Linköping Studies in Science and Technology

Thesis No 733

Performance-Polymorphic Declarative Queries

by

Thomas Padron-McCarthy

Submitted to the School of Engineering at Linköping University in partial fulfillment of the requirements for the degree of Licentiate of Engineering

Department of Computer and Information Science
Linköping University
S-581 83 Linköping, Sweden

Linköping 1998

Performance-Polymorphic Declarative Queries

by

Thomas Padron-McCarthy

December 1998
ISBN 91-7219-354-9
Linköping Studies in Science and Technology
ISSN 0280-7971
Thesis No 733
LiU-Tek-Lic 1998:68

ABSTRACT

Performance polymorphism, where a system can select between several given implementations of the same conceptual operation, has been used in real-time programming languages, such as Flex. The contingency plans used in the active database system HiPAC is a related, but more limited, mechanism. We have introduced performance polymorphism into a declarative database query language. We have shown the feasibility of the concept by implementing a general, performance-polymorphic query optimizer. We show how performance-polymorphic queries are specified and optimized in our system. A number of applications for the technique are suggested.

This work was supported by NUTEK, the Swedish National Board for Industrial and Technical Development, as part of ISIS, the Competence Center for Integrated Systems for Control and Information.

Department of Computer and Information Science
Linköping University
S-581 83 Linköping
Sweden

Contents

Acknowledgments

My thanks to my advisor, professor Tore Risch, and also to the other members of the EDSLAB research group at the Linköping University who have provided valuable advice and discussions. Special thanks to Jonas S Karlsson (who thanked me in *his* lic thesis!) for discussions about hash tables, how to implement *alloca*, and other interesting topics.

This work was funded by NUTEK, the Swedish National Board for Industrial and Technical Development, as part of ISIS, the Competence Center for Integrated Systems for Control and Information.

Chapter 1

Introduction

This work is about database systems, and how to let such systems find optimal solutions to problems by making trade-offs between constrained resources.

Parts of the material have been presented in [37] and [38].

1.1 Overview of this thesis

Chapter 2 gives an introduction to some concepts within the fields of databases, real-time systems, and real-time databases. In section 2.1 we will examine some of the properties of a database management system that are of importance for this work, including active and object-oriented database systems. Section 2.2 will cover some relevant aspects of real-time systems, and in section 2.3 we look at the combination of the two concepts, i. e. real-time database systems. Section 2.4 is about active real-time database systems.

Chapter 3 explains the concept of performance polymorphism, and gives some definitions of the term. In chapter 4 we explore more in depth some related work that has been done using performance polymorphism, in a wide sense of the term. Then, in chapter 5, we will explain how our approach differs from this previous work.

Chapter 6 documents the performance-polymorphic query optimizer that we have implemented in the AMOS system.

Chapter 7 shows how this work can applied to some example application domains.

Chapter 8 suggests some future directions for this research, and chapter 9 gives some conclusions.

Chapter 2

Background

This thesis is intended to show how certain database technology can be used to solve a class of problems for, primarily, real-time applications. As a background, we present in this chapter some important aspects of database systems [13] [47] [11] and real-time systems [8] [50].

We concentrate on those aspects that we believe are important for the understanding of our work in performance polymorphism, so this chapter is not necessarily useful as a first introduction to any of the subjects covered.

2.1 Databases

A *database* is a collection of data. Typically, the amount of data is large, and it may also have a complicated structure. Other requirements may be added to the definition, such as the data being related, consistent, or stored in a computer system.

A *database management system* (DBMS) is a program or set of programs that manages these data. Typical DBMSs provide their users with powerful and flexible ways to define, store and retrieve the data. Sometimes the term *database* is used to refer to DBMSs.

The combination of a database and the DBMS that manages it can be called a *database system*. This term, like *database*, is also sometimes used to refer to DBMSs.

A *schema* is a description of the data that can be stored in the database. The schema is stored explicitly, along with the data in the database, and can be accessed and modified by the DBMS. While it is possible to have a collection of data without an explicitly stored schema, where the information about the data and its structure is instead hard-coded in, for example, a programming language, we believe that the existence of such an explicit schema is one of the most important traits of a database system.

A *data model* defines what kinds of schemas can be used. The most common data models in current research and development are the *relational* model and various *object-oriented* models.

An important ingredient in many DBMSs is a *declarative query language*, where the user can formulate *declarative queries*. A declarative query, or *query* for short, is an operation against a database that is formulated in such a declarative query language, which is a high-level programming language that permits the user to specify complex database operations in a concise manner. A well-known query language is *SQL*. Declarative queries are not directly executable, but must be translated, by the DBMS, into an executable, procedural, program, the *execution plan*. This process is known as *query optimization*, and is done by a DBMS subsystem called the *query optimizer*.

A DBMS will usually add some overhead, in memory usage, disk usage and execution time, compared to an application where the data management has been hand-coded in a traditional programming language such as C++ or Ada. In some cases, this overhead may be large. On the other hand, development time can be much shorter, since advanced functionality is already built in into the DBMS. Execution speeds may also be higher in the database solution, since the DBMS provides advanced data structures and execution modes that are difficult to implement well in a hand-coded application. Also, and perhaps more importantly, the DBMS allows for much greater flexibility than a hand-coded program. It is comparatively easy to change both the logical and the physical structure of the data, and to manipulate the data in new and unforeseen ways. An especially important factor for this power and flexibility is the presence of a declarative query language in the DBMS. Using the query language, it is possible to perform new types of operations on the data in the database, for example previously unexpected types of searches.

2.1.1 Query optimization

There are usually many possible execution plans for a given declarative query, and these plans can have widely varying performance. One execution plan can easily be several orders of magnitude faster than another. It is therefore important that the query optimizer finds a good (i. e. fast) plan, ideally the best.

A traditional database query optimizer [45] works with a single performance measure, the "cost", which usually reflects the expected execution time, which in a disk-based database is dominated by the number of disk accesses.

The optimizer uses an *optimization algorithm* to find an acceptable execution plan. In some cases, such as in a traditional disk-based database with few and well-known data types and operations on these data types, it may be sufficient to use some heuristics to order the steps in the execution plan. For example, for a query expressed in relational algebra, the main such heuristic rule is to reorder the query so that *select* and *project* operations are performed before *join* or other binary operations. The justification for this rule is that *select* and *project* typically decrease the size of the result, and will never increase its size. Therefore, the cost of the entire query execution will be lower.

For more complicated cases, and for more exact optimization, it is necessary to use an explicit *cost model* to estimate the cost of the execution plans. The cost model defines a cost for each step in the execution plan, and a way to calculate the cost for a complete (or partial) plan. In a traditional database the cost model is crude, and only specifies the relative merits of different execution plans, instead of the actual expected execution time. The optimizer then searches the space of possible execution plans, in different orders depending on the optimization algorithm.

With an *exhaustive method*, the optimizer will find the best plan, according to the cost model. This requires the optimizer to examine all possible execution plans, except those that can be guaranteed to be inferior. Typically *dynamic programming* is used, which builds a search tree of possible execution plans using a best-first search.

For complex queries, where the number of possible execution plans can be too large for an exhaustive search, a *heuristic method* can be used. A heuristic method uses some rule-of-thumb to concentrate the search to sets of plans that are likely to be good, and thus will not examine all possible plans. Such methods are not guaranteed to find the best plan, but will, in practice, find an acceptable plan. A *randomized method* [20] inserts an element of randomness, for example by using one or many random starting points for the search.

2.1.2 Active databases

Traditional database systems have been passive, i. e. they only execute queries or transactions that are explicitly submitted by a user or an application program. Sometimes it is necessary to monitor the data in the database, to detect certain situations, and to trigger a response in a timely manner. For example, an inventory control system may need to monitor the quantity of stock for the items in the database, and to detect when the quantity of some item falls below a certain value. Then, the system should initiate some steps to order additional items to fill up the inventory.

An *active database* (or rather, *active DBMS*) is a DBMS that can detect such conditions, and perform different kinds of actions in response ([26], chapter 21; [12]). Typically, *ECA rules* are used, where the user can specify an *E*vent, such as the deletion or insertion of data, a *C*ondition on the deleted or inserted data, and an *A*ction, which can consist of arbitrary operations on data. The DBMS is then responsible for monitoring changes in the database, and to execute the appropriate actions.

There is some support for active functionality in the SQL3 standard, and many modern commercial DBMSs now contain at least limited active functionality.

2.1.3 Object-oriented databases

Traditional database applications have been business-oriented, such as banking or inventory applications. Common to these applications have been large amounts of data with a relatively simple structure, and large numbers of relatively simple transactions.

In order for other areas of computing also to benefit from the flexibility and power of database systems, a new class of applications with different requirements has emerged. Among these are CAD (computer-aided design), CASE (computer-aided software engineering), office automation, and expert systems. These applications handle data with a more complex structure than in traditional databases. Transactions may also be longer and more complex.

A new class of DBMSs have been developed in order to meet these requirements. To model the complex and interrelated data, and the procedural data, of the new applications, they use *object-oriented* data models [9] [26].

One of the advantages of object-oriented DBMSs over, e. g., relational systems, is a decreased so-called *impedance mismatch*. An impedance mismatch means that the database uses another data model for the data than what is natural and efficient for the application program, so the application program has to translate back and forth between its own data representation and the one that the database system uses. Less impedance mismatch makes possible a closer integration between DBMS and application program, and a different and more efficient architecture of the database system.

What is sometimes called *first-generation* object-oriented databases, are systems based on a programming language, usually C++ or Smalltalk, which has been extended with database functionality, primarily persistence.

Second-generation object-oriented databases, also called *object-relational* databases, are instead based on a traditional DBMS, which is extended with object-oriented functionality. The important difference is that second-generation object-oriented DBMSs provide a better declarative query language.

2.1.4 Main-memory database systems

Traditionally, database systems have been disk-based. In such a system, all the data in the database is stored on disk, and retrieved into main memory only when it is needed. This is slow, and response times can be hard to predict, due to the effects of buffering and to the physics of disks.

Modern databases may be entirely stored in main memory, and only use disk for the purpose of recovery and persistence [18]. This is, today, easily feasible for 10-100 megabytes of data, and clearly possible even for one or a few gigabytes on a large, dedicated machine. (Numbers like these are growing rapidly. A popular model for this is *Moore's law*, which usually is taken to say that the capacity of computer systems roughly doubles in eighteen months.)

A main-memory database system typically shows much higher performance than a disk-based system. Main memory is typically 10000 times faster than a disk for processing a block (one or a few kilobytes) of data [52]. The difference is much bigger for accessing a single small object, since disks are block oriented and have a high, fixed cost per access that does not depend on the amount of data that is retrieved during the access. This fixed cost for retrieving a block of data is typically around 10 ms for modern disks (and this figure is *not* decreasing rapidly), while access to main memory can be around 10 ns. This means that for some applications with high performance requirements, either for average response times or for real-time (i. e., worst-case) response times, disk-based databases are simply not possible to use. (With disk striping, as used in some RAID architectures [47], part of this limitation of disks can be alleviated. The maximum data transfer rate can be increased, but the minimum access time is not affected.)

The lay-out of data, especially concerning locality and the possibility for sequential access, is much less critical in a main-memory system than in a disk-based one. Therefore, index structures that are simpler to implement and gives less overhead can be used. (On the other hand, if the computer uses a cache memory between main memory and the CPU, and this cache memory is much faster than the normal main memory, locality and sequential access may still have to be considered.)

Other overhead induced by the DBMS can also be lower. E. g., in some cases locking can be entirely avoided, because the system will never need to wait for a slow disk operation to complete [18]. Another example of reduced overhead can be that while a disk-based system uses a buffer manager, which makes it necessary to copy the data in one or more steps, a main-memory system might access the data directly, referring to it by its memory address.

One disadvantage with main-memory systems is that data in main memory is more vulnerable than disk-resident data to software and hardware errors. Main-memory systems will also still need to use disks for transaction processing purposes, for backup, logging and recovery.

2.2 Real-time systems

A *real-time system* [8] [50] is a system where the operations not only have correctness requirements, but also requirements on timeliness. For example, a task needs to be completed before a given deadline. Real-time systems are not restricted to applications where the time spans in question are short (fractions of a second), although many real-time applications fall in this domain, but also include applications with a longer time range (several seconds or much longer) [51]. The common denominator is that the system must be able to meet the timeliness constraints, either by predicting in advance how long operations will take, and scheduling them accordingly, or by performing some contingency action when a deadline cannot be met. The concept of a *deadline* is common in real-time systems, i. e. a time limit when an operation should be finished.

A real-time system is not the same thing as a high-performance or fast system, although in actual implementations a common way of guaranteeing response times is to simply have a system that is (one hopes) fast enough.

Sometimes a distinction is made between *hard*, *soft* and *firm* real-time systems. The distinction is made by looking at the usefulness of the result, and how this usefulness varies when the deadline is passed. A *hard* real-time system is one where the deadline absolutely must be met. Otherwise, e. g., the plane crashes. A *soft* system is one where the result may still be useful even if it arrives after the deadline. An example of a soft real-time system is a system that shows a film by retrieving individual frames form a repository of some kind, or by computing them. If a frame is not available when it should be presented, it may still be valuable to show it later. The movie will just freeze for a moment. A *firm* systems is one where the result will not be useful after the deadline, but no disaster will occur. An example of this can be a weather forecast.

2.3 Real-time database systems

In traditional database applications, real-time response has been of small importance. With respect to execution times, the focus has been on high through-put, meaning that *average* performance has been optimized instead of the execution time for an individual transaction, or the *worst-case* execution time for an individual transaction. Because of this, the real-time characteristics of a traditional DBMS can be totally unsatisfactory for applications that require a guaranteed response time.

A *real-time database management system* (RTDBMS) [42] [4] [3] is a DBMS that meets timeliness requirements, in addition to the traditional DBMS functionality. Alternatively, it can be defined as a real-time system that includes database capabilities, such as transaction management, index structures, and query capabilities. Despite the apparent similarity, in practice these two definitions are not co-extential, but differ in a way that is similar to what can be said about object-oriented databases. The approach of starting with the DBMS concept, and adding real-time functionality, tends to put emphasis on database facilities, such as transaction processing and defining a declarative query language as interface to the database, while the other approach favors a programming-language interface, typically using C or Ada.

For applications with extremely high requirements on performance, and that use very simple data, solutions in hardware may be necessary. For applications with slightly lower demands, and more complex data, software solutions that are hand-coded in a traditional programming language might be used. A database approach will probably be used for applications with larger amounts of more complex data, with higher requirements on flexibility, and with somewhat less extreme performance requirements.

The best choice for a real-time database systems is probably a main-memory database architecture, with disk-based solutions reserved for systems that use very large amounts of data and that have lower demands on performance and uniformity of response time.

2.4 Active real-time databases

The presence of active rules in a real-time database [1] [2] poses additional problems. Even if it is possible to guarantee response times in a system where activity in the DBMS is explicitly initiated by the user, it may be much more difficult to do so when the DBMS initiates actions on its own. Since these can happen at unpredictable times, and with unpredictable amounts of data, care must be taken to avoid that the execution of a triggered rule breaks a deadline.

One solution may be to define several different actions to be executed when a rule is triggered, and let the system choose among them. The early active database system HiPAC [14] provided for alternative actions, *contingency plans*, which could be chosen when there wasn't time to execute the normal action. This can be viewed as an early form of performance polymorphism, as described later in this thesis.

Chapter 3

Performance Polymorphism

In time-critical applications it is sometimes possible to find a simplified algorithm that can be used when there isn't enough time to run the normal algorithm. This simplified algorithm performs the same conceptual operation as the normal one, but in shorter time. The trade-off is that the result may be of a lower quality in some sense, for example with respect to precision, completeness, or data consistency.

As an example of an application, such different algorithms can be used in a control system that reads input from a slow physical sensor. If the control application doesn't always have time to wait for the next sensor reading, it could instead use an extrapolation of previous values. This will produce a value within the allowed time-frame, but the value may deviate from the actual physical value.

Another example is an iterative numerical computation that may be set to produce results with different precisions depending on the execution time spent on the computation. Thus we have a trade-off between time and precision, and this trade-off can be used in a time-critical application to find an acceptable result within the allowed time-span, instead of a more exact result that arrives too late.

The different implementations can be defined by the programmer, and under some circumstances it is possible for the system to find them automatically, such as is done in CASE-DB [19] [35] [34].

3.1 Contingency plans

A contingency plan is an alternative operation that can be performed when there is not time enough for the normal operation, or when some other condition makes the normal action impossible. The contingency plan may perform the same task as the normal action, but in a faster or cheaper way, and perhaps not as well. An example of this can be to use less precision in a calculation, or to use old data instead of collecting or calculating them. It is also possible for the contingency plan to do something completely different from the normal operation, such as to turn back and land the plane instead of flying to the destination.

Contingency plans were used in HiPAC [14], where the action part of an ECA rule could be specified to have an alternative, which was to be executed if enough time was not available for the normal action part.

3.2 Exceptions

Constraints, which put limits on the allowed performance of the query execution in terms of time, resource consumption, quality of the answer, etc., need to be satisfied and checked by the query execution system. These constraints may be explicitly stated by a user who writes a query, or they may originate from some other source.

The handling of constraints may consist of the system making a choice between alternative actions in order to satisfy the constraints, or that the system monitors the constraints in order to e. g. guarantee that appropriate action is taken before a deadline. Such handling can be done at different times. There are four important cases:

- At compile-time. At this time, the program and the structure of the data is known. In a database context, this means that the query and the schema is known. The data itself may not be fully known, or it may be different from the data that will be used in the actual execution. For example, the size of a relation may be unknown or different from what will be used when the query is run. In some cases it is therefore difficult or impossible to estimate execution time and other resource usages at compile-time, even with a perfect cost model. In other cases data statistics are available. Relational cost-based optimizers rely on this.

- Immediately before execution, that is, when a query has been submitted for execution, or an action has been triggered, but before execution has started. This is sometimes called the *activation phase*. At this time, there will probably be more knowledge available about the data. Database systems can keep track of the size and statistical distribution of data, so it may be possible to use the cost model to find a better estimate of the time and other resources that the query will need.

  The additional resource usage for the constraint checking and for the choice between options, may make this case unsuitable for some applications since these resources have to be added to the resources consumed by the query execution itself. E. g. in a real-time system with high performance requirements (i. e. which requires short response times), the additional delay may not be acceptable.

- During execution. While the execution of the query is in progress, it may be possible to keep track of resource usage by actual measurements. For real time, this simply means to keep track of elapsed time. Time-outs fall in this category. It may also be possible to compare the pre-execution cost estimate with the actual measured execution cost, either to check the validity of the estimate or to find new and improved cost estimates by feeding better estimates to the cost model than what was available before execution.

- After the execution has completed. For some applications, the actual cost of the query may not be available until after execution. One example of this is a real-time database system that runs on top of a non-real-time operating system, where other tasks or disk access may cause unpredictable delays. For a performance measure that is not increasing or decreasing monotonically, it may not even be possible to detect a constraint violation until after the entire execution has completed. It may not be pointless to detect a constraint violation at this late time, since some compensating action may be possible, or the cost model may be improved.

In HiPAC, it seems that the choice between the normal action and the contingency plan was performed before execution.

Other systems choose to continuously monitor the specified constraints, using an *exception* mechanism, which is familiar from the programming languages Ada and C++. One such system is the C++-based programming language Flex. When a constraint is broken, and this is detected by the system, we say that an exception is *raised* or *thrown*. This can be done either automatically by the system, or explicitly by the application program. When an exception has been raised, the normal execution stops, and the system instead executes an exception handler, that has been written by the application programmer. If no exception handler exists, execution is terminated with some form of error message.

3.3 Performance dimensions

Traditional database query optimizers use a single performance measure, the "cost", which usually reflects the expected execution time. The execution time is but one of many different resources that may be of interest, and that may therefore be used as performance measures. Other examples are memory usage, network communication, and monetary cost. When optimizing a query, we might want to keep track of all these, both to minimize resource usage and to enforce constraints on this usage. We can therefore view each performance measure as a *performance dimension*.

The examples of performance dimensions mentioned are resources consumed by the query execution. Another class of measures is connected with the result of the query, such as its precision, its quality or its size. We may want to enforce constraints, or optimize, these too, so we consider them also as performance dimensions.

Both the programming language Flex (see section 4.4 below) and the data model ROMPP (see section 4.5) use several performance dimensions. ROMPP has a mechanism where the application programmer can specify any number of performance dimensions.

In the most general environment, it should be possible for the user to specify new performance dimensions, and it should be possible to use any of the dimensions as the optimization objective. If the system allows the specification of constraints, it should be possible to put constraints on all performance dimensions,

3.4 Performance polymorphism

The term *performance polymorphism* refers to "a scheme were all the versions of a particular computation are identified as candidates for binding to a generic name. We call this technique *performance polymorphism* by analogy to the conventional polymorphism where different functions of the same name operate on different data types" [23]. A similar definition is given in [55].

The essence of the definition is that the system must be able to automatically select among several different given implementations, and that it should find the best, or at least an acceptable, trade-off between the different performance measures. This can be done either at run-time, or at compile-time. A mixture is also possible, where the selections that can be made early are made early, and the rest is deferred to run-time. The simplest solution for implementing this mixture is to let a programming-language compiler perform normal compiler optimizations, thereby eliminating some of the choices where the conditions are known at compile-time. This approach is used in Flex [23]. A more advanced solution could use partial evaluation techniques [21] or have explicit mechanisms to let the compiler leave some choices unmade in the generated execution plan, or generate several alternative execution plans [10].

While these definitions seem to be best suited for use in programming languages or in object-oriented databases, with named functions or methods, the concept could be extended to cover e. g. contingency plans in the ECA-rules of an active database system [14], and the query partitioning in CASE-DB [34]. However, in order to naturally capture this and other forms of polymorphism, an object-oriented data model is advantageous. Several real-time object-oriented data models have been defined, e. g. RTSORAC [41]. Some real-time systems combine performance polymorphism and object orientation, such as the Flex programming language and the ROMPP data model.

Chapter 4

Related Work on Performance Polymorphism

4.1 HiPAC

An early example of a mechanism where different implementations of an operation can be defined by the user, and then automatically chosen by the system, is the contingency plans for alternative executions of reactive rules in the active DBMS **HiPAC** [14], also discussed in some later publications, e. g. [6]. In the ECA rules described in the HiPAC project, more than one version of the action part could be specified. When the time constraints could not be met by the normal action of a rule, the system could instead choose an alternative, the *contingency plan*. While HiPAC provided a declarative query language, the use of different-performance implementations was limited to the ECA rules, and could not be used in other parts of the system. Contingency plans do not fall under the definition of performance polymorphism, but it is a related concept.

4.2 CASE-DB

CASE-DB [19] [35] [34] is a real-time relational prototype DBMS that permits the specification of time constraints for queries expressed in relational algebra.

Given a deadline, the system can automatically partition a query, and then does iterative improvement using these partitions, while handling the risk of over-spending its time budget. First a query is run on the smallest subset, giving a crude approximation in short time. Then a larger subset is used, and so on, until at the end all the data is used.

For aggregate queries, this can be done automatically. In this case, CASE-DB uses query approximation techniques where the result of the query is estimated using statistical estimators and sampling techniques.

For non-aggregate queries, this requires a previous, user-defined partitioning of the data. The user or the database administrator identifies the relations that are likely to be used in time-constrained queries, and (horizontally) divides each relation into three fragments: *required*, *strongly preferred*, and *preferred*.

The choice of relation fragments is completely based on the semantics of the application, and will change with each application. Therefore, this partitioning is entirely left to the user, and CASE-DB gives no help or guidelines on how to do the partitioning.

CASE-DB has no user-declared performance polymorphism, i. e. it is not possible to define multiple implementations of operations. The quality of the answer, and trade-offs between time and quality, is not discussed.

The value of a transaction is mentioned in [34], where there is a brief discussion on how to choose between several transactions that are competing for resources, where some of the transactions can be executed in different versions.

The preferred way to handle competing transactions, according to that same brief discussion, is to let each transaction specify its "optional" and "required" parts (subtransactions), in order to let the DBMS modify transactions (by downsizing them), and make sure that all or most of the transactions can complete within their deadlines.

4.3 Parachute queries

A heterogeneous database system, where a query can retrieve data from several different data sources, suffer from a common problem: if some sources are unavailable during query execution, these systems either silently ignore the unavailable source, or the entire query fails. In environments where there is a large probability that data sources become unavailable (such as the Internet), this behavior is not good enough.

A possible improvement is to generate a partial answer, based on the data that could be retrieved and processed. If such a partial answer is accompanied by a representation of the unfinished work to be done, it can be resubmitted to the system in order to generate the full answer. A secondary query that is designed to complete a partial answer is called a *parachute query.* [5]

This type of query modification is not based on any trade-off between e. g. quality or execution time. The result depends entirely on from which data sources that are available.

4.4 Data models

As mentioned above, an object-oriented data model is advantageous for expressing performance polymorphism. Several real-time object-oriented data models have been defined, e. g. **RTSORAC** [41].

Other real-time systems combine object orientation with a more explicit performance polymorphism, such as the Flex programming language [27] [23] [25] [32] [22] [24] in the Concord project [28] and the ROMPP data model [55] in the MDARTS project [33].

4.5 Flex

Flex is an experimental programming language based on C++, which has been extended with real-time functionality. It contains constructs for stating timing constraints, for performance polymorphism, and for imprecise computations. Flex also has the ability to measure the actual execution times, and analyze this data in order to improve the timing estimates.

4.5.1 Timing constraints in Flex

Flex incorporates primitives for specifying constraints on time and other resources [27] [24]. These constraints can be both absolute, and relative to other computations. Constraints are described using *constraint blocks*:

```
A: constraint block
    (start >= B.start + 5; duration <= 10; temperature <= 100)
~> { /* optional exception statements here */ }
{ /* block's statements here */ }
```

In this case, the constraint block is labeled "A", and the constraints say that this block is not allowed to start until 5 time units after the start of another block labeled "B", and that the duration of this block must be at most 10 time units. Also, the value of the variable "temperature" must at all times be less than 100. If a constraint is broken, and an optional exception handler exists, this exception handler is called.

Note that the specified constraints must be true at all times. There is no mechanism here for a database-type transaction (see e. g. [13] chapter 17) where the constraints are checked only at the end of the transaction. This is probably the right choice for checking the usage of resources, since we probably want such an exception to be raised when the problem occurs and not later, after the execution has completed.

Constraints can be created using the start time of an activation, the finish time, the duration, and the interval between starts of consecutive activations.

Flex handles two resources, *duration* (that is, real execution time) and *count* (the number of executions of a block). The authors state in [22] that "[w]e have not experimented with other resources, but it would be easy to model the amount of memory a program uses, the communications bandwidth it consumes, the number of processors it uses, and so on." Also, in the Concord project [28], which the Flex project is a part of, *precision* is considered as a resource.

The types of resources are built-in in the language, so, at least in this experimental version of Flex, there is no mechanism for an application programmer to declare additional resources.

4.5.2 Execution-time measurements and timing analysis in Flex

There are performance-analysis systems that use analytical methods to determine the expected or worst-case execution times of real-time tasks. Other systems rely on the programmer to supply the required data. Flex instead uses an empirical method, where actual execution times are measured and fed into a timing analyzer. It also incorporates programmer knowledge about timings, since it allows the programmer to state expected timings. [22]

The programmer inserts measuring directives in the program. An example given in [22] is a directive for measuring the execution time of a sorting function. The code might look like this:

```
void isort(int* a, int n)
#pragma measure mean duration defining
    A,B,C in (A*n + B) * n + C safety 2
{ /* function body's statements here */ }
```

isort is a sorting function, which sorts an array *a*, containing *n* integers. The *#pragma measure* directive causes the compiler to insert code that measures the execution time of each invocation of this function. *mean* in the measure directive indicates that we are interested in the mean resource usage, and not the worst case. *duration* is the resource. *A*, *B* and *C* are constants that will be calculated by the timing analyzer, and that appear in the expression *(A*n + B) * n + C*, which models the expected resource behavior.

When the measurement data has been collected, the programmer runs a timing-analysis program, which determines the best fit of the parameters, in this case the constants *A*, *B* and *C*.

The formula for calculating the expected execution time of the function *isort* can now be used, for example for choosing between different sorting functions using performance polymorphism (see below).

4.5.3 Imprecise computations in Flex

A program model that is discussed in the Flex literature [32] [25] is imprecise computations. With such a mechanism, an approximate value can be found when there is not time enough to find an exact result.

Flex contains a statement *impcall*, which can be used instead of a normal procedure call. The called procedure uses the normal *return* statement to return the final result, but before it does that it can use the *impreturn* statement to make a tentative, imprecise result available. If the calculation has to be interrupted due to a timing constraint, the best of these tentative results is then used.

4.5.4 Performance polymorphism in Flex

In Flex it is possible to define several implementations of the same function, with different timing measures, and with different figures of merit. [23] [25]

Using the declaration

```
void sort(int* a, int n) perf_poly;
```

sort is declared to be a performance-polymorphic function, i. e. a name that can be bound to one of a number of different implementations, with different performance characteristics.

Given that declaration, different implementations can be provided:

```
provide isort for void sort(int* a, int n);
provide hsort for void sort(int* a, int n);
```

Given these implementations, and performance models calculated by timing measurements and timing analysis as described above, in a code fragment looking like the following,

```
A: constraint block (duration <= 100)
{
    #pragma objective minimize duration
    sort(some_array, array_length);
}
```

the system will, at run-time, choose an implementation that fits within the given time constraint. If several implementations are feasible (i. e., fit within the given time constraint), the one with lowest duration will be chosen. The directive *#pragma objective minimize duration* in this example tells the compiler to choose the implementation that, according to the model, has the smallest duration.

(The choice will not necessarily always be the same. A sorting algorithm that has the asymptotical time complexity $O(n \log n)$ will, for sufficiently large ns, always be faster than an algorithm with the time complexity $O(n^2)$, but for smaller ns, the $O(n^2)$ algorithm may be faster.)

An alternative way of choosing between different feasible implementations is for the programmer to specify a formula giving a *figure of merit* for each candidate implementation. The system will then choose the implementation with the highest merit.

The performance-polymorphic binding is done at run-time, but if some constraints and other data are known at compile-time, the compiler can perform normal compiler optimizations on the code for choice of implementation, and some of the binding decisions can therefore be done at compile-time.

[23] introduces the term performance polymorphism, and contains a discussion of the concept and how to implement it.

4.5.5 Trade-offs in performance polymorphism in Flex

Since Flex isn't a declarative query language, it does not do any query optimization. The emphasis is on a single choice between the elements in a set of performance-polymorphic candidates, to meet a set of constraints. However, the "allocation problem" is discussed: when more than one performance-polymorphic function is to be executed, more than one choice has to be made, and the trade-off between them has to be considered.

In [23], the authors show that, given some assumptions, the allocation problem is equivalent to a "knapsack sharing" problem, which can be solved in linear time. However, when we relax these assumptions, it becomes a combinatorial bin-packing problem. Here the authors suggest the use of approximate algorithms, or specific techniques that may solve a subset of the problem for a certain application.

In the end, Flex does not provide a general allocation mechanism. Instead, the application programmer has to provide this allocation himself[1], using one constraint block for each performance-polymorphic function call.

4.5.6 Combining resource measurements in Flex

If a program contains several performance-polymorphic function calls within the same constraint block, the system must have a way to combine the performance values for these calls. How this combination should be done, depends on the type of resource, and of the structure of the program.

[22] notes that most resource types fall into one of two categories: *time-like* and *space-like* resources.

For example, if two statements a and b are executed sequentially, the usage of a time-like resource (such as time) is the usage for a plus the usage for b. For a space-like resource (such as memory space) the usage is the maximum of the usage for a and the usage for b.

Other resources, such as precision, are more difficult to handle.

4.6 ROMPP

ROMPP (*Real-time Object Model with Performance Polymorphism*) [55] uses a solution that is similar to, and more general than, the one used in Flex, but from a database approach with an explicit schema that describes the data. ROMPP is a conceptual data model, and is not dependent on any specific implementation. [55] uses C++ in the examples.

ROMPP has a mechanism of envelope and letter classes to handle performance polymorphism in several specialization dimensions, not just (or even necessarily) time.

The envelope/letter structure used in ROMPP requires a pair of classes that are used in combination: an outer class (the envelope class) that provides the visible interface to the application programmer, and

[1] "Himself" is used gender-inclusively.

an inner class (the letter class) that buries implementation details. Several such letter classes may exist for each envelope class, and the system will then choose one of them to provide the functionality that is required by the interface in the envelope class. That is, letter classes are not explicitly accessed by the application developer. Instead, they are manipulated and selected by the system, based on the performance requirements.

In the following example (from [55]), an envelope class *Sensor* is declared. The declaration for *Sensor* contains two methods: *sample* and *process*. There are also two specialization dimensions: *STime*, which is the execution time for the method *sample*, and *PTime*, which is the execution time for the method *process*. In this envelope class, no implementations are given for the methods *sample* and *process*.

When an object of type *Sensor* is used in a program, the system will instead create an object of one of the two letter classes *Sensor1* and *Sensor2*. Each of these two classes provides implementations for the methods *sample* and *process*. The implementations have different performance, which is seen by the different values given to the specialization dimensions *STime* and *PTime*.

```
// @EC: Sensor
class Sensor {
public:
    Sensor();
    // @DIM: int sample() = STime
    virtual int sample();
    // @DIM: void process() = PTime
    virtual void process();
    ....
};

// @LC: Sensor1 OF Sensor
class Sensor1 : public Sensor {
    Sensor1();
    // @DIM: STime = 10 ms
    int sample();
    // @DIM: PTime = 6 ms
```

```
        void process();
        ....
    };

    // @LC: Sensor2 OF Sensor
    class Sensor2 : public Sensor {
        Sensor2();
        // @DIM: STime = 20 ms
        int sample();
        // @DIM: PTime = 3 ms
        void process();
        ....
    };
```

In the following declaration of another class, which contains as an object of type *Sensor* along with a statement of the performance requirements, an object of *Sensor1* will be constructed by the system. *Sensor1* satisfies the constraints on *STime* and *PTime*, while *Sensor2* does not. If, in the future, the application adjusts the timing requirements for the Sensor object to `"STime < 22 ms, PTime < 5 ms"`, the system will automatically select another *Sensor2* as the implementation object for *Sensor*. The process of rebinding is handled by the system, and is transparent to the application developer.

```
    class Foo {
    public:
        ....
    private:
        Sensor s("Time <= 15 ms, PTime < 7 ms");
        ....
    };
```

In contrast to Flex, it seems that all resolution of performance-polymorphic functions in ROMPP is intended to be done at compile-time.

ROMPP does not provide a declarative query language, and thus no optimization.

[55] mentions “value propagation” of the specialization dimensions, i. e. the combination of performance characteristics, as an “open question to be answered”.

[55] also contains an overview of previous work on performance polymorphism.

4.7 CHAOS

CHAOS [44] is a system for developing and executing real-time applications. CHAOS has support for different implementations of an operation, and for configuring and re-configuring an application by replacing these implementations. While this re-configuration can be done “dynamically” at run time, the system cannot do this automatically.

Chapter 5

Our Approach to Performance Polymorphism

In this chapter we describe our approach to performance polymorphism, how it differs from the work described in the previous section, and the advantages gained.

In short, our system handles declarative queries, and we allow the user to specify any number of performance dimensions. Constraints can be specified for any of the performance dimensions. and it is possible to use any of the dimensions as the optimization objective.

5.1 Performance dimensions

The concept of performance dimensions is introduced in section 3.3. Here we give some examples on performance dimensions that may be used by our optimizer.

Many real-time applications would use at least two dimensions: some kind of time dimension (for example worst-case time) and some kind of quality, e. g. value precision, and allow for a trade-off between these two.

5.1.1 Different kinds of time

Nothing prevents us from defining several different time measures, for example the expected execution time (which can be minimized by the optimizer), and the guaranteed worst-case execution time (which we might put constraints on in a hard real-time system).

In a soft or firm real-time system, where we can allow some transactions to miss their deadlines, we may want to use an "expected worst-case execution time". Most transactions will finish within this predicted execution time, but we allow a small number of transactions to be too late.

5.1.2 Quality

If the application should work with a quality dimension, the characteristics of this quality measure will probably vary between different application domains. Some general types of quality may be the precision of the answer, and the age of the data used in the calculation.

5.1.3 Bayesian probability as quality

For some applications, the quality in a quality/performance trade-off can be a Bayesian probability [39], for example the probability that the value of a boolean predicate is correct.

For some applications, such a one-dimensional probability measure is not sufficient. It may be necessary to handle false positives separately from false negatives.

If you are searching for sea-shells on the beach, you could examine each square foot of the beach to see if there are any shells there. If a typical beach consists of one million square feet, and there is a sea-shell on 0.01 percent of the squares, then there are 100 sea shells on the beach. Say that for each square you find a boolean value, indicating if there is believed to be sea-shell there. If there is a 1 percent chance that each value is wrong, you can expect to find 99 of the 100 sea shells, but these will be difficult to find among the 9999 false positives. Therefore, you would want the probability of a false positive to be lower than the probability of a false negative.

In medicine, this applies to clinical tests. The term *diagnostic sensitivity* refers to the conditional probability that a person having a disease will be correctly identified by a clinical test, i. e., the number of true positive results divided by the number of true positive and false negative results. The term *diagnostic specificity* refers to the conditional probability that a person not having a disease will be correctly identified by a clinical test, i. e., the number of true negative results divided by the number of true negative and false positive results. It is important to consider both the sensitivity and the specificity of clinical tests, both when determining which tests to use and how to interpret the results. [16]

In a database application, were external procedures in the database refer to clinical tests, queries could be optimized along performance dimensions of sensitivity, specificity, monetary cost, time required for the tests, and the number of tuples in the result.

5.2 Performance-polymorphic queries

By a *performance-polymorphic query* we mean a query that is formulated in a declarative query language, and that involves operations that may exist in several implementations with different performance.

A *performance-polymorphic query optimizer* is, in addition to the functionality of a traditional query optimizer, required to choose between the different implementations of each performance-polymorphic operation.

To the best of our knowledge, all previous work concerning object-oriented performance polymorphism where it is possible for the user to define multiple versions of a function or method with different performance, has concentrated on providing a programming-language interface. No declarative query language, and therefore no query optimization, has been provided.

While a programming-language interface may be sufficient for many applications, there are important advantages with a declarative query language, such as a simpler interface, increased data independence, and the possibility for better optimization than for hand-coded procedural programs, especially for large amounts of data and non-trivial schemas.

Since the system includes a cost model for the optimizer, it can use this to automatically estimate the execution time.

5.3 Optimization

Since query optimization is a potentially time-consuming task, it is usually important for a real-time database with a declarative query language to do the optimization at an early time, so the optimization time does not have to be included in the time constraints at execution time. Even if there exists applications where this is tolerable, for example when the ranges of the time in the time constraints are very large (minutes), optimization should be done early, if possible.

5.4 The cost model

The choice between different implementations of an action is done at compile-time, and is entirely based on the cost model. It is therefore very important that the predictions in the cost model accurately reflect the actual behavior of the system.

Chapter 6

The Implementation

In this chapter we describe how our approach to performance polymorphism, which itself is described in the previous chapter, has been implemented within our research platform, AMOS. We describe the optimization algorithm, and give some details of the implementation.

We have implemented a performance-polymorphic query optimizer within our research platform AMOS [31] [48] [49] which is a main-memory object-oriented active DBMS, with a relationally complete, object-oriented query language. The optimizer uses dynamic programming [45], which has been modified to handle operations that are polymorphic in any number of user-defined performance dimensions, e. g. time, precision, quality. The performance dimensions can have both numeric and symbolic values. Constraints can be stated on all performance dimensions, and any one of these can be used as the optimization objective.

6.1 AMOS

Our research platform AMOS [31] [48] [49] is a main-memory object-oriented active DBMS, with a relationally complete, object-oriented query language.

6.1.1 Query optimization in AMOS

In AMOS, a declarative query is first translated to an internal form, a domain calculus language called ObjectLog [31], which is a variant of Datalog where facts and Horn Clauses are augmented with type signatures. The predicates in this internal form are, at this stage, type-resolved but not binding-resolved and not ordered.

The optimizer then orders the predicates, while at the same time resolving their bindings. The result, the execution plan, will be run as a nested-loop join.

AMOS allows for the use of different optimization algorithms for ordering the predicates. Exhaustive and heuristic methods have been implemented.

6.2 The optimizer

The performance-polymorphic optimizer that has been implemented is based on the normal exhaustive optimizer, which uses dynamic programming.

6.3 Specifying performance dimensions in our implementation

For each performance dimension, the user has to supply the following characteristics:

- its name,
- its starting value, which is used as the value of this performance dimension for an empty execution plan, i. e. a partial plan to which no operations have been added yet,
- a default value, which is used when the value of this performance dimension for a certain operation is not specified,

- a combination function that combines the performance values of two operations, to be used when more steps are added to a partial execution plan,

- a comparison function that determines which of two values of this performance dimension is better,

- a switch indicating if the value of this performance dimension is monotonically worsening as the incomplete plan grows by adding operations to it, if it is improving, or if its monotonicity is unknown.

As an example, the performance measure *time* will typically have the starting value 0, no default value, a combination function that numerically adds two values, and will use the function *less-than* as a comparison function. Since an execution plan can never be made faster by adding operations, the value is monotonically worsening.

A typical use, in the context of a real-time system, is to define one performance measure called *time*, which expresses either the expected or the worst-case actual execution time, and another measure called *quality*, which expresses the quality or "goodness" of the result. The optimizer is then typically required to either choose the execution plan that gives a result with the highest possible quality within some given time limit, or to choose the execution plan with the fastest execution time, given some minimum quality.

Any number of performance measures can be defined.

6.4 The algorithm

During optimization, the space of possible execution plans is investigated by building a search tree using best-first search with respect to the performance measure that is used as objective. Each node in this search tree represents an execution plan for the declarative query that is being optimized. The leaf nodes represent complete, executable plans, while the internal nodes of the tree represent partial plans.

6.4.1 Traditional dynamic programming

In the traditional algorithm, as described in [45], the search tree is generated until a complete plan is found. Because of the best-first search, and because appending operations to an (incomplete) plan can never decrease the cost, it will not be possible to construct another complete plan with better cost than the one that was found. Other plans with the same cost can be found, though, but since there is no reason to prefer them, the first one is used.

6.4.2 Modifications in our algorithm

Our algorithm handles several performance dimensions, compared to the single "cost" in the traditional algorithm, but since one of the dimensions is chosen as the optimization objective, this modified algorithm is similar to the traditional one.

The differences are due to the presence of *constraints* on the values of the performance dimensions, and because the chosen optimization objective may not always be monotonically worsening.

6.5 Pseudo-code for the optimizer

This section contains (somewhat simplified) pseudo-code for the performance-polymorphic query optimizer. The actual Lisp source code is available on request.

6.5.1 Initial values

When the optimization procedure is called, the variables *unresolved-query-predicates*, *performance-dimensions*, *optimization-constraints*, *optimization-objective*, and *performance-polymorphic-predicates* are assumed to be set to appropriate starting values.

The variable *unresolved-query-predicates* contains the list of predicates in the query. These need to be (1) ordered in an execution order, (2) type-resolved and (3) performance-resolved. Type-resolution means that a call to a function name F is replaced by a call to a specific implementation of F that is specialized for certain argument types. As an example, a call to a generic PRINT function might be replaced by a call to PRINT-INTEGER. Analogously, performance-resolution means that a call to a function name F is replaced by a call to a specific implementation of F with specific performance characteristics.

Performance-dimensions is a set of performance dimensions. Each such performance dimension contains its characteristics: its name, starting value, default value, combination function, comparison function and monotonicity.

Optimization-constraints is a set of constraints on the values of the performance dimensions, which have been specified for this query.

Optimization-objective is the performance dimension used as optimization objective for this query. The optimizer will try to find the best possible execution plan with respect to this dimension, given the constraints.

Performance-polymorphic-predicates is the set of all performance-polymorphic predicates. Each performance-polymorphic predicate is connected to the set of all its implementations. With each such implementation, the performance values along the different performance dimensions are stored. The query optimizer will, for each performance-polymorphic predicate that appears in a query, choose one of the existing implementations, depending on the optimization objective and the optimization constraints.

6.5.2 Some other variables

Partial-plans-queue is a best-first priority queue of partial plans, i. e. the "frontier" of the search tree. Each "plan" in this queue contains not only the actual partial plan, i. e. an ordered list of type- and performance-resolved function calls, but also the expected performance values after executing the existing part of the plan.

Best-complete-plan is the best complete plan that we have found so far, if any. If the optimization objective isn't monotonically worsening, we need to continue investigating even after finding the first complete plan.

6.5.3 The optimization procedure

```
BEGIN
  /* Input data */
  unresolved-query-predicates :=
    the (unordered and unresolved) predicates in the query;
  performance-dimensions :=
    all existing  performance dimensions, with their characteristics;
  performance-polymorphic-predicates :=
    the "virtual" predicates, with their implementations;
  optimization-objective :=
    the performance dimension to optimize;
  optimization-constraints :=
    constraints on the values of performance dimensions;

  /* Initialize local variables */
  partial-plans-queue := empty queue of partial plans;
  best-complete-plan := empty list of function calls;
  partial-plans-queue :=
    a list containing the root node, i. e. a single search tree node
    with starting values for all performance-dimensions;

  FOREVER DO BEGIN
    IF a "best-complete-plan" exists,
       AND "partial-plans-queue" is empty
    THEN RETURN "best-complete-plan";
                /* since it is the only possible solution */

    IF a "best-complete-plan" exists,
       AND "optimization-objective" is monotonically worsening
    THEN RETURN "best-complete-plan";
         /* since no other plan can be better */

    IF there is no "best-complete-plan",
       AND "partial-plans-queue" is empty
    THEN FAIL;
         /* since the "unresolved-query-predicates"
```

```
            couldn't be optimized */

    oldplan := the first element of "partial-plans-queue";
          /* i. e. the best partial plan so far,
             according to the "optimization-objective */

    remove that first element from "partial-plans-queue";

    IF "oldplan" is a complete plan,
       AND "optimization-objective" is monotonically worsening
    THEN RETURN "oldplan"; /* since no other plan can be better */

    oldrem := remaining predicates from "unresolved-query-predicates";
      /* i. e. those that haven't been used in "oldplan" */

    oldplan-performance :=
      the expected performance values after executing
      the partial plan "oldplan";

    FOR EACH remaining unused predicate expression,
             called "virt_pred_expr", IN "oldrem", DO BEGIN
      IF the function name in "virt_pred_expr" predicate expression
         is a placeholder for one or more performance-polymorphic
         function implementations,
      THEN implementations :=
             all the implementations of that function name;
      ELSE implementations := the function name itself;

      FOR EACH function implementation, called "implementation",
               IN "implementations", DO BEGIN
        IF the predicate can be executed at this point in the plan,
        THEN BEGIN
          newplan := "oldplan", extended with a call to
            the function "implementation";
          predicate-performance :=
            these performance values
            (or functions to calculate them)
          of this predicate
          for the different performance dimensions;
        newplan-performance :=
          empty set of performance dimension values;
        FOR EACH performance dimension, called "dim",
                 IN "predicate-performance" DO BEGIN
          new-value :=
            apply the "combine-function" for "dim" on the
            performance dimension value of "oldplan-performance"
            and "newplan-performance";
          add "new-value" to "newplan-performance";
        END /* for each "dim" */

        IF any values in "newplan-performance" breaks
           a constraint in "optimization-constraints" on any
           monotonically worsening performance dimension
        THEN throw away this partial plan;
        ELSE IF there is no "best-complete-plan",
                OR "newplan" is better than "best-complete-plan";
        THEN best-complete-plan := newplan;
        ELSE insert "newplan" into "partial-plans-queue";
      END /* if the predicate can be executed */
    END /* for each "implementation" */
  END /* for each "virt_pred_expr" */
 END /* forever */
END
```

6.6 A real-time telecom example

In this example we will show how the AMOS system translates a declarative query into an unoptimized execution plan, how the possible execution plans can be represented as a search tree, and how the query optimizer finds an optimal execution plan by partially constructing, and traversing, this tree.

A mobile telephone network consists of a number of base stations, each covering an certain area, and a number of mobile telephones. At all times, each base station needs to know which of the mobile telephones are present in the area it covers.

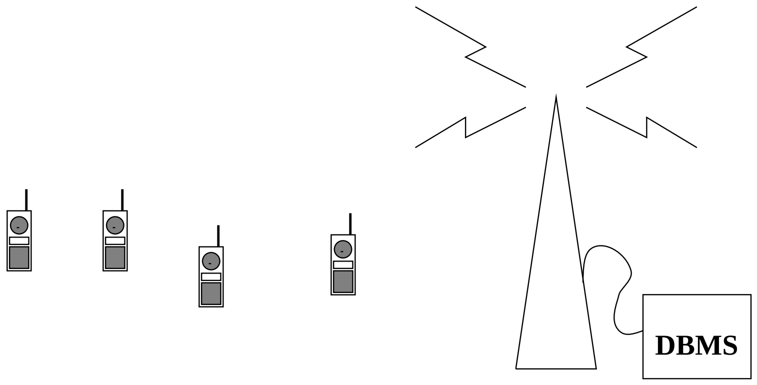

Figure 6.1: A cell in a mobile telephone network

We assume that the base station has the ability to find a certain mobile telephone by sending out a radio message that the telephone responds to, if that telephone is present in the area. We call this operation `present`.

We also assume that the base station can use a different operation, `signal_strength`, to determine the strength of the signal received from the telephone.

In this scenario, it can be useful to have multiple implementations of both these operations. For example, it will often be enough to know that a telephone was present in the area some time ago, and thus a previously stored value can be used, but at other times it will be necessary to be more certain, which requires actually sending a radio message to the mobile telephone to receive a reply. This is expensive from a battery consumption and frequency utilization point of view, in addition to the time required.

Therefore we assume that the conceptual operation `present` has been implemented in three different ways, each with a different performance measure for time (**t**) and quality (**q**):

- `p1`: the procedure `was_present` gets the previously stored value ($\mathbf{t} = 0$, $\mathbf{q} = 0.2$)

- p2: search_once sends one radio message ($\mathbf{t} = 0.2$, $\mathbf{q} = 0.6$)
- p3: search_many sends several radio messages ($\mathbf{t} = 3.0$, $\mathbf{q} = 0.99$)

We also assume that the conceptual operation signal_strength has been implemented in two different ways:

- s1: old_signal_strength gets the previously stored value ($\mathbf{t} = 0$, $\mathbf{q} = 0.2$)
- s2: measure_signal_strength measures the actual signal strength by sending a radio message and measuring the reply ($\mathbf{t} = 0.3$, $\mathbf{q} = 0.9$)

In this example, radio communication is very slow in comparison with internal data lookup and calculations. We can therefore assume that internal operations take time 0.

The quality measure $\mathbf{q}$ that is used here varies between 0 (lowest quality) and 1 (highest quality), and is combined using the function **MIN**. The starting value and the default value are both 1.

We also assume that the number of telephones in the database is 100, and that previous values of present and signal_strength have been stored for 10 of these (for use by p1 and s1).

As an example query, we need to find which of the mobile telephones that are present in the area but have a signal strength less then 25. Assuming that the data type telephone and the performance-polymorphic functions present and signal_strength have been defined, this query can be formulated using AMOS' query language, AMOSQL:

```
select p
for each telephone p
where present(p)
      and signal_strength(p) < 25
with t better than 2.0
optimize q;
```

(In the current implementation the performance specifications, i. e. "t better than 2.0" and "optimize q", are given through AMOS' Lisp interface.)

We want the query to be executed in at most 2 seconds, with the best possible quality within that time limit. We have therefore defined a constraint `t <= 2.0`. We have also declared the quality `q` as the optimization objective. This means that the optimizer will attempt to find the execution plan with the best quality that has an estimated execution time of less than 2 seconds.

The AMOSQL query is compiled into a domain calculus language called ObjectLog [31], which is a variant of Datalog where facts and Horn Clauses are augmented with type signatures:

$\texttt{answer}_{telephone}\texttt{(P) :-}$
    $\texttt{signal\_strength}_{telephone,integer}\texttt{(P, \_G1) \&}$
    $\texttt{present}_{telephone}\texttt{(P) \&}$
    $\texttt{<}_{object,object}\texttt{(\_G1, 25).}$

The functions in ObjectLog are not only performance-polymorphic, but also both type-polymorphic in the regular way (notice the subscripts) and *binding-polymorphic* (with different implementations depending on whether the arguments are bound or free).

This representation of the query is still declarative and not directly executable, since the order among the predicates and the bindings are not determined. The predicates will be re-ordered by the optimizer, and the polymorphic functions will be resolved. The result, the execution plan, will be run as a nested-loop join.

The query optimizer starts by creating an empty execution plan, which is used as root of the search tree (figure 6.2). The space of possible execution plans will then be traversed as the tree is constructed. In the figure, **plan** denotes the list of functions in the partial plan, **t** denotes the estimated execution time for the partial plan, and **q** denotes its quality. **Fanout** is the expected number of objects in the result of an operation when the plan is executed. The fanout is operator-dependent, and fanout is actually treated as yet another performance dimension in our system.

The optimizer has six choices as the first step in the execution plan: the operation `<` (less-than), the three implementations of `present`, and

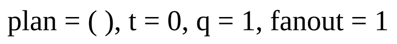

Figure 6.2: The root of the search tree

the two implementations of `signal_strength`. So far, no parameters are bound, and the operation `<` cannot be executed with two unbound parameters. All the implementations of `present` and `signal_strength` can be executed.

For each of the five partial plans that result from adding one of these operations to the empty plan, **t**, **q** and **fanout** are calculated by looking up the specified values of these performance dimensions for the operation, and calling the combination function for that performance dimension. Of the five plans, all but two will break the time constraint `t <= 2.0`. Those that don't are added to the search tree, which then consists of two partial plans (figure 6.3).

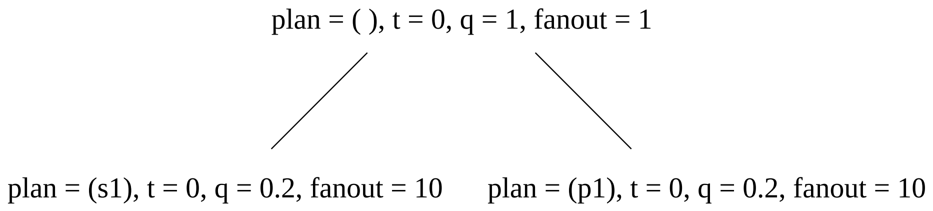
plan = ( ), t = 0, q = 1, fanout = 1
plan = (s1), t = 0, q = 0.2, fanout = 10
plan = (p1), t = 0, q = 0.2, fanout = 10

Figure 6.3: The search tree after the first iteration

In the next iteration we continue building on the partial plan with the best quality **q**, since this is the optimization objective. In this case both plans have the same quality, so it doesn't matter which one is chosen.

We take the plan that consists of the operation `s1`, and expand it. It already contains an implementation of `signal_strength`, so we can expand it with either the operation `<` (less-than) or one of the three implementations of `present`. `p3` would break the time constraint `t < 2`, so it is not used. The three new partial plans are added to the search tree (figure 6.4).

In the next iteration, the partial plan `(p1, s1)` is added (figure 6.5).

In the iteration after that, the partial plan `(s1, p2)` is expanded to `(s1, p2, <)`, which is a complete execution plan (figure 6.6).

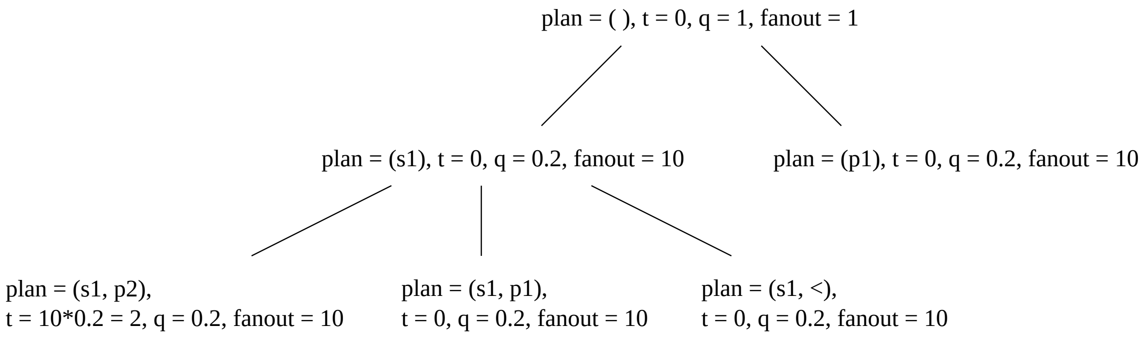
plan = ( ), t = 0, q = 1, fanout = 1
plan = (s1), t = 0, q = 0.2, fanout = 10
plan = (p1), t = 0, q = 0.2, fanout = 10
plan = (s1, p2),
t = 10*0.2 = 2, q = 0.2, fanout = 10
plan = (s1, p1),
t = 0, q = 0.2, fanout = 10
plan = (s1, <),
t = 0, q = 0.2, fanout = 10

Figure 6.4: The search tree after the second iteration

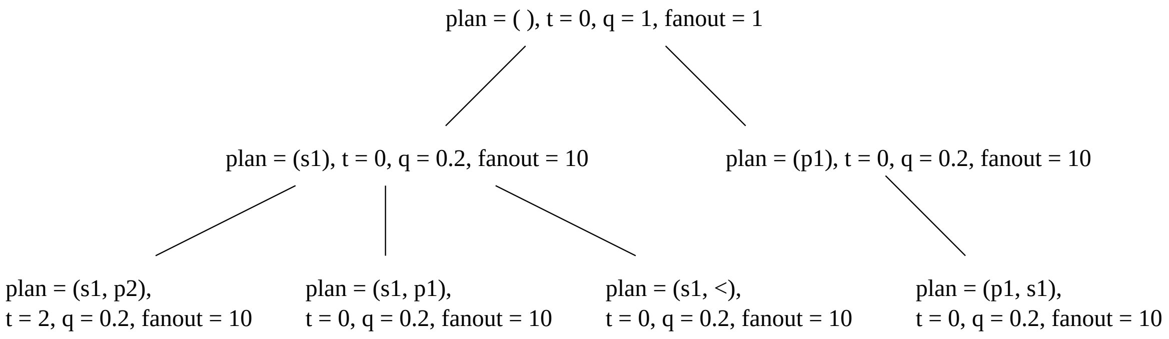
plan = ( ), t = 0, q = 1, fanout = 1
plan = (s1), t = 0, q = 0.2, fanout = 10
plan = (p1), t = 0, q = 0.2, fanout = 10
plan = (s1, p2),
t = 2, q = 0.2, fanout = 10
plan = (s1, p1),
t = 0, q = 0.2, fanout = 10
plan = (s1, <),
t = 0, q = 0.2, fanout = 10
plan = (p1, s1),
t = 0, q = 0.2, fanout = 10

Figure 6.5: The search tree after the third iteration

Since we have done a best-first search with respect to the quality measure **q**, and we know that **q** is monotonically worsening as the plan grows, we can return this result. None of the partial plans can be extended to a complete plan with better quality than this plan.

If we instead use a quality measure with a value that can improve as the plan grows, the algorithm will continue expanding the partial plans in the search tree even after finding this plan. This is a difference from the standard dynamic programming algorithm [45].

A possible simplification to our algorithm is to return the first found complete plan no matter what the monotonicity of the optimization objective. This plan is within the time limit, and with some probability it does also have a good quality compared to the other plans (because of the best-first search).

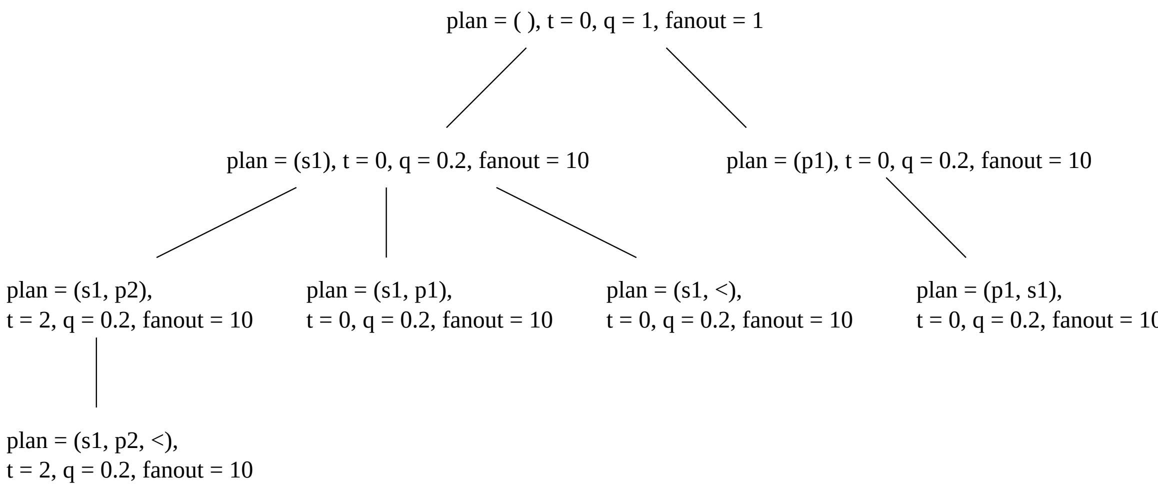
plan = ( ), t = 0, q = 1, fanout = 1
plan = (s1), t = 0, q = 0.2, fanout = 10
plan = (p1), t = 0, q = 0.2, fanout = 10
plan = (s1, p2), t = 2, q = 0.2, fanout = 10
plan = (s1, p1), t = 0, q = 0.2, fanout = 10
plan = (s1, <), t = 0, q = 0.2, fanout = 10
plan = (p1, s1), t = 0, q = 0.2, fanout = 10
plan = (s1, p2, <), t = 2, q = 0.2, fanout = 10

Figure 6.6: The search tree after the fourth iteration

Chapter 7

Applications

In this chapter, we look at applications for our approach to performance polymorphism. First, we try to identify the characteristics of a suitable application, and then we give some examples of possible application domains.

Even if we in this work concentrate on the real-time aspect of performance polymorphism, where a trade-off in quality is made in order to get the operation done within a time limit, performance polymorphism is not limited to real-time applications. If we e. g. are searching for information from sources on the Internet, with different monetary costs and data quality, this can be modeled using performance polymorphism, and a performance-polymorphic query optimizer can be used to find an acceptable trade-off between monetary cost and data quality. In the general case, the implementations of the performance-polymorphic operations are specified along any number of performance dimensions.

7.1 Suitable applications

There are applications where a database solution in general, and therefore also the technique presented here, is probably not suitable. Applications whose data has a simple structure, and where the operations on that data are few, simple, constant, and known in advance, can sometimes be better handled in a traditional programming language.

Also, applications with very high requirements on through-put and response times may also be unsuited for the overhead that is usually imposed by a database management system. For some applications, it may even be necessary to implement certain functionality in hardware.

On the other hand, a database solution allows for a much more flexible system. It is easy to accomodate changes in the structure of the data, the amounts of data, and the operations that are to be performed.

Performance polymorphism adds another dimension of flexibility, and we believe that the approach presented here, with performance-polymorphic queries, can benefit any application where there can be alternative operations, or where there exists possible alternative implementations of the operations.

Among possible application domains are real-time systems of different kinds (except, perhaps, those with the very highest performance requirements), but also those non-real-time systems where there are constraints on some other resource, for example network bandwidth or monetary cost.

7.2 Real-time systems

The original motivation for performance polymorphism comes from the real-time domain. This includes not only the "traditional" real-time applications such as robot control and factory automation, but also e. g. certain types of Internet lookup. There are probably very few applications where the time dimension is totally unimportant.

7.2.1 Graphics-intensive simulations

If you have played graphics-intensive computer games such as Quake or Unreal, you may have noticed that when more objects are shown on the screen, fewer frames are shown per second. When there are more objects, it takes more time to generate each frame. This can be irritating, because when many objects are shown at the same time, the player is usually in a difficult situation in the game. That, of course, is exactly when a high and constant frame rate is most needed. The screen resolution and the amount of detail in the graphics can usually be changed, but only by the user and not dynamically by the game. The user has to choose between bad graphics all the time, or slow updates when many objects are shown.

A better solution may be to let the game automatically switch to lower-quality graphics, perhaps selectively omitting unimportant objects or the background, in order to maintain a high frame rate.

Performance polymorphism could be used to model this problem. The constraints are that a picture has to be computed within a certain time limit (such as 100 ms), and that certain objects have to be shown. The performance dimensions are the amount of detail, or the presence, of each of the objects. The optimization objective is the overall quality of the picture, perhaps measured by the number of objects shown or the number of surfaces used when computing the picture.

This particular example may not be best solved by database technology with declarative queries, but it serves to indicate a class of non-traditional real-time applications. Also, as the size and complexity of e. g. games grows, databases technology may become more and more relevant and even necessary.

7.2.2 Time ranges

As for real-time database technology in general, performance-polymorphic queries may not be feasible for applications with the very highest performance requirements. Depending on the implementation, the time range that is of interest is probably one or a few milliseconds, assuming a main-memory database.

Also, for queries where optimization has to be done on the fly, the time for optimization has to be included in the response time, and this puts further restrictions on the type of applications. On the other hand, this usually applies to ad hoc queries formulated or generated by the user, and then an additional response time of a few milliseconds, or even seconds, may not be important.

7.3 The Internet and the World Wide Web

The Internet, with all its diverse applications, and with the varying speed and reliability of its different parts, seems to be well suited as an environment for database technology and performance-polymorphic operations.

The deadline semantics is usually soft or firm instead of hard, and response times are usually in the order of seconds instead of milli- or microseconds. This makes it possible and economical to perform even advanced pre-processing of queries in connection with their execution.

7.4 Multidatabases and mediators

The information in a database may exist in several copies, and the cost of retrieving or updating data may vary between these copies. One example of this a distributed database, where the same data may be replicated on different hosts in a computer network. If there are several performance dimensions, such as execution time and quality, it may be useful to find a trade-off between them.

7.4.1 Distributed databases

A *distributed database* can be defined as "a collection of multiple, logically interrelated databases distributed over a computer network", and a *distributed DBMS* as "the software system that permits the management of the [distributed database system] and makes the distribution transparent to the users" [36]. It is usually understood that the term distributed database refers to a situation where the different nodes ("servers", "hosts") have little autonomy, and where the internal functioning of the different nodes in the database system (such as cuncurrency control) is accessible from the network [7]. Transactions that originate locally are not given preference at a node over transactions that originate from some other node. An architecture where the nodes are more independent may be called a "federated database" or "multidatabase".

Since a distributed database system is tightly integrated, there is a global query optimizer, with a cost model that can differentiate between the costs of accessing data at different nodes [36]. Traditionally, network communication has been the dominant factor, at least in wide-area networks. If data is replicated in several locations, it has therefore been considered cheaper to access data locally than to get it from some other node. However, in modern high-speed networks (both local-area and wide-area networks), it may be cheaper to get data that is cached in main memory at another node, than it is to retrieve it from local disk storage [43].

7.4.2 Distributed databases and performance polymorphism

If some or all of the data in a distributed database is replicated at several nodes, the query optimizer must choose which of these copies to use in each query. With normal replication, the different copies contain exactly the same data, and one only has to find the copy that is cheapest to access (taken into account that the choice of node to use may affect the costs of choices in other parts of the query, and that processing costs may vary). This may be seen as a simple form of performance polymorphism, with cost as the single performance dimension, since the system chooses between different implementations of the same conceptual operation (getting the data, from any of the places where it is stored).

It does, however, become more interesting if we introduce more than one performance dimension. If the data in the different nodes are not simple copies of each other, but have different characteristics, we can define several performance dimensions. For example, the data in a node may be incomplete (with some tuples or fields missing), or it may have lower precision, its temporal validity may vary (it being older than data on other nodes), and so on. We may want to keep the cost (a more or less rough estimate of the execution time) as one of the performance measures.

Now we can use our model of performance polymorphism (as described above in chapter 5) to specify performance-polymorphic queries where we want to optimize one of these dimensions, while we put constraints on some of the other dimensions. For example, we may want a result with as high a precision as possible, with the constraint that the query execution time may not exceed some given value. A performance-polymorphic query optimizer would then find an execution plan that retrieves the best (most precise) data copies that it has time to do within the given time constraint.

With the common data model and tight integration of a distributed database system, the characteristics of the copies of replicated data can be controlled, and it will (under some conditions) be possible to construct a cost model that accurately describes these characteristics. However, in current research and practice of distributed databases, the focus has been on a replication which means exact copies, with no differences in characteristics except varying access cost. One reason for this may be lack of a good approach to use in choosing between (non-equivalent) copies of data, and handling the trade-offs between multiple performance measures.

The performance-polymorphic approach may be more useful for multidatabases, as described below, where the differences between replicated (or rather, "similar") data can be much larger.

7.4.3 Multidatabases

A *multidatabase* is a collection of autonomous databases, whose data can be manipulated through a common system, a *multidatabase system* [30]. The individual databases in the multidatabase system do not expose their low-level interfaces to other nodes, which instead have to access the data through the normal DBMS user interface [7].

Typically, the databases existed previously, before they were collected into one or more multidatabases, thus, making it possible to manipulate their data through a common interface. This means that their schemas have been designed independently, even using different data models. The same facts about the world may be simultaneously present in several of the databases, but in very different form.

7.4.4 Multidatabases and performance polymorphism

There are many problems that need to be handled to use databases with different and independent schemas (name differences, format differences, structural differences, missing or conflicting data, many kinds of semantic differences). However, there are some areas that seem especially interesting if we want to apply performance polymorphism to multidatabases. The following may all be identified as performance dimensions:

- Response time. The response times of the databases may vary. Even if exactly the same data is available in two individual databases, the expected performance can be very different, due to different hardware, different load, different local policies, and for reasons of the network.
- Reliability. We may expect that the probability of us being able to access an individual database at all may vary between the databases, for much the same reasons as given for performance above.
- Access policies. Different sites may enforce different access policies, so external users, or some external users, may be denied access to data in some of the databases. (Even if not proper databases, some ftp sites disallow users from outside the country to download certain software, for legal reasons.)
- Data quality. The quality of the data may vary between databases in the multidatabases. Thus, precision, correctness, validity, consistency, and age of the data, can vary between the databases.
- Monetary cost. There may be a fee for accessing some of the individual databases.

It is probably not uncommon to want to optimize or constrain some of these performance measures. (As fast as possible! No monetary cost!) Being able to state them explicitly, and let an optimizer handle them in a uniform way, may be very useful.

One example is prices of stocks on some stock exchange. These prices may be found on the Internet for free, if one can accept a certain delay. It is also possible to get the prices immediately ("in real-time"), but for this one has to pay a (large) fee.

7.4.5 Performance-polymorphic MSQL

As an example on how a multidatabase query language could be extended with performance polymorphism, and how a query may look, we will use the multidatabase query language MSQL [30], and show how a hypothetical extension to MSQL may be used to formulate a simple performance-polymorphic multidatabase query.

Assume that there are three different databases, all of them containing information about stock prices at a certain stock exchange. These databases are called `stock_de_luxe`, `stockprices_on_line`, and `gnu_stock`. The externally accessible data in these databases is delayed by different time intervals (0, 10 seconds, and 15 minutes, respectively), and each retrieval has a monetary cost (1.00, 0.01, and free). We now need to find the current price of Microsoft, but we do not want to pay more than 0.50 for the result. For simplicity, we assume that each of the three databases contains a table called "stock", with columns "name" and "price".

We also assume that we have, through some interface in the multidatabase system, defined the performance dimensions *age* (of the data) and *dollars* (monetary cost) according to the model in chapter 5, and with the appropriate default values, combination functions, etc.

It will also be necessary to specify the values for *age* and *dollars* of the different databases to the multi-dimensional cost model of the multidatabase. This will have to be done in terms of primitives on a lower level than the MSQL language provides.

First we use normal MSQL to create a multidatabase that uses the three databases:

```
CREATE stockprices
FROM stock_de_luxe, stockprices_on_line, gnu_stock
```

Then we need a way to state that the three "stock" tables can be used to find the same information, but with different performance values. This could be a possible syntax:

```
CREATE PERFORMANCE_POLYMORPHIC VIEW stock
AS SELECT name, price
FROM [stock_de_luxe.stock, stockprices_on_line.stock,
      gnu_stock.stock];
```

The query may then look like this, along the pattern from the AMOSQL query in section 6.5:

```
USE stockprices

SELECT price FROM stock
WHERE name = 'Microsoft'
WITH dollars BETTER THAN 0.50
OPTIMIZE age
```

In this case, the optimizer will choose an execution plan that reads the requested value from database number 2, `stockprices_on_line`, the one with a cost of 0.01 and a delay of 10 seconds.

7.4.6 Mediators

In a large computer network, such as the Internet, with various data sources with different formats of data and different semantics, applications may find it useful to have an intermediate system to mediate between them. Such a *mediator* can be defined as "a software module that exploits encoded knowledge about certain sets or subsets of data to create information for a higher layer of applications" [53].

The mediator module has knowledge, for example in the form of rules, about the different formats of data, and their semantics. The mediator can perform not only simple translation, but also other processing of the data, such as creating different abstractions of the same data. Several previously existing databases may be accessed from one application, through one or more mediators that sits inbetween the systems.

Even if mediators originally have been mostly intended to translate and in various ways interpret data, it will probably be interesting or even necessary to be able to choose between more or less equivalent data that represent the (more or less) same real things.

In a heterogeneous environment, where many existing databases, that perhaps are managed by different organizations, are to be used as data sources for mediators, it seems that an important part of the information processing done in a mediator may be to choose between alternative sources for the same data, sources that provide that data with differences in quality, cost, speed, and other characteristics.

7.5 Server-side optimization

A server may have to handle several concurrent transactions. If not all these transactions can be performed with the highest available quality, some could be “down-graded”. It is then necessary to be able to handle several implementations of the “same” operation, but with different performance. Performance polymorphism may be a way to handle these multiple implementations and the choices between them. Also, there may be some required quality of service for the transactions, and this translates to the constraints in our model of performance polymorphism.

An example is a central server for electronic commerce, which handles commercial transactions, and which guarantees that each transaction is completed within, say, one second. The optimization objective in such a system should not be to minimize the execution time for each transaction, but to maximize the number of transactions that can be completed within the one-second time limit. In order to maximize this number, we may have to slow down some individual transactions.

If the entire set of queries that has to be executed (in a certain time period) is optimized as one large query (as seen by the optimizer), with performance-polymorphic choices for (some of) the steps in each plan, constraints can be put on each transaction’s cost. This could lead to a very large “query conglomerate” that has to be optimized, and a better way may be to view each individual query as one operation, with a performance-polymorphic choice between different, previously generated, execution plans for this query.

7.6 Telecom databases

Telecom databases [43] are databases used in the operation of a telecom network or as parts of applications in the telecom network.

The first telecom databases provided number translations for various services. Another early application was databases that keep track of mobile phones. Other telecom databases are used for management of the network, especially for real-time charging information.

Database servers for telecom application have in common that they have to answer massive amounts of rather simple queries (e. g. 10000-20000 requests per second) and that they have (soft real-time) requirements on short response times (e. g. 5-15 ms). Some of them also need large storage, and some need to send large amounts of data to the users. They also have to be very reliable, for example with unavailability required to be less than 30 seconds per year, and no scheduled downtime allowed.

A complication is that all telecom networks are built with the assumption that not all subscribers are active at the same time. This means that overload situations can occur.

If the requirements on response times and availability are to be met, even in such situations, telecom databases must therefore have a scheme to handle overload situations.

Chapter 8

Future Work

In the present implementation, the optimizer should be rewritten with more regard to efficiency, robustness and elegance of the program code. The integration with other parts of AMOS should be better.

8.1 Late and early optimization

The optimization of a declarative query is a combinatorical problem, and performance polymorphism adds additional complexity. n predicates can be ordered in $n!$ different orders, and for each of these predicates that exists in several performance-polymorphic versions, the expression $n!$ has to be multiplied by the number of different versions of that predicate.

Since optimization is a hard problem in this sense, and the optimizer itself in its present implementation is not time-constrained, query optimization and the resolving of performance-polymorphic predicate implementations is expected to be done early, at query compile time. In some cases, however, late binding is advantageous, [17] and then strategies are needed to estimate the performance of late bound performance-polymorphic function calls. The query optimizer should automatically choose early binding when possible. When late binding cannot be avoided the system can optimize each resolvent separately and then estimate the time to execute the performance-polymorphic call in terms of the actual time to execute its resolvents. Such partial evaluation of the estimates of cost and quality would minimize the amount of work that has to be done at run time.

8.2 Alternative optimization algorithms

For complex queries, the (pseudo-)exhaustive search done by dynamic programming will not be feasible. The complexity of queries in relational DBMSs is increasing. One reason for this is that modern graphical user interfaces to database systems enables the users to pose very complex queries, where the textual or internal form of the query is automatically generated. [40]. Therefore, alternative optimization algorithms should be investigated, such as randomized and heuristic algorithms. Among the candidates are hill-climbing with multiple random starting points, and simulated annealing [20].

One problem here is how to handle the combination of constraints and a non-exhaustive optimization algorithm, in the cases where the constraints can't be satisfied. An exhaustive algorithm will, in that case, try all possible plans (or, as in our dynamic-programming-based algorithm, after pruning unnecessary parts from the search tree), and then report a failure. If the non-exhaustive algorithm simply proceeds until it finds a feasible solution, i. e. one that satisfies the constraints, the non-exhaustive algorithm may degenerate into an unusually inefficient exhaustive algorithm. We should handle this in some way, instead of letting the optimizer work for maybe hours or days before it reports the failure. Perhaps a simple time-out will be sufficient. Perhaps we can apply performance polymorphism to the optimizer itself?

8.3 Measurements of quality

The present work leaves the choice of a quality measure, or several quality measures, to the application implementer. However, different quality measures can be studied. Among these are the rules of fuzzy logic [54] [15], or some ad-hoc measure, like the certainty factors of EMYCIN [46].

8.4 Refining and validating the cost model

Since the choices between different implementations of actions are done at compile-time, these choices are entirely based on the cost model. It is therefore important for the cost model to be correct.

For the system to be useful in real-time applications, timing estimates should be developed for the internal operations This includes operations, such as lookup and insertion, on the DBMS's internal data structures. These data structures should be modified for an improved and well-analyzed worst-case behavior.

The cost model that is used to find timing estimates should be verified against actual, measured execution times [25] [29].

8.5 What if the cost model was wrong?

Even if great care is taken to guarantee the accuracy of the cost model, it could turn out to be wrong in certain cases. It may be difficult to find accurate estimates, for example due to inherent unpredictability in the application domain, or to changing conditions. Unexpected events may occur, such as if a server suddenly doesn't respond.

This means that if the cost model is wrong, the generated execution plan may not be optimal, or it may not comply with the stated constraints on performance dimensions.

To handle a situation like this we can monitor execution, and compare the actual performance with the one predicted by the cost model.

The simplest case is to detect whenever a constraint is overrun, and then raise an exception and call an exception handler. A more advanced system could keep track of resource usage during each step of plan execution, and perform some appropriate action if the actual measured performance deviates from the cost model.

Chapter 9

Conclusions

9.1 Concept

We have defined the concept of performance-polymorphic queries, and compared it to other similar approaches.

Performance-polymorphic queries can be used in situations where it is possible to define one or more performance dimensions, such as different types of cost or quality, and where one or more operations can exist in more than one version, and where these versions differ in their performance along one or more of the performance dimensions.

Previous work has either used a programming-language approach, or severely limited the use of performance polymorphism. We give the users the power and flexibility of a declarative query language, and allow general use of performance-polymorphic functions in the language.

Our approach seems to handle the combination of performance values better than previous work, especially the programming language Flex and the data model ROMPP. In contrast to these, our optimizer is able to find a trade-off between choices when a program or query contains more than one performance-polymorphic operation

9.2 Implementation

We have extended a query language with performance-polymorphic queries, and we have developed a performance-polymorphic query optimizer based on extensions to an object-oriented query optimizer.

We have therefore shown that the idea is possible to implement, and that performance-polymorphic declarative queries can be compiled, optimized and executed.

9.3 Applications

The technique seems to be useful on a number of applications. Medium- to low-speed real-time applications is one class of such applications. Since any number of performance dimensions can be defined and handled by the optimizer, the technique is general, and can also be used outside the real-time domain.

Bibliography

[1] M. Berndtsson and J. Hansson, editors. *Active and Real-Time Database Systems (ARTDB-95)*, Berlin, 1995.

[2] M. Berndtsson and J. Hansson. Issues in active real-time databases. In *International Workshop on Active and Real-Time Database Systems (ARTDB-95)*, pages 142–157, Sweden, June 1995. University of Skövde, Sweden, Springer-Verlag.

[3] A. Bestavros and V. Fay-Wolfe, editors. *Real-Time Database Systems - Research Advances.* Kluwer Academic Publishers, 1997.

[4] A. Bestavros, K.-J. Lin, and S. H. Son, editors. *Real-Time Database Systems - Issues and Applications.* Kluwer Academic Publishers, 1997.

[5] P. Bonnet and A. Tomasic. Partial Answers for Unavailable Data Sources. *Proceedings of the Third International Conference on Flexible Query Answering Systems, FQAS'98*, 1495:44–55, 1998.

[6] H. Branding and A. P. Buchmann. On providing soft and hard real-time capabilities in an active DBMS. In *International Workshop on Active and Real-Time Database Systems (ARTDB-95)*, pages 158–169, Sweden, June 1995. University of Skövde, Sweden, Springer-Verlag.

[7] M. W. Bright, A. R. Hurson, and S. H. Pakzad. A Taxonomy and Current Issues in Multidatabase Systems. *Computer*, 25(3):50–60, March 1992.

[8] G.C. Buttazzo. *Hard Real-Time Computing Systems - Predictable Scheduling Algorithms and Applications.* Kluwer Academic Publishers, Dordrecht, The Netherlands, 1997.

[9] R. G. G. Cattell. *Object Data Management. Object-Oriented and Extended Relational Database Systems.* Addison Wesley, 1991.

[10] R. L. Cole and G. Graefe. Optimization of dynamic query evaluation plans. In Richard T. Snodgrass and Marianne Winslett, editors, *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, pages 150–160, Minneapolis, Minnesota, 24–27 May 1994.

[11] C. J. Date. *An Introduction to Database Systems.* Addison-Wesley, Reading, Mass., 6 edition, 1995.

[12] U. Dayal. Ten Years of Activity in Active Database Systems: What Have We Accomplished? In *Proceedings of the 1st International Workshop on Active and Real-Time Database Systems*, Workshops in Computing, pages 3–22. Springer, 1995.

[13] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems.* Addison Wesley, Redwood City, CA, 2 edition, 1994.

[14] U. Dayal et al. The HiPAC project: Combining active databases and timing constraints. *SIGMOD Record*, 17(1):51–70, March 1988.

[15] R. Fagin. Combining Fuzzy Information from Multiple Systems. In *Proceedings of the Fifteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS 1996*, pages 216–226, Montréal, Canada, June 1996. ACM Press.

[16] P. Fernlund, G. Fex, A. Hanson, J. Stenflo, and B. Lundh. *Laurells Klinisk kemi i praktisk medicin.* Studentlitteratur, Lund, Sweden, 1991.

[17] S. Flodin and T. Risch. Processing Object-Oriented Queries with Invertible Late Bound Functions. In *Proceedings of VLDB-95*, 1995.

[18] H. Garcia-Molina. Main memory database systems: An overview. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):590–516, December 1992.

[19] W.-C. Hou, G. Özsoyoğlu, and B. K. Taneja. Processing aggregate relational queries with hard time constraints. In *Proc. of ACM SIGMOD Conf. 1989*, pages 68–77, Portland, Oregon, May 1989.

[20] Y. E. Ioannidis and Y. C. Kang. Randomized algorithms for optimizing large join queries. In H. Garcia-Molina and H. V. Jagadish, editors, *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, pages 312–321, Atlantic City, NJ, May 1990.

[21] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall International, International Series in Computer Science, June 1993. ISBN number 0-13-020249-5 (pbk).

[22] K. Kenny and K.-J-Lin. Measuring and Analyzing Real-Time Performance. *IEEE Software*, 8(5):41–49, September 1991.

[23] K. Kenny and K. J. Lin. Structuring large real-time systems with performance polymorphism. In *Proc. 11th IEEE Real-Time Systems Symp.*, pages 238–246, Orlando, FL, December 1990.

[24] K. Kenny and K.-J. Lin. Implementing and Checking Timing Constraints in Real-Time Programs. *Microprocessing and Microprogramming*, (38):477–484, 1993.

[25] K. B. Kenny and K.-J. Lin. Building Flexible Real-Time Systems Using the Flex Language. *Computer*, 24(5):70–78, May 1991.

[26] W. Kim, editor. *Modern Database Systems. The Object Model, Interoperability, and Beyond.* Addison Wesley, New York, NY, 1995.

[27] K.-J. Lin and S. Natarajan. Expressing and Maintaining Timing Constraints in FLEX. In *Proceedings of the 9th IEEE Real-time*

Systems Symposium, pages 96–105, Los Alamitos, CA, July 1988. IEEE Computer Society Press.

[28] K. J. Lin, S. Natarajan, J. W. S. Liu, and T. Krauskopf. Concord: A system of imprecise computations. In *Proc. 1987 IEEE Compsac*, October 1987. Japan.

[29] S. Listgarten and M.-A. Neimat. Cost model development for a main memory database system. In A. Bestavros, K.-J. Lin, and S. H. Son, editors, *Real-Time Database Systems - Issues and Applications*, pages 139–162. Kluwer Academic Publishers, 1997.

[30] W. Litwin, A. Abdellatif, A. Zeroual, B. Nicolas, and P. Vigier. MSQL: A multidatabase language. *Information Sciences,2,3*, 49(1):59–101, 1989.

[31] W. Litwin and T. Risch. Main memory oriented optimization of OO queries using typed datalog with foreign predicates. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):517–528, December 1992.

[32] J. W. S. Liu, K.-J. Lin, W.-K. Shih, A. Chuang shi Yu, J.-Y. Chung, and W. Zhao. Algorithms for Scheduling Imprecise Computations. *Computer*, 24(5):58–68, May 1991.

[33] V. Lortz. An object-oriented real-time database system for multiprocessors. Technical Report CSE-TR-210-94, University of Michigan, April 1994.

[34] G. Özsoyoğlu, S. Guruswamy, Kaizheng Du, and Wen-Chi Hou. Time-constrained query processing in CASE-DB. *IEEE Transactions on Knowledge and Data Engineering*, 7(6):865–884, December 1995.

[35] G. Özsoyoğlu, K. Du, S. Guru swamy, and W.-C. Hou. Processing real-time, non-aggregate queries with time-constraints in CASE-DB. In F. Golshani, editor, *Proceedings of the International Conference on Data Engineering*, volume 8, pages 410–417, Los Alamitos, CA, February 1992. IEEE Computer Society Press.

[36] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[37] T. Padron-McCarthy and T. Risch. Performance-Polymorphic Execution of Real-Time Queries. In *Proceedings of the First International Workshop on Real-Time Databases*, pages 50–53, Newport Beach, CA, USA, March 1996.

[38] T. Padron-McCarthy and T. Risch. Optimizing Performance-Polymorphic Declarative Database Queries. In A. Bestavros and V. Fay-Wolfe, editors, *Real-Time Database Systems - Research Advances*, pages 311–326, Dordrecht, The Netherlands, 1997. Kluwer Academic Publishers.

[39] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, San Francisco, 2 edition, 1994.

[40] H. Pirahesh, T. Leung, and W. Hasan. A rule engine for query transformation in starburst and IBM DB2 C/S DBMS. In *Proceedings of the 13th International Conference on Data Engineering (ICDE'97)*, pages 391–401, Washington - Brussels - Tokyo, April 1997. IEEE.

[41] J. J. Prichard, L. C. DiPippo, J. Peckham, and V. F. Wolfe. RTSORAC: A real-time object-oriented database model. *Lecture Notes in Computer Science*, 856:601–610, 1994.

[42] K. Ramamritham. Real-time databases. *Distributed and Parallel Databases*, 1(2):199–226, April 1993.

[43] M. Ronström. *Design and Modelling of a Parallel Data Server for Telecom Applications*. Linköping Studies in Science and Technology, Dissertation No. 520, Linköping, 1998.

[44] K. Schwan, P. Gopinath, and W. Bo. CHAOS – Kernel support for objects in the real-time domain. *IEEE Transactions on Computers*, 36(8):904–916, August 1987.

[45] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access Path Selection in a Relational Database Management System. In *Proc. of SIGMOD Conf. 1979*, pages 23–34, Boston, MA, 1979.

[46] E. H. Shortliffe. *Computer-Based Medical Consultations: MYCIN.* Elsevier/North-Holland, Amsterdam, London, New York, 1976.

[47] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database System Concepts.* McGraw-Hill, New York, 1997.

[48] M. Sköld, E. Falkenroth, and T. Risch. Rule Contexts in Active Databases — A Mechanism for Dynamic Rule Grouping. In *RIDS'95 (Rules in Database Systems)*, Athens, Greece, September 1995.

[49] M. Sköld and T. Risch. Using partial differencing for efficient monitoring of deferred complex rule conditions. In *Proceedings of the 12th International Conference on Data Engineering*, pages 392–401, Washington - Brussels - Tokyo, February 1996. IEEE Computer Society.

[50] S. H. Son, editor. *Advances In Real-Time Systems.* Prentice Hall, Englewood Cliffs, NJ, 1995.

[51] J. A. Stankovic. Misconceptions About Real-Time Computing. A Seriuos Problem for Next-Generation Systems. *Computer*, 21(10):10–19, October 1988.

[52] H. S. Stone. *High-Performance Computer Architecture.* Addison-Wesley, 3 edition, 1993.

[53] G. Wiederhold. Mediators in the Architecture of Future Information Systems. *Computer*, 25:38–49, March 1992.

[54] L. A. Zadeh. Fuzzy Sets. *Information and Control*, 8(3):338–353, June 1965.

[55] L. Zhou, E. A. Rundensteiner, and K. G. Shin. Schema evolution for real-time object-oriented databases. Technical Report CSE-TR-199-94, University of Michigan, March 1994.

Linköping Studies in Science and Technology

Thesis No 733

Performance-Polymorphic Declarative Queries

by

Thomas Padron-McCarthy

Licentiatavhandling

som för avläggande av teknologie licentiatexamen vid Linköpings universitet kommer att offentligt presenteras i seminarierum Belöningen, hus B, Linköpings universitet, tisdagen den 8 december 1998, kl 13:15.

Abstract

Performance polymorphism, where a system can select between several given implementations of the same conceptual operation, has been used in real-time programming languages, such as Flex. The contingency plans used in the active database system HiPAC is a related, but more limited, mechanism. We have introduced performance polymorphism into a declarative database query language. We have shown the feasibility of the concept by implementing a general, performance-polymorphic query optimizer. We show how performance-polymorphic queries are specified and optimized in our system. A number of applications for the technique are suggested.

This work was supported by NUTEK, the Swedish National Board for Industrial and Technical Development, as part of ISIS, the Competence Center for Integrated Systems for Control and Information.

Department of Computer and Information Science
Linköping University
S-581 83 Linköping
Sweden

ISBN 91-7219-354-9 ISSN 0280-7971

Department of Computer and Information Science
Linköping University

Linköping Studies in Science and Licentiate Theses at the Faculty of Arts and Humanities

No 17 **Vojin Plavsic:** Interleaved Processing of Non-Numerical Data Stored on a Cyclic Memory. (Available at: FOA, Box 1165, S-581 11 Linköping, Sweden. FOA Report B30062E)
No 28 **Arne Jönsson, Mikael Patel:** An Interactive Flowcharting Technique for Communicating and Realizing Algorithms, 1984.
No 29 **Johnny Eckerland:** Retargeting of an Incremental Code Generator, 1984.
No 48 **Henrik Nordin:** On the Use of Typical Cases for Knowledge-Based Consultation and Teaching, 1985.
No 52 **Zebo Peng:** Steps Towards the Formalization of Designing VLSI Systems, 1985.
No 60 **Johan Fagerström:** Simulation and Evaluation of Architecture based on Asynchronous Processes, 1985.
No 71 **Jalal Maleki:** ICONStraint, A Dependency Directed Constraint Maintenance System, 1987.
No 72 **Tony Larsson:** On the Specification and Verification of VLSI Systems, 1986.
No 73 **Ola Strömfors:** A Structure Editor for Documents and Programs, 1986.
No 74 **Christos Levcopoulos:** New Results about the Approximation Behavior of the Greedy Triangulation, 1986.
No 104 **Shamsul I. Chowdhury: S**tatistical Expert Systems - a Special Application Area for Knowledge-Based Computer Methodology, 1987.
No 108 **Rober Bilos:** Incremental Scanning and Token-Based Editing, 1987.
No 111 **Hans Block:** SPORT-SORT Sorting Algorithms and Sport Tournaments, 1987.
No 113 **Ralph Rönnquist:** Network and Lattice Based Approaches to the Representation of Knowledge, 1987.
No 118 **Mariam Kamkar, Nahid Shahmehri:** Affect-Chaining in Program Flow Analysis Applied to Queries of Programs, 1987.
No 126 **Dan Strömberg:** Transfer and Distribution of Application Programs, 1987.
No 127 **Kristian Sandahl:** Case Studies in Knowledge Acquisition, Migration and User Acceptance of Expert Systems, 1987.
No 139 **Christer Bäckström:** Reasoning about Interdependent Actions, 1988.
No 140 **Mats Wirén:** On Control Strategies and Incrementality in Unification-Based Chart Parsing, 1988.
No 146 **Johan Hultman:** A Software System for Defining and Controlling Actions in a Mechanical System, 1988.
No 150 **Tim Hansen:** Diagnosing Faults using Knowledge about Malfunctioning Behavior, 1988.
No 165 **Jonas Löwgren:** Supporting Design and Management of Expert System User Interfaces, 1989.
No 166 **Ola Petersson:** On Adaptive Sorting in Sequential and Parallel Models, 1989.
No 174 **Yngve Larsson:** Dynamic Configuration in a Distributed Environment, 1989.
No 177 **Peter Åberg:** Design of a Multiple View Presentation and Interaction Manager, 1989.
No 181 **Henrik Eriksson:** A Study in Domain-Oriented Tool Support for Knowledge Acquisition, 1989.
No 184 **Ivan Rankin:** The Deep Generation of Text in Expert Critiquing Systems, 1989.
No 187 **Simin Nadjm-Tehrani:** Contributions to the Declarative Approach to Debugging Prolog Programs, 1989.
No 189 **Magnus Merkel:** Temporal Information in Natural Language, 1989.
No 196 **Ulf Nilsson:** A Systematic Approach to Abstract Interpretation of Logic Programs, 1989.
No 197 **Staffan Bonnier:** Horn Clause Logic with External Procedures: Towards a Theoretical Framework, 1989.
No 203 **Christer Hansson:** A Prototype System for Logical Reasoning about Time and Action, 1990.
No 212 **Björn Fjellborg:** An Approach to Extraction of Pipeline Structures for VLSI High-Level Synthesis, 1990.
No 230 **Patrick Doherty:** A Three-Valued Approach to Non-Monotonic Reasoning, 1990.
No 237 **Tomas Sokolnicki:** Coaching Partial Plans: An Approach to Knowledge-Based Tutoring, 1990.
No 250 **Lars Strömberg:** Postmortem Debugging of Distributed Systems, 1990.
No 253 **Torbjörn Näslund:** SLDFA-Resolution - Computing Answers for Negative Queries, 1990.
No 260 **Peter D. Holmes:** Using Connectivity Graphs to Support Map-Related Reasoning, 1991.
No 283 **Olof Johansson:** Improving Implementation of Graphical User Interfaces for Object-Oriented Knowledge-Bases, 1991.
No 298 **Rolf G Larsson:** Aktivitetsbaserad kalkylering i ett nytt ekonomisystem, 1991.
No 318 **Lena Srömbäck:** Studies in Extended Unification-Based Formalism for Linguistic Description: An Algorithm for Feature Structures with Disjunction and a Proposal for Flexible Systems, 1992.
No 319 **Mikael Pettersson:** DML-A Language and System for the Generation of Efficient Compilers from Denotational Specification, 1992.
No 326 **Andreas Kågedal:** Logic Programming with External Procedures: an Implementation, 1992.
No 328 **Patrick Lambrix:** Aspects of Version Management of Composite Objects, 1992.
No 333 **Xinli Gu:** Testability Analysis and Improvement in High-Level Synthesis Systems, 1992.
No 335 **Torbjörn Näslund:** On the Role of Evaluations in Iterative Development of Managerial Support Sytems, 1992.
No 348 **Ulf Cederling:** Industrial Software Development - a Case Study, 1992.
No 352 **Magnus Morin:** Predictable Cyclic Computations in Autonomous Systems: A Computational Model and Implementation, 1992.
No 371 **Mehran Noghabai:** Evaluation of Strategic Investments in Information Technology, 1993.
No 378 **Mats Larsson:** A Transformational Approach to Formal Digital System Design, 1993.
No 380 **Johan Ringström:** Compiler Generation for Parallel Languages from Denotational Specifications, 1993.
No 381 **Michael Jansson:** Propagation of Change in an Intelligent Information System, 1993.
No 383 **Jonni Harrius:** An Architecture and a Knowledge Representation Model for Expert Critiquing Systems, 1993.
No 386 **Per Österling:** Symbolic Modelling of the Dynamic Environments of Autonomous Agents, 1993.
No 398 **Johan Boye:** Dependency-based Groudness Analysis of Functional Logic Programs, 1993.
No 402 **Lars Degerstedt:** Tabulated Resolution for Well Founded Semantics, 1993.
No 406 **Anna Moberg:** Satellitkontor - en studie av kommunikationsmönster vid arbete på distans, 1993.
No 414 **Peter Carlsson:** Separation av företagsledning och finansiering - fallstudier av företagsledarutköp ur ett agentteoretiskt perspektiv, 1994.
No 417 **Camilla Sjöström:** Revision och lagreglering - ett historiskt perspektiv, 1994.
No 436 **Cecilia Sjöberg:** Voices in Design: Argumentation in Participatory Development, 1994.

No 437	**Lars Viklund:** Contributions to a High-level Programming Environment for a Scientific Computing, 1994.
No 440	**Peter Loborg:** Error Recovery Support in Manufacturing Control Systems, 1994.
FHS 3/94	**Owen Eriksson:** Informationssystem med verksamhetskvalitet - utvärdering baserat på ett verksamhetsinriktat och samskapande perspektiv, 1994.
FHS 4/94	**Karin Pettersson:** Informationssystemstrukturering, ansvarsfördelning och användarinflytande - En komparativ studie med utgångspunkt i två informationssystemstrategier, 1994.
No 441	**Lars Poignant:** Informationsteknologi och företagsetablering - Effekter på produktivitet och region, 1994.
No 446	**Gustav Fahl:** Object Views of Relational Data in Multidatabase Systems, 1994.
No 450	**Henrik Nilsson:** A Declarative Approach to Debugging for Lazy Functional Languages, 1994.
No 451	**Jonas Lind:** Creditor - Firm Relations: an Interdisciplinary Analysis, 1994.
No 452	**Martin Sköld:** Active Rules based on Object Relational Queries - Efficient Change Monitoring Techniques, 1994.
No 455	**Pär Carlshamre:** A Collaborative Approach to Usability Engineering: Technical Communicators and System Developers in Usability-Oriented Systems Development, 1994.
FHS 5/94	**Stefan Cronholm:** Varför CASE-verktyg i systemutveckling? - En motiv- och konsekvensstudie avseende arbetssätt och arbetsformer, 1994.
No 462	**Mikael Lindvall:** A Study of Traceability in Object-Oriented Systems Development, 1994.
No 463	**Fredrik Nilsson:** Strategi och ekonomisk styrning - En studie av Sandviks förvärv av Bahco Verktyg, 1994.
No 464	**Hans Olsén:** Collage Induction: Proving Properties of Logic Programs by Program Synthesis, 1994.
No 469	**Lars Karlsson:** Specification and Synthesis of Plans Using the Features and Fluents Framework, 1995.
No 473	**Ulf Söderman:** On Conceptual Modelling of Mode Switching Systems, 1995.
No 475	**Choong-ho Yi:** Reasoning about Concurrent Actions in the Trajectory Semantics, 1995.
No 476	**Bo Lagerström:** Successiv resultatavräkning av pågående arbeten. Fallstudier i tre byggföretag, 1995.
No 478	**Peter Jonsson:** Complexity of State-Variable Planning under Structural Restrictions, 1995.
FHS 7/95	**Anders Avdic:** Arbetsintegrerad systemutveckling med kalkylkprogram, 1995.
No 482	**Eva L Ragnemalm:** Towards Student Modelling through Collaborative Dialogue with a Learning Companion, 1995.
No 488	**Eva Toller:** Contributions to Parallel Multiparadigm Languages: Combining Object-Oriented and Rule-Based Programming, 1995.
No 489	**Erik Stoy:** A Petri Net Based Unified Representation for Hardware/Software Co-Design, 1995.
No 497	**Johan Herber:** Environment Support for Building Structured Mathematical Models, 1995.
No 498	**Stefan Svenberg:** Structure-Driven Derivation of Inter-Lingual Functor-Argument Trees for Multi-Lingual Generation, 1995.
No 503	**Hee-Cheol Kim:** Prediction and Postdiction under Uncertainty, 1995.
FHS 8/95	**Dan Fristedt:** Metoder i användning - mot förbättring av systemutveckling genom situationell metodkunskap och metodanalys, 1995.
FHS 9/95	**Malin Bergvall:** Systemförvaltning i praktiken - en kvalitativ studie avseende centrala begrepp, aktiviteter och ansvarsroller, 1995.
No 513	**Joachim Karlsson:** Towards a Strategy for Software Requirements Selection, 1995.
No 517	**Jakob Axelsson:** Schedulability-Driven Partitioning of Heterogeneous Real-Time Systems, 1995.
No 518	**Göran Forslund:** Toward Cooperative Advice-Giving Systems: The Expert Systems Experience, 1995.
No 522	**Jörgen Andersson:** Bilder av småföretagares ekonomistyrning, 1995.
No 538	**Staffan Flodin:** Efficient Management of Object-Oriented Queries with Late Binding, 1996.
No 545	**Vadim Engelson:** An Approach to Automatic Construction of Graphical User Interfaces for Applications in Scientific Computing, 1996.
No 546	**Magnus Werner :** Multidatabase Integration using Polymorphic Queries and Views, 1996.
FiF-a 1/96	**Mikael Lind:** Affärsprocessinriktad förändringsanalys - utveckling och tillämpning av synsätt och metod, 1996.
No 549	**Jonas Hallberg:** High-Level Synthesis under Local Timing Constraints, 1996.
No 550	**Kristina Larsen:** Förutsättningar och begränsningar för arbete på distans - erfarenheter från fyra svenska företag. 1996.
No 557	**Mikael Johansson:** Quality Functions for Requirements Engineering Methods, 1996.
No 558	**Patrik Nordling:** The Simulation of Rolling Bearing Dynamics on Parallel Computers, 1996.
No 561	**Anders Ekman:** Exploration of Polygonal Environments, 1996.
No 563	**Niclas Andersson:** Compilation of Mathematical Models to Parallel Code, 1996.
No 567	**Johan Jenvald:** Simulation and Data Collection in Battle Training, 1996.
No 575	**Niclas Ohlsson:** Software Quality Engineering by Early Identification of Fault-Prone Modules, 1996.
No 576	**Mikael Ericsson:** Commenting Systems as Design Support—A Wizard-of-Oz Study, 1996.
No 587	**Jörgen Lindström:** Chefers användning av kommunikationsteknik, 1996.
No 589	**Esa Falkenroth:** Data Management in Control Applications - A Proposal Based on Active Database Systems, 1996.
No 591	**Niclas Wahllöf:** A Default Extension to Description Logics and its Applications, 1996.
No 595	**Annika Larsson:** Ekonomisk Styrning och Organisatorisk Passion - ett interaktivt perspektiv, 1997.
No 597	**Ling Lin:** A Value-based Indexing Technique for Time Sequences, 1997.
No 598	**Rego Granlund:** C^3Fire - A Microworld Supporting Emergency Management Training, 1997.
No 599	**Peter Ingels:** A Robust Text Processing Technique Applied to Lexical Error Recovery, 1997.
No 607	**Per-Arne Persson:** Toward a Grounded Theory for Support of Command and Control in Military Coalitions, 1997.
No 609	**Jonas S Karlsson:** A Scalable Data Structure for a Parallel Data Server, 1997.
FiF-a 4	**Carita Åbom:** Videomötesteknik i olika affärssituationer - möjligheter och hinder, 1997.
FiF-a 6	**Tommy Wedlund**: Att skapa en företagsanpassad systemutvecklingsmodell - genom rekonstruktion, värdering och vidareutveckling i T50-bolag inom ABB, 1997.
No 615	**Silvia Coradeschi**: A Decision-Mechanism for Reactive and Coordinated Agents, 1997.
No 623	**Jan Ollinen:** Det flexibla kontorets utveckling på Digital - Ett stöd för multiflex? 1997.

No 626	**David Byers:** Towards Estimating Software Testability Using Static Analysis, 1997.
No 627	**Fredrik Eklund:** Declarative Error Diagnosis of GAPLog Programs, 1997.
No 629	**Gunilla Ivefors:** Krigsspel coh Informationsteknik inför en oförutsägbar framtid, 1997**.**
No 631	**Jens-Olof Lindh:** Analysing Traffic Safety from a Case-Based Reasoning Perspective, 1997
No 639	**Jukka Mäki-Turja:**. Smalltalk - a suitable Real-Time Language, 1997.
No 640	**Juha Takkinen**: CAFE: Towards a Conceptual Model for Information Management in Electronic Mail, 1997.
No 643	**Man Lin**: Formal Analysis of Reactive Rule-based Programs, 1997.
No 653	**Mats Gustafsson**: Bringing Role-Based Access Control to Distributed Systems, 1997.
FiF-a 13	**Boris Karlsson:** Metodanalys för förståelse och utveckling av systemutvecklingsverksamhet. Analys och värdering av systemutvecklingsmodeller och dess användning, 1997.
No 674	**Marcus Bjäreland:** Two Aspects of Automating Logics of Action and Change - Regression and Tractability, 1998.
No 676	**Jan Håkegård**: Hierarchical Test Architecture and Board-Level Test Controller Synthesis, 1998.
No 668	**Per-Ove Zetterlund**: Normering av svensk redovisning - En studie av tillkomsten av Redovisningsrådets rekommendation om koncernredovisning (RR01:91), 1998.
No 675	**Jimmy Tjäder**: Projektledaren & planen - en studie av projektledning i tre installations- och systemutvecklingsprojekt, 1998.
FiF-a 14	**Ulf Melin**: Informationssystem vid ökad affärs- och processorientering - egenskaper, strategier och utveckling, 1998.
No 695	**Tim Heyer**: COMPASS: Introduction of Formal Methods in Code Development and Inspection, 1998.
No 700	**Patrik Hägglund:** Programming Languages for Computer Algebra, 1998.
No 712	**Christina Wennestam:** Information om immateriella resurser. Investeringar i forskning och utveckling samt i personal inom skogsindustrin, 1998.
No 719	**Joakim Gustafsson:** Extending Temporal Action Logic for Ramification and Concurrency, 1998.
No 725	**Erik Larsson:** High-Level Testability Analysis and Enhancement Techniques, 1998.
No 730	**Carl-Johan Westin:** Informationsförsörjning: en fråga om ansvar - aktiviteter och uppdrag i fem stora svenska organisationers operativa informationsförsörjning, 1998.
No 731	**Åse Jansson:** Miljöhänsyn - en del i företags styrning, 1998.
No 733	**Thomas Padron-McCarthy:** Performance-Polymorphic Declarative Queries, 1998.