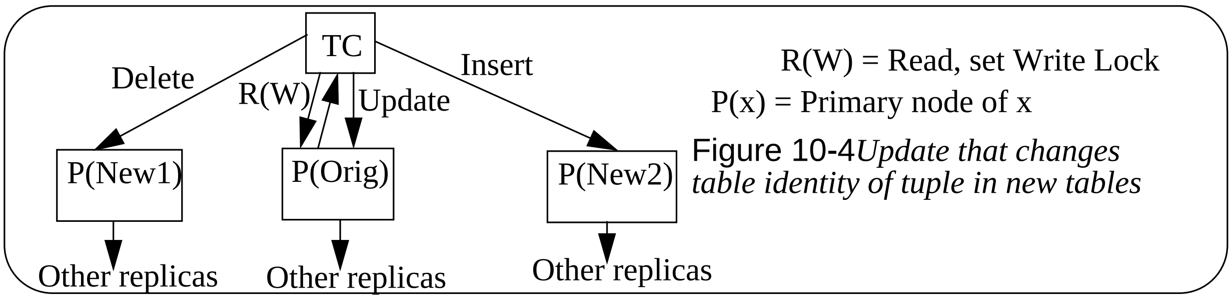


Figure 10-4*Update that changes table identity of tuple in new tables*

Finally the fourth trigger is used by transactions using the new tables. These updates must also be propagated to the original table. This is performed by using the foreign key in the new table that refers to the original tuple in the original table. Inserts into the new table should also be performed in the original table.

All of these changes are made as one transaction to change the schema information and an entry in the Saga Table is also created that represents the UNDO-information. The schema change is soft. This information specifies the attributes, tables, foreign keys and secondary indexes that should be removed if the complex schema change is aborted.

Phase 2: When the triggers, the new indexes and the new foreign keys have been inserted, then the scan processes can start. It scans the original table and inserts the tuples into the new tables. When a tuple is read from the original table and inserted into the new table, then the foreign key in the original table is updated to reflect the connection between the original tuple and the new tuple. If the new foreign key of the original table was not NULL, then some transaction has already performed the work needed by the scan process and the transaction need not be performed. The update of the new foreign key in the original table fires the trigger that updates new foreign keys in foreign tables. Secondary indexes of the new tables are also updated when tuples are inserted, updated and deleted in the new tables.

When the scan process is completed, the new tables exist, the new foreign keys are updated to reflect the new tables, and secondary indexes on the new tables exist. The triggers will ensure that the original table and the new tables are kept consistent. Now the completion of the schema change follows the description in section 10.4.1.

Error handling is such that if any of the primary replicas fails during the scan processes, then the scan process is restarted. The foreign keys in the original table ensures that it is not necessary to REDO work performed before the node failure.

In some cases it might be desirable that the original secondary index should be kept on all the resulting tables instead of creating a new for each new table. This does not create any special problems; this secondary index must, however, be created as the others since the entries will refer to several tables. This index will then be a multi-table index where each entry contains the tuple key plus a table identifier.

10.4.5 Vertical Split Table

A vertical split is actually simpler than a horisontal split. Foreign keys that now refer to the table should refer to only one of the tables. This table should keep its table identifier and then there are no problems with foreign tables. Hence it is not necessary to create this table, it is only necessary to remove some attributes at the end of the complex schema change.

The next case that caused problems was secondary indexes. When a table is split the secondary index normally follows only one of the new tables. Hence it is only necessary to update the table identifier of the secondary index or of the entries in the secondary index. An easy conversion of the table identifier is thereby needed. If several tables need the same secondary index, then another secondary index with the same attributes is created that can follow the other table.

If this approach is not feasible it is also possible with a more complex schema change using the same basic ideas with triggers, transaction and use of foreign keys.

The work is performed in the normal five phases as described in section 10.4.1.

Phase 1: The new tables and their attributes are added to the schema information. A foreign key is also needed for each of the new tables. These are set to NULL to start with. When the value is not NULL then the new tuple has been created.

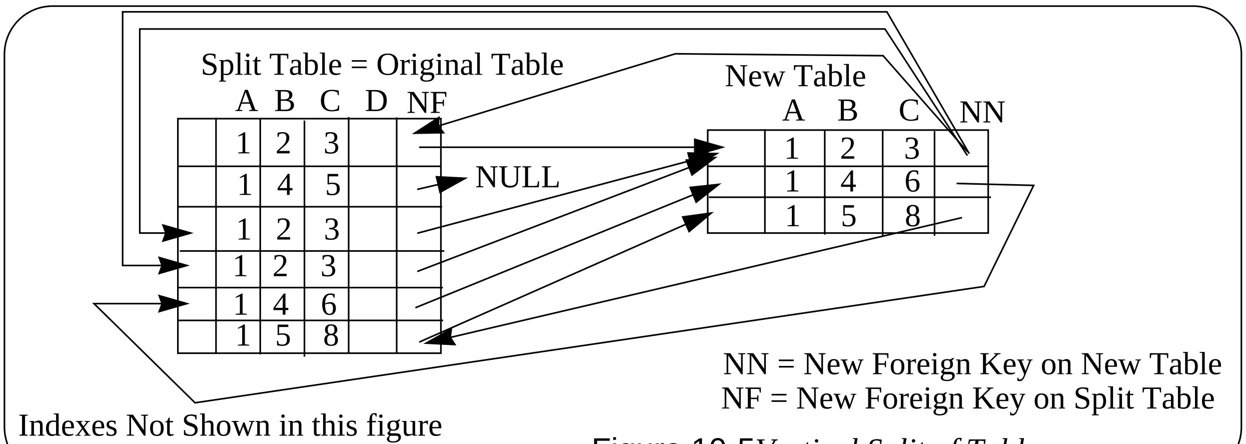

Split Table = Original Table
A B C D NF
1 2 3
1 4 5
1 2 3
1 2 3
1 4 6
1 5 8
NULL
New Table
A B C NN
1 2 3
1 4 6
1 5 8
NN = New Foreign Key on New Table
NF = New Foreign Key on Split Table
Indexes Not Shown in this figure

Figure 10-5*Vertical Split of Table*

A trigger is needed to ensure that updates on the original table are also reflected in the new tables. If the foreign keys in the original table is NULL, then no update is needed in the new tables since the tuple has not been created yet. If a foreign key exists, then all updates on the original table must also be reflected in the new tables. At insertion the new tables are also updated and the foreign keys are updated through the trigger. The foreign key must also be defined to delete the new tuples when the tuple in the old table is deleted.

A similar trigger and foreign key is also placed in the new table to ensure that transactions using the new table also update the original table.

A complication is possible if the original table contains tuples that create duplicates in the new table. In this case updates of the new table must update all tuples which refer to the new tuple.

If a transaction on the original table updates attributes such that the new tuple is also affected, this must be done with care. If others refer to the same tuple, then the tuple cannot be deleted, only the reference to the tuple is removed. Thus the update creates a new tuple in the new table. This tuple might however already exist and, if so, only a reference to this tuple is created.

An entry in the Saga Table is created that ensures that the added attributes, foreign keys, tables and indexes are removed if the schema change is not to be performed.

Phase 2: This phase contains a scan of the old table that creates the tuple copies in the new tables and updates the foreign keys in the old table. When inserting into new tables during the scan process, the tuple might already exist in the new table. The reason for the split could be that a third normal form of the tables is needed and then replicated information is removed from the new tables as shown in Figure 10-5. Thus inserts that find duplicates should not insert any tuple in the new tables. They should only update the foreign key to refer to the duplicate tuple. Hence duplicates are removed in the scan process.

The last phase performs the actual schema change to be used by user transactions as described in section 10.4.1.

10.4.6 Horisontal Merge of Tables

Merging a number of tables horisontally is similar to a horisontal split. The major complexity comes from the fact that two tuples from the original tables might have an identical tuple key. This problem occurs when the tuple key is a primary key. When the tuple key is a tuple identifier it contains a table identifier. Thus two tuples in two different tables cannot be equal.

Thereby it can be necessary to include some extra information in the tuple key in the new table. The description assumes that this problem exists. If the tuple key from the original tables is usable without risk of duplication (i.e. using a tuple identifier), then the problem is simpler. In this case the tuple key of the original tables and the new table also functions as a foreign key between the original tables and the new table. Those foreign keys that refer to only one of the original tables need only a change of the table identifier that the foreign key refers to. If they refer to both and the foreign key includes some table identifier, then this needs to be removed or changed during the merge as described below.

Phase 1: The new table is created with its secondary indexes and its attributes. A foreign key to the original tables is needed. The original tables need a foreign key to the new table. Foreign tables referring to the original tables must also be equipped with a new foreign key that refers to the new table.

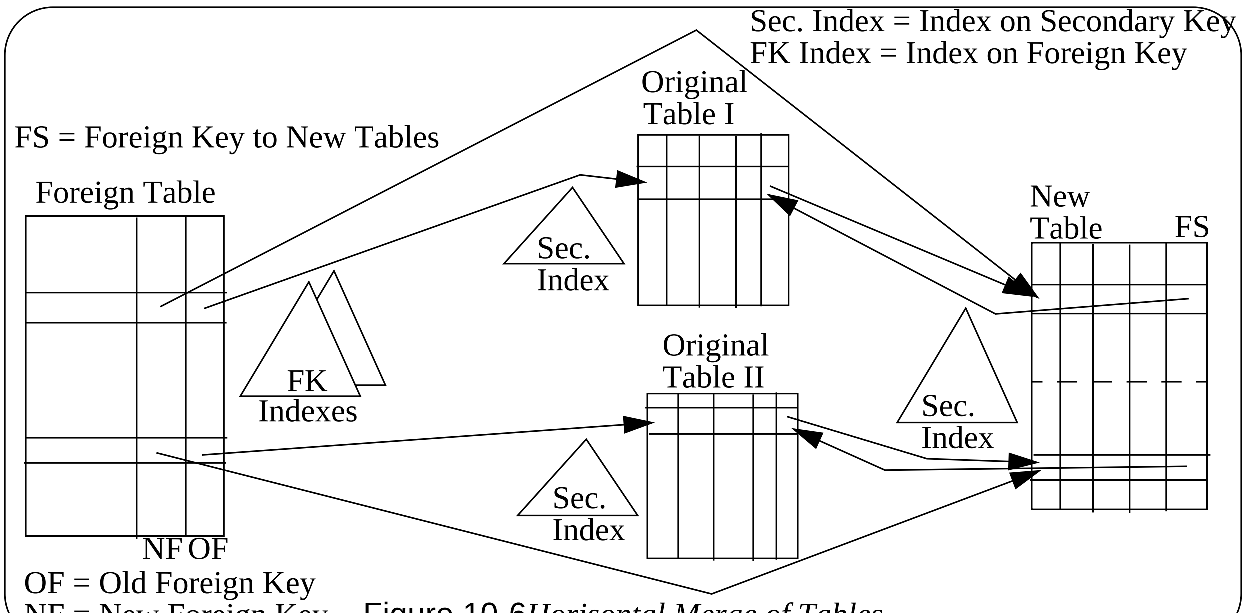


Figure 10-6*Horisontal Merge of Tables*

A trigger is set on the original tables such that updates on them are also propagated to the new table if the foreign key of the original table is not NULL. Insert transactions always update both the original table and the new table through this trigger to simplify the scan process.

A trigger is also set on updates of the foreign key attribute in the foreign tables. This trigger follows the updated foreign key to the original table and reads the new foreign key to update the new foreign key in the foreign table.

A trigger is set on updates of the new table to ensure that the original table is kept consistent with the new table.

As usual an entry in the Saga Table is also created that provides UNDO-information to remove all new foreign keys, tables, indexes and attributes.

Phase 2: The old tables are scanned and the tuples are copied to the new table if the foreign key is not NULL.

Then the normal completion is performed as described in section 10.4.1.

10.4.7 Vertical Merge of Tables

A vertical merge of a number of tables can be seen as adding a number of attributes to an already existing table, called Merge Table, and derive the attribute values from an old table, called Originating Table. The Originating Table can then either be dropped after the merge or kept. To perform the merge the tables must either have the same tuple key or the Merge Table must have a foreign key to the Originating Table. The situation with a common tuple key can actually be seen as a special case of a foreign key. This solution also works if there are more than one Originating Table.

Foreign keys to the Merge Table need not be handled at all since the table remains and the tuples still have the same tuple key. Foreign keys to the Originating Table can be handled in several ways. If the Originating Table is not dropped after the merge, then no action at all is needed. If the Originating Table is dropped, then also the foreign keys might be dropped. The foreign keys could also be moved to the Merge Table. In the description below we assume that the Originating Table is kept and that no action is needed on foreign keys referring to this table.

The merge is performed in the three phases since the change only installs new attributes and does not remove any attributes. Thereby as soon as the scan process is completed, new transactions can start using the new attributes.

Phase 1: Add the new attributes to the Merge Table. Also, a similar attribute as used in adding an index is used. This attribute is NULL if the tuple has been copied from the old table, otherwise it is NOT NULL without any specific value. A trigger on updates of the Originating Table is specified. Updates of the Originating Table thereby affect all tuples in the Merge Table that refer to the updated tuple. During such an update also the attribute which specifies whether the copy has been performed is set. Deletion of a tuple in the Originating Table should set new attribute values to NULL in all the tuples that referred to the deleted tuple.

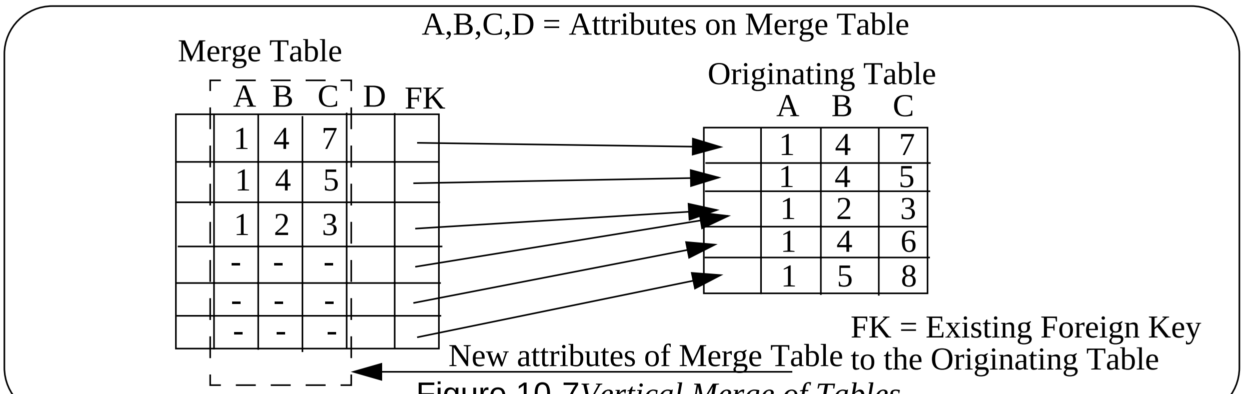

A,B,C,D = Attributes on Merge Table
Merge Table
A B C D FK
1 4 7
1 4 5
1 2 3
- - -
- - -
- - -
Originating Table
A B C
1 4 7
1 4 5
1 2 3
1 4 6
1 5 8
FK = Existing Foreign Key to the Originating Table
New attributes of Merge Table

Figure 10-7 *Vertical Merge of Tables*

Phase 2: The Merge Table is scanned and in each tuple the new attributes are updated by using the foreign key to access the Originating Table. Each tuple update is one transaction.

Phase 3: When the scan is completed, the schema change can be performed. The schema change can be made soft. The schema change includes dropping if necessary Originating Table, dropping the triggers of phase one and so forth. It also includes deleting the entry in the Saga Table.

10.5 Schema Changes in the Backup System

The only schema changes that actually affect the backup system are changes to tables and attributes. These affect the backup system since new attributes must be represented in the storage structures and new tables must be handled both with a number of new log channels and new storage structures. The rest of the schema changes are only necessary for updating the schema information. The schema information is not used to perform any checks in the backup system, if not requested to do so by the application. These checks are performed by the primary system. It is, however, necessary to ensure that updates to schema information are serialised with updates to the application information.

10.5.1 Schema Information in the Backup System

It is actually desirable to have two types of schema information in the backup system. Activated schema information and deactivated schema information. Deactivated schema information is activated in case the backup system is promoted to primary system. Actually activated schema information could also be marked to indicate that it is only used in a backup system. This type of schema information is deactivated when the backup system is promoted to primary system.

An example of deactivated schema information could be all types of constraints that are performed in the primary system and not deemed necessary to perform in the backup system as well. A materialised view that represents statistics used by management systems is an example of information that could be marked as only active for the backup system. The idea is to move processing power from the primary system to the backup system. Care must, however, be taken here since the backup system is not allowed to abort transactions in failure cases. The only possibility to affect the primary system is to report failures that cause the promotion of the backup system to primary system.

The primary system could also contain schema information that is not replicated in the backup system. An example of this is the special attribute used when creating a secondary index. Secondary indexes are independent of each other in different systems. Each system defines its own set of indexes. Normally they would have the same because of requirements by the application. But from a database point of view it is not necessary for them to be equal.

10.5.2 Log Channel of Schema Information

To send transactions to the backup system a log channel is needed for all fragments. We define the schema information as one fragment. The version number is then used as the ticket number. All write transactions, including their read operations should be sent in this log channel too.

Instead each transaction specifies which schema version it has read. The log records also specify the attributes used. All log records are sent in a log channel that specifies the fragment and the table of the log records, hence the tables used are defined. Using the schema version and the attribute identity the actual attribute to update can be derived. The problem is that attribute identifiers can be reused after deletion of an attribute. Since log records only contain attribute identity and no attribute name the schema version must also be specified to deduce which attributes is to be updated.

Actually by ensuring that attribute identities cannot be reused until the previous user of the attribute identity is dropped the only schema information needed is the information that specifies how attributes are stored.

10.5.3 Simple Schema Changes

The schema changes that are important in the backup system are those adding/dropping tables and attributes. These affect the execution in the backup system. The other operations that affect schema information are only performed to replicate schema information so that the backup system can be used as primary system if needed. Much of this information is deactivated in the backup system until promotion to primary system.

When a table is added, all fragments need to set up a log channel. After this log records can start to be shipped to the backup system. Actually these log channels should be set up in phase one, in the preparatory phase of the schema change.

Dropping of a table and its log channels is not performed until all transactions using it have completed in all systems.

Attributes are added as part of the preparatory phase and then when the transaction that changes the schema is committed the transaction is sent in the log channel of the schema information.

Information about dropping of an attribute is sent on the log channel of the schema information. The actual dropping of the attribute is not performed until the clean up phase. When all software entities and all backup systems have reported the removal of the attribute, then the attribute identity can be reused.

10.5.4 Complex Schema Changes

Complex changes use the Saga Table; the Saga Table is part of the schema information and is also replicated in the backup system. The Saga Table is used at promotion to primary system to UNDO all changes that were not completed.

10.6 Remarks

The complex schema changes have one drawback and that is consumption of computation resources such as memory and processing resources. To avoid this is very difficult. One possible method could be to change one fragment of a table at a time during the complex schema change. This can only be used if both the old and the new schema can be used with only one copy of the data. Often this is the case. Of course this also makes the change more complicated and it has impact on many more low-level details of the database software.

Another method is to install extra hardware when a schema change is needed. Thus it is not necessary to dimension the system for schema changes. Equipment can be temporarily used when performing schema changes. To achieve this it must be possible to perform on-line reorganisation of the system without disturbance.

A DBMS which uses the methods described in thesis should have a very efficient implementation of triggers.

10.7 Related Work

Most research articles on schema changes are dealing with schema evolution in object-oriented databases (e.g. [PET95]).

The concept of soft schema change has not been found in any paper by the author of this thesis. It is common to treat updates to schema information as a normal transaction. This means that only one schema version can be active at a time, thus a hard schema change is achieved. A similar concept, soft software change, is found in [NILS92]. Software changes which involves changes of data structures are very similar in nature to schema changes. Thus many ideas can be reused in the area of schema changes.

In [Salzberg92] a number of small transactions are used to change the Record ID of tuples. If the Record ID contains a physical page ID, then it becomes very difficult to reorganise the database. By using a transaction per record it was possible to perform changes of Record ID in a seamless fashion. [Dayal90] describes in general how triggers and transactions can be combined to perform long-running transactions. In this thesis we have applied those approaches to solve the problem with complex schema changes, i.e. schema changes which require a database reorganisation.

[Mohan92a] uses a different method to create indexes in an on-line system. They use a log-based approach. Triggers combined with transactions and log-based approaches are two techniques that are the most common techniques to perform on-line reorganisations. In this thesis we have almost exclusively used triggers and transactions. The reason is its simplicity. The solutions becomes very elegant and easy to understand.

Combining soft schema changes with network redundancy is even more a new area. In [KING91] it is briefly mentioned that schema changes will use a separate log channel between the primary system and the backup system. It is implicit in this paper that only hard schema changes are used since a normal transaction is used to update schema information.

Our method have used the SAGA concept [Garcia87] to handle schema changes in conjunction with system crashes and network redundancy. A SAGA table is used to find UNDO and REDO-information of ongoing complex schema changes after a crash of a system. UNDO of complex schema changes can often be performed by simply dropping the tables and attributes that were created in the first phase. Thus UNDO of complex schema changes is considerably simpler than the general SAGAs.

[Ferrandina95] describes the implementation of schema changes in O_2. Some complex schema changes can be handled by using a combination of user-defined conversion functions and migration functions. The schema change is performed in a different order than in this thesis. It begins by performing the schema change. The actual change of the objects is performed when an object actually access an object. The possibility to perform soft schema changes is not mentioned and it is therefore assumed that only hard schema changes are supported. In our method the conversion functions are applied before the data becomes accessible to the applications. This provides the possibility to execute test transactions before the schema change is generally accessible to all applications. Our method also briefly describes how schema change is integrated with software change.

[Lerner90][Lerner94] treats complex schema changes. It develops a special-purpose language that specifies the propagation of data from the old schema to the new schema. In [Clamen94] a distinction is made between change propagation through conversion and through emulation. Conversion actually converts the old objects into new objects and emulation provides the new or old schema through an interface that converts to the proper schema. The method in this thesis makes it possible to use both of those approaches. If software can be changed simultaneously with the schema, then a conversion is most appropriate. If old software needs to function even after changes, then emulation is more appropriate. Emulation in this approach would mean that the old schema (or new schema) is provided through views, methods and stored procedures.

[Kim88] treats versions of schemas. Our approach provides the possibility to use several schema versions concurrently. In this respect it is similar to [Kim88]. [Kim88] does, however, only treat simple schema changes (adding/removing attributes, adding/removing classes/superclasses/sublasses). There is no discussion on conversion functions in [Kim88].

Most databases support simple schema changes that add/remove tables, attributes and methods.

DBS is a semi-relational database used in AXE [DBS91]. It supports on-line hard schema changes which are rather complex. It also does so by providing a special-purpose language, FUSQL, that reads data from the old schema and transfers it to the new schema.

Very few reports have been written concerning how to perform complex schema changes without stopping the system. Many reports have dealt with the managerial problems in maintaining schemas and which schema evolutions that are allowed. Most of the work on on-line schema changes has been performed in implementations where this is essential and these systems are rarely reported in the literature.

The solution in this section has received inspiration from work in software change management. Similar ideas to these can be found in [NILS92].

This section contains results rarely reported of very complex schema changes. There are many items that need to be developed further, such as a complete check of all possible schema evolutions if they can be supported. It should be possible to specify the algorithms in this section in a rather simple fashion since they only deal with changes on tables, attributes, foreign keys and indexes. The resulting implementation of the algorithms can then be left to the system.

Creating indexes on-line is discussed in a number of reports [IBM96][Zou96][Hvasshovd96] and many more. Some of those approaches handle the creation of an index without any outage. Our approach does so by using the concept derived from [CHAMB92] with some minor modifications.

10.8 Conclusion

A number of protocols to change schemas have been developed. It is seen that the same general approach can handle very complex schema changes. This approach therefore has a good chance of being usable for any type of schema change. This solution has been derived by making use of a set of related simple schema changes together with a set of scan processes that create the new tables, attributes, indexes and foreign keys according to the new schema.

There are many more possible changes that could be made. An important issue to consider is when combining these different schema changes in the same schema change transaction. There might be side-effects in these combinations that are not easily predicted. Therefore there are many open research issues in this area.

11 A new variant of LH*, LH^3

In this section we will describe ***LH^3***, an extension of LH*LH as described in [KARL96]. LH*LH is an extension of LH* described in [Litwin93a] and LH* is a distributed version of LH [Litwin80]. The main difference is the treatment of the local linear hashing function. Hence this idea could also be used to improve the data structure of linear hashing. Extensions of the original idea on ***LH^3*** has been developed in cooperation with Ataullah Dabaghi who has reported more details of the implementation of ***LH^3*** in [ATTA98].

The data structure could also be extended with storage of tuples in the index. Using that approach it would be possible to find tuples with one disk read. Our main focus is, however, to decrease the number of cache misses when using an index in main memory. Accessing storage structures is very likely to cause a cache miss at the first access. The cost of cache misses are equivalent to the cost of hundreds of assembler instructions without cache misses in next generation processors. Thus it is important to decrease the number of cache misses as well as the number of disk I/Os.

Before describing the extension of LH*LH we will briefly describe LH, LH* and LH*LH. LH is a hash algorithm which provides a dynamic growth and decline without any hiccups. This is achieved by performing the growth in small steps rather than as one big step. A hash function is used on a key which provides a hash value. This hash value should now be mapped to a bucket. If the number of buckets is 390, then we could use the following algorithm to determine the bucket.

A bitwise AND of the hash value and 255 (2^8-1), is performed. Hence we only keep the eight least significant bits of the hash value.

If the resulting value was equal to or greater than 390-256, then the resulting value is used as the bucket number. If the resulting value is smaller than 390-256, then we perform a new bitwise AND with 511 (2^9-1 = 0x000001FF) and the hash value. This value is used as the bucket number.

Each time a key is added, we increase a counter by one and each time we remove a key we decrease the same counter by one. When the counter goes over a certain threshold, then the next bucket is split and when the counter goes below a certain threshold two buckets are merged.

In the example above bucket number 390-256 = 134 is the next bucket to be split. If a merge occurs instead then bucket 133 and 389 are merged.

A split of a bucket occurs by calculating a hash value on each key in the bucket, then performing a bitwise AND of the hash value and 256 (0x00000100). Those keys that then get the value 0 are kept in this bucket and the others are moved to bucket number 390.

The major extension of LH* compared to LH is that a bucket is a processor node instead of a local data structure (e.g. a page). LH* also involves more advanced schemes to perform the splits and how the clients can know the parameters of the LH* algorithm. These features of LH* have not been used in this thesis since all splits of fragments are performed as an atomic action.

LH*LH extends LH* by also using a LH scheme internally in the processor node. The same hash value is still used and it is necessary to be careful in how to use the hash value bits. They are used in the same way as in LH^3; the extension of LH^3 over LH*LH is to use a static number of bits of the hash value to find an even more direct hit of the element.

11.1 Handling of Hash Value Bits

In Figure 11-1 it is shown how the bits of the hash value are used in the LH^3 variant. As the name suggests, the hash value is used in three steps. The first step is to access the LH* bits. These bits specify the fragment identifier of the tuple. By using the distribution information this can be used to find all replicas of the fragment.

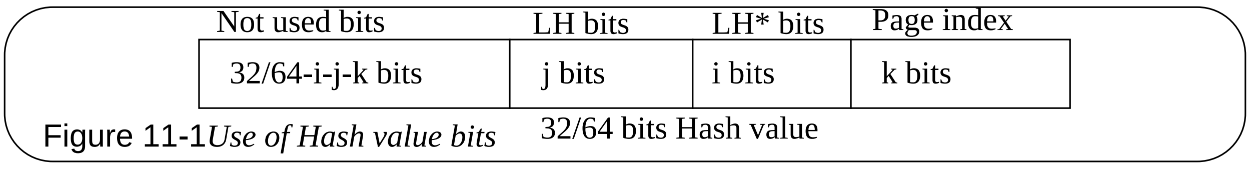


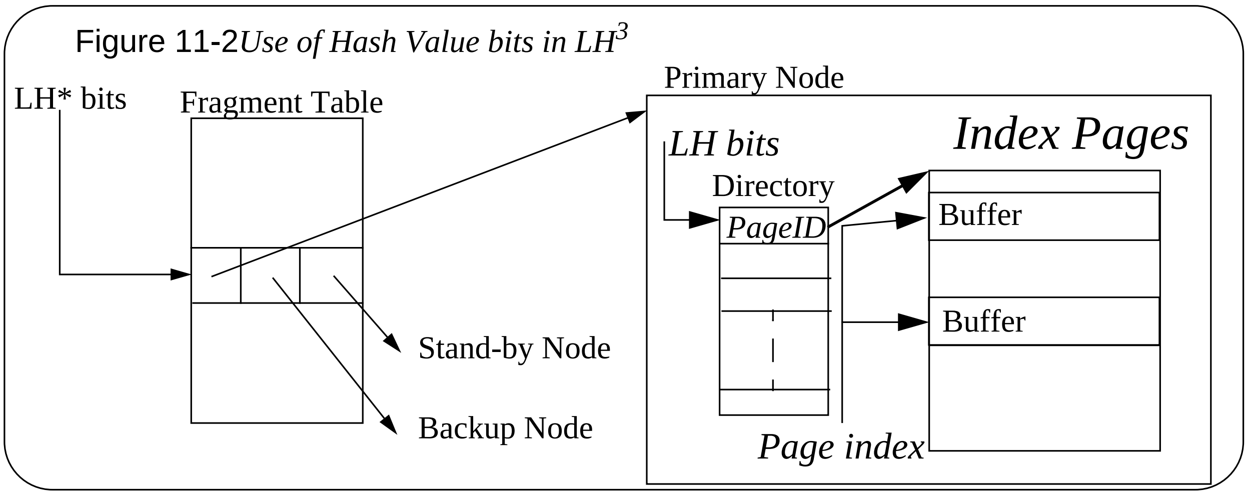
Figure 11-1*Use of Hash value bits*

Normal LH* uses a static distribution. Thus there is a simple function that translates the LH* bits into a node address. The requirements on reliability means that the possibility to dynamically move fragments must be available. Thereby in LH^3 the LH* bits are used as a fragment identifier. The fragment identifier is used as an index into a table. From this table all replicas of the fragment can be found as shown in Figure 11-2.

The next step is to use the LH bits. These are used as a pointer in a simple directory as shown in Figure 11-2. The directory is a simple array of page identifiers.

The page index is used to find the correct buffer on the correct page by a predefined calculation. The page index has a fixed number of bits. As an example assume that the page index is 6 bits and the page size is 8 kByte. The page index of 6 bits means that 64 buckets are stored in each page.

Figure 11-2*Use of Hash Value bits in LH^3*



Bucket splits and merges are performed as with a normal linear hashing function where the LH bits and the page index together form the bits of the local linear hashing function.

The number of bits for the page index is fixed at creation of the table. It should be fixed to a large value as the number of tuples in the table will be large. Otherwise six is a good figure which always allocates a page of 8 kByte at a time when more memory is needed. The number of LH bits dynamically grows and shrinks without affecting the number of LH* bits. This is the reason why these bits reside on the left side of the LH* bits. LH* bits can also grow and shrink. When the number of LH* bits grows, the number of LH bits decreases. Actually when an LH* split occurs, one bit from the LH bits is transferred from the LH bits to the LH* bits.

What will happen at a split is simply that all pages with an even LH bit number will stay and the pages with an odd LH bit number will be moved to the new storage node. Thus it is easy to see how easy it is to split a fragment. It is simply a matter of splitting the directory.

From this approach we can see that finding the correct bucket does not entail searches in any large data structures. Only if the number of fragments grows very large can this become a problem. When the local index is searched, the directory is accessed. The directory is only a few kByte at most and should normally be found in the cache memory. The access to the pages is likely to entail a cache miss. In this case prefetching can be performed to ensure that the full bucket is fetched from main memory. Thereby only one cache miss is needed to access the index. Of course many more data structures are needed to find the proper directory and page and so forth. Details of these data structures have been jointly developed with Ataullah Dabaghi and are found in [ATTA97]. However these, too, access small data structures that are often used and should normally be found in the cache memory.

11.2 Data Structure of LH^3

Ordinary linear hashing used a directory to find a linked list of disk pages. No special handling of the internal structure of the pages was described [Litwin80]. In [Larson88] an implementation of linear hashing for main memory was developed. It used a data structure as shown in Figure 11-3. The lowest bits of the linear hashing value are used as a pointer in the segment table and the higher bits are used as a pointer in the directory table. Then the elements are traversed and the key is compared to the searched key. This is obviously a very fast method to find data in main memory as long as the linked list does not become too long.

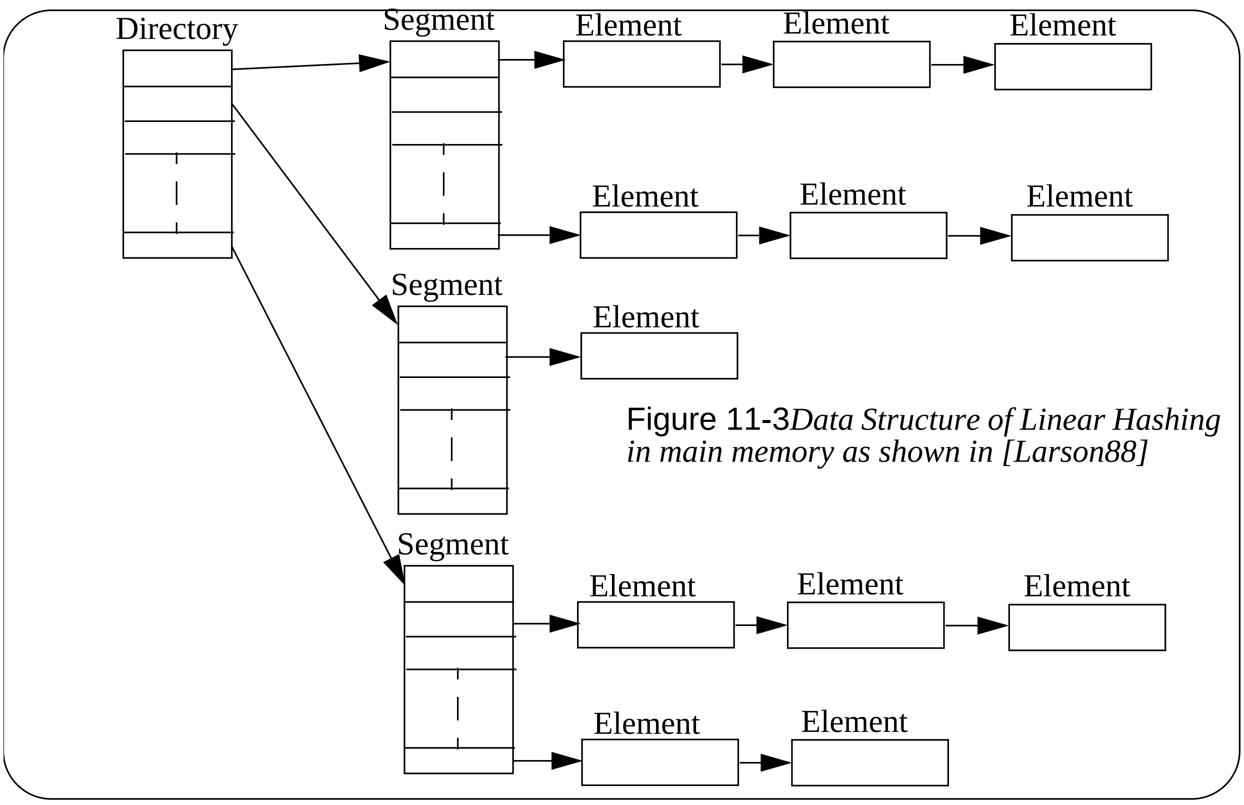


Figure 11-3*Data Structure of Linear Hashing in main memory as shown in [Larson88]*

By analysing the data structures using a cached computer architecture, we find that the directory should be in the cache most of the time. The segments are, however, less likely to be found in the caches since there are many segments in each hash table. In a large table consisting of 100,000 records there are about 100 segments of 1 kByte each. This means that there could easily be several MBytes of segments in one processor node with GBytes of memory. The elements are not likely to be found in the cache memory and so each element access is likely to cause a cache miss. Thereby the performance of this search is hampered by a number of cache misses that will cost performance in modern processor architectures.

To solve the problem with a cache miss for each element access we put a number of elements into a container. This container can then be more suitable in size to the size of cache lines. The problem with the possible cahe cmiss on segment access is solved by the page index part of the linear hash value. Each bucket has a predefined place in a page which can easily be calculated. One could describe this solution as a method where the elements are actually placed in the segment. The data structure becomes somewhat more complex but executes faster. The resulting data structure is shown in Figure 11-4.

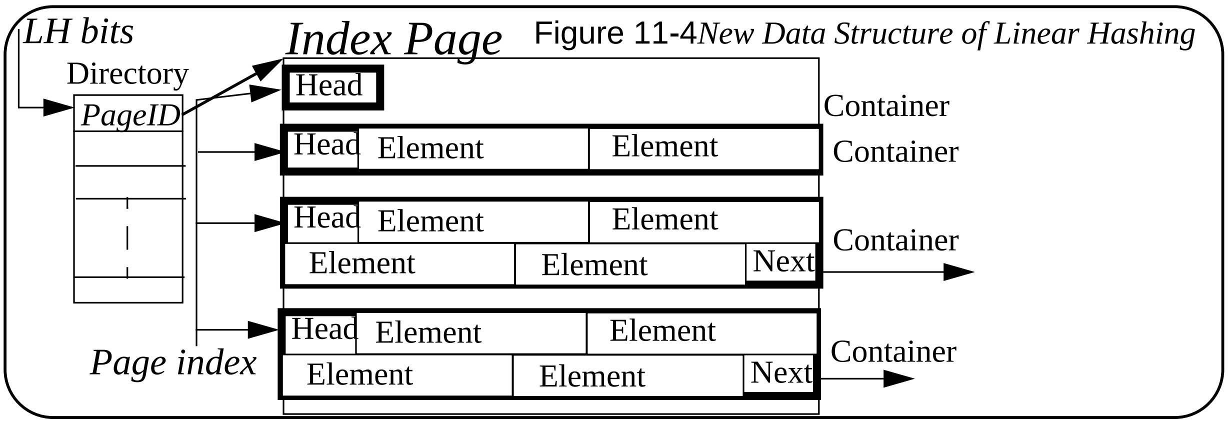


Figure 11-4*New Data Structure of Linear Hashing*

By analysing this data structure we see that the directory structure is still likely to be found in the cache memory. The accesses to the containers are likely to be cache misses. Since the probability of a cache miss is so high, one could also use containers that are larger than cache lines. Then the full container is prefetched from main memory to the cache memory to ensure that there is only one cache miss per container access. Thus the number of cache misses using this data structures has been substantially decreased compared to the normal data structure.

The use of containers with a fixed size can easily lead to a waste of memory. The reason of that is that many containers will contain zero or only one element and thus waste a lot of memory. To counter this problem each container of fixed size can have two containers. One container in the beginning and one at the end. This is described in more detail in section 11.4.3.

A side benefit is that we also get a data structure that is usable for disk access as well. It is even possible to pursue this idea and also integrate the tuples into this structure. This would, however, increase the size of the elements so that the point of using containers is lost. Also almost all indexes except the charging database can store all indexes in main memory. Hence it is better to let the index structure be such that it is very fast to access an element. Then the access of the tuples should also be very fast with at most one cache miss, see chapter 13.

A negative effect of this algorithm is that when a bucket is split, it creates more work. In the previous approach it was enough to relink the elements. The number of cache misses is, however, less in our method even when splitting. More computing resources are used but these are cheap in next generation processors as long as no cache misses occur.

11.3 Copy Fragment Protocol

Before we study the details of the data structures we will describe the copy fragment protocol. This creates some requirements on the data structures of the containers and the elements.

The fragment is created by the primary replica node after executing a state transition protocol where the system is informed of the new copy. The basic idea of the algorithm to copy a fragment is rather simple. The sending node sends the fragment, tuple by tuple until the last is sent. When the last is sent, a new state transition protocol is executed to inform the system of a new copy with data.

The receiving node receives all write transactions after the state transition protocol has been executed. If the tuple has already arrived when a write transaction is performed, it applies the write transaction to the tuple and if the tuple has not arrived, it ignores the write but sends acknowledgements in all phases of the two-phase commit protocol. Insert operations are always performed in the new replica. The idea of this protocol is a modification and an extension of the Dynamic Data Distribution protocol described in [CHAMB92] and the LH*LH bucket transfer protocol described in [KARL96].

If any of the sender and the receiver node fails during the copy process, the whole copy process is aborted and must be restarted with new sender and/or receiver nodes.

11.3.1 Alternative Solutions

Another possible solution to the copy fragment protocol is that it is handled by the primary replica all the time until the copy has been created. This simplifies the action taken by the receiver node during the copy process. The tricky part is for the primary replica to hand over control of the new copy to the system. This is performed by a state transition protocol and after performing this, the new copy is involved in all committed actions after that. The problem is that if a transaction is committed before the state transition protocol is executed and does not reach the primary replica before the state transition protocol is completed, then if the primary replica fails before this commit message arrives the new copy will never be informed of the commit message. Therefore the new copy must be able to find out where the coordinator of the transaction is, if the primary replica fails. It must also be able to find the coordinator before the coordinator flushes the transaction state to the log. If this happened, it would mean that the data must be locked until the log can be searched to find the transaction state.

Another solution is that the primary replica can inform the coordinator of the new copy in the prepare phase. This is only necessary at the end of the copy process; when the state transition protocol is started all outstanding transactions must have added or will add the new copy during the prepare phase.

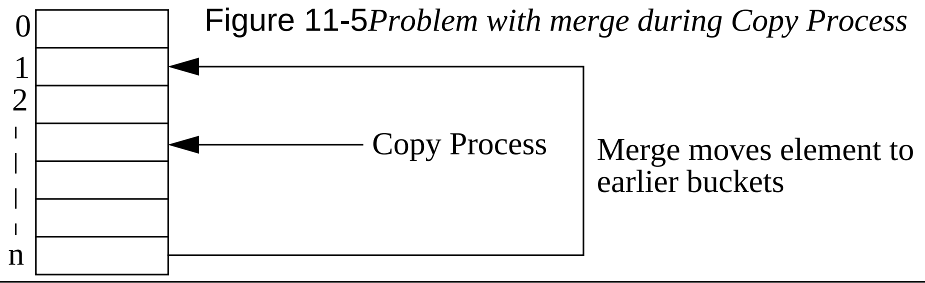
11.3.2 Algorithm Descriptions

We show here how to achieve this protocol using our new data structure LH^3. The idea is that the fragment consists of a number of buckets in the LH^3 algorithm. One bucket at a time is transferred. One simple way to proceed would then be to simply lock the bucket and all records within it. Then after the locks are granted, the tuples are transferred. This creates a simple way to move the records, yet not locking the whole table.

We will however opt for an even more fine-grain algorithm where one tuple at a time is transferred and where the copy process will never cause any deadlocks. Deadlocks can occur if a whole bucket is to be locked since there could be other transactions trying to lock several of the tuples within the bucket.

During the copy process splits and merges of buckets must be controlled. If the scan process starts from the first bucket and proceeds to the last, then it is not allowed to merge buckets during the copy process. The reason is that a merge moves elements to a bucket that is earlier in the list of buckets as seen in the figure below.

Figure 11-5*Problem with merge during Copy Process*



This problem can be overcome by keeping all elements that move backwards in a special list and thereby ensuring that the tuples that these elements refer to are moved to the new replica. Splits do not have any problems since elements are always moved to the last bucket.

The receiver does not need to bother about splits and merges. The sender specifies how many buckets it has at the start of the transfer and their fill level. It could also notify the receiver if changes occur during the transfer. The receiver then sets up the proper number of buckets in his hash table at the start of the copy process. When all tuples have been received by the receiver, it can again split or merge a proper amount of buckets to get the proper fill level of the buckets. After this the split and merge can proceed according to a normal linear hash function.

There are four algorithms that are necessary for this protocol. At both the sender and the receiver, there must be an algorithm to handle the copy process and an algorithm that handles the write transactions that occur during the copy process. These algorithms are shown below.

Before the loop that transfers the bucket starts there must also be a set-up phase. This phase involves setting up the receiver so that he can understand the messages bearing the tuples when they arrive. This is necessary if the tuples are transferred as an array of bytes. Then the descriptive information must be transferred in the set-up phase.

```
SENDER NODE: SCAN ALGORITHM
Perform set-up between sender and receiver
State Transition Protocol: Create new copy;
```

```
Set State of Fragment to ‘Moving’;
Set Tuple state of all tuples to “Not moved”;
WHILE (MORE BUCKETS) DO
 WHILE (MORE TUPLES IN BUCKET) DO
     Acquire Simple Read Lock on Tuple;
     WHEN Lock Granted DO
        IF Tuple State = ‘Not Moved’ THEN
          Read and Send Tuple;
         WAIT FOR Acknowledge from Receiver Node;
          Set Tuple State “Moved”;
        END IF;
        Release Lock;
     END WHENDO;
   END WHILEDO;
  END WHILEDO;
Set State of Fragment to ‘Not Moving’;
State Transition Protocol: New copy created;
```

```
RECEIVER NODE: COPY FRAGMENT ALGORITHM
Start when State Transition Protocol: Create new copy discovered
End when State Transition Protocol: New copy created discovered
WHEN (Tuple Received from Sender Node) DO
   IF (Tuple NOT Exist) THEN
     Acquire Exclusive lock;
     Insert Tuple;
     Release Lock;
   ENDIF
   Send Acknowledge to Sender Node;
END WHENDO
```

The major problem during the copy process is inserts. To ensure that these tuples are not missed we use the simple method of always performing them in the receiver node. Thereby there is no risk that the scan process in the sender misses any tuples. To avoid unnecessary work in the copy process the sender also makes a “best effort” to ensure that inserted tuples are not transferred. As shown above the receiver will check for duplicate tuples and thereby sending a tuple already existing in the receiver is no catastrophe.

The sender node checks the list of nodes involved in the operation to find out if the receiving node is involved in the transaction. The receiving node could be involved without the sender node knowing it due to the commit protocol. As mentioned above there is no harm done in this case. The sender will then transfer the tuple in the scan process, which will be discovered by the receiver node as a duplicate.

```
SENDER NODE: WRITE TRANSACTION ALGORITHM
IF Insert AND Receiving Node is involved in transaction THEN
   Set Tuple State = ‘Moved’;
ELSE
  Perform Normal Operation;
ENDIF
```

The receiver node performs all insert operations. It also performs all read/update/delete operations where the tuple exist since it has arrived previously. Operations on tuples that does not exist since they have not yet arrived are handled as if they were successful and reported as such to the transaction controller. It can ignore these operations since they will be reflected in the tuple when it arrives. This is possible since each tuple copy is a mini-transaction which locks the tuple before moving it. Thus the copy process is serialised with normal transactions.

```
RECEIVER NODE: WRITE TRANSACTION ALGORITHM
IF Insert THEN
  Perform Normal Insert;
ELSE IF (Tuple exists) AND (Update OR Delete) THEN
  Perform normal write transaction;
ELSE IF (Tuple not exists) AND (Update or Delete) THEN
  Ignore Writes, send positive acknowledgements in all two-phase commit messages;
END IF
```

11.4 Detailed Data Description

Now that we have described the algorithm to copy a fragment we are prepared to describe the details of the data in the containers and in the elements.

11.4.1 Container Data

To better understand the details of the container we will describe its data structure in more detail. The header contains four information items. It contains the length of the container to allow for variable container length. The maximum size of the container is 256 Bytes with a 6-bit container length since space is always allocated in at least 4 byte chunks. The header has an indicator that specifies if the next container is on the same page or on an overflow page. If the container is on another page, then the next pointer contains a page index and a page reference. This means that 2 GBytes of pages can be accessed with this next pointer. If this is not enough in the future, the next pointer could also be a pointer into a table of page references. This table is specific for each fragment. In this case each fragment can address up to 2 GBytes of overflow pages. If even this is not enough, then the header must be extended to be 8 bytes instead of 4 bytes. Another option is also to use a larger page size of the overflow pages.

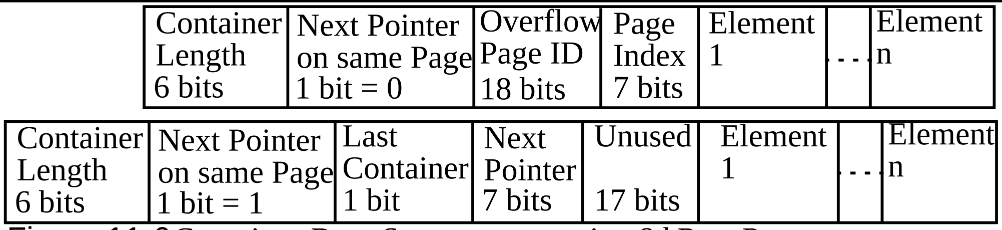


Figure 11-6*Container Data Structure assuming 8 kByte Pages*

The container must have an indicator of whether there are more containers in this bucket. To indicate that it is the last container is achieved by setting the bit for next pointer on the same page and the bit for last container. This method is used to save bits in the container header. The reference to an overflow page ID is mapped through a table to a physical page ID. This makes it possible to have location and replication independent index. In a system restart several fragments which were created in different nodes can be created in the same node.

Then there are a number of elements in the container.

Overflow pages are simple to handle since each page has its own next pointer. Thus one overflow page can be used at a time. To simplify recovery there should, however, be one overflow page per fragment. Otherwise the fragment cannot be recovered until all overflow pages have been recovered.

11.4.2 Element Data

The element data structure is shown in Figure 11-7. To ensure a small overhead we use a cameolont data structure. If the tuple is not locked, then there is no need of a transaction pointer and thereby the full 32 bit header can be used for hash value, lock mode, tuple status, lock status, local key length and tuple key length. When the tuple is locked, then a transaction pointer is needed. Also to enable fast searching through the hash table we need to keep the local and the tuple key length even when the tuple is locked. To make place for a transaction pointer we simply move the hash value, the tuple status and the lock mode to the transaction record. Thereby we get 23 bits to use as a transaction pointer which should be more than enough. The transaction pointer is used to find the transaction record of the lock owner and the lock queue is found from the transaction record.

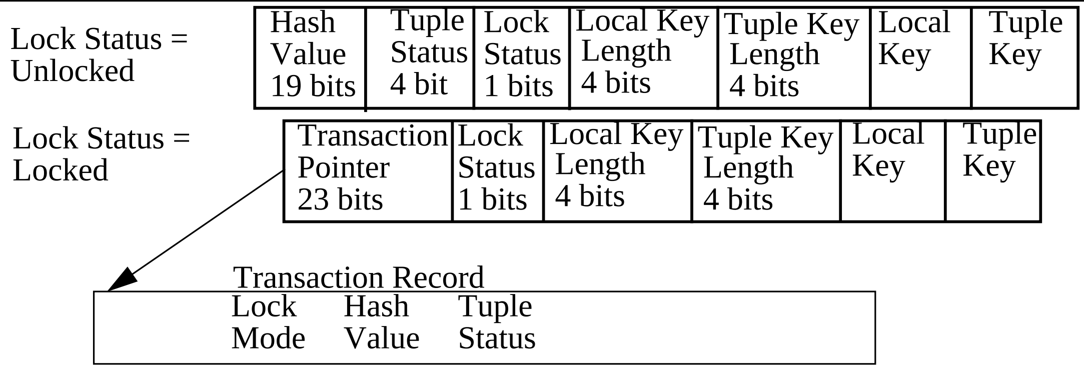


Figure 11-7*Element Data Structure*

The tuple status is used by the copy fragment protocol to indicate whether the tuple has moved or not. To ensure that more than one concurrent copy process can proceed simultaneously we have four bits. Thereby four concurrent copy processes can be handled simultaneously. Each copy process can also handle sending the fragment to more than one node.

The lock status contains information on whether the tuple is locked and, if locked, the lock mode specifies the type of lock. The local key length contains the length of the local key. The local key is the reference used to find the tuple within this machine. It contains a page identifier and a page index, the size of the local key is mostly 4 but needs to be 8 bytes with very large database tables. If the index is used as a secondary index or if the index is the table, then the local key is the sought attribute. The tuple key length contains the length of the tuple key. Both these lengths are counted in number of words of 4 bytes. If the size is zero, this means that the first word of the key is the length of the key. Thereby keys of any length can be handled although short keys are handled very efficiently in terms of storage overhead.

The hash value is also placed in the element, the reason being to speed up split and merge processing. This optimisation of linear hashing was suggested in [PETT95]. Since storing the full hash value for this purpose creates a large memory overhead, we are only storing a window of nineteen bits of the hash value. This should be enough for most purposes. The k bits and the minimum number of LH* bits (e.g. 3 bits for 8 fragments) are never needed by the split processing. Also most fragments use a minimum number of entries in the directory table. This should be 8 bits even in the minimum case. Then the 19 bits can be used to grow the hash table 524288 times before the hash value window is missed. Thereby the window should only be missed when using 64-bit hash values and after a tremendous growth of the index.

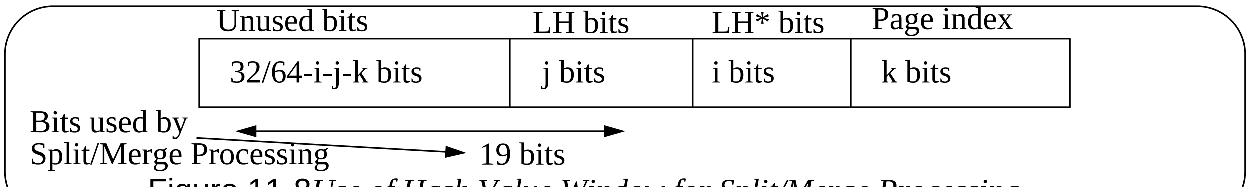


Figure 11-8*Use of Hash Value Window for Split/Merge Processing*

If the window gets misplaced, the window can be moved. This is performed by first setting the fragment status so that the window of the hash value bits are not used. This means that the hash value needs to be computed during split and merge processing. Also the fragment record describes the new window. This is used by insert operations when storing elements.

Then a background process starts to move the window. One bucket at a time is processed and the window is changed by computing the hash value and storing the new window. This can be performed even if the element is locked since it does not interfere with transaction processing.

When the background process has completed, the status of the fragment is updated again and the new window of the hash value can be used by split and merge processing.

11.4.3 Page Memory Handling

Not all buckets in a hash table are of the same size and we are preallocating storage for all buckets. This makes it necessary to be careful with the memory area of the pages. At allocation of a new page for storage of buckets, all buckets are assigned to their positions and at least 4 bytes for the container header will always be there. When elements are inserted, the access manager tries to allocate more memory consecutively up to a limit. If this is successful, the size of the container simply grows by the allocated amount.

The limit is the space between consecutive containers. This spacing should be dependent on the cache line size of the computer. In current computers the cache line size is mostly 64 bytes and this would be a suitable spacing. Most processors (e.g. UltraSparc-II and Alpha 21164) can handle at least two outstanding cache misses simultaneously and hence we can use 128 bytes instead in current systems. In future generations the cache line size will increase and then the spacing can also increase. It is, however, not appropriate to increase it so that the processing time becomes larger than the cache miss time. Thereby the spacing should be carefully tuned in each generation. There is also a need to have a header area in each page; this has the advantage that we also get an overflow area. Thereby the spacing should be slightly smaller than 128 bytes.

With 16-byte key size, as is the most common in the benchmarks defined, a normal element size will be 24 bytes with a 4-byte local key. Thereby the size of zero elements is only the first four bytes, the size of one element would be 28 bytes, two elements 52 bytes, three elements 76 bytes, four elements 100 bytes and five elements 124 bytes. Using 8-byte keys we would have seven keys in 112 bytes. Using 4 byte keys we would have ten keys in 124 bytes and using 12-byte keys we would have six keys in 124 bytes.

From this discussion we derive that 124 bytes is a suitable spacing. This allows room for a 8 byte header of the page and an overflow area consisting of two 124-byte buffers.

Each 124-byte buffer can contain two containers. One container grows from the beginning, referred to as the left container, and the other one grows from the end of the buffer, referred to as the right container. The right containers are used for overflow containers. When a buffer has a container in both ends, it is no longer on any free list. The free space within the buffer is handled locally by the two containers.

When a page is allocated, there is a free list of the two 124 byte free buffers and another free list with the right containers. When one of the two overflow buffers are allocated, its right side is moved to the free list of the right containers.

When the left container grows over the high limit, then the right container is removed from the free list. The left container can then grow all the way up to 124 bytes. The limit should be about 60 bytes.When the left container grows below the low limit the right side of the container is put back on the free list. The page data structure is shown in Figure 11-9.

The page data structure gives us two different containers. The left container grows to the right and the right container grows to the left. They look the same but grow in different directions.

The page size can easily be extended to larger sizes. If the size is 32 kBytes then there will be 256 containers, 8 free containers and a page header of 32 bytes. When the cache line size grows, the buffer size can also grow. Another almost perfect fit comes when using a buffer size of 244 bytes. In this case with 4-byte tuple keys there will be 20 keys in each container, with 8 byte tuple keys there will be 15 keys in each container, with 12-byte keys there will 12 keys in each container, with 16 byte keys there will be 10 keys in each container and in all cases there is a perfect fit.

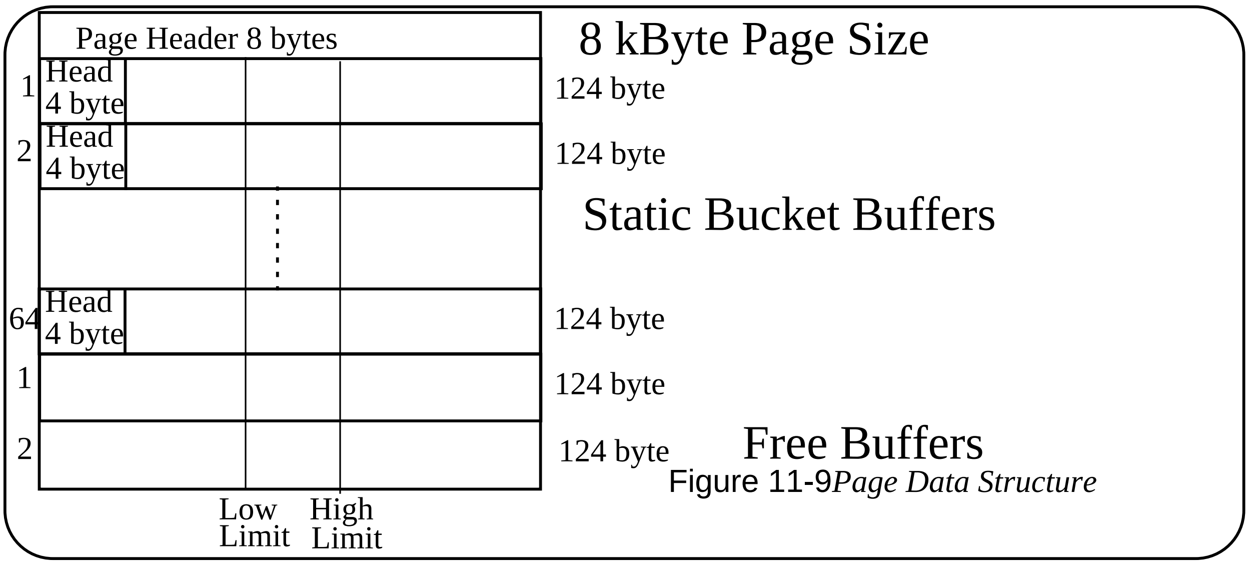


Figure 11-9*Page Data Structure*

11.4.4 Element Split between Containers

The buckets are organised in a linked list of containers as described above. An important reflection is that an element can cross the boundary of two or more containers. This makes it easier to handle large keys in this data structure. With small keys in the table it could, however, be an option to avoid this element split. Catering for the element split will always cost a little bit in performance and is therefore a suitable parameter of the hash table.

No marker indicating that a container begins in the middle of an element is necessary. The reason is that all processing starts at the first container and proceeds to the next. During this processing the context can be held in local variables so that no markers are needed.

11.4.5 Overflow Handling

The overflow pages are arranged in the same manner as the original pages. An overflow page can be used by any page since the container has a page reference in the next pointer. To simplify recovery it is, however, desirable that an overflow page is only used by one fragment. This makes overflow handling rather simple since one overflow page at a time is allocated.

11.5 Split and Join Handling

11.5.1 Local Bucket Split and Join

Instead of using a maximum load factor with the number of elements in the fragment, the element sizes are used as the maximum load factor. With a load factor of 85% this gives 105 bytes per bucket as the maximum load factor. When more than 105 bytes have been added, a bucket split is initiated and when the load have decreased with 105 bytes a bucket join is initiated.

When a bucket is split using Larson’s method [Larson88], it is only necessary to rearrange the pointers. Thereby the amount of instructions to perform this action should be small. The number of cache misses can, however, be substantial since it is necessary to access many different memory areas. With the new method, the elements that are moved during the split must be copied to the new container. The bucket that is split must be compressed also. This means that the new method uses more instructions than Larsons method. However, most of these instructions are performed in a small number of cache lines and thus only a few cache misses should occur.

11.5.2 Fragment Split and Join

Splitting a fragment is very easy with this model. It is simply a matter of splitting the directory. All directory entries that end in a zero bit stay in the same directory in a new position and those ending with a one bit move to the new directory. Even with a very large data structure consisting of 2.000 page references (16 MB if one page per reference, 1 GByte if 64 pages per reference), this split only affects the original directory of size 8 kBytes and the new directory of size 4 kBytes. The process is a simple copy process and is therefore performed in less than 100 microseconds. The hash table must then remember that it has been split until the split information has been sent to all nodes in the distributed database.

11.6 Lock Handling

Since the lock data structure is combined with the index structure, it is essential that all accesses to the tuples go through the index of the tuple key. Locks on tuples stored on disk can also be handled in this manner by ensuring that disk pages are not removed from main memory until all locks on the page are released. Otherwise the same data structure could be used, where only the locked tuples are in a main memory index as in a normal lock data structure [Gray93].

11.7 Secondary Indexes

11.7.1 Unique Secondary Index

When this structure is used as unique secondary index, the only difference is that the tuple key is the secondary key and the local key is the sought tuple key.

11.7.2 Non-unique Secondary Index

There are two options here, either the index is used only as a non-unique secondary index or it is used as a non-unique primary index. If it is used as a non-unique index then the major difference is that there can be several elements with the same key. There are different ways to handle this. Either as above with a list of all elements, otherwise there could be a search structure within each bucket.

If the index is used as the primary access to the tuples, the index is used as a normal index of the tuple key. The difference is that tuple key specified in LH^3 consists of both the tuple key of the table and a secondary key. It is necessary to provide both the tuple key and the secondary key to be able to find a tuple.

11.8 Handling of Physiological UNDO Log

As shown in chapter 9 it is necessary to have a physiological UNDO log in connection with a fuzzy checkpoint to be able to create an action-consistent checkpoint that can be used by a logical fragment log. This must be supported by all index structures in the data server.

When the system crashes, all ongoing transactions are aborted. Thereby when the system restarts no locks on tuples should be set. The information stored in our index structures is basically a mapping from a key to a local key in connection with lock information of the key.

The actions that need to be logged are then inserts of a key in a container, deletion of a key from a container, locking a key in a container, deleting an overflow container, creating an overflow container, allocating an overflow page, deallocating an overflow page, moving elements between containers (at split and merge), and when elements are updated (e.g. when the local key is updated).

Unlocking a key is not necessary to log since this is mandatory after a crash. If a key is unlocked, the lock key entry can be removed from the UNDO log. To ensure that all locks are removed at restart the locks at the time of the checkpoint are logged in the UNDO log. The last item in the UNDO log of a checkpoint should also specify the size of the directory at the time of the checkpoint and also the size of the directory of the overflow pages. Accesses to overflow pages go through a directory to ensure that checkpoints are location independent.

11.9 Conclusion

This structure of linear hashing is especially suitable for computers with cache memory where a cache miss is equivalent to a large number of executed processor instructions. It also fits small key sizes, 4-16 byte keys as is mostly used in benchmarks developed here and other benchmarks used. It also reflects most real life applications where there is normally a rather short key. It is, however, also good for use with larger keys. Variable sized keys are also supported.

However, indexes for, for instance, http-addresses and file names are better provided by other index types. Http-addresses are preferably handled by a B-tree structure with compression as will be shown in Chapter 12. The B-tree structure with compression also results in a much smaller storage space and this linear hashing structure is then used for fast access and the B-tree structure can be used for small storage overhead.

Indexes of telephone numbers use this structure if speed is the most important and use the compressed B-tree structure if memory space is more important.

The optional parts of the hash table are stored in a record describing the hash table. This record is accessed during all types of accesses. The number of these records is fairly small, so these accesses should not incur very many cache misses.

This linear hashing method is used for all types of tuple keys, that is primary keys and tuple identifiers. If a primary key is very long, then a tuple identifier should be assigned as tuple key instead. The primary key can then be a secondary key of a unique index instead. This index can then use another index if needed. A typical example of this is the news-on-demand application where documents are accessed by document name. This is not a good tuple key; this should then be a unique secondary index instead. Unique secondary indexes can use this linear hashing method. Non-unique secondary key indexes can use this method with the adaptation described above.

12 A Compressed and Distributed B+tree, HTTP-tree

There is also a need for ordered data structures in the applications, B+trees being a popular variant of this. We will use a distributed variant of the B+trees where the root page is fully replicated in all processor nodes of the database. Since the root page is fully replicated, it is possible to find the processor node immediately. The root page changes when B+tree fragments join or split and when B+tree fragments move. Those changes are handled by a state transition protocol.

Using an ordered index means that it becomes possible to compress the keys for certain keys, such as http-addresses, file names, timestamps, telephone numbers and so forth. We will therefore show how to accomplish such a compressed distributed B+tree. The ideas on compression are, of course, also valid in a local B+tree.

The distributed B+tree can be used for all types of secondary indexes. Using it for tuple keys will not be studied in this thesis. The data structures described below are useful both as a unique secondary index and as a non-unique secondary index.

The ideas in this chapter are a development of the prefix B-trees [Bayer77][Gray93]. It has been developed with some extensions to make it useful.

It is important to note is that nodes are used in this chapter to describe a part of a page. It is a node in the B-tree. This is in contrast to the other chapters where node has been used as a short form of processor node.

12.1 Compressed B+tree for Long Names

This index uses the fact that names that are organised in an alphabetical order often do not differ very much between neighbours in the list. This can be used to compress the data structure by a factor of five or more. This could also speed up processing since the data structure will use less memory and thereby have a better hit rate in the cache memory.

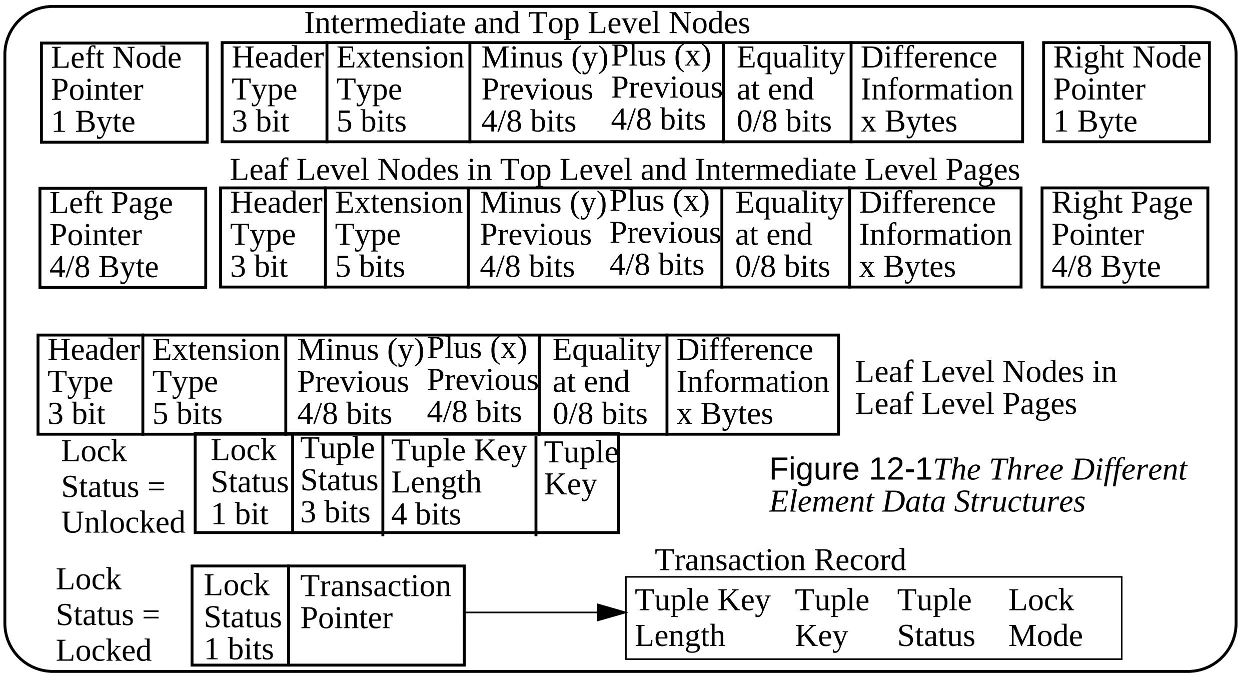
The starting point of this structure is to use a normal B+tree structure. Instead of storing the full name of each entry, only the difference from the previous entry is stored. The index maps the name to a tuple key (or local key if it is used as a tuple key index). This tuple key can then be used as a file handle when using the data server to implement scalable file systems. An example of the compression algorithm is provided here:

START =http://www.ericsson.se/ndb/compbtree/description.html
2) http://www.ericsson.se/ndb/compbtree/nodedescription.html
3) http://www.ericsson.se/ndb/compbtree/system_arch.pdf
4) http://www.ericsson.se/ndb/compbtree/system_arch.ps

To describe this with the compressed algorithm we need a start value, this is equal to START. Then 2), 3) and 4) are described as:

2) Equality at end (1 bit), 4 bit minus and plus (1 bit), difference information exists (1 bit), extension = .html (5 bits), -0 (4 bits), +4 (4 bits), Last 11 equal (8 bits), “node” (4 bytes)
3) Not equal at end (1 bit), 4 bit minus and plus (1 bit), difference information exists (1 bit), extension = .pdf (5 bits), -15 (4 bits), +11 (4 bits), “system_arch” (11 bytes)
4) Not equal at end (1 bit), 4 bit minus and plus (1 bit), difference information does not exist, extension = .ps (5 bits)

A total of 21 bytes for 2), 3) and 4) instead of 158 bytes originally. The amount of compression is, of course, dependent on the naming style and so forth. A substantial compression is, however, likely in most cases. The B+tree is organised as a normal B+tree with a number of pages organised in a hierarchical structure. Within the pages there is also a B+tree structure to find a key within the page. Here nodes of smaller size than pages are used instead (e.g. 128 bytes). Within a node a sequential search is performed using this compression algorithm. This means that there are three types of nodes. The first type are top and intermediate level nodes within the page, these are used to find the leaf level nodes within a page. The leaf-level nodes are either leaf-level nodes of a top or intermediate level page or a leaf-level node of a leaf level page. The data structure of these three types of nodes are shown in Figure 12-1.



Intermediate and Top Level Nodes
Left Node Pointer 1 Byte
Header Type 3 bit
Extension Type 5 bits
Minus (y) Previous 4/8 bits
Plus (x) Previous 4/8 bits
Equality at end 0/8 bits
Difference Information x Bytes
Right Node Pointer 1 Byte
Leaf Level Nodes in Top Level and Intermediate Level Pages
Left Page Pointer 4/8 Byte
Header Type 3 bit
Extension Type 5 bits
Minus (y) Previous 4/8 bits
Plus (x) Previous 4/8 bits
Equality at end 0/8 bits
Difference Information x Bytes
Right Page Pointer 4/8 Byte
Header Type 3 bit
Extension Type 5 bits
Minus (y) Previous 4/8 bits
Plus (x) Previous 4/8 bits
Equality at end 0/8 bits
Difference Information x Bytes
Leaf Level Nodes in Leaf Level Pages
Lock Status = Unlocked
Lock Status 1 bit
Tuple Status 3 bits
Tuple Key Length 4 bits
Tuple Key
Lock Status = Locked
Lock Status 1 bits
Transaction Pointer
Transaction Record
Tuple Key Length
Tuple Key
Tuple Status
Lock Mode

Figure 12-1*The Three Different Element Data Structures*

The header type contains three bits: the first bit specifies whether there is any byte that counts how many bytes are equal at the end of the word, the second bit specifies whether 4 bits or 8 bits are used for minus and plus information and finally the third bit specifies whether there is any difference in information at all. Thus in some cases the descriptive information becomes one single byte in some cases. As an example, an index on telephone numbers will have as extensions 0,1,2,3,4,5,6,7,8,9,00 and some other common ends of telephone numbers. Then a list of telephone numbers will mostly contain only 1 byte per telephone number.

In the table below the different uses of the header information is shown.

Table 12-1*Possible ways of using the header information*

Equality at the end	Difference information exists	4 bit or 8 bit used for minus and plus	Minus	Plus	Number of equals at the end	Extension type	Difference Information
0	0	Not used	Not there	Not there	Not there	0-31	Not there
0	1	0 (4 bit)	0-15	0-15	Not there	0-31	Upto 15 bytes
0	1	1 (8 bit)	0-255	0-255	Not there	0-31	Upto 255 bytes
1	0	Not used	Not there	Not there	0-255	0-31	Not there
1	1	0 (4 bit)	0-15	0-15	0-255	0-31	Upto 15 bytes
1	1	1 (8 bit)	0-255	0-255	0-255	0-31	Upto 255

The information at the bottom of the tree is similar to what is found in the LH^3 data structure. There is a tuple status used when moving the B+tree-fragments, a tuple key length and a tuple key and finally a lock status. When the information is locked, a transaction record pointer is put there and the tuple key, the tuple key length, the tuple status (used to indicate moved or not moved during fragment split) and the lock mode are temporarily saved in the transaction record.

An alternative solution is to place the lock information in another data structure which allows more room for the tuple key length. If the copy fragment protocol transfers a bucket at a time, then there is no need for any tuple status either. Then finally, if the tuple key length is static for a B+tree, we only need the tuple key stored in the element.

If we need to make an index for mapping telephone numbers to an identifier specifying in which country and state the owner of the number resides (2 bytes), then there would be about 3-4 bytes per entry in this index. Even with all telephone numbers on earth, the index would fit in 3-4 GByte of memory.

An index for a city of 1 million subscribers would only need 1 byte of identifier per telephone number. Thereby 2-3 bytes * 1 million = 2-3 MByte of storage for the full index. This will fit in the cache memory of a high-end computer.

12.1.1 Searching the HTTP-tree

A search starts from a root page and proceeds downwards. Since only difference in information is stored in the entries in the B+tree, it is necessary to have a start value at all levels of the search. By using the start value and the difference in information, the real key stored is found. At the root node in the root page the start value is NULL. As a search is always from the top to the bottom it is not necessary to store start information at all. The search will always be able to calculate the start value at all levels. It is likely that management of the HTTP-tree is easier if start values are stored at least for each page. This requires further study into the details of the HTTP-tree.

As seen in Figure 12-2 the searched key is then compared to key1 and if smaller than or equal to key1 it follows the left pointer, otherwise it follows compares against the next key, if smaller than or equal to this key it will follow pointer1. The start value of the node that pointer1 points to is then key1 since all keys are greater than key1 in this node. In this manner one proceeds down the nodes in a page.

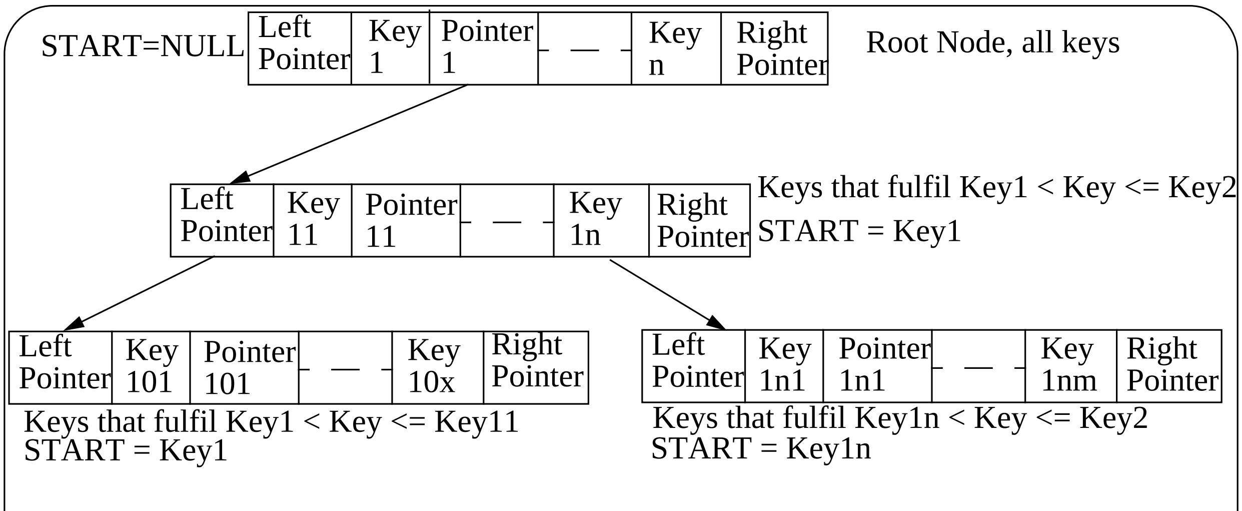


Figure 12-2*Assignment of START values during search in Compressed B+tree*

When one moves to a new page, this page has the start value of the root node in the page as if one had proceeded down the tree without respect to pages. This means that the start value of a page is not the leftmost item, the start value is rather the value which all keys are greater than, as seen in the figure below.

As can be seen from this analysis, the start value is implicitly known all the time when a search in the index is performed. Therefore no storage of start values is needed on behalf of searches.

When splits of nodes occur and when pages are split then the left node/page has the same start value as the node/page split. The right node/page receives the start value which is equal to the rightmost key in the left node/page. This value is then inserted in the level above the split level.

Since we are using a B+tree structure, nodes/keys can also be shipped between nodes/pages. In this case the rightmost key of the remaining node/page is also used to update the level above the ship level.

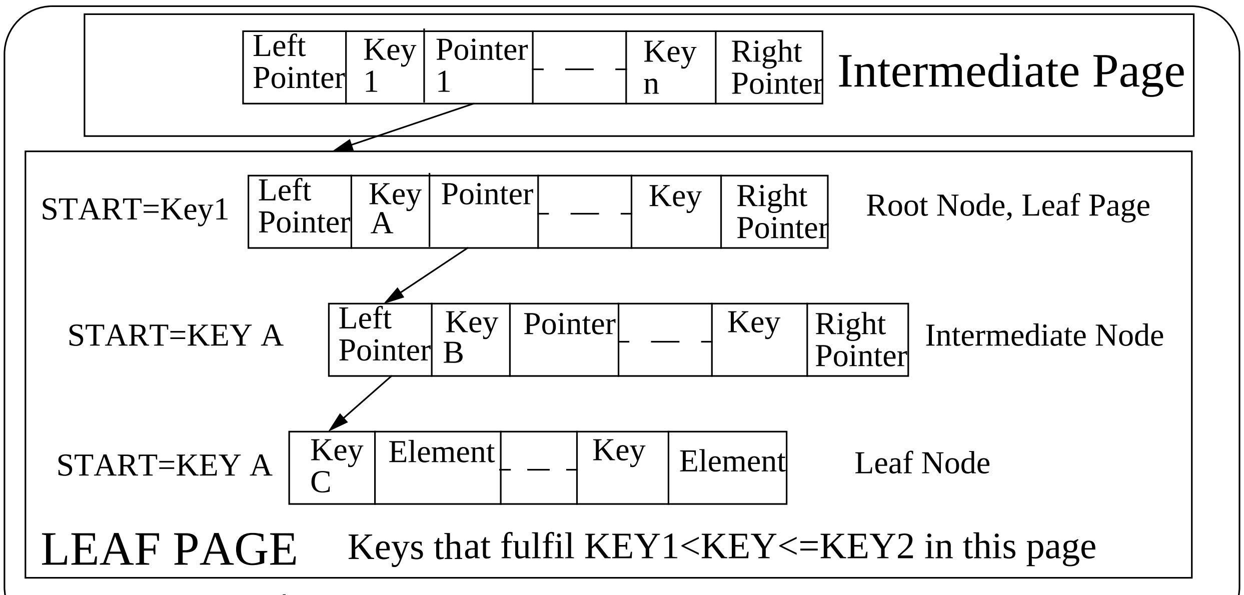


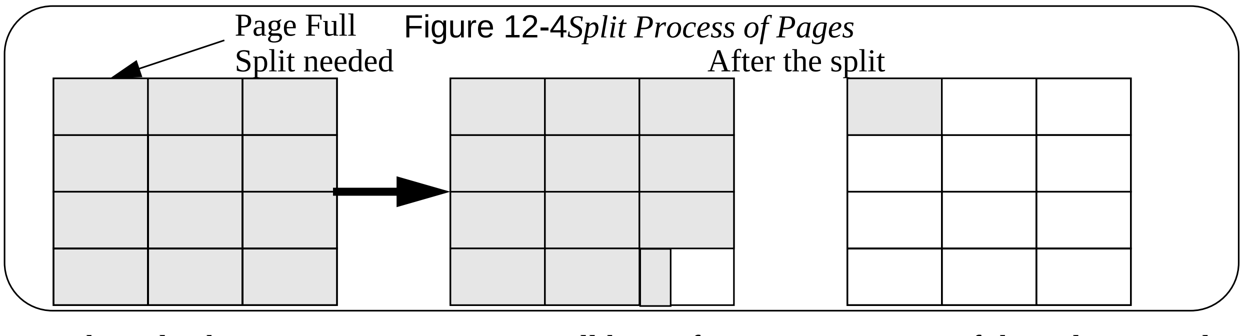
Figure 12-3*Assignment of START Values to Pages*

Finally an index can also be used in scan operations. In this the operation start from the bottom of the tree at the leftmost node. This has the start value NULL. The start value of any element is the entry before and thereby scans can keep track of the start value.

When elements are inserted and deleted, the entry to the right must always be updated since the difference to the left neighbour has changed. Inserted records are inserted by using the start value as calculated during the search. If the last key of a node/page is deleted, then the start value of the next node/page is updated which affects both the first element in the node/page and the parent node/page. Thus both those needs to be updated

The B+tree uses pointers in both directions to ensure that the fill level is high enough. When a page is split anyway, the page which is split only ships one node to the new page. The reason is to improve real-time performance when a split of all pages up to the root level occurs. Each time an overflow of the page then occurs, it will move another node over to the new page. Thereby the split activity is spread over a longer time for better real-time behaviour.

Figure 12-4*Split Process of Pages*



Since the index becomes very compact, it will better fit in main memory. If the index is used to map a global telephone number into a country code only about 4-5 bytes per number is needed and thereby all numbers on the earth (about 1 billion) can fit in 4-5 GBytes.

12.2 Memory Handling

Since the architecture should perform well in cached computer architectures, the nodes should be of a size that fits well with the size of cache lines. Thereby a node size of 128 bytes is appropriate (see reasoning in section 11.4.3). When the size of the node after an insert exceeds 128 bytes then it causes a split or a shipping of elements to a neighbour node. If a large key is inserted that is larger than 128 bytes, two or more nodes of size 128 bytes are used. Since normally the size of the elements should be rather small, this scheme should be good enough.

12.3 Split/Merge of B+tree Fragments

The split is performed by first performing a proper split of the root page of the B+tree-fragment. A part of the B+tree is replicated in all processor nodes containing a reference to the B+tree; this part includes the root page. Then two new pages are allocated as root pages. The two root pages are assigned as B+tree-fragments internally in the processor node. Also a new root page that refers to the split pages is created. After this split the processor node can now accept requests to search both with the old non-split fragment and with both the split fragments. When all replicas of the B+tree-fragment have performed the split, the split is performed on a distributed level through a state transition protocol. Since both old and new fragment identifiers can be used, there is no particular requirement on stopping traffic during the state transition protocol. When all operations using the old fragment identifiers have completed, the created root page can be removed and the two B+tree-fragments are now independent fragments. Thus a move of the fragments can be performed if needed.

A merge is performed in another order. First the B+tree-fragments to be merged are moved to the same processor node for all replicas. Then a local merge is performed followed by an operation to update the distributed information.

12.4 Copy B+tree Fragment

When it is necessary to copy a fragment of the B+tree, this must as before be performed simultaneously with a lot of transactions that update the B+tree fragment. Since it is only necessary to move keys and not tuples here, it is possible to use coarser locking than with the linear hashing protocol. We will therefore copy one leaf level node at a time to the new fragment copy.

By starting the copy process in the lowest node, it is possible to easily find out whether a key has been received in the new fragment or not. If the key is between the start and end value, the key has been copied, otherwise not.

At the receiver the upper levels of the B+tree are created by building an independent B+tree structure. The received nodes are only seen as new keys to be inserted in the B+tree. The major difference from normal inserts is that those inserts will always be inserted at the end of the B+tree.

As in the previous copy fragment protocol, the sender of the fragment is the primary copy owner. In many cases this is the only copy of the fragment. Since the indexes are copies of some attributes in the table, the index can be created from the tuples in the table. Thereby reliability of the index is not as critical as with the tuples. Therefore it is not necessary to support stand-by copies with indexes, only support of backup copies is needed.

As before there are four algorithms needed, one to scan the fragment at the receiver, one to receive these scanned nodes and one algorithm at the sender and the receiver to handle operations on the index while copying the fragment.

```
SCAN ALGORITHM AT SENDER:
Inform all of copy process starting through state transition protocol
START: WHILE NOT END OF FRAGMENT
Set Read Lock on Next Leaf Level Node;
WHEN Read Lock on node Acquired THEN
Read Node and Send Node;
WHEN Ack from Receiver THEN
Release Lock on node
GOTO START:
END: Copy Fragment Finished
```

The locking of a node needs to be done carefully since otherwise deadlocks can easily occur. If a transaction owns a lock in a node it should be allowed to acquire new locks; other transactions should only be allowed to set read locks. One problem that could occur is that nodes are split and that keys are shipped to neighbour nodes/pages. This is solved by only using a next pointer in those situations. Thereby splits always create new nodes to the right. Thus if one is trying to lock a node, a split of this node will never create a new node to the left. If this was possible, this new node could be missed in the copy process. Splits that occur in a part already copied create no problems since the inserts that triggered the split will be performed independently at the receiving node as part of the transaction. Problems with merging are easily solved by not allowing any merges to occur during the copy process. Page splitting and merging do not affect nodes and cause therefore no problems during the scan.

The algorithm at the receiver to receive the node is very simple. It inserts the new node and updates the upper levels of the B+tree fragment; in particular it will update the root node to reflect the new end of fragment.

Operations at the sending node continue as usual, they only need to check the node status to ensure that it is not read locked when inserting and deleting.

Operations at the receiver node are performed as usual if they are between the start and end key of the B+tree fragment; if not the operation is acknowledged but not performed.

Operations on indexes are performed starting at the primary as with normal tuple operations.

12.5 Node Data Structure

From the above discussion it is obvious that each node needs to have a status bit and a next pointer. The lock status bit is only useful for leaf level nodes.

12.6 Handling of Physiological UNDO Log

The UNDO log in the http-tree is similar to handling of the LH^3. The difference is that instead of containers we have nodes.

12.7 Related Work

[Bayer77][Gray93] reports work on prefix B-trees and suffix B-trees. This B+tree implementation is an extension of prefix B-trees. It has been extended by having a B-tree structure also within pages. This is very important to achieve the usefulness gained by prefix B-trees. Searching a whole page sequentially using prefix B-trees will simply not be a good idea. By implementing nodes within a page with the size adapted to cache lines only a node needs to be searched. Since modern processors are extremely fast at processing when working within the cache, this structure will be very useful. Also extensions have been added that use common endings which are common in Http-addresses and file names. Also, commonalities between neighbours at the end can be handled. The prefix method is used on all levels in the B+tree. Mainly the leaf level benefits but also certain commonalities such as “http://www” can be removed on higher levels.

12.8 Conclusion

A new compressed distributed B+tree has been described that will be very useful in developing secondary indexes for long names, telephone numbers, timestamps and so forth. This is useful when implementing a distributed file system using a parallel data server. The secondary index can then be used to translate the file name to a file handle and the file handle can then be used to get direct access to the file and its attributes.

13 Data Structure for Tuple Storage

The ideas described in this section have been developed in cooperation with David Jacobsson and Joakim Larsson. The details of the data structures are found in [LARS97].

The aim of this chapter is to describe how the design of the tuple storage is made. This includes:

1) Use of copies of tuple parts to avoid the need of a logical UNDO log.

The use of an action-consistent checkpoint provides the opportunity to remove the need of a logical UNDO log by using copies of the tuple parts as “UNDO log records”

2) Data structure of tuples.

A B-tree structure is used for the tuples to enable very flexible tuple structures.

3) Details on handling of pages.

4) Use of the UNDO log for tuple storage.

5) Handling of very large attributes.

Ideas from [Lehman89] are shown how to integrate in the tuple storage to support very large attributes.

13.1 Recovery Support in Tuple Storage

Our recovery model provides facilities for advanced data structures for the tuple storage. It also provides the possibility to avoid saving logical UNDO information in the log. Since we are using a logical fragment log, it is necessary to have some UNDO information to ensure that data structures are consistent when the fragment log is executed. Only a small physiological UNDO log is needed in connection with the logical fragment log which is a REDO log.

In most main-memory databases it is common to use a REDO-only log [DALI97], [DBS91], [DBN94]. When a record is updated, a copy of the record is created. The copy is destroyed when the update is committed. This is possible since most main-memory databases use a transaction-consistent checkpoint. Thereby all data structures are intact after recovery. A common procedure for achieving the transaction-consistent checkpoint is to copy all committed records to disk pages. The data structures on disk do not necessarily look the same as the data structures used within main memory.

In disk databases it is most common to use fuzzy checkpoints [ARIES92]. This means that after recovery the internal data structures are not consistent. Thus the reference to the tuple copy might be pointing to the wrong place. Only references within a page can be trusted. Hence most disk-based databases use a UNDO/REDO logging scheme [ARIES92], [Hvasshovd96], [Informix94]. A possibility exists to use REDO-only log for disk-based databases. This requires that no uncommitted data is written to disk. This is undesirable since the buffer manager cannot choose to write pages to disk in a flexible way.

Our recovery scheme provides an action-consistent checkpoint. This was necessary to provide support of a logical fragment log. This was implemented by performing fuzzy checkpoints in conjunction with a physical UNDO-log. The physical UNDO-log makes it possible to restore all data structures that were present at the checkpoint time. The aim is to keep the UNDO log which is sent to disk as small as possible.

The use of action-consistent checkpoints has interesting implications on the data structures used for tuple storage. An implication is that we can also trust references between pages in the database. The reason for this is that we create a consistent copy of all the pages in the database. All pages have the value which they had at the time of the checkpoint. Also, action-consistent implies that no actions were ongoing and thereby there are no problems due to tuples being in the middle of an update. That we can trust references between pages implies two things:

- The same scheme used in main-memory based databases can also be used in disk-based databases. Thus we can have a REDO-only log and still have allow the buffer manager to write pages to disk in a flexible manner. It also implies that a new copy of changed parts of the tuple are stored in the database pages.
- The storage structure of a tuple can span many pages. Hence we can allow a totally flexible data structure of the tuple.

13.1.1 Copy of Tuple Parts

The various applications put different requirements on our tuples. Many applications in telecommunications use rather large records with many attributes. Many times the applications use a data model which is highly optimised for performance. Also, it is always desirable to build data structures which do not contain unnecessary known limitations. Thus we desire a very flexible data structure which also performs well in the simple cases.

This flexible structure does also means that we can avoid creating very large copies of tuples which only update small parts of the tuple. Only the affected parts need to be copied.

Our decision is then to store tuples in a data structure as shown in Figure 13-1.

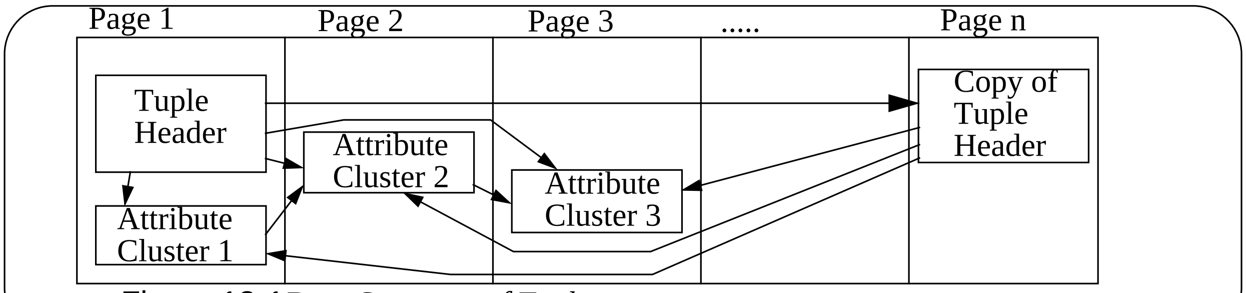


Figure 13-1*Data Structure of Tuples*

As shown in this figure the copy of the ***tuple header*** contains all the necessary UNDO information. In this case only the tuple header was updated and thus no copies of the three tuple clusters are needed in this transaction. As soon as the transaction is completed, the copy of the tuple header can be removed. This scheme also provides simple means for querying old versions of the tuples. In this case one keeps the copies until it is certain that they are not needed by any querying transaction.

An optimisation of this scheme is to allocate the copies in a small number of pages and use the same pages all the time. This means that accesses to those pages are likely to hit in some cache level. It also means that the references to the copies will use very little memory space.

The concept of using a B+-tree structure for tuple storage makes it possible to have arbitrarily large tuples. In Figure 13-2 the general structure of a tuple is shown. All the entities can contain part of the data of the tuple.

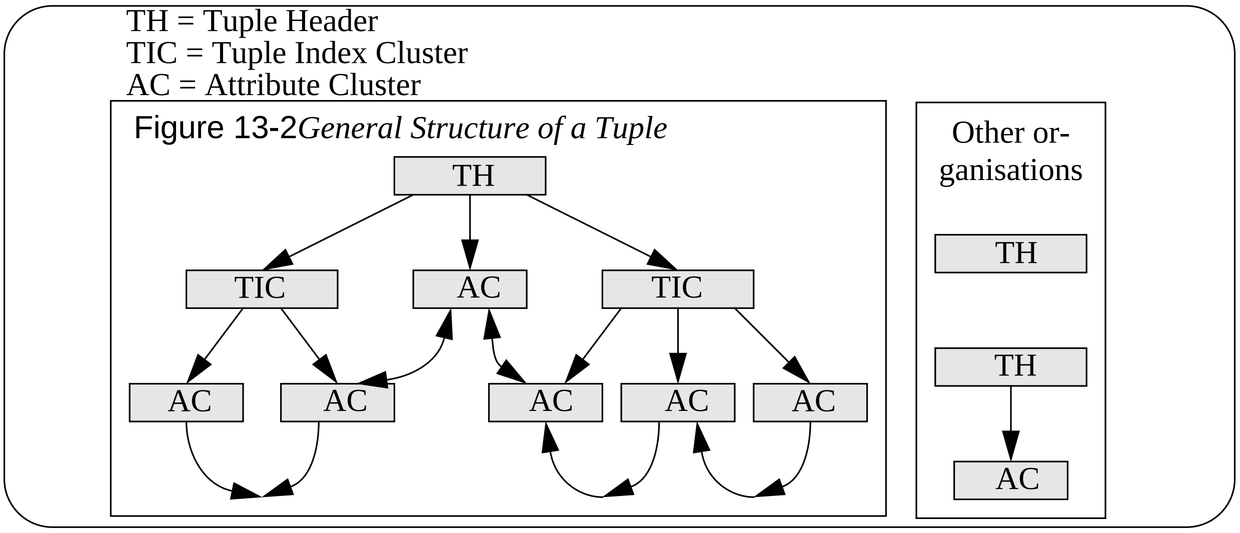


Figure 13-2*General Structure of a Tuple*

13.2 Data Structure within a Tuple

Internally within a tuple two basic organisations are possible [Gray93]. The first structure as shown in Figure 13-3 uses fixed positions of attributes. This means that an additional data structure is needed that specifies these fixed positions. Of course, attributes with variable length need special treatment in this scenario.

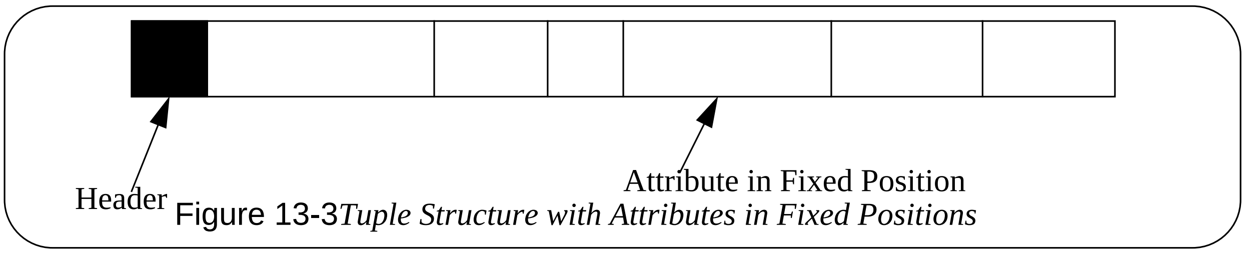


Figure 13-3*Tuple Structure with Attributes in Fixed Positions*

The second structure is shown in Figure 13-4. In this case each tuple contains a directory. Each attribute has a fixed position within this directory and each entry in the directory points to the attribute within the tuple. This means that no extra data structure is needed to show where the attributes are stored.

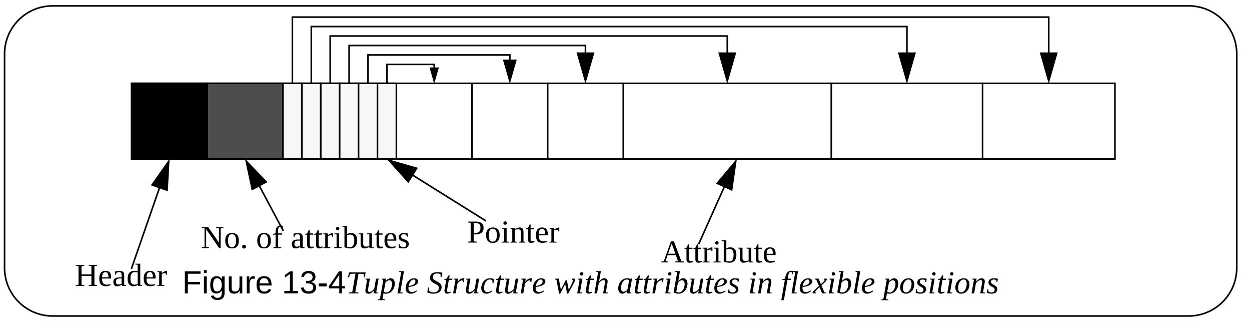


Figure 13-4*Tuple Structure with attributes in flexible positions*

The flexibility of the second data structure does, however, have a severe cost in memory space and also it introduces a possible second cache miss in reading the directory. Thus we will study more closely the scheme where attributes are given a fixed position.

13.2.1 Structure of a Tuple Header

To show an example of the tuple header is organised we will briefly describe its various fields. We will describe it in a pseudo-language.

```
TuplerHeaderType = {
HeaderType Int8, /* The Type of the Header */
SizeOfNULLAttributeField Int8, /* How many words are used to store NULL bits */
SizeOfAttributeFieldInTupleHeader Int8,
/* How many words are used to store Attribute Fields in the Tuple Header */
NumberOfReferencesInTupleHeader Int8,
/* How many references to other Tuple Index Clusters or Attribute Clusters are there in the Tuple Header */
TupleCopyReferenceType Int2, /* What type of Reference is used to the copy of the Tuple Header */
TupleCopyReference Reference ReferenceType
/* The reference to the copy of the Tuple Header, used during up-dates of the Tuple */
DynamicAttributeReferenceType Int2 /* Type of Reference to the Dy-namic Attributes */
DynamicAttributeReference ReferenceType, /* The reference to the dynamic attributes, this is an optional attribute */
SET OF {
NULLBitWord Int32, /* A word consisting of 32 NULL bits, optional attribute */
},
SET OF { SEQUENCE OF {
FirstAttributeOffset Int32, /* Offset of start word in Tuple Struc-ture for the cluster referenced */
ClusterReferenceType Int2, /* Type of Reference to the referred cluster */
ClusterReference ReferenceType /* The Reference to the referred Cluster */
}},
SET OF {
AttributeWord Int32 /* The Attribute Field in the Tuple Header */
}}
```

The other clusters and headers discussed in this chapter have similar structures. The most common tuple structures will have a very simple structure. A common tuple in the benchmarks is a tuple with a fixed size of 100 bytes. If it contains any attribute that can be NULL, there are three words of header information, reference to a copy of the tuple header and NULL bits. Then comes the 100 bytes of Attribute information. This means that the Tuple Header becomes 112 bytes and the overhead per tuple is 12 bytes. It is possible to decrease the overhead even more.

The basic idea of all tuple headers, tuple clusters, and attribute clusters is to use a cameolont data structure. Given the type the data structure is known. Thus the only common information in the data structure is the first byte that specifies the type.

13.2.2 Support of Various Data Types in a Tuple

There are many different data types that can be foreseen in a database. First there are fixed size attributes with various lengths (e.g. 1, 2, 4, 8, 16, 32, 64 and 128 bits). Then there are fixed arrays of fixed size attributes and there are arrays of fixed size attributes where the array is of variable length.

The fixed size attribute could be of different types as well. It could be an unsigned integer, signed integer, floating point number, decimal number, datetime, character and so forth. This type does only has a meaning to the database if operations are performed on those attributes. If they are simply stored and retrieved, the database does not really care about the attribute type.

The only complication of these data types are the variable arrays (~BLOB's). This type of attribute is needed to store multimedia objects as was shown in many of the applications described in this thesis.

These arrays cannot be put in a fixed start and end position. To allow for many of these we provide only a reference in the fixed part of the tuple. This reference has a fixed position in the tuple and the actual variable attribute is in the dynamic part of the tuple. Using a structure as in Figure 13-4 will cause problems when the size of the array are increasing/decreasing. It will be necessary to reallocate all data after the attribute when the size is increasing/decreasing. Instead we opt for a reference to a separate data structure for this particular variable array element. This structure reuses the same structure as a tuple as shown in Figure 13-5.

Those attributes described can be either NULL-able or NOT NULL-attributes. If they can be NULL, one bit is assigned as the NULL bit in the Tuple. However, even if they are NULL, they will take up space in the fixed area.

A table could be used to store attributes for classes in an object-oriented database. One example of this could be that a table is used to store the base class and all inherited classes. All classes should then have a common tuple key which is normally an object identifier (tuple identifier in our terminology). If this scenario is used, then different tuples in the same table will contain different attributes. Hence it is not a good idea to put all attributes in the fixed area. This would waste memory

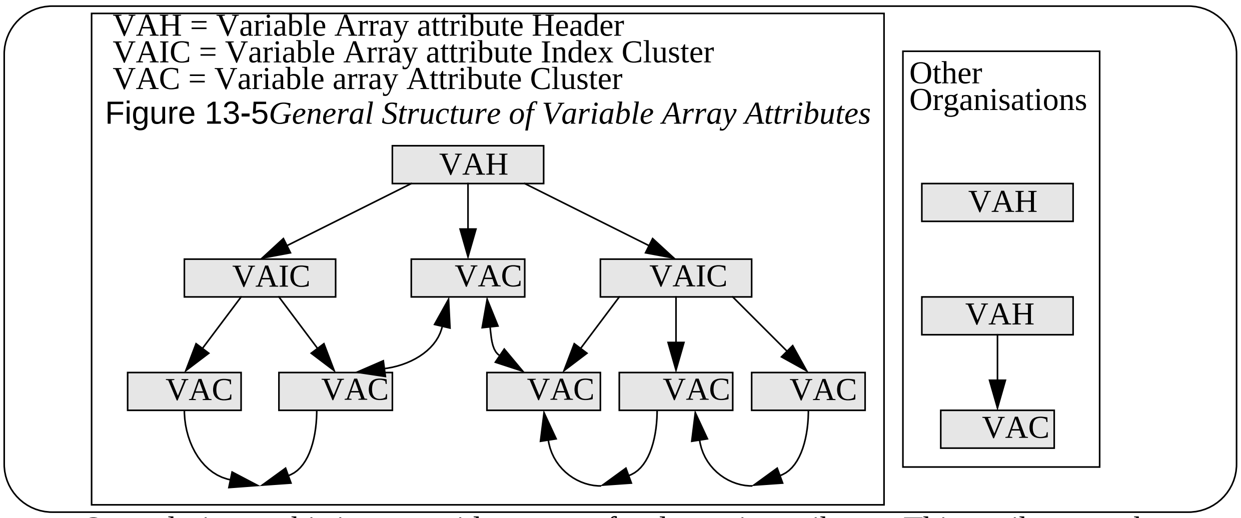
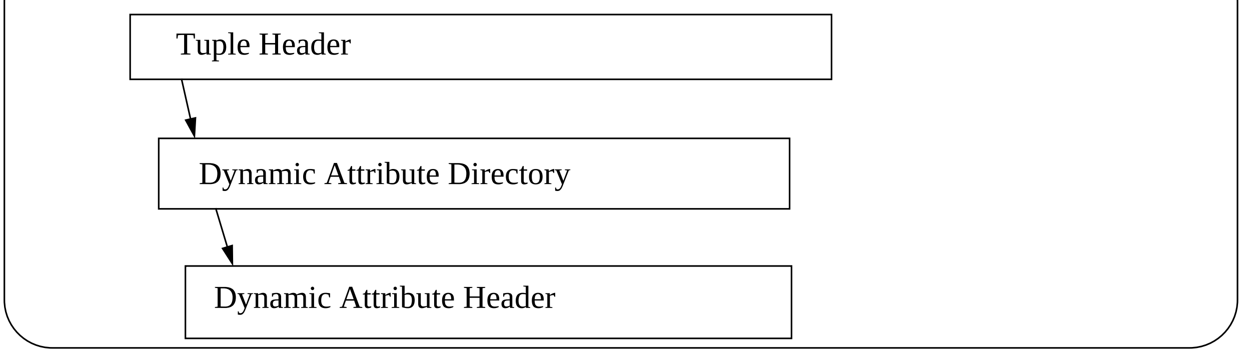


Figure 13-5*General Structure of Variable Array Attributes*
VAH = Variable Array attribute Header
VAIC = Variable Array attribute Index Cluster
VAC = Variable array Attribute Cluster
VAH
VAIC
VAC
VAIC
VAC
VAC
VAC
VAC
VAC
Other Organisations
VAH
VAH
VAC

space. Our solution to this is to provide support for dynamic attributes. This attribute can be a part of the tuple but need not be. All these attributes are put in a special dynamic part of the tuple. One such application that needs this functionality is the charging server described in Section 2.5.

The dynamic part of the tuple is referenced from the tuple header. The dynamic part is handled as a separate tuple basically. It has the same structure as the tuple with Tuple Header, Tuple Index Cluster and Attribute Clusters. Within these there could also be references to Variable Attribute Arrays. The difference is that the size of the fixed part is given in the dynamic part. The start of the dynamic part is a directory where each existing attribute has a pair installed. The pair shows the attribute identifier and the pointer within the fixed area. The structure is shown in Figure 13-6. The Dynamic Attribute Header and below is treated in the same manner as a tuple.

Figure 13-6*Handling of Dynamic Attributes*



Another type of structure that is also useful in representing unstructured information is to use untyped information. This is very useful in providing support for storage of home-pages which are unstructured. Another example is a table that stores transcribed records from a genealogical record. Since different genealogical records store information in different ways, it is useful to have the records specify the structure instead of the schema. It basically means that much of the schema information is integrated in the tuple. In [LOREL95] more information on the use of semistructured data can be found. By providing the possibility that an attribute could refer to a dynamic attribute directory, we can store very flexible tuples.

Actually one could also foresee that there will be special data structures and search structures that cannot be foreseen at the time when the database is implemented. To provide support of extensibility, the database should be equipped with a procedural language. Since this language should not need to know the structure of tuples, an interpreting language is very suitable for this task.

13.3 Page Structure

The structure of pages are another important item in designing a database with the above architecture. As the description of the applications have shown, the most common type of table is a table with many tuples where each tuple is rather small and consists mainly of fixed size attributes. Thereby it is important to optimise access for these although supporting a general data structure.

A tuple structure is accessed by a pair consisting of a page identifier and a page index. To allow for reorganisation of the pages the page index is not a direct pointer to the data; it is rather a pointer to a page directory [Gray93]. However to allow for fast access to tuple header we have two indexes on a page. One contains an index with pointers to tuple structures and the second directory contains tuple headers. The size of tuple headers depends on the table. A table of size 100 bytes should have a tuple header where all 100 bytes fit. A table with larger tuple sizes could use 128 bytes as a proper size of the tuple header.

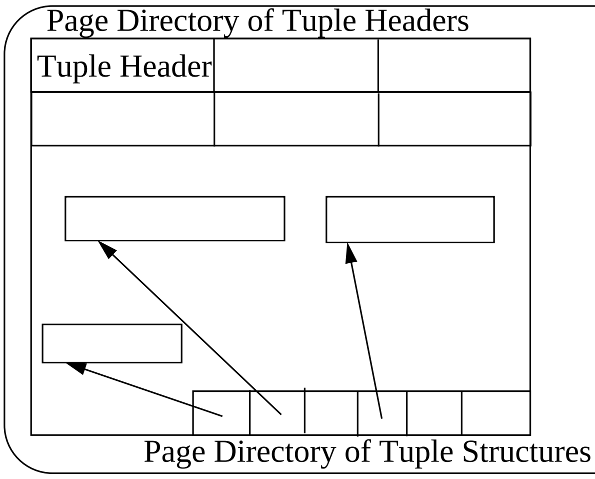

Page Directory of Tuple Headers
Tuple Header
Page Directory of Tuple Structures

Figure 13-7*Page Structure*

Which directory to use is shown by the last bit of the page index.

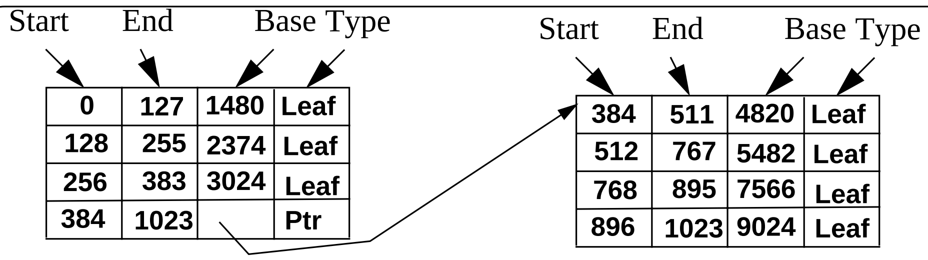
13.3.1 Page Identifiers

To find a page is also an issue. In a disk-based database it is not known which pages reside in main memory. Thereby a flexible data structure is needed to find out if page is in main memory. The most common is to use a hash table. This is one of the reasons why a disk-based database is slower if used as a main memory database. Each page access requires a search in a hash table.

Since we need to support both disk-based data and main memory based data we need different solutions on disk-based pages compared to main memory based pages. Disk-based pages uses a hash based structure as is normal. Main memory pages cannot use a page identifier that refers to a memory page. The reason is that we want the checkpoint to be location independent. This means that a node could restart a fragment from a checkpoint created by another node. Thus the same physical page can have been used by two fragments whose checkpoints were created at different nodes.

Hence we need a logical page identifier also for main memory pages. We assume that pages are allocated to fragments in this discussion. This must then be translated into a physical page identifier.

This mapping can be performed by a data structure similar to a B-tree. It consists of a mapping table with start and end of intervals of logical page identifiers. Then the base address of this interval specifies the number of the start page in the logical page interval. A small such table with four entries should cater for most needs since often the needed allocation size is known. However, at least two cases can complicate things. The first is if the memory is fragmented. Then it might not be possible to allocate such large chunks and a few smaller chunks must be allocated instead. The second problem is if the DBMS does not know the size to be allocated properly and starts allocating too small chunks. To avoid these problems giving any major impact the data structure is organised as a B-tree. Thus even a fragmented memory can be supported with good performance. The mapping table is shown in Figure 13-8.



Start	End	Base	Type
0	127	1480	Leaf
128	255	2374	Leaf
256	383	3024	Leaf
384	1023		Ptr

Start	End	Base	Type
384	511	4820	Leaf
512	767	5482	Leaf
768	895	7566	Leaf
896	1023	9024	Leaf

E.g. Logical Page ID = 158 => Physical Page ID = 2404
E.g. Logical Page ID = 552 => Physical Page ID = 5522

Figure 13-8*Mapping Logical Page Identifiers to Physical Page Identifiers for Main Memory Pages*

13.4 Disk-based Attributes and Main Memory based Attributes

The database can support both disk-based data and main memory based data. The selection of what data to put on disk and which data to put in main memory can be very flexible since the tuple structure can be very flexible. A page must be either disk-based or main memory based as our algorithms are performed. However, the choice can be on attribute level. A typical example where this is needed is the table to store files. The attribute which stores the file (a variable array of 8 bits) should be disk-based while some of the descriptive attributes might be better suited for storage in main memory.

This was useful in the News-on-Demand application, the Multimedia email application, and in the genealogy application as seen in Chapter 2.

13.5 Handling of the UNDO Log

As described in chapter 9 it is necessary to create a physiological UNDO log so that the logical fragment redo log can execute on a consistent data structure. This UNDO log is used together with a local checkpoint to create an action consistent checkpoint. This checkpoint must include both the tuple structures and the index structures. The UNDO log of the index structures was described in chapter 11.

The local checkpoint is used to execute the fragment redo log. This means that all committed updates after the local checkpoint is going to be performed by executing the fragment redo log. Also during crash recovery the data is not read. Thereby it is not necessary that the UNDO log restores the attribute values that were present when the local checkpoint was taken. It is only necessary to restore the data structure that was present when the local checkpoint was taken. This means that the size of the UNDO log can be diminished. This is especially interesting for disk-based data since these data need to use the UNDO log all the time.

Attribute values changed by transactions that abort shall be handled in a different manner. For these operations the UNDO log must restore the attribute values since there is no UNDO part of the fragment log.

To actually release all memory connected to a deleted tuple can be a long operation. Therefore one should write into the start checkpoint record a reference to all tuples that are currently deallocated. Otherwise the tuple storage might lose the memory of those tuples after a crash.

13.5.1 Changes of Tuple Structures

The various operations update, create and delete all change the data structure of tuples. However, this change is divided into the update phase and the commit phase (or the abort phase). Also updates of attributes of fixed size differ from the updates of attributes with variable size. We will go through these one by one:

1) Update Attribute of fixed size (or variable size without change of size) =>
- Create Tuple copy of updated parts of the tuple
- Change Attribute values

2) Commit Update Attribute of fixed size (or variable size without change of size) =>
- Delete Tuple copy of updated parts of the tuple

3) Abort Update Attribute of fixed size (or variable size without change of size) =>
- Copy Data from Tuple copy to Tuple original
- Delete Tuple copy of updated parts of the tuple

4) Update Attribute of variable size (increasing size) =>
- Create Tuple copy of updated parts of the tuple
- Allocate new parts to the tuple
- Update reference to new parts of the tuple

5) Commit Update Attribute of variable size (increasing size) =>
- Delete Tuple copy of updated parts of the tuple

6) Abort Update Attribute of variable size (increasing size) =>
- Copy Data from Tuple copy to Tuple original
- Deallocate new parts of the tuple
- Delete Tuple copy of updated parts of the tuple

7) Insert
- Allocate all parts of the tuple
- Write into attribute values

8) Commit Insert
- Do nothing

9) Abort Insert
- Deallocate all parts of the tuples

10) Delete
- Do nothing

11) Commit Delete
- Mark Tuple as Deleted
- Deallocate all parts of the tuple
- Unmark Tuple when all memory of the tuple has been released

12) Abort Delete
- Do nothing

As shown above it is not necessary to log updates of committed attribute values in the UNDO log. All changes of data structures and changes of references between data structures must be logged. This means that create tuple copy, delete tuple copy, allocate new parts of a tuple, deallocate parts of a tuple, and update reference to new parts of the tuple must be logged in the UNDO log. Marking and unmarking of the tuple as deleted needs to be logged in the UNDO log. Also deallocation of the deleted tuples must be logged.

Attribute values of aborted operations must be logged to ensure that the UNDO log can restore the committed values. This means that creation of tuple copies must also log the attribute values in the UNDO log. Also when copying tuple copies to the original it is necessary to write this into the UNDO log.

It is possible to decrease the size of the UNDO log even more, however. After performing the action create tuple copy and delete tuple copy belonging to the same operation nothing has changed in the data structure. Thus we have two possibilities to log those in a correct manner. Either log both of them or none. The same applies to create tuple copy and copy tuple copy to original tuple.

It is important when deleting entries from the UNDO log that it is performed in the correct order. The last added tuple part in an operation must be the first removed from the UNDO log.

The same reasoning holds true for allocate new parts of a tuple and deallocate parts of a tuple after aborting a transaction.

This means that normally only committed insert, delete operations, and update operations that change the size of variable attributes will be stored in the UNDO log.

Allocation and deallocation of blocks should also be logged in the UNDO log.

13.6 Special Handling of Very Large Attributes

The method we have described is capable of handling all types of data in a dynamic manner. It is, however, optimised for tuples which are not extremely large. The problem with very large attributes that reside on disk is that it is necessary to write those large blocks both into the fragment log and into the data pages. Sometimes data has to be written to the UNDO log as well. [Lehman89] reports an interesting solution to this problem:

For very large attributes the data part is located in a special region of the data pages. The references to these pages are located in a region which is handled according to the above scheme. The data pages are handled according to a shadow page scheme. Thus updates are not allowed to ovewrite the old values. Updates are always placed in a new area.

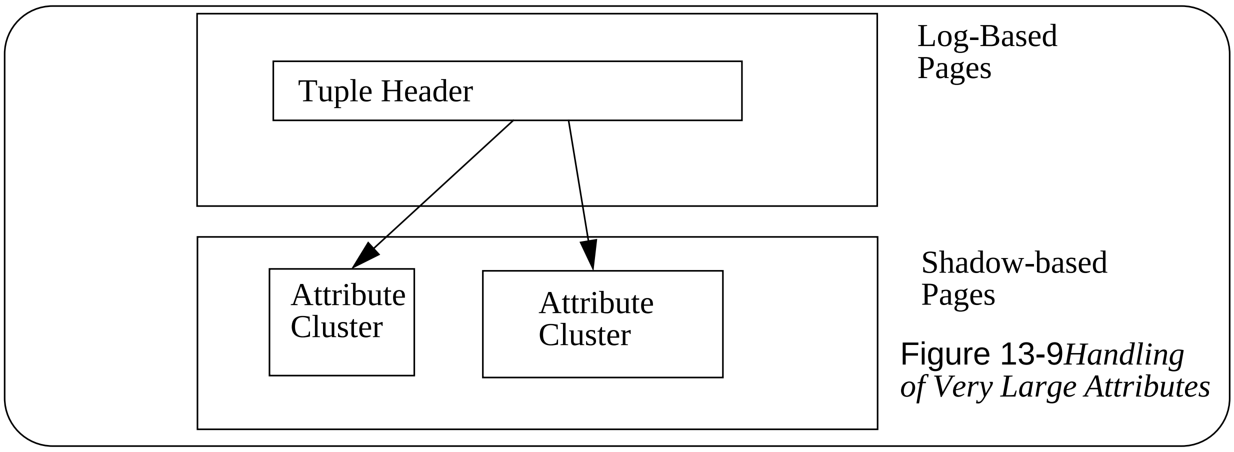


Figure 13-9*Handling of Very Large Attributes*

By using this scheme only a single write is needed. This write to disk must be performed before the transaction commits.

13.7 Related Work

[Gray93] contains a thorough description of common techniques used to store tuples in disk-based databases. In [Garcia92] an overview of main memory based databases shows that most of them use main memory data structures with a heavy use of pointers. It is also common to allow the application to access data directly in main memory [DALI97]. In our case we have a distributed database and the database and the application are not likely to reside on the same processor node. Hence it is better to use the database through an API as with disk-based databases.

[Informix94] describes a solution to a similar problem where a logical REDO/UNDO log was used in connection with a physical UNDO log. [Hvasshovd96] describes a scheme with a logical REDO/UNDO log in connection with a log that handles local events such as block splits and so forth.

No reports on using REDO-only log techniques with disk-based databases are known to the author. This is a new result of this thesis. Using B-tree structures for storing very large tuples are reported in [Gray93]. In [Lehman92] the choice of main memory or disk based is performed per table. [Sullivan92] reports the data structure used by POSTGRES. They use a structure where old copies of the attributes are kept in storage to provide queries on old versions. Instead of using an action consistent checkpoint they assume the use of force-at-commit (by using stable main memory). Thereby no log at all is needed.

13.8 Conclusion

The major achievement in this chapter is the development of a tuple structure which removes the need of storing UNDO information in the log. It also shows a very flexible structure of tuples. These structures are not assumed to be new but are rarely reported in the database literature.

However, the connection of these advanced data structures with the particular log strategy that is presented in this thesis is a step forward in designing a DBMS. It achieves very small volumes of the log compared to other strategies.

14 Modelling of a Parallel Data Server

In this section we will make some performance analysis of various ideas in the architecture of the Parallel Data Server. This is necessary in order to understand when the different features developed can be used. We will start by describing a computer architecture on which the performance modelling is based. Then we will analyse performance of the replication structure using stand-by nodes; we will analyse the new transaction protocol. We will also analyse the recovery performance in various situations. Both after a total crash and after failures of nodes such that on-line recovery can be performed.

The performance figures are based on early estimates on a prototype developed at Ericsson Utveckling AB; some figures are based on descriptions of Sun systems [SUN95]. Using these figures simulation has been performed to understand the interworking of various features.

14.1 Computer Architecture Used in Performance Modelling

The estimates in this chapter are based on execution times during usage of a cluster of Ultra 2 Servers equipped with two UltraSparc-I processors at 200 MHz. Each server contains two 2.1 GByte disks (recent releases of Ultra 2 Servers use 4.2 GByte disks and can have two UltraSparc-II processors at 300 MHz)

In [SUN95] a 2.1 GByte disk is said to have maximum data rate of 4 MBytes/sec. A new disk from Seagate reports average sustained data rates of 7.9 MBytes/sec with maximum data rate of almost 15 MBytes/sec [http.//www.seagate.com/]. It is also shown in [SUN95] that the actual data rate is very much dependent on the size of the buffers sent to the disk. With 8 kByte buffers only 432 kBytes/sec with a slightly faster disk. With 56 kByte buffers the data rate goes up to 1864 kBytes/sec. This is the largest size of the buffers that can be handled with Solaris [SUN95].

During recovery large buffers can be read from disk since all files are large sequential files. There are three types of files read during recovery. Data buffers, fragment logs, and the local logs. All these are read sequentially during recovery. Hence we can assume that with some knowledge of how to set parameters in Solaris it should be possible to reach 6 MBytes/sec on each disk during recovery.

If the database is used entirely for main-memory based data then the activities that are performed to disk are writing checkpoints (which writes data buffers), writing to fragment logs, and writing to local logs. Thus we can also here assume that 6 MBytes/sec can be reached. Disk-based data is, however, read and written continuously with a random pattern. Hence the data rate to the disk goes down. However, certain applications such as the charging server are create only and only use data for searches. Thus these disk accesses also become sequential.

Since we have two disks we can assume that we have a data rate of 12 MB/sec available.

14.2 Performance Comparisons

Some crude estimates on performance of the data server can be made to deduce the overhead of a stand-by replica and deduce decreased overhead in error cases. These estimates are based on a prototype developed at AXE Research & Development based on this thesis. The prototype is still in an early phase and the figures are therefore crude. In the figures we estimate the cost of sending a message at 10 microseconds. This is based on achievable figures using SCI and Memory Channel with a simple message interface. In [OMANG97] a report on implementation of message passing using SCI and Memory Channel is given. Even current versions of the hardware can achieve down to 8.9 microseconds delay of messages up to 63 bytes in size. This is actually the delay for the reader to receive the message. The sending processor node can continue with other activities even sooner than this. The execution time to receive a message is almost neglible since the message is read from local memory.

The figures relate to an execution of a simple write transaction that updates one record using a primary node, a backup node and a stand-by node. The complete phase is ignored in this discussion. It is handled as part of the background processing. The load on the transaction coordinator is 50 microseconds in the prepare phase, 20 microseconds in the commit phase and 20 microseconds when completing the commit phase and sending the response to the application. The load on the primary and backup node is 100 microseconds in the prepare phase and 50 microseconds in the commit phase. The stand-by node uses about 20 microseconds in the prepare and commit phases. This results in a total of 540 microseconds including execution time of messages. Of these 40 microseconds are executed by the stand-by node. The overhead caused by the stand-by node is thus 10% for write transactions. Read queries are not affected at all by the stand-by node. The complete phase has been ignored in those performance comparisons.

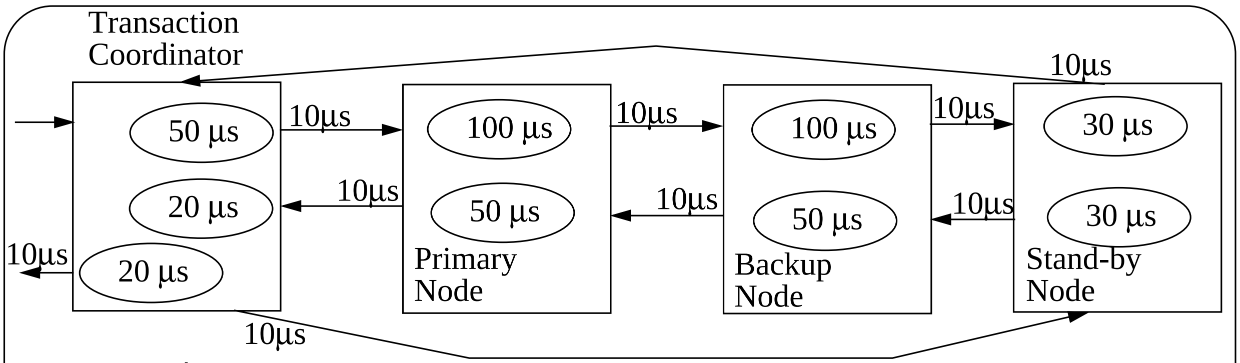


Figure 14-1*Costs Divided on the Participating Processor Nodes in a Normal Write Transaction*

These figures can also help us in estimating the delay in various scenarios. In this scenario we also need to consider read queries. We estimate the load to be 50 microseconds on the transaction coordinator and 100 microseconds on the primary node. In total 170 microseconds with communication costs added.

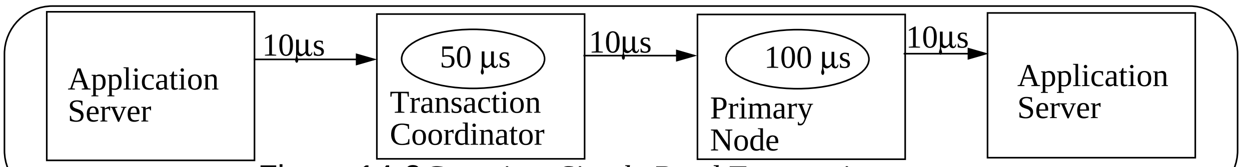


Figure 14-2*Costs in a Simple Read Transaction*

The mixture used here is 47% write transactions and 53% read transactions based on the UMTS benchmark. We assume here a simple model with a load of 75% in the processor nodes.

We have performed simulations based on those figures. These simulations measure the impact of stand-by nodes on the delay time and also, the impact on load and delays of using the new two-phase-commit protocol.

The simulation model consists of a set of 16 servers. It is assumed that communication bandwidth is not a bottleneck.

In the table below simulations using a stand-by node is shown. It is obvious from these figures that using a stand-by node does not affect the response time in any particularly negative way. We can also deduce that the new two-phase commit protocol gains about 6% in better performance by adding 0.3 ms delay to write transactions. This trade-off should certainly be acceptable.

Table 14-1

	New 2PC 75% load	Normal 2PC 75% load	Normal 2PC Same load as when New 2PC
Mean Delay of Read	414	437	512
Reads/sec	15405	14562	15405
Mean delay of Write	1662	1297	1544
Writes/sec	17372	16421	17372
99.9% confidence interval of Read Delay	1086	1184	1424
99.9% confidence interval of Write Delay	3543	3124	3776

In the table below figures when not using a stand-by node are shown. The performance gain by the new two-phase commit protocol is slightly higher. The delay does not go up more than 0.2 ms for write transactions in this scenario. The extra cost of using stand-by nodes is 13%.

Table 1:

	New 2PC 75% load	Normal 2PC 75% load	Normal 2PC Same load as when New 2PC
Mean Delay of Read	441	452	550
Reads/sec	17424	16352	17424
Mean delay of Write	1459	1267	1569
Writes/sec	19648	18440	19648
99.9% confidence interval of Read Delay	1182	1226	1555
99.9% confidence interval of Write Delay	3355	3118	4014

From this discussion we derive that if the customer is willing to pay for 13% extra capacity he can increase the reliability of the system and avoid having problems with the delay requirements in failure situations. The choice should, however, be the customer's and hence the replication of tables should be configurable in the interface of the data server.

It is likely that the stand-by replicas will be used in very large configurations where the probability of two replicas failing simultaneously rises.

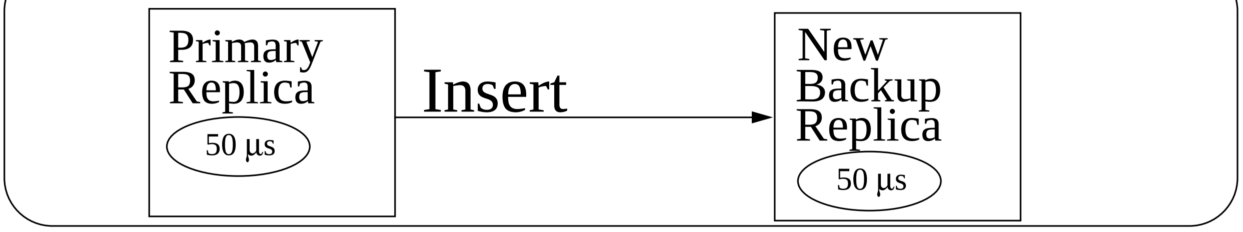
The comparison of the normal two-phase commit protocol and the linear two-phase commit protocol shows that the delay is normally smaller with the normal two-phase commit protocol. The conclusion is that the performance benefit of our version of the two-phase commit protocol does not cause any problems with the delay requirements.

Another benefit of our version of the two-phase commit protocol is that it simplifies handling of recovery. Since the messages arrive to the replicas in order, there are fewer possible states in which the failures can occur. This simplifies handling of node failures. As shown in chapter 8 it also simplifies integration of network redundancy in the two-phase commit protocol.

14.3 Performance of On-line Recovery

Performing On-line Recovery where a new backup replica is created is very similar to performing an insert operation. The main difference is that a great deal of protocol overhead can be avoided. Hence we assume that each tuple can be read and sent from the primary replica in 50 microseconds. The same time is assumed for reception of the tuple. This means that two nodes can recover about 2 MBytes/sec if they are both entirely working on recovery (~200 bytes per record). However, in a working system this is not possible. We assume that there exists a pool of processor nodes that are ready to use when a processor node fails. Thus the role of the primary replica is divided among the survivors. The processor of the new backup replicas are devoted to restarting the new backup replicas.

Figure 14-3*On-line Recovery Model*



With larger tuples it is possible to transport more than 2 MBytes/sec since the overhead per tuple is a major part of the cost.

If a machine contains 1.5 Gbytes data, then the time for recovery will be at least 12 minutes. Since the new backup replica will also be involved in transactions when it starts the new backup replicas, the time is longer. We estimate that it will take about 20 minutes to recover after a processor node failure.

This figure makes it obvious that using stand-by nodes is often worthwhile to avoid the risk of a system crash during this time.

14.4 Performance of Crash Recovery

In a complete restart after a crash it is necessary to read the entire database from disk and apply the logs. As described in section 9.2 the restart consists of three phases. The first phase reads all pages belonging to a fragment from the disk. This data was produced by a local checkpoint, therefore no data in this disk copy is older than the time between local checkpoints. The next step is to execute the local UNDO log from the most recent entry backwards until the time of the local checkpoint. The third step is then to execute the fragment log from a global checkpoint which was concluded before the local checkpoint was produced.

The major consumer of resources is to read the database fragment from disk. Using the computer architecture described above we have two disks that can perform reads in parallel.

Using the UMTS benchmark with 8 processor nodes we will have about 1.5 GBytes of data to restore in each processor node. This will take a little more than 2 minutes with 12 MBytes/sec of disk bandwidth. A possibility for shortening the time of crash recovery is to have more than two disks per processor node.

We assume here that only a primary replica is created in the first phase of crash recovery. After that production of a backup replica can start as soon as the primary replica is recovered. This has the benefit that the system can be restarted quicker.

The time to perform a local checkpoint is the same as the time to recover all data since it involves writing all data pages to disk. There is no use in trying to detect which pages are dirty or not. Since writing large buffers to disk is so much more efficient, it is better to write all data to disk at a local checkpoint. Hence we assume that local checkpoints are taken once every 3 minutes.

The fragment log must be read from the last completed global checkpoint before the last local checkpoint. Global checkpoints are taken more often than local checkpoints since achieving a global checkpoint is a small operation not involving any major work compared to producing a local checkpoint. Thus we can approximate that we need to read the fragment log since the last local checkpoint.

In a UMTS database we produced 20,000 transactions/sec and of those a little more than half were updates. If we assume that each log records consists of 300 bytes of data, then we need to log 3 MB/sec in the total system. This means about 240 MBytes of fragment log to handle during recovery. These are also divided on all processor nodes in the system. The time it takes to read these logs from disk is obviously almost neglible compared to the 10-12.5 Gbytes of data pages.

Applying the log uses the same mechanisms as performing on-line recovery.

The local logs for main memory data are even smaller than the fragment logs since they are only written between the start of the checkpoint and the writing of the page to disk. Since one fragment at a time is checkpointed, this time is only a few seconds which means only approximately 10 MBytes of local logs in the total system.

The time to apply those logs is then assumed to be neglible compared to all other activities during crash recovery.

14.5 Conclusion

From the discussions on performance of normal operations and recovery operations it can be seen that the requirements of the UMTS application can be met. The architecture has good possibilities for being useful for all the other applications, too. More research is needed on performance modelling of the system, particularly for the disk-based applications. Also more research on modelling of load balancing algorithms is a necessity to be operational in a real-time system such as a telecom network.

IV: Conclusions and Future Work

15 Summary

We will here summarize the most important work related to this thesis, summarize the conclusions and the contributions that this thesis has produced. Also future work in this area will be described.

15.1 Related Work

A number of projects have worked on developing either products or prototypes that have focused on databases in telecom applications. We will here report these and compare our approaches.

In Trondheim, Norway a team of database researchers has worked on these issues for a long time. They are now developing a product, ClustRa. This project is reported in [ClustRa95], [Hvasshovd92], [Hvasshovd96], [Hvasshovd93a], [Hvasshovd93b], [Tor95], [Brat96], [Tor96]. The author has had close contact with this team and has learned much from the ideas developed here and has extended some of these ideas.

ClustRa uses a primary and hot stand-by (called backup in this thesis) architecture. Both the primary and hot stand-by are involved in the transactions. The hot stand-by does, however, not update its data immediately. The log records are shipped to the hot stand-by and are updated in order on the hot stand-by but with a delay. Only the primary is used for reading, the hot stand-by is merely used for reliability reasons. The hot stand-by takes over in 300 ms [Tor96] after a failure of the primary node. When the primary node fails, the log of the hot stand-by is copied to another node to decrease the risk of system failure [ClustRa95].

The nodes communicate by message passing using ATM, FDDI or Ethernet. The cost of communication is very high; each message costs 0.8 ms with Sparc 5/85 compared to 0.1 ms using SCI with Sparc 5/85. The software has been developed with Unix processes with a self-developed thread package in it.

The parallel data server developed in this thesis also uses a primary and hot stand-by architecture. It is, however, extended with stand-by replicas. These ensure that not even double faults will crash the system. Also, backup replicas are updated within the transaction and are thereby immediately available to take over when the primary node has failed. It is even possible for the backup replica to take over without aborting any transactions. This is possible if updating transactions also set read locks in the backup replicas. Since the backup replicas are updated within the transaction, it is also possible to let simple read transactions access the backup replicas. Since simple read transactions are very common in telecom applications, this has great benefits in the algorithms for load balancing.

Major differences are found in the underlying software architecture. The major difference is the communication where the parallel data server assumes the use of message passing implemented through shared memory between the nodes (e.g. SCI or Memory Channel). This decreases the communication costs by a factor of about 10, especially for short messages that are very common in such applications. Also, the parallel data server uses a run-time system with a concurrent programming language which is highly modular and very efficient in all sorts of communication [Ronstrom97].

[ClustRa95] have reported the mean delay of a simple write transactions to 5.16 ms (ClustRa send acknowledgement to the application after prepare phase). The simulations in this thesis show that it should be possible to achieve down to 1.46 ms, even when waiting for the commit phase to complete before sending the acknowledgement. Even adding a stand-by node does not increase the delay more than 1.66 ms. It is, however, difficult to compare simulations based on uncertain values with real measurements. Furthermore the machines used by the ClustRa measurements were not as powerful as the one used for the parameters of the simulations.

Delays of simple reads in this architecture are even below 1 ms. This means that even if several database transactions are performed as part of one application message, the database will not cause any problems due to the delay.

The performance of ClustRa on simple write transactions was reported to be 2268 transactions per second on a 16-node cluster using HP 9000/735 99 Mhz workstations interconnected with a 100 Mb/s FDDI network [ClustRa95].

The conclusion is that the use of SCI decreases the delay even more. However, the figures achieved by ClustRa were excellent. The major difference is, however, that using SCI removes the performance impact of distribution almost completely. This is clearly visible in that we expect to achieve 32,777 UMTS transactions per second with a 4-node cluster of Ultra 2 Servers (two Ultra Sparc-I processors at a frequency of 200 Mhz). The major reason that we expect so good performance figures is the use of SCI in cooperation with a distributed real-time run-time system which is described in [Ronstrom97].

Additional features reported in this thesis are the on-line schema changes. In the parallel data server No UNDO/REDO logging is used in combination with a physiological UNDO scheme of the data structures. ClustRa and many other distributed databases use UNDO/REDO logging in combination with a local physical log of changes of data structures.

Prototype work and theoretical work at IBM has been of great value for the development of many ideas for dynamically creating new replicas [CHAMB92]. This work only deals with tuples accessible by a record identifier. It does not handle crash recovery, only on-line recovery. These ideas have been used in this thesis and extended with secondary indexes, foreign keys and schema changes. Also ideas from [CHAMB92] and [KARL96] have been further developed to achieve a copy fragment algorithm where only one tuple at a time is locked and shipped. [CHAMB92] locks the whole fragment and ships it and [KARL96] ships one bucket at a time.

[FRIED96] reports work on a prototype for Intelligent Network applications. It is optimised for simple read transactions. The communication is efficient using an IBM SP2 communication switch between 12 RS/6000 workstations. On top of the communication they use Horus, a group communication technology. Horus was not optimised for small messages and hence they had to group many messages together and send a message each 15 ms. They achieved a throughput of 1,670 reads/sec per processor node where a processor node was a RS/6000 with a processor frequency of 66 MHz. Some interesting ideas were used in this work where reads that waited for some time were simply retried on the other replica.

Our estimated results shows that 6,000 simple read transactions per second per processor node should be achievable with an Ultra 2 Server. This is achieved without packing messages and without an architecture which is highly optimised for simple reads.

Discovering a failed node took six seconds in [FRIED96], 100 ms in [ClustRa95]. In this work we assume that we will be able to discover failed nodes in 20-30 ms. [ClustRa95] achieve a take-over time of 0.3 seconds which is excellent. Our scheme performs the take-over in 1 ms after the failed node has been discovered. This take-over means that new transactions can start using the new scheme. Dealing with the transactions that are waiting for the failed node takes somewhat longer. However, it should not take more than 100 ms in any case.

Updating transactions can then simply wait for the fail message and retry as described in section 6.2.1. Simple read transactions can use the ideas from [FRIED96] but only once, otherwise these resends would be disastrous in an overload situation. Actually one should check that an overload situation does not exist to avoid less real work being performed in overload situations.

Another major source of inspiration for this work has been [KING91]. This work has been only slightly extended with a method of assigning the transaction coordinator in the backup system. The major work in this thesis has been to show how to integrate the work presented in [KING91] with the ideas used in the local replication protocols. Also, the work has been extended with on-line schema changes that also change the schema in the backup system.

Many ideas on distributed databases were developed in the 70's and the 80's. [Skeen85] is representative of these ideas. It describes a system with support of on-line recovery. It used a scheme where the log was applied to an old replica residing on disk. It does not handle reconfigurations of the replicas. In this thesis the scheme from [CHAMB92] is used where all updates are applied in the recovering fragment and simultaneously a copy process sends all tuples to the new backup replica. This simplifies the protocol to declare when a new replica is up-to-date. It assumes, however, that communication is cheap. If the fragment is disk based and very large and the recovering node has an old copy, then it is a good idea to use the protocol developed in [Skeen85] instead.

An internal database product within Ericsson has also been a major source of inspiration (DBS, Database Subsystem, used in AXE switches delivered by Ericsson). This database is specialised for use by applications written for the AXE switch. It uses a preprocessor to the compiler to implement very efficient read queries. Reading a tuple with five attributes in DBS can be performed in about 1-2 microseconds if the table is stored within the same module (this query uses a pointer and does not use an index, which is the normal case in telecom switches).

The work on linear hashing has been based on development of distributed linear hashing [Pram90], [Litwin93], [KARL96], [Litwin96]. This work has contributed to scalable distributed data structures. This thesis has extended this work by integrating it into a replicated database with transactions and also by enabling dynamic moving of fragments. Also, an algorithm that enables shipping one tuple at a time has been developed using LH*. The original data structure of linear hashing [Litwin80] has also been extended with the use of a double linear hashing internally in the processor node. This enables the number of cache misses to be normally only one to find an entry in the linear hashing index.

15.2 Achievements of This Thesis

15.2.1 Part 2

The achievement of this part is that it serves as a vision of what future telecom databases need to handle in the timeframe 1998-2005. It studies the impact on the network and its databases given that most people use mobile telephones, use email, read news by use of electronic means and that all payment is handled by on-line electronic systems. It also presents a study of a genealogy database as an example of an application with requirements on massive data storage and also a very complex schema which needs to be globally defined.

These studies provide important requirements that can be used in developing next generation database management systems. We also provide a new set of benchmarks to be used when developing these database management systems.

15.2.2 Part 3

One of the achievements of this part is a design that enables division of the telecom database in various servers. These are the application servers, where the application logic resides. The management servers where the operation and management of the applications in the telecom database is handled, which is an interface to the service and information creation environment. The database server has been divided in two servers, the data server and the query server. One of the reasons for this is that most applications of telecom databases require one simple query interface and one complex query interface. The complex query interface is used for management operations and also some application queries. The query server handles the complex query interface by using the simple query interface of the data server. The data is stored in the data server.

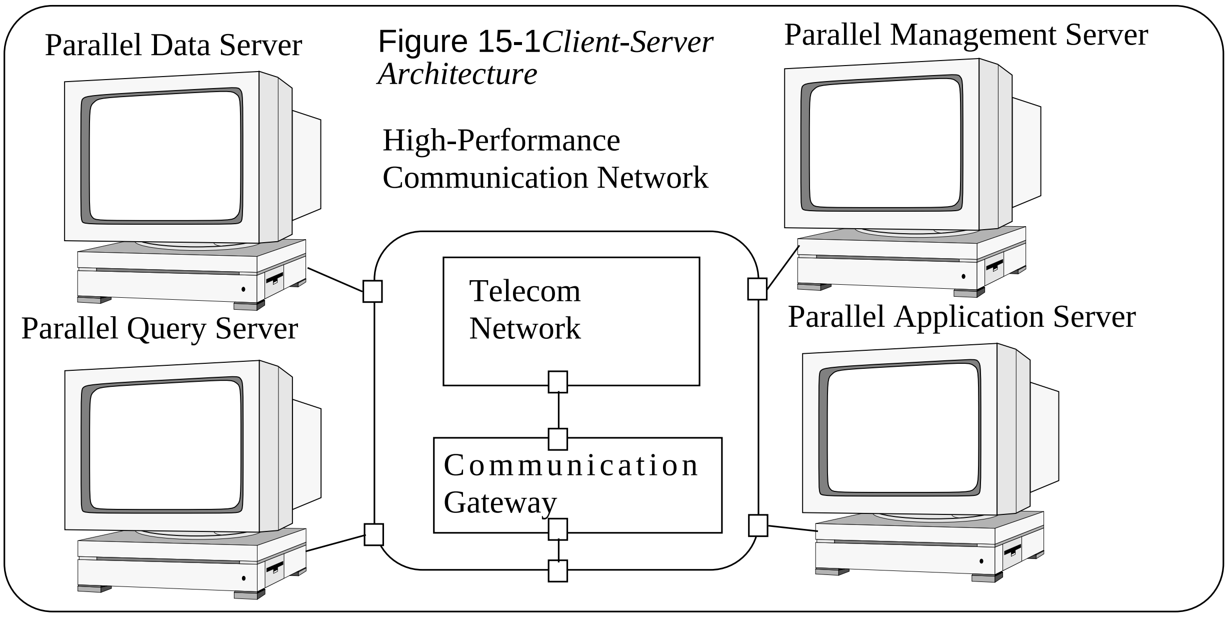

Parallel Data Server
Parallel Management Server
High-Performance Communication Network
Telecom Network
Parallel Query Server
Parallel Application Server
Communication Gateway

Figure 15-1*Client-Server Architecture*

The benefits of making this logical division also a physical division are several. First the data server is not damaged by malfunctioning applications and other clients, which means that the data server also has better control over its processor resources. This means that the Data Server can better handle the real-time requirements that the applications put on it. It can also better control the usage of memory spaces, cache memories and other essential processor resources. This makes the implementation of an efficient Data Server a much easier task. This architecture development is possible due to the development of high-performance communication networks (e.g. SCI) that make it possible to communicate without intrusion by the operating system. The communication is performed using a distributed shared memory handled by the hardware of the processor nodes and the SCI communication network.

By using this architecture the full responsibility of the reliability of the applications can be given to the parallel data server. Since this server implements reliability by using software methods, it is not necessary to use any hardware reliability. Normal cheap workstations/PCs can be configured in clusters to achieve scalable systems with high reliability.

Another achievement is the development of the Replication Algorithm using stand-by nodes. This development was enabled by the development of main-memory databases and by the development of nWAL algorithm [Hvasshovd92] and finally by new efficient communication technology (e.g. SCI). By using a primary and backup nodes that have an up-to-date copy of the database, very high availability of the database is achieved. By also sending log records to the stand-by nodes an extremely high reliability of the database is achieved. The stand-by nodes only consume small processor resources and small main memory resources, thus providing extra reliability at a cheap price.

The replication strategy is also combined with a new variant of the two-phase commit protocol which combines efficiency with good real-time properties. It is shown how this two-phase commit protocol can be combined with a protocol for network redundancy as developed in [KING91].

Index structures designed especially to be suitable in modern RISC processors were developed with index structures suitable for web addresses, file names, tuple identifiers and telephone numbers. Both a time-optimised index and a space-optimised one were developed. These keys are the dominating types of keys in telecom databases. This improves the search time in indexes. This is an important achievement since searching indexes is the most commonly used action in innermost parts of the Data Server. It is used in accessing the Dictionary several times and used to access the data.

The LH*LH data structure developed in [KARL96] has been extended into LH^3. It is shown how to integrate it in a parallel data server with full support of transaction and automatic recovery handling. The LH^3 data structure is an integral part of the recovery protocols. A new result in this area is a method of performing split and join of fragments even when transactions are executing and in a system where knowledge of fragment distribution is fully replicated.

The Http-tree is a new distributed B+tree that results in a compressed size of indexes with very large keys, such as http-addresses, file names and so forth. It uses the ordering of a B+tree and the repeatable directory structures often found in http-addresses and file names to compress the keys by a large factor. This provides better use of main memory and should also provide more efficient search algorithms. The efficiency should be better both because of a smaller working set because of compression and a smaller comparison key since only a delta change is needed to check per key.

The LH^3 structure is the most appropriate when performance of the index is most important. The Http-tree is useful for very large indexes. These indexes might not fit in main memory if LH^3 is used. By using the Http-tree they can fit into main memory and thus the performance is very much better since disk accesses are avoided.

A final a result in this thesis is the use of a new data structure in pages storing tuple information. This data structure provides the possibility of using No UNDO/REDO-logging although a totally flexible scheme of writing pages to disk is used. This is an important result of this research.

15.3 Future Work

This thesis has focused on developing the basic data access functions using a very reliable system. Much work has been performed on showing to perform many changes without interrupting system operation. Also basic data structures needed by the applications have been developed.

There are many openings for future work. One is to test the algorithms developed in this thesis in a real implementation. This provides an opening for testing all the benchmarks developed in this thesis. Work in this direction has already started as described in [Ronstrom97a]. Also more work is needed on load balancing of the system. This involves both simulation studies and analytical models that study the behaviour of the database. Simulations should also be done on the low level software of the database. This will find the real bottlenecks that lead to cache misses and too much processing.

Future work is also needed to extend the ideas into active databases. The active components must be maintained without sacrificing reliability. Examples of work in this area can be found in [Ceri96], and [STRIP96]. Another extension is to support unstructured data [LOREL95]. The data structures developed for the buffer manager can be extended to support unstructured data.

Abbreviations

AAL5: ATM Adaption Layer 5
ACID: Atomic, Consistent, Isolation, Durable
AMPS: American Mobile Phone System
ATM: Asynchronous Transfer Mode
AXE: Switch developed by Ericsson
BLOB: Binary Large Object
BSS: Base Station System
CCITT: Global Standardisation organisation
CLR: Compensating Log Record
D-AMPS: Digital AMPS
DAVIC: Organisation for standardising multimedia services
DBS: Database Subsystem, part of AXE
DBMS: Database Management System
DECT: European System for Cordless Terminals
FDDI: Fiber Distributed Data Interface, LAN and WAN network
Ethernet: LAN network
GIF: Format for storing pictures in files
GSM: Groupe Special Mobile (European Mobile Telephone System)
HLR: Home Location Register. A node in a mobile telephone network which contains subscriber data.
HTTP: Hyper-Text Transfer Protocol
HyTime: Extension of SGML for real-time multimedia documents
IN: Intelligent Network
INAP: Intelligent Network Application Part, runs on top of TCAP
I/O: Input/Output
IP: Internet Protocol
ITU-T: New name on CCITT
JPEG: Compression standard for pictures
LAN: Local Area Network
LE: Local Exchange
LH: Linear Hashing
LH*: Distributed variant of Linear Hashing
LH*LH: Variant of LH* where Linear Hashing also used locally.
LH^3: Variant of LH and LH* developed in this thesis.
LSN: Log Sequence Number
MAP: Mobile Application Part, runs on top of TCAP
Memory Channel: Cluster Interconnect developed by Digital
MONET: Research project on UMTS within RACE
MPEG: Compression standard for video
MTP: Link layer and part of network layer in #7-stack
MTBF: Mean Time Between failure
MTTF: Mean Time To failure

MTTR: Mean Time To Repair
NMT: Nordic Mobile Telephone System
nWAL: neighbour WAL
OO7: Object Database Benchmark
PCS: American System for Mobile telecommunication
PDC: Japanese Mobile Telephone System
PDH: Plesichronous Digital Hierarchy
PHS: Japanese System for Cordless Terminals
Q.3: Standard protocol for communication for management services in telecom applications
RACE: European Research projects on telecommunications
RAID: Redundant Disk Arrays
RAM: Random Access Memory
RISC: Reduced Instruction Set Assembler
ROWAA: Read One Write All Available
SCCP: Part of Network Layer in #7-stack
SCI: Scalable Coherent Interface, Cluster Interconnect Standard that uses a shared memory model
SCP: Service Control Point. A physical node in an Intelligent Network.
SDH: Synchronous Digital Hierarchy
SDP: Service Data Point. A physical node in an Intelligent Network.
SGML: Standard Graphic Markup Language
SMTP: Email protocol
SQL: Standard Query Language (for Databases)
TACS: British Mobile Telephone Systems
TCAP: Transaction based protocol on top of #7-stack
TCP/IP: Safe transport protocol on top of IP
TE: Transit Exchange
TIFF: Format for storing documents with pictures in files
TPC-B: Database Benchmark
TPC-C: Database Benchmark
UDP: Unsafe transport protocol on top if IP
UMTS: Universal Mobile Telephone Systems. A third generation mobile telephone system developed in Europe.
WAL: Write Ahead Log
WAN: Wide Area Network
WWW: World Wide Web
X.25: Network Protocol based on OSI (Open Systems Interconnect)

Glossary

Telecom Database:

A database used for telecom applications

System:

In this thesis a system is defined as a cluster of processor nodes that together acts as a parallel data server. Two systems can cooperate to provide network redundancy as primary system and backup system.

Processor Node:

A node that acts as one node in a cluster. Is normally a multi-processor machine (e.g. workstation)

Application:

The system that uses the database and a set of communication services to provide a service to end-users or other systems.

References

[AND95]T.E. Anderson, D.E. Culler, D.A. Patterson, *A Case for NOW (Networks of Workstations)*, IEEE Micro, Feb 1995.

[ARIES92]C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, P. Schwarz, *ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging*, ACM Transactions on Database Systems, vol. 17, no.1, march 1992, p. 94-162.

[ATTA98]A. Dabaghi, *Design of Access Manager in a Network Database*, Master Thesis at University of Stockholm, to be completed 1998.

[Bayer77]R. Bayer, K. Unterauer, *Prefix B-trees*, ACM Trans. on Database Systems, vol 2, no 1, p. 11-26, 1977.

[BERN93]A.J. Bernstein, P. M. Lewis, *Concurrency in Programming and Database Systems*, Jones and Bartlett and Publishers, ISBN 0-86720-205-X.

[Blakey86]M. Blakey, *Partially Informed Distributed Databases: Conceptual framework and knowledge model*, Tech.Rep no. 80, Monash University, Australia, Dec 1986.

[BHG87]P.A. Bernstein, V. Hadzilacos, N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley 1987, ISBN 0-201-10715-5.

[Brat96]S.E. Bratsberg, S-O Hvasshovd, Ø. Torbjørnsen, *Location and Replication Independent Recovery in a Highly Available Database*, submitted to VLDB'96.

[Carino92] F. Carino, P. Kostamaa, *Exegesis of DBC/1012 and P-90 - industrial supercomputer database machines*, Parallel Architecture and Languages Europe, Paris 1992.

[Ceri96]J. Widom, S. Ceri (editors), *Triggers and Rules for Advanced Database Processing*, Morgan Kaufmann 1996, ISBN 1-55860-304-2.

[CHAMB92]D.D. Chamberlin, F.B. Schmuck, *Dynamic Data Distribution in a Shared-Nothing Multiprocessor Data Store*, (VLDB 1992).

[Clamen94]S.M. Clamen, *Schema Evolution and Integration*, Distributed and Parallel Databases, vol 2, no. 1 Jan 1994, ISSN 0926-8782.

[ClustRa95]S.O. Hvasshovd, Ø. Torbjørnsen, S.E. Bratsberg, P. Holager, *The ClustRa Telecom Database: High Availability, High Throughput and Real-Time Response*. Proc. 21st VLDB (Very Large Data Bases), Zurich, Switzerland, Sep 1995.

[Comer79]D. Comer, *The Ubiquitos B-Tree*, ACM Computing Surveys, June 1979.

[CVET96]Z. Cvetanovic, D. Bhandarkar, *Performance Characterisation of the Alpha 21164 Microprocessor using TP and SPEC workloads*, The Second Int'l Symposium on High-Performance Computer Architecture (HPCA 96), p. 270-280, 3-7 Feb 1996, San Jose, California, IEEE.

[deWitt90] D. de Witt, S. Ghandeharizadeh, D.A. Schneider, A. Bricker, H-I Hisiao, R. Rasmussen, *The Gamma Database Machine Project*, IEEE Trans. on Knowledge and Data Eng. vol 2 no 1 March 1990.

[DALI94]H.V. Jagadish, D. Liuwen, R. Rastogi, A. Silberschatz, S. Sudarshan, *Dali: A High Performance Main Memory Storage Manager*, Proc' 20th VLDB Conf. Santiago, Chile Aug 1994.

[DALI97]P. Bohannon, D. Liuwen, R. Rastogi, S. Seshadri, A. Silberschatz, S. Sudarshan, *The Architecture of the Dali Main-Memory Storage Manager*, Memoranda from Lucent Technologies, http://www.bell-labs.com/project/dali/papers.html.

[Dayal90]U. Dayal, M. Hsu, R. Ladin, *Organizing Long-Running Activities with Triggers and Transactions*, ACM 089791-365-5/90/0005/0204.

[DBN94]Internal design documents within Ericsson describing a distributed main-memory database.

[DBS91]Internal design documents within Ericsson describing a database implemented in AXE, a switch developed by Ericsson.

[deWitt92] D. de Witt, J. Gray, *Parallel database systems: The future of high performance database systems*, Comm. of the ACM, June 1992.

[DOLPH96]*The Dolphin SCI Interconnect*, White paper, Feb 1996.

[ENG96]D.R. Engebretsen, D.M. Kuchta, R.C. Booth, J.D. Crow, W.G. Nation, *Parallel Fiber-Optic SCI Links*, IEEE Micro vol. 16 no. 1, p 20-26, Feb 1996.

[FELT96]E.W. Felten, R.D. Alpert, A. Bilas, M.A. Blumrich, D.W. Clark, S.N. Damianakis, C. Dubnicki, L. Iftode, K. Li, *Early Experience with Message-Passing on the SHRIMP Multicomputer*, The 23rd Ann. Int'l Symp. on Computer Architecture, p. 296-307.

[Ferrandina95]F. Ferrandina, T. Meyer, R. Zicari, G. Ferran, J. Madec, *Schema and Database Evolution in the O_2 Object Database System*, VLDB 1995, Zurich.

[FLAN96]J.K. Flanagan, B.E. Nelson, G. D. Thompson, *Transaction Processing Workloads - A Comparison to the SPEC Benchmarks using Memory Hierarchy Performance Studies*, Proc. of the Fourth Int'l Workshop on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '96), p. 152-156.

[FRIED96]R. Friedman, K. Birman, *Using Group Communication Technology to Implement a Reliable and Scalable Distributed IN Coprocessor*, submitted to "TINA" conference.

[Garcia87]H. Garcia-Molina, K. Salem, *SAGAS*, ACM 0-89791-236-5/87/0005/0249.

[Garcia92]H. Garcia-Molina, K. Salem, *Main Memory Database Systems: An Overview*, IEEE Transactions on Knowledge and Data Engineering, vol. 4, no. 6, Dec 1992.

[GILL96]R. B. Gillett, *Memory Channel Network for PCI*, IEEE Micro vol. 16 no. 1, p. 12-19, Feb 1996.

[GOLD96]G. Goldman, P. Tirumalai, *UltraSPARC-II: The Advancement of UltraComputing*, Digest of Papers, Compcon Spring '96, Feb 1996.

[Gottemukkala92]V. Gottemukkala, T. J. Lehman, *Locking and Latching in a Memory-Resident Database System*, VLDB Aug 1992.

[Gray79]J.N. Gray, *Notes on Data Base Operating Systems, in Operating Systems: An Advanced Course*, R. Bayer, R.M. Graham, G. Seegmuller (eds.), Springer Verlag, p. 393-481, 1979.

[Gray81]J.N. Gray, P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, F. Putzulo, I. Traiger, *The Recovery Manager of the System R Database Manager*, ACM Computing Surveys 13(2):223-242, June 1981.

[Gray91]J. Gray, *The Benchmark Handbook for Database and Transaction Processing Systems*, Morgan Kaufmann 1991, ISBN 1-55960-159-7.

[Gray93]J. Gray, A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann 1993.

[GS91]J. Gray, D. P. Siewiorek, *High-availability computer systems*, IEEE Computer, vol. 24, no. 9, page 39-48, Sep 1991.

[HELAL96]A.A. Helal, A.A. Heddaya, B.B. Bhargava, *Replication Techniques in Distributed Systems*, Kluwer 1996, ISBN 0-7923-9800-9.

[Herlihy92]M. Herlihy, J. E. B. Moss, *Transactional Memory: Architectural Support for Lock-Free Data Structures*, CRL 92/07, DEC Cambridge Research Lab, Dec 1992.

[Hvasshovd92]S-O Hvasshovd, *A Tuple Oriented Recovery Method for a continously available distributed DBMS on a shared-nothing computer*, Ph.D. Thesis, NTH Trondheim, 1992, ISBN 82-7119-373-2.

[Hvasshovd93a]S-O Hvasshovd, Ø. Torbjørnsen, *A Hardware Architecture for a Continously Available Shared-Nothing Parallel DBMS Based on ATM Technology*, SINTEF DELAB STF40 A93119, 10 Oct 1993.

[Hvasshovd93b]S-O Hvasshovd, Ø. Torbjørnsen, *A Software Architecture for a Continuosly Available Shared-Nothing Parallel DBMS Based on ATM Technology*, SINTEF DELAB STF40 A93118, 10 Oct 1993.

[Hvasshovd96]S-O Hvasshovd, *Recovery in Parallel Database Systems*, Vieweg, ISBN 3-528-05411-5.

[IBM96]G.H. Sockut, B.R. Iyer, *A Survey of Online Reorganization in IBM Products and Research*, Bulletin of the Technical Committee on Data Engineering, Jun 1996, vol 19, no.2, available on Internet.

[Informix94]*INFORMIX-OnLine Dynamic Server, Database Server*, Informix Software Inc., Dec 1994.

[Ingres93]*ASK OpenIngres Replicator User's Guide. Product documentation REP10-9(9)-16200*, The ASK Group, Inc. Dec 1993.

[Jaga95]H.V. Jagadish, I.S. Mumick, A. Silberschatz, *View Maintenance Issues for the Chronicle Data Model*, PODS'95.

[KARL96]J.S. Karlsson, W. Litwin, T. Risch, *LH*LH: A Scalable High Performance Data Structure for Switched Multicomputers*, 5th Int'l Conf on Extending Database Technology (EDBT '96).

[Kim88]W. Kim, H.-T. Chou, *Versions of Schema for Object-Oriented Databases*, VLDB 1988, Los Angeles.

[KING91]R.P. King, N. Halim, H. Garcia-Molina, C.A. Polyzios, *Management of a Remote Backup Copy for Disaster Recovery*, ACM Trans. on Database Systems vol. 16, no. 2, Jun 1991, p. 338-368.

[LARS97]J. Larsson, Design of Tuple Manager, Master Thesis work at Royal Technical Institute (KTH) in Stockholm 1997.

[Larson88]P-Å Larson, *Dynamic Hash Tables*, Communications of the ACM, April 1988.

[LATA]*LATA Switching Systems Generic Requirements*, Bellcore Technical Reference TR-TSY-000517, Issue 3, March 1989, Sec. 17, Table 17.6-B: “Traffic Capacity and Environment.”.

[Lehman92]T.J. Lehman, E.J. Shekita, L-F Cabrera, *An Evaluation of Starburst’s Memory Resident Storage Component*, IEEE Transactions on Knowledge and Data Engineering, vol. 4, no. 6, Dec 1992.

[Lehman89]T.J. Lehman, B.G. Lindsay, *The Starburst Long Field Manager*, VLDB 1989, Amsterdam.

[Lerner90]B.S. Lerner, A.N. Habermann, *Beyond schema evolution to database reorganization*, Proc. ACM OOPSLA/ECOOP ‘90 p. 67-76 Ottawa, Canada, Oct 1990.

[Lerner94]B.S. Lerner, *Type Evolution Support for Complex Type Changes*, Technical Report UM-CS-94-71, University of Massachusetts, Amherst, 31 Oct 1994.

[Litwin80]W.Litwin, *Linear hashing: a new tool for file and tables addressing*, Reprinted from VLDB-80 in Readings in Database Systems, 2nd ed, M. Stonebraker (ed.), Morgan Kaufmann 1994.

[Litwin93]W. Litwin, M-A Neimat, D. A. Schneider, *LH* - Linear Hashing for Distributed Files*, ACM SIGMOD Int’l Conf. on Management of Data, 1993.

[Litwin96]W. Litwin, M-A Neimat, *High-Availability LH* Schemes with Mirroring.*

[LOREL95]S. Abiteboul, D. Quass, J. McHugh, J. Widom, J. Wiener, *The Lorel Query Language for Semistructured Data*, Technical Report from Department of Computer Science, Stanford University.

[MALIK97]S. Malik, An Efficient and Reliable Dynamic Data Distribution (ERD3) method, Master Thesis work at University of Uppsala 1997.

[MARK96]E.P. Markatos, M.G.H. Katevenis, *Telegraphos: High-Performance Networking for Parallel Processing on Workstation Clusters*, The Sec. Int’l Symp. on High-Performance Computer Architecture, p 144-153, Feb 1996, IEEE.

[MICL94]M. Larsson, *Network Aspects in UMTS*, Master Thesis work at Ericsson Telecom AB, 1994.

[MMDB92]IEEE Transactions on Knowledge and Data Engineering, vol 4, no. 6, Dec 1992. A special issue on Main Memory Databases.

[Mohan83]C. Mohan, *Efficient Commit Protocols for the Tree of Processes Model of Distributed Transactions*, Proc. 2nd ACM Symp. on Principles of Distributed Computing, p. 76-88, 1983.

[Mohan92]C. Mohan, *Supporting Very Large Tables*, Proc. 7th Brazilian Symp. on Database Systems, Porto Alegre, May 1992.

[Mohan92a]C. Mohan, I. Narang, *Algorithms for Creating Indexes for Very Large Tables Without Quiescing Updates*, ACM SIGMOD 1992.

[MONET053]R2066/OTE/MF4/DS/P/053/b1, *UMTS Charging and accounting algorithms*, RACE MONET deliverable, Dec 1994.

[MONET061]R2066/PTT NL/MF1/DS/P/061/b1, *Implementation Aspects of the UMTS Database,* RACE MONET deliverable, 30 May 1994.

[MONET075]R2066/ERA/NE2/DS/P/075/b1, *Evaluation of Network Architectures and UMTS Procedures,* RACE MONET deliverable, 31 Dec 1994.

[MONET099]R2066/PTT NL/MF/DS/P/099/b2, *Baseline Document on Functional Models,* RACE MONET II deliverable, 31 Dec 1995.

[MONET108]R2066/SEL/UNA3/DS/P/108/b1, *Distributed database for UMTS and integration with UPT*, RACE MONET II deliverable, 31 Dec 1995.

[MONET109]R2066/SEL/UNA3/DS/P/109/b1, *Performance Evaluation of Distributed Processing in UMTS*, RACE MONET II deliverable, 31 Dec 1995.

[NILS92]Rikard Nilsson, Ulf Markström, Leif Klöfver, *System For Changing Software during Computer Operation*, USA Patent No. 5,410,703, filed 1 Jul 1992.

[OMANG96]K. Omang, B. Parady, *A Performance Analysis of UltraSparc Multiprocessors connected by SCI*, Internet.

[Pax94a]V. Paxson, *Growth trends in Wide-Area TCP Connections*, Lawrence Berkeley Laboratory.

[Pax94b]V. Paxson, *Empirically-Derived Analytic Models of Wide-Area TCP Connections: Extended Report*, Lawrence Berkeley Laboratory.

[PAPA86]C.H. Papadimitrou, *The theory of Database Concurrency Control,* Computer Science Press 1986, ISBN 0-88175-027-1.

[PET95]R.J. Peters, M.T. Özsu, *Axiomatization of Dynamic Schema Evolution in Objectbases,* 11th Int'l Conf on Data Engineering, Mar 1995, p. 156-164.

[Pettersson93]M. Pettersson, *Main-Memory Linear Hashing - Some Enhancements of Larson's Algorithm*, University of Linköping, Sweden, March 1993.

[POLY94]C.A. Polyzois, H. Garcia-Molina, *Evaluation of Remote Backup Algorithms for Transaction-Processing Systems,* ACM Transactions on Database Systems, Sep 1994, p 423-449, vol 19, no. 3.

[Ronstrom93]M. Ronström, *Signal explosion due to mobility*, Nordiskt Teletrafikseminarium 1993, Aug 1993.

[Ronstrom97]M. Ronström, *A Portable and Distributed Real-time Run-time System*, Internal Ericsson document.

[Ronstrom97a]M. Ronström, *The NDB Cluster, A Parallel Data Server for Telecommunications Applications,* Ericsson Review 1997, no. 4

[RRDF93]*Remote Recovery Data Facility, Operations Guide, Version 2 Release 1*. Technical Report, E-Net Corporation, Nov 1993.

[RSR95]*IMS/ESA Administration Guide: System, Version 5, First Edition*. Technical Report SC26-8013-00, IBM, Apr 1995.

[Salzberg92]B. Salzberg, A. Dimock, *Principles of Transaction-Based On-Line Reorganization,* VLDB 1992, Vancouver.

[Skeen83]D. Skeen, M. Stonebraker, *A Formal Model of Crash Recovery in a Distributed System*, IEEE Trans. Software Engineering, May 1983, p. 219-228.

[Skeen85]D. Skeen, A. Chan, *The Reliability Subsystem of a Distributed Database Manager*, Technical Report CCA-85-02, Computer Corporation of America, 1985.

[Stonebraker81]M. Stonebraker, *Operating System Support for Database Management*, Communications of the ACM, vol 24, no 7, p: 412-418.

[STRIP95]B. Adelberg, H. Garcia-Molina, J. Widom, *The STRIP Rule System For Efficiently Maintaining Derived Data*, Technical Report from Department of Computer Science, Stanford University.

[Sullivan92]M.P. Sullivan, *System Support for Software Fault Tolerance in Highly Available Database Management Systems*, Ph.D Thesis.

[SUN95]A. Cockcroft, *SUN Performance and Tuning*, Sun Microsystems, ISBN 0-13-149642-5.

[Sybase95]*Replication Server Technical Publications*, Sybase Inc, Mar 1995.

[Sybase96]T.K. Rengarajan, L. Dimino, D. Chung, *Sybase System 11 Online Capabilities*, Bulletin of the Technical Committee on Data Engineering, Jun 1996, vol 19, no.2, available on Internet.

[Tandem91]J. Guerro, *Rdf: An overview*. Tandem Systems Review, 7(2), Oct 1991.

[Tandem96]J. Troisi, *NonStop SQL/MP Availability and Database Configuration Operations*, Bulletin of the Technical Committee on Data Engineering, Jun 1996, vol 19, no.2, available on Internet.

[Tor95]Ø. Torbjørnsen, *Multi-Site Declustering strategies for very high database service availability*, Ph.D. Thesis, University of Trondheim, 1995, ISBN 82-7119-759-2.

[Tor96]Ø. Torbjørnsen, *An overview of the ClustRa DBMS*, Report from Telenor FoU Trondheim.

[TREMB96]M. Tremblay, J.M. O'Connor, *UltraSparc-I: A Four-Issue Processor Supporting Multimedia*, IEEE Micro, vol 16 no 2, p. 42-50, Apr 1996.

[Tuite93]D. Tuite, *Cache architectures under pressure to match CPU performance*, Computer Design, March 1993.

[Valduriez93] P. Valduriez, *Parallel Database Systems: Open problems and new issues*, Distributed and Parallel Databases vol 1 no 2 April 1993, Kluwer.

[Vittal94]C. Vittal, M.T. Özsu, D. Szafron, G. El Medani, *The Logical Design of a Multimedia Database for a News-on-Demand Application*, TR 94-16, The University of Alberta, Dec 1994.

[Vittal95]C. Vittal, *An Object-Oriented Multimedia Database System for a News-on-Demand Application*, Technical Report TR 95-06, The University of Alberta, June 1995.

[ÖZSU91]M.T. Özsu, P. Valduriez, *Principles of Distributed Database Systems*, Prentice Hall 1991, ISBN 0-13-691643-0.