Management of 1-D Sequence Data – From Discrete to Continuous

by

Ling Lin

Mar. 25th, 1998

Abstract

Data over ordered domains such as time or linear positions are termed *sequence data*. Sequence data require special treatments which are not provided by traditional DBMSs. Modelling sequence data in traditional (relational) database systems often results in awkward query expressions and bad performance. For this reason, considerable research has been dedicated to supporting sequence data in DBMSs in the last decade. Unfortunately, some important requirements from applications are neglected, i.e., how to support sequence data viewed as *continuous* under user-defined interpolation assumptions, and how to perform subsequence extraction efficiently based on the conditions on the value domain. We term these kind of queries as *value queries* (in contrast to *shape queries* that look for general patterns of sequences).

This thesis presents pioneering work on supporting *value queries* on 1-D sequence data based on arbitrary user-defined interpolation functions. An innovative indexing technique, termed the *IP-index*, is proposed. The motivation for the IP-index is to support efficient calculation of implicit values of sequence data under user-defined interpolation assumptions. The IP-index can be implemented on top of any existing ordered indexing structure such as a B^+-tree. We have implemented the IP-index in both a disk-resident database system (SHORE) and a main-memory database system (AMOS). The highlights of the IP-index — fast insertion, fast search, and space efficiency are verified by experiments. These properties of the IP-index make it particularly suitable for *large* sequence data.

Based on the work of the IP-index, we introduce an extended SELECT operator, σ^*, for sequence data. The σ^* operator, $\sigma^*_{cond}(\mathrm{TS})$, retrieves *subsequences* (time intervals) where the values inside those intervals satisfy the condition *cond*. Experiments made on SHORE using both synthetic and real-life time sequences show that the σ^* operator (supported by the IP-index) dramatically improves the performance of value queries. A cost model for the σ^* operator is developed in order to be able to optimize complex queries. Optimizations of time window queries and sequence joins are investigated and verified by experiments.

Another contribution of this thesis is on physical organization of sequence data. We propose a multi-level dynamic array structure for *dynamic*, *irregular* time sequences. This data structure is highly space efficient and meets the challenge of supporting both *efficient random access* and *fast appending*. Other relevant issues such as management of large objects in DBMS, physical organization of secondary indexes, and the impact of main-memory or disk-resident DBMS on sequence data structures are also investigated.

A thorough application study on “terrain-aided navigation” is presented to show that the IP-index is applicable to other application domains.

Acknowledgements

First of all, I would like to express my deep gratitude to my supervisor, Prof. Tore Risch, for guiding me into the area of databases and being supportive during my four years of graduate study at Linköping University. His extensive knowledge and endless enthusiasm in research work has always been an inspiration for me. I am also indebted to him for his unflagging words of encouragement whenever he thinks I needed them, and his warm recommendations which helped me to find my ideal job in industry (although he would prefer me to stay in academia).

I would also like to thank my co-supervisor, Prof. Zebo Peng, for providing valuable comments for both my Licentiate and doctoral theses and showing interest in my work. I also appreciate the interesting discussions with Dr. Per-Olof Fjällström on algorithms and complexity during the early stage of my research. Special thanks to Ivan Rankin for checking the English in this thesis.

I would like to thank all my (current and former) colleagues for contributions and help. Among them, I am especially grateful to Martin Sköld and Magnus Werner, who have shown the utmost patience with my many questions during the early stage of my research and study. For technical discussions, I would like to thank Olof Johansson for suggesting the “slab method” which led to the idea of the IP-index, and Henrik André-Jönsson for discussions on similarity search on time sequences. Many thanks to Martin Sköld who has shared with me all the frustration and excitement during my research, and offered his kindly help whenever he could.

I had the great fortune to meet Prof. Richard Snodgrass (Univ. of Arizona) at a summer school in Italy in 1995. Prof. Snodgrass has provided me insightful comments on my work and helped shape my research direction despite his busy schedule. I would like also to thank Prof. Christian S. Jensen at Aalborg University, Denmark, for reading my doctoral thesis and for willing to be my opponent.

Being a member of the ISIS project, I would especially like to thank Prof. Lennart Ljung and Niclas Bergman at the Dept. of Electrical Engineering (ISY) for pleasant and inspiring discussions on applying database techniques in terrain-

aided navigation.

I am also grateful to all staffs at the Dept. of Computer Science (IDA), especially Lillemor Wallgren and Anne Eskilsson, for creating a friendly, warm environment for foreign students. I am indebted to Dr. Anders Törne, who accepted me as a guest student at the very beginning (otherwise this thesis would not have existed).

I consider myself very fortunate because of the friendship I have experienced. Among my former colleagues, I would like to thank Magnus (and his wife Pernilla) for being supportive when I needed friendship; and Gustav, for teaching me tennis and sharing laughter. Among my Chinese friends, I would like to thank YuCheng, RongFeng, Dan, Li, TianRuo, Man, and many others, for being supportive and sharing a colourful life away from China. I am also indebted to my swedish "host family" — Ms. Christina Grill, who has shared my tears and laughter all these years.

Among the many blessings that have happened to me during the last few years when I was pursuing my doctoral degree, the best part is that I met a wonderful man — my fiancé Martin Sköld. Martin has helped me a great deal in every aspect of my life in Sweden, including my study, my research, and my personal life. Where would I be without him?

Finally, I would like to thank my parents and my sister for their encouragement, their endless support, and always believing in me.

Ling Lin

Linköping
December 1998

Contents

Chapter 1

Introduction

The motivation for this thesis is to address a neglected issue in database management systems — to support sequence data viewed as *continuous* under arbitrary user-defined interpolation assumptions. For understanding of this thesis work, this chapter provides background knowledge such as the evolution of database technology, the application domains for sequence data, and the concept of indexing. The main contributions and thesis outline are listed at the end of this chapter.

1.1 DBMS

In concept, a *database management system* (DBMS) is a general-purpose software system that facilitates the processes of storing and manipulating data. The primary goal of a DBMS is to provide the user with both *convenient* and *efficient* access to large volume of complex data. In addition, a DBMS must provide the safety of the information stored in case of system crashes or unauthorized access. Other services such as data integrity, concurrent access of shared information, are also carried out by the DBMS.

1.1.1 Historical Overview

The evolution of DBMS technology is illustrated in Fig. 1.1.

In the 1960s, data management was carried out at *file processing* level by conventional operating systems. Indexed files provided a simple way to store

records on disk with fast look up facility. The advanced indexed file systems provide the most basic features of modern database systems such as managing fixed-length records with various types, providing persistent storage and indexes, and performing locking for concurrent access.

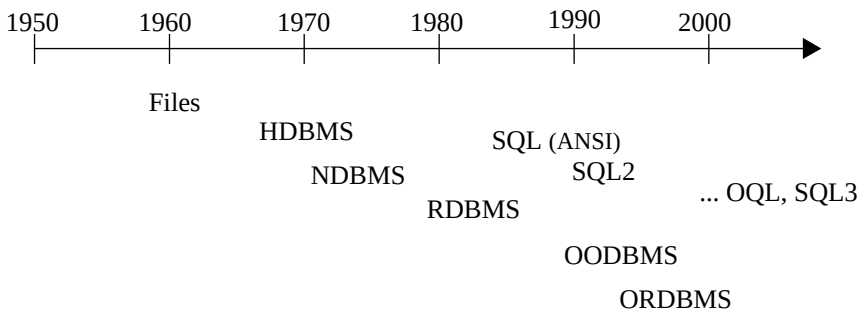


Fig. 1.1: The evolution of DBMS technology

In the 1970s, the first complete database management systems appeared, using the *network* (NDBMS) and *hierarchical* (HDBMS) data models [46]. They provided the previous facilities of file systems plus several more sophisticated ones: record identifiers and link structures between records, multiple files treated as a single database, user authorization, and transactions for database recovery and consistency.

In the 1980s, *relational DBMSs* (RDBMSs) [46] started to dominate the database product market. The relational data model is based on *relations*. A relation is a table consisting of *rows* and *columns*, where each column contains a particular data type. Therefore, a *relational database* consists of a set of tables, and each table consists of a set of columns. An example of a relational table would be *employee(name, age, salary)*. This table manages information about employees in a certain company.

Operations on relational tables constitute the *relational algebra* [34]. The major operations in the relational algebra include *selection*, *projection*, and *join*. A *selection* retrieves the rows in a table where the columns in those rows satisfy some predicates (an example predicate could be "*name* = 'John'"). A *projection* returns its argument table with certain attributes left out. A *join* combines tables by connecting their rows where the columns of these rows satisfy some predicates (an example of a join predicate could be "employee.name = manager.name").

The relational DBMS achieved its popularity by its *simplicity* in database

design, its data *independency* between physical and logical level, and a high-level *query language (SQL)*. Compared to earlier database management systems such as NDBMSs or HDBMSs, the query language SQL frees the end-user from getting into the low-level storage details. In this way the application programs do not need to be rewritten when low level implementations change.

Starting from late 1980s, new application domains such as computer-aided design, scientific and statistic applications, multimedia applications for image, audio, and video data, require more capabilities than relational DBMSs provide. These new requirements include complex structures for objects, new data types, and new access methods. As a consequence, a new generation of database technology — the *object-oriented DBMS* (OODBMS) [39] emerged.

Most OODBMSs are implemented by extending some object-oriented programming languages (e.g., C++ or Smalltalk) with database functionalities such as data persistence, concurrency control, and recovery. Compared to RDBMSs, OODBMSs are more powerful in data modelling and have higher performance. For example, objects in OODBMSs may have arbitrary complex structures; while information about a complex object in RDBMSs is often *scattered* over many relations or records, leading to a loss of direct correspondence between a real-world object and its database representation. However, OODBMSs normally do not have the *declarative* access power and data *independency* as RDBMSs do. Data *security* is also a problem in OODBMSs because application programs and the database share the same address space [39].

It can be seen that RDBMSs and OODBMSs both have their strength and weakness. RDBMSs are more suitable for traditional business applications which are characterized in ad-hoc queries, short-duration transactions, high throughputs, and high security. OODBMSs are more suitable for applications with complex data, high performance and long transactions such as CAD.

1.1.2 Object-Relational DBMS

To combine the strength from both RDBMSs and OODBMSs, a new generation of database technology, the *object-relational DBMS* (ORDBMS) emerged in the 1990s. ORDBMSs can be seen as a marriage of the declarative access power from the relational world and the complex data modelling power from the object world.

In the relational world, the *abstract data type* (ADT) [128] is the key technology to extending relational DBMSs with more powerful modelling and processing capabilities. The relational query language standard, *SQL2*, is under way to be extended to a new standard, *SQL3*, which supports object-oriented concepts such as object identifiers, classes, type hierarchies and inheritance. According

to the preliminary draft of SQL3 (which is likely to be finished in a few years), columns in a relation can have *complex types*, the structure of a relation is extended to allow *nested relations*, type hierarchies and inheritance are also supported. See [118] for details about the extensions. These extensions bring challenges to almost every aspect of a relational DBMS, including its data model, query processing techniques, and storage management. In fact, a relational DBMS has to be redesigned from scratch in order to achieve good performance. An example of this kind of system is Illustra [64].

In the object world, the Object Data Management Group (ODMG) is proposing a standardized query language for OODBMSs, named *OQL*. OQL has features in common with SQL3. It can be seen that relational DBMSs (RDBMSs) and object-oriented DBMSs (OODBMSs) are merging into object-relational DBMSs (ORDBMSs), which is the future trend of the database technology.

The ORDBMS technology is not mature yet. In fact, many extensible database systems claim that they are object-relational but they are not. According to Stonebraker [39], an object-relational database system should include the following features: 1) support for base type extensions in an SQL context; 2) support for complex objects in an SQL context; 2) support for inheritance in an SQL context; 4) support for a production rule system.

Among the new data types that ORDBMSs support, the most widely studied one is *time series*, serving business applications such as stock price indexes and bank interest rates, or scientific applications where data are generated from sensors. From the discussion of the next section, we shall see that some important requirements from the time series applications are *overlooked* in both research prototypes and commercial products.

1.2 Sequence Data

Data over ordered domains such as time or linear positions are termed *sequence data*. The most commonly seen sequence data are *time sequences (time series)*[1] where data are ordered by *time*. Time sequences appear in many application domains: 1) business applications such as stock price, product sales, or bank interest rates; 2) scientific data from sensor readings such as climate measurements or collision of particle beams; 3) medical data such as temperature readings of patients or cardiology data; 4) event sequences in automatic control, process supervision, or telecom network monitoring. An example time sequence is shown in Fig. 1.2, which represents the temperature reading of a patient in a hospital.

1. The subtle difference between these two terms will be discussed in Chapter 2. In this chapter, these two terms are used interchangeably.

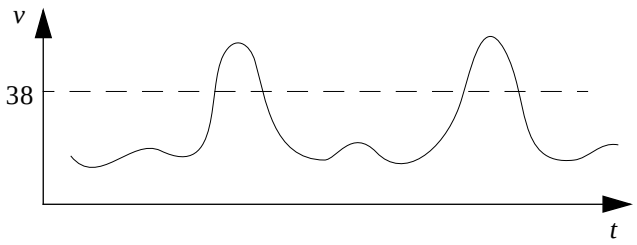


Fig. 1.2: The temperature reading of a patient in a hospital

Sequence data require special treatments which are not provided by traditional DBMSs. Modelling sequence data in traditional (relational) database systems often results in awkward query expressions and bad performance [110]. In fact, most sequence data in real-life applications are managed by *file systems* or *special purpose management systems* instead of by DBMSs.

Realizing this deficiency, considerable research has been dedicated to supporting sequence data in DBMSs in the last decade. In what follows, we shall give an overview of that work and point out what is missing. This will bring out the motivations for the thesis.

1.2.1 Active Research on Sequence Data

Segev and Shoshani [105] proposed the "time sequence" data model to model temporal information. A *time sequence (TS)* is denoted as <*s*, *(t, a)**> where *s* denotes a *surrogate* and *(t, a)** denotes the sequence of values associated with the surrogate. A collection of time sequences for the same surrogate class is defined as a *time sequence collection (TSC)*. Operations over TSCs were defined in [105], such as *extraction* of sub-sequences, *aggregation*, *composition* of two sequences, etc. Storage management of time sequences was studied in [117]. The time sequence data model is independent of any existing data models (such as the relational data model) and serves as the data model of this thesis work. This data model will be discussed more in detail in Chapter 2.

An active research area on time sequences deals with *similarity search*, i.e., finding similar patterns in different time series. Similarity search is essential in discovering and predict the risk, causality, and trend associated with a specific pattern. Several approaches have been suggested. Agrawal et al. [4][51] use Discrete Fourier Transform and compare the first few coefficients in a multi-dimensional space to check the similarity of time series. In [112] time series are transformed into some feature-preserved functions to achieve efficiency in storage and indexing. In [6] a shape language has been defined to express time series and feature queries.

Seshadri et al. [110] developed a sequence database system named SEQ. Various issues in managing sequence data in DBMSs such as data models, query languages, and implementations were investigated. In SEQ, sequence data were modelled as an *abstract data type* (ADT), and supported by common operators such as *subsequence extraction, aggregate, and composition*. A sequence query language, *SEQUIN*, was developed to specify these operations. Important issues such as query optimization were investigated. This system was later evolved to PREDATOR [111] which supports other types of non-traditional data types such as image, audio, and spatial data.

Supporting of sequence data has been an active research subject in ORDBMSs. In fact, many commercial database systems have been extended to support the *time series* data type. Examples are Informix's TimeSeries DataBlade [65], Oracle's TimeSeries DataCartrige [95], and IBM's TimeSeries DataExtender. Clearly there is a high demand from applications to support sequence data.

1.2.2 What is Missing?

It can be seen from the above overview that supporting *sequence data* has attracted substantial research interest during the last few years. Unfortunately, an important requirement from applications is *neglected*, i.e., how to support sequence data viewed as *continuous* under arbitrary user-defined interpolation assumptions. In most research literature, sequences are treated as discrete points and operations on sequences are defined on the discrete model. Among the few research papers ([21][32][33]) that address interpolation issues, only the "step-wise constant" interpolation is assumed, while most real-life applications require more sophisticated interpolation functions such as linear interpolation or moving average. Therefore, it is important that a DBMS should understand the semantics of a continuous sequence and support user-defined interpolation functions.

Another interesting issue is how to extract sub-sequences based on the condition on the *value* domain. Most research on constructing sub-sequences is based on the condition on the *ordering* domain (such as *time* domain). However constructing sub-sequences based on conditions on the *value* domain is highly desirable for real-life applications. Two examples are given below.

1. The Temperature Sequence shown in Fig. 1.3.

   In [112] an example is given to find the pattern of "goalpost fever" in a patient's temperature reading. "Goalpost fever" is one of the symptoms of Hodgkin's disease, behaving as two consecutive fevers during 24 hours. This query was formulated as a shape query in [112] as "finding those subsequences with exactly two peaks".

   However, since a "fever" means the body temperature is higher than 38°C,

this query can also be formulated as "finding the two time intervals when the *values* inside the intervals are greater than 38 and the distance between them is less than 24 hours".

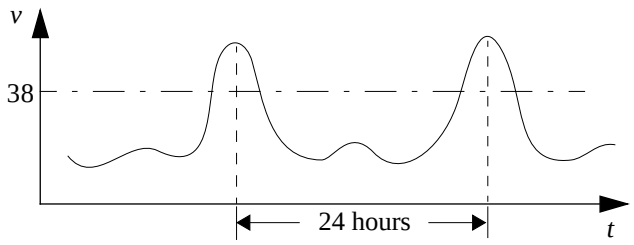


Fig. 1.3: The "goalpost fe ver" pattern

2. The Engine Sequence shown in Fig. 1.4.

   Fig. 1.4 shows a periodic time sequence representing the pressure of a cylinder inside an engine. The data is collected by a sensor in a real-life application [48]. The pressure of the cylinder changes with its angle periodically (360°) and reaches a peak once in every period. On monitoring the behaviour of the engine, an interesting query would be "when did the pressure reach its peak in every period?" [48].

   It can be seen from Fig. 1.4 that all peaks have the property that v > 1.5. So this query could be reformulated as "when were the *values* greater than 1.5?".

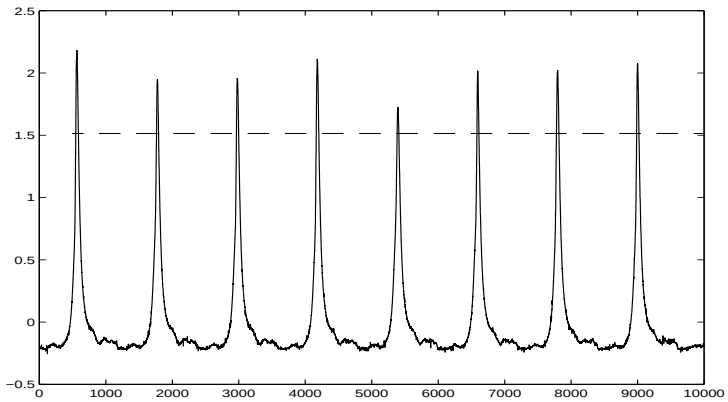


Fig. 1.4: The pressure sequence

These two examples demonstrate the importance of supporting sub-sequence extraction based on the conditions on the value dimension. We term the queries

concerning the value dimension of a sequence as *value queries* (in contrast to *shape queries* that look for general patterns in the sequence). In most applications, value queries involve *implicit values* introduced by interpolation assumptions on the sequence.

It is difficult to support value queries *efficiently* because *linear scanning* seems like the only obvious solution, which will result in bad performance, especially for large sequences. Clearly, to speed up processing time of value queries, some kind of secondary index [118] has to be built for sequence data. [117] is one of the very few research papers mentioning the issue of value queries. Unfortunately, it claims that "a secondary index over the data values is not needed in most applications — such an index can potentially be very expensive in terms of storage, because the number of entries for such an index is in the order of the number of data value". It also claims that "such indexes (if absolutely necessary) would use conventional indexing methods".

Although the above claims seem reasonably true at first glance, we argue in this thesis that they are actually not. First of all, we claim that a secondary index is *definitely needed* in the value dimension for most sequence data. This is because queries on a sequence are often based on the conditions on the *value* dimension rather than on the ordering dimension, and the efficiency issue is essential for long sequences. Secondly, we will show that a conventional secondary index *cannot* deal with value queries. The main reason is that a conventional secondary index does not preserve the ordering semantics of a sequence and cannot support interpolation.

In this thesis we propose an innovative indexing technique, termed the *IP-index*, to efficiently support value queries based on user-defined interpolation functions. Surprisingly, the IP-index is not expensive in terms of storage as [117] predicated. In fact, the size of the IP-index is generally small even for large sequences. Meanwhile, the IP-index is very efficient, i.e., the index insertion and search time is very small, even for large sequence data.

The *efficiency* requirement for sequence data is essential because most application sequences are large. Management of sequence data should remain efficient regardless of the growing of the sequences. This implies challenges in almost every aspect of a DBMS design, from physical storage organization, index design, to query optimization. We shall address these issues in the corresponding chapters.

1.3 Indexing

Since an essential part of this thesis is on indexing sequence data, this section provides background knowledge on the concept of indexing in DBMSs.

1.3.1 Introduction to Indexing

As we all know, data in databases are usually of large volume; retrieval of data can be very *slow* without an index. Basically, an index for a DBMS is like a catalogue for a library. When we look for a book in a library, we look in the catalogue based on the author's name, or the title, etc.

The idea of indexing in DBMSs is best illustrated by a simple example. Suppose we have a relational database *employee(name, age, salary)*. We need to find employees according to their *ages*. Then, we can build an index as shown in Fig. 1.5, where the values in the *age* field are used as *keys*, and every key is associated with data pointers that point to the corresponding employee records. Therefore, if we want to find all employees that are of age 30, we can search the index to find the entry "30" and follow the points to the data. Without this "age" index, we have to scan the entire file, which will be very slow when the file is large.

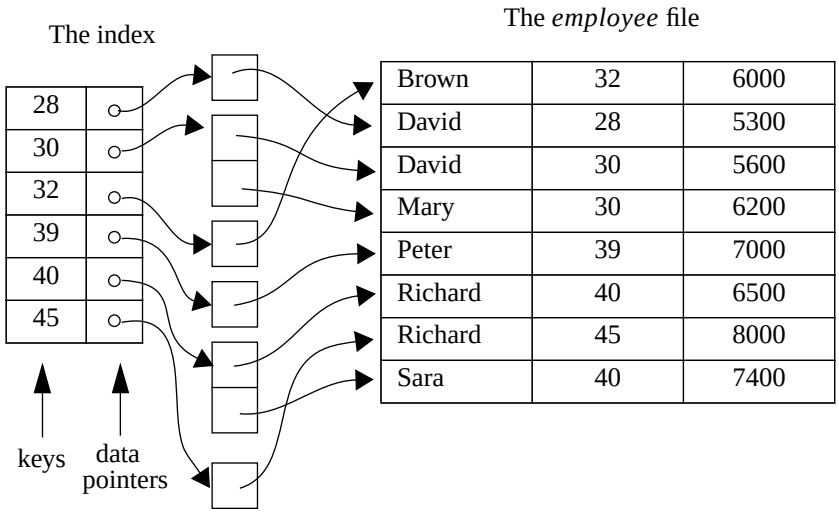


Fig. 1.5: An index on *employee* file, on the field *age*

Many kinds of indexes exist in DBMSs. These indexes can be classified into two categories: *ordered indexes* and *hash indexes*. The above example (Fig. 1.5) is an ordered index where the index is based on *the ordering of the keys* (i.e., the *age*). Hash indexes, on the other hand, are based on distributions of keys in different buckets according to some hash function. We will not discuss hash indexes here because they are not directly relevant to this thesis work.

Several structures can be used to implement an ordered index, such as an array or a tree. The most popular index structure in DBMSs is the B^+-tree [35]. Basically, a B^+-tree is a balanced tree in which every path from the root to a leaf is of the same length. All the keys appear in the leaf nodes. Each key is associated with data pointers that point to the corresponding data items in the file. The nonleaf nodes of a B^+-tree form a multilevel (sparse) index on the leaf nodes. The B^+-tree implementation of the example index (Fig. 1.5) is shown in Fig. 1.6. An important property of a B^+-tree is that each node is at least half full. Leaf nodes are linked together to allow sequential access.

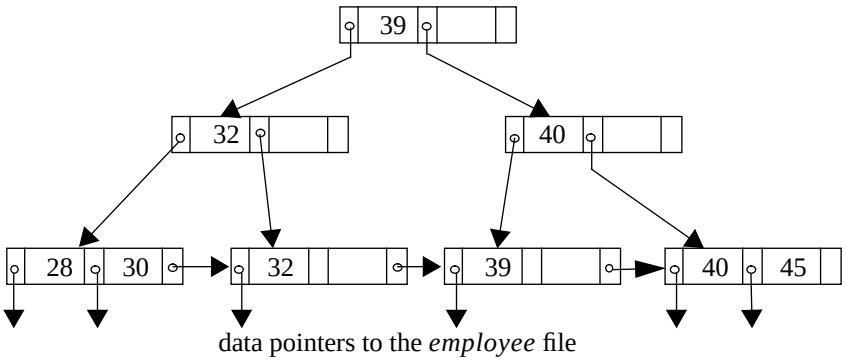


Fig. 1.6: The B^+-tree implementation of the *age* index on Fig. 1.5

The reason why the B^+-tree is so popular in DBMSs is that it is a dynamic, ordered index structure which maintains efficiency despite its insertion and deletion.

1.3.2 Criteria of a Good Index

As mentioned before, many kinds of indexes exist in DBMSs. How do we choose between different kinds of indexing methods for an application? What is a good index design? The most important criteria are the following:

- Access time — the time taken to find a particular data item using the index. A good index should take very little time to find the data item needed, no matter how large the database is.
- Insertion time — the time taken to insert a new data item into the index. The insertion time of a good index is expected to be small.
- Deletion time — the time taken to remove a data item from the index. As for the case of insertion time, the deletion time of a good index should be small.

- Space overhead — the additional space occupied by the indexing structure. By having an index, we gain performance improvement by sacrificing space. Usually it is worth doing so since the performance improvement is usually substantial. But space inefficiency of an index will inhibit it from practical use.

The property of small insertion and search time is especially important for time sequence applications. This is because time sequences are usually very long, the property of small insertion and search time should *scale up*[1] with the growing of sequences. On the other hand, index deletion time is less interesting because deletion of a data item in time sequences occurs very seldom.

1.4 Main Contributions and Thesis Outline

In this section we summarize the main contributions and present the thesis outline.

1.4.1 Main Contributions

This thesis presents pioneering work on supporting *value queries* on 1-D sequence data based on arbitrary user-defined interpolation functions. An innovative indexing technique, the IP-index, is proposed. The motivation of the IP-index is to support efficient calculation of implicit values of sequence data under user-defined interpolation functions. The idea of the IP-index is general, and it can be implemented on top of any ordered indexing structure such as a B^+-tree.

We have implemented the IP-index in both a disk-resident database system [22] and a main-memory database system [49]. We measured the insertion and search time of the IP-index and show that: for a time sequence with limited range and precision (most real-life time sequences have this property), the *insertion* and *search* time of the IP-index remains *small* regardless of the growing of the sequence. This indicates that the performance of the IP-index scales up gracefully with the cardinality of the time sequence. We also investigated the space usage of the IP-index to show that it is practical to build IP-indexes for large sequences.

Based on the work of the IP-index, we introduce an extended SELECT operator, σ*, on a time sequence (TS). The σ* operator, σ^*_{cond}(TS*)*, retrieves *subsequences* (time intervals) where the values inside those intervals satisfy the

1. The term “scale up” here means the insertion and search time should stay *small* regardless of the growing of the time sequence.

condition *cond*. The σ* operator supports arbitrary user-defined interpolation functions on TS. Experiments made on SHORE [22] using both synthetic and real-life time sequences show that the σ* operator (supported by the IP-index) dramatically improves the performance of value queries. The performance gain is even more dramatic for large sequences with *small answer sets*, while most submitted value queries in real-life applications *are* for small answer sets. Another promising observation is that the performance of σ* for *the first few answers* is stable, regardless of the *positions* where the first few answers appear in the time sequence. This shows that the IP-index is essential in the situations when the time sequence is long and the query processing time is limited.

On query optimization, we develop a cost model for the σ* operator in order to be able to optimize complex queries. An interesting observation is that the cost of a range query $\sigma^*_{v>v'}$(TS) is nearly the *same* as the cost of the exact query $\sigma^*_{v=v'}$(TS). This indicates that processing *range queries* is very efficient using the IP-index, especially for large sequences. We also investigate optimizations of time window queries and complex value queries. Time window queries can be optimized by pushing the *time window* into the IP operator (a component of the σ* operator), thus reducing the number of anchor-states retrieved. Complex queries (sequence joins) can be optimized by choosing a good join order according to the cost and the selectivity factors [107] of the σ* operators involved. Experiments are performed to verify the above optimization strategies.

On physical organization, we propose a multi-level dynamic array structure for *dynamic, irregular* time sequences. The highlight of this data structure is that it is highly *space efficient* and supports both *efficient random access* and *fast appending*. Other relevant issues such as management of large objects in DBMSs, physical organization of secondary indexes, and the impact of main-memory or disk-resident DBMSs on sequence data structures are also investigated.

Related work is discussed in depth in this thesis. Extensive research on similarity search on time series complements our work nicely. Similarity search is based on the general *shapes* (features) of a sequence, while our work is based on individual *values* of a sequence. These two aspects of support for sequence data are both highly needed in real-life applications. Research on object-relational DBMSs to support the abstract data type of *time series* is covered. Our work contributes specifically to this area by the extension of supporting *value queries* on time series, which was a neglected issue up till now. A sequence database system, SEQ, also indicates the need of the IP-index.

A thorough application study [83] on “terrain-aided navigation” is presented to show that the IP-index is applicable to other application domains. The IP-index improves the performance of terrain-aided navigation by finding the *starting*

positions for the matching algorithm (the bayesian approach [20]) efficiently. Experiments on a real terrain map and simulated track files are performed to verify the efficiency of the approach.

The work presented in this thesis is based on the following publications:

I. L. Lin, T. Risch, M. Sköld, and D. Badal
Indexing Values of Time Sequences
Proceedings of 5th International Conference on Information and Knowledge Management, Rockville, USA, Nov. 1996.

II. L. Lin and T. Risch
Using a Sequential Index in Terrain-aided Navigation
Proceedings of 6th International Conference on Information and Knowledge Management, Las Vegas, USA, Nov. 1997.

III. L. Lin
Implementing the IP-index in SHORE
Linköping Electronic Press, Vol. 2, No. 17, 1997.

IV. L. Lin
Study of Supporting Sequences in DBMS — Data Models, Query Languages, and Storage Management
Linköping Electronic Press, Vol. 3, No. 4, 1998.

V. L. Lin and T. Risch
Querying Continuous Time Sequences
Proceedings of 24th International Conference on Very Large Data Bases, New York City, USA, August, 1998.

Publication I is the first paper that appeared at an international conference where the idea of the *IP-index* is introduced.

Publication II presents how the IP-index can be applied to *terrain-aided navigation* to improve the real-time performance.

Publication III documents how the IP-index is implemented in a persistent object system SHORE [22] using SDL (a variant of the ODMG ODL) and C++.

Publication IV summarizes recent research on supporting *sequence data* in DBMSs, covering issues such as data models, query languages, query optimization, and storage management. A sequence database system SEQ is described.

Publication V proposes the extended SELECT operator, σ* *operator,* for the abstract data type of *time sequences* in an extensible DBMS. The σ* *operator* retrieves *sub-sequences* (time intervals) in a time sequence TS where the values inside those sub-sequences satisfy some conditions. User-defined interpolation functions are supported. The σ* operator is efficiently supported by the IP-index. Query optimization issues are investigated and verified by experiments on SHORE.

1.4.2 Thesis Outline

This thesis is organized as follows:

Chapter 1 (this chapter) provides background knowledge for understand of this thesis work and points out the motivations and main contributions of this thesis.

Chapter 2 introduces the *time sequence* data model, which serves as the data model of this thesis. Different types of time sequences are classified based on properties such as *regularity, static/dynamic*, and the *interpolation function* applied.

Chapter 3 is the essence of the thesis. It introduces the idea of the *IP-index* based on the time sequence data model. The central concept is the *anchor-state sequence*. The insertion and search algorithms of the IP-index are presented. The important relationship between the IP-index and the precision of values v_is are investigated. The IP-index is compared with conventional secondary indexes and related indexes in the area of temporal databases, spatial databases and computational geometry.

Chapter 4 demonstrates the highlights of the IP-index: *fast insertion, fast search,* and *space efficiency.* These properties show that the IP-index is not only an elegant idea but also a practical solution for large sequence data. Experimental results on both a main-memory database system and a disk-based database system are presented. This chapter shows that an index on the value domain of a time sequence is not necessary expensive and impractical, as claimed by [117].

Chapter 5 shows how to solve various kinds of *value queries* efficiently by using the IP-index. In particular, it demonstrates the importance of the IP-index for *range queries* (i.e., sub-sequence extraction based on a value range). Other queries that benefit from the IP-index include time window queries, and amplitude-sensitive shape queries.

Chapter 6 introduces the extended SELECT operator, σ*, which retrieves *sub-sequences* (time intervals) where the values inside those intervals satisfy some

conditions. Performance measurements made on SHORE [22] using both synthetic and real-life time sequences with large cardinalities are presented to demonstrate the efficiency of the $\sigma*$ operator. The $\sigma*$ operator is compared to related operators proposed in the area of temporal databases.

Chapter 7 investigates relevant physical organization issues, including physical organization of time sequences, physical structure of secondary indexes, and physical organization of anchor-state sequences. We propose a *multi-level* array structure for dynamic, irregular time sequences. This data structure meets the challenge of supporting both *fast appending* and *efficient random access*. Other relevant issues such as storage management of large objects and the impact of main-memory or disk-resident DBMSs on sequence data structures are also investigated.

Chapter 8 is about query optimization. The cost model of the $\sigma*$ operator is developed. Optimizations of time window queries and complex sequence queries are investigated and verified by experiments.

Chapter 9 discusses related work in depth.

Chapter 10 presents a thorough application study where the IP-index is applied to *terrain-aided navigation*.

Chapter 11 concludes this thesis and discusses future work.

Chapter 2

Time Sequences

This chapter introduces the data model of *time sequences*. This data model aims at capturing the ordered semantics of temporal data and defining operators over them. It is independent of any existing data model (such as the relational data model or the object-oriented data model) and serves as the basic data model of this thesis work. Properties of time sequences are studied in this chapter, such as time granularity, lifespan, regularity, static/dynamic, and interpolation assumptions.

A relevant issue is the modelling of *time* in DBMSs. We discuss discrete/continuous time models and point out why interpolation is important for both models.

2.1 The Time Sequence Data Model

The data model of "time sequence" was proposed in the middle of 1980s when research on temporal databases [134] started. This data model first appeared in the paper "Temporal Data Management" [117] and later appeared as a chapter [105] in the first book on temporal databases [134]. In this data model, a temporal data value for an object is defined as a triplet <*s, t, a*>, where *s* is the object *surrogate*, *t* is the time, and *a* is the attribute value for *s* at time *t*. Thus, the history of data values for the object *s* is defined as <*s, (t, a)**>. The sequence *(t, a)** is termed a *time sequence* (TS). An example time sequence is the following: suppose that an employee 'John' has been working in a certain company for many years, and he has received salary raises several times, then his salary information could form a time sequence as the following:

Example 2.1: (01/05/95, 1200), (09/20//96, 1400), (05/13//97, 2300),... □

A collection of time sequences for the same surrogate class is defined as a *time sequence collection (TSC)*. For example, the salary histories of all employees in a certain company would form a time sequence collection.

Compared to most research work on temporal databases that extends existing data models (e.g., the relational data model, the object-oriented data model, or the functional data model) to support temporal features, the work of [105][117] takes a different approach. They start with the understanding and specification of the semantics of temporal data, thus leading to a precise characterization of the properties of temporal data, and define operators over them. In this way their work is not influenced by traditional models that were not specially designed for modelling temporal information.

Several properties of time sequences were studied in [105][117]. They are:

- *Time Granularity*

  The time granularity specifies the granularity of the time points of a TS, that is, the points in time that can potentially have data values. Two time granularities were identified in [106] — *ordinal* and *calendar*. The ordinal representation simply signifies that the potential time points are counted by integer ordinal position (1^{st}, 2^{nd}, 3^{rd},....). The calendar representation can assume the usual calendar time hierarchy values: year, month, day,..., second, and so on.

- *Lifespan*

  The lifespan of a TS is specified by a *start_time* and an *end_time* defining the range of valid time points of that TS. The *start_times* and *end_times* can be represented as either ordinal or calendar.

- *Regularity*

  A time sequence can be either regular or irregular. A *regular* TS is a TS where the values are measured in regular time intervals. Otherwise the TS is irregular. The regularity of TS is very important and most relevant to our work. It is further discussed in Section 2.1.1.

- *Type*

  The type of a TS determines the data values of the TS for time points that do not have explicit data values. In general, there is an interpolation function associated with each TS. Some of the interpolation functions are very common, which will be discussed in Section 2.1.3.

Data manipulation on time sequences is defined by operators over time sequence collections (TSCs) in [106]. Every operator over one or more source TSCs will produce a single target TSC. The most important operators in [106] are:

- *Selection*

  The selection operator extracts parts of a TSC that satisfy a predicate referencing *s* and/or *t* and/or *a* values. For interpolated time sequences, selection based on a predicate referencing *a* values would need the IP-index.

- *Aggregation*

  The aggregation operator can be applied over groups in the time dimension or the surrogate dimension. For aggregation over the time dimension, the new time points in the target TSC will be of granularity higher than that of the source TSC.

- *Composition*

  The composition operator enables manipulation of related data that are part of two TSCs (in pair-wise manner). For example, composition of a daily *price* and a daily *quantity* time sequence (for a certain product) will give a daily *revenue* time sequence (for this product).

In the rest of this section we discuss those properties of time sequences that are relevant to our work.

2.1.1 Regularity

As mentioned above, a *regular* TS is a TS where the values are measured in regular time intervals [117]. Examples of regular time sequences are stock prices (where values are obtained for every business day), scientific experiments or simulations (where values are *pulled* at regular time intervals by some mechanism or computer programs), etc. *Irregular* time sequences usually result from manual measurements or from unpredictable events, such as the failure of a detector. Business transaction data, such as items sold in a store, or the salary history of an employee, are typically irregular time sequences as well.

The reason for distinguishing between regular and irregular TSs is that this property affects the physical design of a time sequence profoundly. For example, a regular time sequence $(t_i, a_i)^*$ can be stored as an array $v[i] = a_i$ where the time stamps are “factored out”. This is because t_i can be *computed* by $t_i = t_0 + i * \Delta t$ (Δt is a constant for a regular TS). The retrieval of any data value given its time stamp t_i is easy because the time stamp t_i can be easily mapped into the corresponding position i in the array, i.e., $i = (t_i - t_0) / \Delta t$. On the other hand, irregular time sequences are more difficult to support. First of all, all time

stamps have to be explicitly stored. Secondly, the retrieval of any data value given its time stamp has to be supported by some *indexes* on the time domain. Physical organization of time sequences is further studied in Chapter 7.

Note that in regular time sequences data values can be *null* (missing or unknown). A regular TS containing a large number of *null* values is considered as an *irregular* TS. The reason is that it is preferable to consider this TS as composed of those *non-null* (t_i, a_i) pairs (an irregular TS) instead of as a regular TS with many *null* values.

In addition to affecting physical organization, regularity implies semantic values as well. In fact, most statistical methods for analyzing time sequences can only be applied to regular TSs. This issue will be further discussed in Chapter 2.1.4, where we discuss the differences between a time sequence and a time series.

2.1.2 Static/Dynamic

Another aspect of a time sequence which affects its physical organization is whether it is static or dynamic. By *static* we mean the sequence is fully collected, no more data will be added at the end of the sequence. By *dynamic* we mean the sequence is continuously growing.

Static TSs are easy to implement since we can allocate storage space in advance based on the size of the TS. Dynamic TSs are difficult to implement since the size of storage cannot be determined, yet we would still like to have fast random access to any element in the sequence. This challenging issue will be further explored in Chapter 7, where we develop a multi-level dynamic array structure for dynamic, irregular time sequences.

2.1.3 Interpolation

As mentioned earlier, interpolation functions are often needed in order to derive those values that are not explicitly stored in a TS. In [105][117], the following interpolation assumptions are classified as the most commonly used ones:

- *Step-wise constant —*

  If (t_i, a_i) and (t_k, a_k) are two consecutive pairs in a TS such that $t_i < t_k$, then $a_j = a_i$ for $t_i \leq t_j < t_k$. An example of step-wise constant TS is the salary history of an employee (salary value stays the same until the next change occurs), see Fig. 2.1.

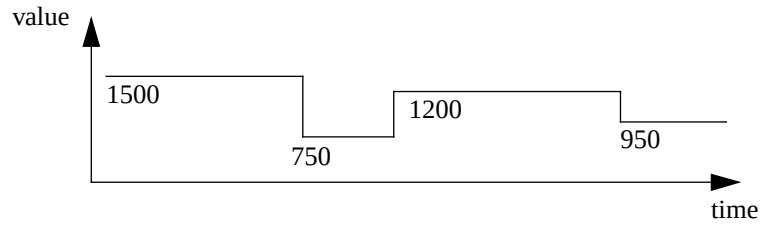


Fig. 2.1: Salary history — step-wise constant

- *Continuous* —

  A continuous function is assumed between (t_i, a_i) and (t_k, a_k) that assigns a_j to t_j $(t_i \leq t_j \leq t_k)$ based on a curve-fitting function. An example of a continuous TS is the sensor data representing measurements of a magnetic field at regular intervals, see Fig. 2.2.

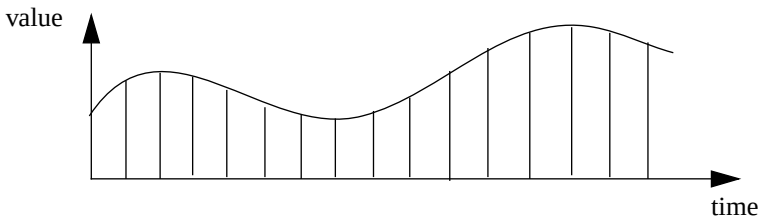


Fig. 2.2: Magnetic field — continuous

- *Discrete* —

  Implicit values cannot be interpolated. An example of a discrete TS is stock price, where values between two sold points cannot be interpolated, see Fig. 2.3.

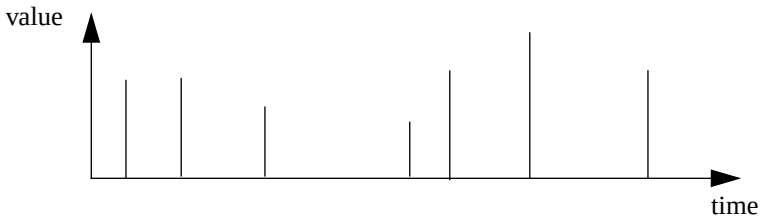


Fig. 2.3: Stock price — discrete

- *User-defined* —

Implicit values in a TS are computed based on user-defined interpolation functions. For example, some applications only require a simple interpolation function such as linear interpolation while other applications may require a higher degree of interpolation function such as moving average or least square.

Therefore, it is important for a DBMS to understand the semantics of a time sequence and support system-defined or user-defined interpolation assumptions.

Actually, the importance of associating interpolation methods with temporal data was pointed out in the early 1980s by Clifford et al. [33] as the "*Comprehension Principle*", i.e. "under any reasonable interpretation a historical database defined over a sequence of states $<S_1, S_2,..., S_n>$ should be considered as modelling an enterprise completely over the entire closed interval $[S_1, S_n]$". It was also mentioned that the mapping from a finite set of moments $< S_1, S_2,..., S_n >$ into the dense interval $[S_1, S_n]$, termed as the "*Continuity Assumption*", could be a accomplished by various interpolation methods.

Although it was pointed out long ago that it is important to support interpolation assumptions, very few *implementation* issues have been addressed. For example: how to support queries based on arbitrary user-defined interpolation assumptions; how to process these queries *efficiently*, especially when the time sequences are very long. These are the motivations for this thesis.

2.1.4 Time Sequences or Time Series?

The difference between the term *time series* and *time sequence* has not been very clear in research literature. Some researchers use these two terms *interchangeably* in the literature. Here we try to clarify the differences between these two terms.

Generally speaking, the term *time sequence* is more *general* than the term *time series*. As mentioned in Section 2.1, time sequences can be classified into *regular* and *irregular* ones. Regular time sequences are those time sequences where values are measured in regular time intervals. The term *time series* refers only to those *regular* time sequences.

Note: The term *time series* refers only to regular time sequences.

Examples of time series are stock prices (collected on every trading day), or scientific data (collected by sensors on regular time intervals). The term *time series* has actually been used for a long time by statisticians. Most analytical methods (such as those provided by special purpose systems such as FAME

[52]) can only be applied to *time series*, not to irregular time sequences. The reason is that these methods assume some regularity between data points in the sequence. For example, the method "moving average" would assume that the data values under manipulation are measured in regular intervals. The method "cross-correlation" would assume that the two source sequences are not only regular but also based on the same time calendar.

Stock data are normally collected on the time unit of *day.* Notice that stock data do not exist on weekends, they only exist on business days. If one generates a time series for the stock market, all the days that the market is closed (weekends and holidays) are removed. In this a regular time sequence (time series) is obtained. Therefore, the *regular* property of stock data is interpreted on the base of *business calendars* only.

Throughout this thesis, we shall use the term *time sequences* when describing our work. This is because our work applies to general time sequences including *regular/irregular*, and *static/dynamic* ones. The work on management of regular time sequences, i.e., *time series,* will be covered separately in Section 9.3.

2.2 Discrete or Continuous Time?

Another issue related to the time sequence data model is to understand the semantics of *time*. As Clifford and Tansel [32] point out:

> *Time is something so taken for granted that its exact nature is highly elusive. It might not be technically difficult to come up with a consistent model having various algebraic operations defined, intuitively it is far from obvious which operations are appropriate, meaningful, and correct.*

The basic questions concerning the modelling of time include the following: what kinds of objects are the members of the set of times? What properties will this set have?

One property of time with little dissension is: time has a *linear* order, i.e., for any two time points t_1 and t_2, either t_1 equals t_2, t_1 is-less-than t_2, or t_2 is-less-than t_1. This ordering is perhaps time's essential property.

As for the members of the set of times, there has been less agreement. According to "A Glossary of Temporal Database Concepts" [67], *three* models of time have been defined: *discrete, dense,* and *continuous*.

Intuitively, *discrete* models of time are isomorphic to the natural numbers, i.e.,

every time moment corresponds to a natural number, which has an unique successor. *Dense* models of time are isomorphic to either the real or rational numbers, with the property that between any two moments of time there is always another time moment. *Continuous* models of time are isomorphic to the real numbers, i.e., both dense and also, unlike the rational numbers, with no "gaps".

Most research literature has adopted the *discrete* time model. Clifford and Tansel [32] argue two reasons for this: 1) it is clear that any recording instrument must have at best a finite sampling quantum; 2) any practical domain (or language) that we might define for time attributes in a historical database would have at most a countably infinite set of names for time moments or time intervals. In [32], it is argued that while it may be philosophically or theoretically interesting to consider a continuum of moments of time, from a practical standpoint the natural numbers seem a more useful candidate for modelling database time.

On the other hand, some research chooses the *dense* or *continuous* time model, such as [33]. The reason is simple, the *discrete* time model is inadequate in the face of the generally accepted notion of continuous time. Clifford and Warren proposed two assumptions — the *Comprehension Principle* and the *Continuity Assumption* [33] (see Section 2.1.3) to view a historical database as modelling an enterprise completely over an interval of the real-time line, and to answer crucial questions such as what are the values of those implicit states that are not explicitly defined (stored).

In a survey by Chomicki on temporal query languages [29], it is argued that the *dense* temporal domain is very useful in many applications but is difficult to implement efficiently since the set of time instances is very large. By developing the IP-index, we provide the ability to *derive* the dense instances from the original discrete sequence, saving both storage space and query processing time.

2.2.1 Interpolation for Discrete/Continuous Time Model

At first glance it might seem that interpolation is only needed for the continuous time model. But this is not true. Interpolation is also important for the discrete time model. Why? Recall that a regular TS can have missing values (Section 2.1.1). For example, a stock price sequence may have missing values for some trading days. The only way to obtain those values is to apply some interpolation function. For irregular time sequences, interpolation is also important. For example, a salary history is interpreted as "step-wise constant" (Fig. 2.1).

Therefore, supporting interpolation is required for both the continuous and dis-

crete time model. Clifford and Tansel [32] propose a discrete time model and emphasize the importance of supporting interpolation for "time-varying attributes" (attributes that vary over time, such as "salary"). They point out:

> *Users must be able to query the database at will with respect to time points or periods, and yet the database cannot possibly store values for every attribute at every point in time. Thus, each attribute must have an associated interpolation function, so that the database system can reconstruct an entire time series over the lifespan of each object from the partial specification store.*

Among the very limited research that addresses interpolation issues on temporal data, most of them assume the simple interpolation function — "step-wise constant". And most of this work addresses semantic issues only, no implementation is done. Our contribution to this area consists of two aspects: 1) we address *implementation* issues such as physical organization and query optimization; 2) it is the first time that *user-defined interpolation functions* are truly supported.

2.2.2 Precision of Time Points

A tricky issue in the modelling of *time* is the *precision* of time points. Since all measurement instruments (such as clocks) have a certain precision, all time stamps stored in the database have a limited precision. Furthermore, all implicit time points that can be interpolated from the explicit time points also have a certain precision limit due to the precision of the explicit points and the precision of real numbers in the computer.

The precision of time points was mentioned in "A Glossary of Temporal Database Concepts" [67]. There a "time point" or "time moment" was named as a *chronon* and defined as "the shortest duration of time supported by a temporal DBMS — it is a nondecomposable unit of time". A particular *chronon* is a subinterval of fixed duration on the time-line. The reason for naming it a *chronon* is, according to [67], clocking instruments invariably report the occurrence of events in terms of time intervals, not time points. Hence, events, even so-called "instantaneous" events, can best be measured as having occurred during an interval.

In Section 6.1.1 we define an interpolated time sequence as an *infinite set*. The reasons are: 1) the *set* concept comes from the fact that time points in the continuous time model will constitute a set eventually due to the precision limit; 2) the *infinite* concept comes from the fact that the precision can be as high as the measurement instruments and the computers possibly have. We do not assume any precision beforehand.

2.3 Summary

This chapter introduced the *time sequence* data model. This data model is independent of any existing data models (such as the relational data model or the object-oriented data model) and serves as the basic data model of this thesis work. Different properties of time sequences were studied, such as *regularity*, *static/dynamic*, and the *interpolation assumptions*. The importance of *interpolation* on time sequences was stressed. The terms *time sequence* and *time series* were clarified based on the author's understanding of relevant literature.

We also discussed two different time models: *discrete* and *continuous*. The interesting point is that interpolation is needed for both the discrete and continuous time models.

Chapter 3

IP-index

This chapter introduces the idea of the IP-index. Generally speaking, the IP-index is designed to support efficient calculation of *implicit* (interpolated) values in large time sequences (or any 1-D sequence data) under user-defined interpolation functions. The insertion and search algorithms of the IP-index are presented. The important relationship between the performance of the IP-index and the precision of values in time sequences are investigated. A comparison of the IP-index with conventional secondary indexes is given. The IP-index is compared to related indexes such as temporal indexes, spatial indexes, and indexes in computational geometry.

3.1 Motivation

As pointed out in Chapter 1, time sequences appear in many application domains. Examples of time sequences include stock price indexes, scientific measurements collected from sensors, or temperature readings of patients in a hospital. In concept, a time sequence (TS) can be modelled as a sequence of states S_i* where $S_i = (t_i, v_i)$ (recall the *time sequence* data model in Section 2.1).

Many applications require time sequences to be seen as *continuous* under arbitrary user-defined interpolation functions. In other words, the *discrete* sequences need to be seen as *continuous* where implicit values can be derived from explicit values by the interpolation assumption. For example, suppose Fig. 3.1 represents a patient's temperature reading in a hospital, a physician will be interested to know:

- When did the patient have the temperature 38°C?

It can be seen from Fig. 3.1 that there are no explicit (stored) time points in the temperature sequence when the values are equal to 38. However, if we apply linear interpolation on this TS, there will be three time points t', t'' and t''' that satisfy this query.

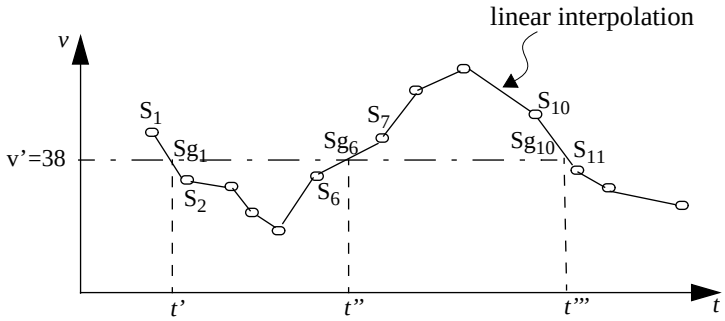


Fig. 3.1: Illustration of a value query

These kind of queries are termed *value queries* [86] (Section 1.2.2). We introduce two notations to denote value queries — **F(t')** and **F^{-1}(v')**: F(t') denotes the value at any time point t', and F^{-1}(v') denotes the time point(s) when the value(s) are equal to v'. Therefore, the above query would be denoted as F^{-1}(38).

The key to processing value queries is to find the neighbor-states where the user-defined interpolation function can be applied. For example, the time point t' in Fig. 3.1 can be calculated by applying linear interpolation on the neighbor states S_1 and S_2, the time point t'' can be calculated by applying linear interpolation on the neighbor states S_6 and S_7, and the time point t''' can be calculated by applying linear interpolation on the neighbor states S_{10} and S_{11}. How can we find these neighbor states efficiently? Obviously, without a suitable index, the only solution is to scan the whole sequence to find those states S_is where S_i.value ≤ v' < S_{i+1}.value. This is very slow when sequences are long. For this reason, we developed the IP-index.

3.2 IP-index

The idea of the IP-index is as follows. Each state S_i in TS is viewed as a point in the two-dimensional plane *t-v* as shown in Fig. 3.2. Each consecutive states S_i, S_{i+1} constitute a line segment Sg_i. Then, if we can find all segments Sg_i that intersect the line v = v', we can answer value queries. For example, in Fig. 3.2, the segments which intersect the line v = v' are <Sg_2, Sg_3>. The answer to the

query $F^{-1}(v')$ will then be:

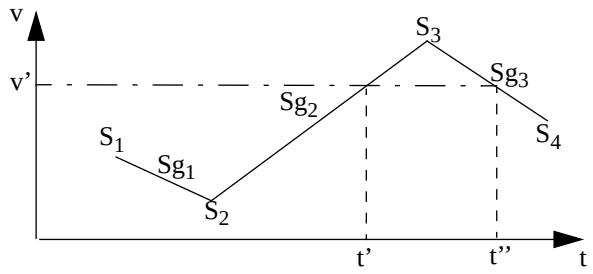


Fig. 3.2: An example of a value query

- If the "step-wise" constant or "discrete" assumption is applied, then $F^{-1}(v') = nil$, since there is no value defined between S_2, S_3 and S_3, S_4 respectively.
- If the "continuous" or "user-defined" interpolation assumption is applied, then $F^{-1}(v') = <t', t''>$, where t' and t'' are calculated by applying some interpolation function (e.g. linear interpolation, or least square) on the states around the segments Sg_2 and Sg_3 respectively.

So, the problem of value queries is transformed into the problem of *finding all the intersecting segments for the line v = v'*. A naive way to solve this problem is to scan the entire TS to check if any two consecutive states S_i, S_{i+1} "contain" v', i.e. if $v_i \le v' < v_{i+1}$, or $v_{i+1} \le v' < v_i$. Such an algorithm, however, has the complexity of $\Theta(N)$, where N is the cardinality of the TS. Below we propose an indexing technique to speed up value queries.

3.2.1 Anchor-State Sequence

If we project each line segment Sg_i on the v-axis, we get non-overlapping intervals $I_j = [k_j, k_{j+1})$, where each k_j is a distinct value of v_i (see $k_1 ... k_4$ in Fig. 3.3). We can see that all values that belong to one interval have the same sequence of intersecting segments (marked to the left in Fig. 3.3). We propose an index, termed the *IP-index*, in which each interval $[k_j, k_{j+1})$ is associated with all segments Sg_i that span[1] it. A simple illustration of the IP-index is shown in Fig. 3.3, where we associate each interval $[k_j, k_{j+1})$ with the sequence of span-

1. We say a segment Sg_i spans an interval I_i when the projection of Sg_i on the v-axis spans the interval I_i, i.e. if $Sg_i = ((t_s, v_s), (t_e, v_e))$ and $I_i = (v_a, v_b)$, then $v_s \le v_a$ and $v_e \ge v_b$.

ning segments Sg_i.

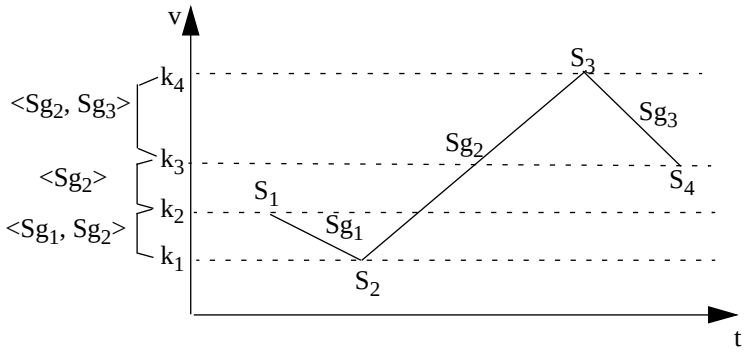


Fig. 3.3: Illustration of anchor-state sequences

Since the segments are consecutive, each segment Sg_i is uniquely identified by its starting state S_i. We use S_i to represent the segment Sg_i in the IP-index. We term the starting states of each segments that intersect the line v = v' as the *anchor-states* of *v'*. Then, the sequence of intersecting segments can be represented as the sequence of anchor-states, which is termed the *anchor-state sequence*. The anchor-state sequence of the queried value v' is denoted as **A(v')**. An anchor-state sequence is a state sequence ordered by time.

Since each interval $[k_j, k_{j+1})$ is uniquely identified by its starting point k_j, we use k_j to represent the interval $[k_j, k_{j+1})$ in the IP-index.

Suppose that $k_1 < k_2 < ... < k_j < ...$ are the ordered, distinct values of v_i in TS. Then each index entry N_j in the IP-index has the form (key, anchors) where

- N_j.key = k_j.
- N_j.anchors is the anchor-state sequence for all v' such that v' ≥ k_j and v' < k_{j+1}. It is also denoted as anchors(k_j).

For example, the anchor-state sequences for the simple TS in Fig. 3.3 are:

anchors(k_1) = <S_1, S_2>
anchors(k_2) = <S_2>
anchors(k_3) = <S_2, S_3>
anchors(k_4) = *nil*

The *cardinality* of an anchor-state sequence is also stored in the IP-index,

denoted as *card(A(k_i))*. For example, in the above IP-index in Fig. 3.3, we have: card(A(k_1))=2, card(A(k_2))=1, card(A(k_3))=2, and card(A(k_4))=0.

The reason for storing cardinality information is that it can be used in query optimization. This will be illustrated in Chapter 8 where optimizations on sequence queries are discussed.

3.2.2 The Limitation of the IP-index

We should point out that if the interpolation method introduces new extreme points (and thus introduces extra segments) to the original time sequence, the IP-index needs to be modified to include the extra segments as well. For example, applying least square interpolation in Fig. 3.4 (TS = <S_1, S_2, S_3, S_4>) leads to some interpolated values (such as v') in the time interval [t_2, t_3) exceeding the range [v_2, v_3]. One way to fix this problem is to include the new extreme point P in the IP-index, i.e., include the new segments S_2P and PS_3 (replacing the old segment S_2S_3). Another possible solution is to replace the query F^{-1}(v') by the approximate query F^{-1}(v'-e<v<v'+e) (see Fig. 3.4) so that the anchor-state of v' can be located. The choice of the value of e' is dependent on the application data. The bigger the value e' is, the more possibility there is to cover the extreme point P.

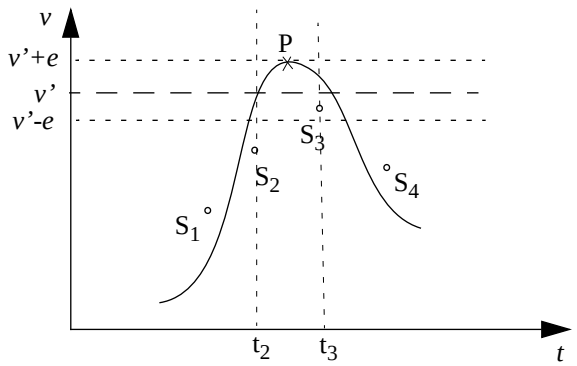


Fig. 3.4: The limitation of the IP-inde x

Fortunately, most applications assume simple forms of interpolation functions such as step-wise constant or linear interpolation. These interpolation functions will not introduce extreme points, thus the IP-index works perfectly well for them. For moving average interpolation functions, a 2-point moving average is no problem because the average of value v_i and v_{i+1} will always be between the range [v_i, v_{i+1}] (suppose $v_i < v_{i+1}$). A 3-point moving average, on the other hand, may introduce extreme points.

3.3 Algorithms

This section presents the insertion and search algorithms of the IP-index.

3.3.1 Insertion Algorithm

Suppose that in Fig. 3.3 the first three states S_1, S_2, and S_3 have already been inserted into the IP-index. Now let us see how to insert the new state S_4. According to the definition of the IP-index (Section 3.2), we already have three index entries with keys v_1 (= k_2), v_2 (= k_1), and v_3 (= k_4) respectively, and we also have anchors(k_1) = <S_1, S_2>, anchors(k_2) = <S_2>, and anchors(k_4) = *nil*. To insert the state $S_4 = (t_4, v_4)$ we need to do the following:

1. The new state S_4 creates a new index entry with the key v_4 (= k_3). This inex entry divides the existing interval [k_2, k_4) into two intervals, [k_2, k_3) and [k_3, k_4).

   The segments that span the new interval [k_2, k_3) are the same as the segments that spanned the old interval [k_2, k_4) (which are already present in the IP-index), i.e., anchors(k_2) = <S_2> stay unchanged.

   The segments that span the new interval [k_3, k_4) are the segments that spanned the old interval [k_2, k_4) plus the new segment Sg_3, i.e., anchors(k_3) = anchors(k_2) $+^1$ S_3 = <S_2> + S_3 = <S_2, S_3>.

2. For all the entries in the IP-index with keys inside the interval [k_3, k_4) (in Fig. 3.3 there happens to be no such key), append S_3 to the end of their associated anchor-state sequences. This is because Sg_3 spans all the sub-intervals inside the interval [k_3, k_4).

The result of the insertion conforms with Fig. 3.3.

The pseudo-code for the IP-index insertion is given in Fig. 3.5. The notation and functions used in the algorithm are:

- *tree* — the index tree (e.g. a B^+-tree) storing the IP-index.
- N — an index entry in the IP-index tree.
- *ts* — the array storing the time sequence.

---

1. We use '+' to denote adding a new element to the end of a sequence.

- $S_i = (t_i, v_i)$ — the new state to be inserted into the IP-index.
- *insert_ts*(*ts*, i, S_i) — inserts the state S_i into array *ts* where ts[i]=(t_i, v_i).
- *exist_key*(*tree*, v_i) — returns TRUE if there already exists an index entry in the IP-index with the key v_i.
- *get_lower*(*tree*, v_i) — searches the IP-index tree to find the index entry N_L where N_L.key= max{N_i.key | N_i.key ≤ v_i, 1 ≤ i ≤ size(*tree*)}.

  This function is used to find the existing interval which needs to be split into two parts; e.g. in Fig. 3.3, *get_lower*(*tree*, v_4).key = k_2. The function returns *nil* if no index entry is found.
- *insert_entry*(*tree*, *k*) — inserts a new index entry into the IP-index tree with key = *k*.
- N.anchors — the anchor-state sequence associated with the index entry N.
- N.anchor_card — the cardinality of N.anchors.
- *append*(*seq*, S_i) — appends the state S_i at the end of the state sequence *seq*.

3.3.2 Search Algorithm

To search the IP-index is to find the index entry which records the anchor-state sequence of the value v', i.e., we need to find the index entry N_L where

$$N_L.key = max\{N_i.key \mid N_i.key \le v', 1 \le i \le size(tree)\}$$

Then N_L.anchors contains the anchor-state sequence for the value v'.

How to find the index entry N_L? The search algorithm is actually dependent on how the IP-index is implemented. Suppose the IP-index is implemented as a B^+-tree, then the entry N_L can be found by using the "index scan" facility which is provided by most B^+-tree implementations. An index scan opens a "cursor" to indicate the index entries that are inside a specified key range (lower_bound, upper_bound). If only upper bound is specified, the lower bound can be denoted as *nil*. The conditions for the upper_bound or the lower_bound can be specified as '<', '>=', etc. The pseudo-code of the IP-index search algorithm for a B^+-tree implementation is given in Fig. 3.6. The notations used in the algorithm are similar to those used in Fig. 3.5. The new functions are explained as follows:

- *open_inverse_index_scan*(*tree, b1, b2, c1, c2*) — performs an inverse index scan with lower_bound *b1*, upper_bound *b2*, lower_bound condition *c1*, upper_bound condition *c2*. This function opens a "cursor" to indicate the

```
insert_ip(tree, ts, t_i, v_i):
    S_i = (t_i, v_i)
    insert_ts(ts, i, S_i)
        /* insert the state into the array which stores the time sequence */
    if not exist_key(tree, v_i)
        N_L = get_lower(tree, v_i)                          (part 1)
        if N_L = nil
            N = insert_entry(tree, v_i)
            N.anchors = nil
            N.anchors_card = 0
        else
            N = insert_entry(tree, v_i)
            N.anchors = N_L.anchors
            N.anchors_card = N_L.anchors_card
                /* insert a new index entry, copy the anchor-state
                   sequence from the "lower" index entry  */
        endif
    endif
    if i > 1
        /* if not the first state in the time sequence */
        for each entry N_j where N_j.key lies inside        (part 2)
                the interval (min(v_i-1,v_i), max(v_i-1,v_i))
            N_j.anchors = append(N_j.anchors, S_i-1)
            N_j.anchors_card = N_j.anchors_card + 1
                /* add the new anchor state to the corresponding anchor
                   state sequences */
        end for each
    endif
```

Fig. 3.5: The IP-index insertion algorithm

current index entry which is inside the specified range. This function needs to be activated by the next function *next(iter)* in order to get the first element.

- *next(iter)* — retrieves the next index entry that is inside the range (specified by the cursor *iter*). The first call of this function retrieves the first index entry inside the specified range.

```
search_ip(tree, v'):                          lower bound   upper bound
        iter = open_inverse_index_scan(tree, nil, v', nil, '<=');
        N = next(iter);
              /* find the first index entry that is inside the range */
        return N.anchors;
```

Fig. 3.6: The IP-index search algorithm

The C++ code corresponding to this pseudo-code (for the implementation on SHORE [22]) can be found in the appendix.

3.4 IP-index versus the Precision of v_is

An interesting observation is that the insertion of the IP-index can be made very fast regardless of the growing of the TS, if the precision of values in the TS is limited.

3.4.1 How Does the Precision of v_is Affect the IP-index?

The algorithm in Fig. 3.5 contains two parts. Algorithm analysis shows that Part 1 takes $\Theta(LogM)$ time (M is the total number of index entries in the IP-index) since they are actually IP-index tree search operations. Furthermore, Part 2 takes *m*append_time* where *m* is the number of intervals which are spanned by the new segment and the *append_time* is the time taken to add the new state to the end of an anchor-state sequence. The *append_time* can be made *almost constant* by using a data structure which supports fast appending (for example, the multi-level dynamic array structure in Section 7.3.2).

Furthermore, if we limit the parameters M and m, we can reduce the insertion time of the IP-index. This can be achieved by limiting the precision of the measured values. The reason is: for a time sequence with range = R and preci-

sion = *P* in the value domain, the number of index_entries will be less than *R / P.* So, the lower the precision (the larger the value of *P*) is, the smaller the value of *M* and *m* will be. Thus, we can reduce the insertion time by limiting the precision of the values. This speculation will be verified by experiments in Chapter 4.

The above observation is practical since 1) normally measured time sequences have a limited range on the value domain, 2) the original precision of the measured data can always be limited due to errors and uncertainty in measurements. Furthermore, some applications do not need very high precision. For example, when measuring temperatures of a patient, the value range is the temperatures that the human body can possibly be alive at, and at a precision that represent changes that affect the well-being of the patient. Therefore, even if the thermometer used to measure the temperature of a patient has the precision of 0.001°C, we can still limit the precision to 0.1°C, which will both improve the performance of the IP-index and still be reasonable for the application.

The conclusion is that the insertion of the IP-index can be made *efficient* regardless of the size of the time sequences. This is done by limiting the precision of the values in the time sequence.

Related work by Lum et al. [89] suggested a “linked list” data structure to store historical data. It pointed out that an additional access path (a secondary index) for the linked list is needed to support fast random access of elements in this linked list (to avoid scanning). The IP-index is perfect for this purpose. Lum et al. [89] also mentioned that when the index tree grows too large, the values can be grouped into intervals. For example, all values between 0 and 5 have index value 1, and 5 to 10 have index value 2, etc. This is similar to our idea of limiting the precision of values (in order to limit the size of the IP-index tree).

From the above discussions we can see that the IP-index is not suitable for some unusual time sequences, e.g. periodic time sequences with *unlimited* precision, or signals which oscillate with an *increasing* amplitude over time (which makes the *M* parameter large). It is also not suitable for those time sequences with many “big jumps” in v_is (since this will make the parameter *m* large). Fortunately, most time sequences from real applications do not have these properties.

3.5 Comparison with a Conventional Secondary Index

This section explains why the IP-index is needed even though conventional secondary indexes are available. The reason why the IP-index is compared with conventional secondary indexes is that the IP-index is essentially a secondary index as well. A secondary index is a “nonclustering index”, as defined in

[118]. TSs are normally clustered by time stamps, not by values. Therefore, all indexes on the value domain of a TS are considered to be secondary indexes.

Suppose that Fig. 3.7 represents a patient's temperature reading sequence (TS), and linear interpolation is assumed to transform the TS into a continuous function. A conventional secondary index on the value domain will use the distinct values of v_is as keys k_j and record all the (t_i, v_i) pairs where v_is equal to the key k_j. By contrast, the IP-index associates the keys k_js with their anchor-state sequences (Section 3.2.1). Let us compare the IP-index with the conventional secondary index in dealing with the following value queries:

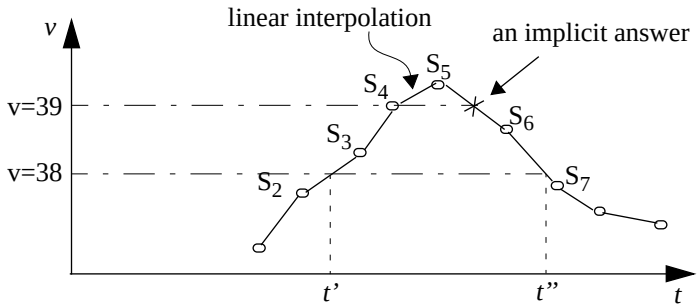


Fig. 3.7: Comparing the IP-index with a conventional secondary index

1. When did the patient have the temperature 38°C?

   A conventional secondary index will return *nil* since there are no *explicit* values equal to 38. By contrast, by using the IP-index we will get <t', t''>.

2. When did the patient have the temperature 39°C?

   A conventional index will only return t_4 (suppose $v_4 = 39$), while the correct answer (if we want to support the interpolation assumption) should include an implicit point as well, that is between S_5 and S_6 (marketed as a cross in Fig. 3.7).

3. During what time period did the patient have a temperature higher than 38°C (i.e., have a fever)?

   By using the IP-index, this query will return the time interval (t', t''). There is no way to return this interval by using conventional indexes since t' and t'' are implicit.

   Now let us drop the "continuous" assumption and assume that the time sequence is discrete. Then the answer to this query would be $[t_3, t_6]$, where *no implicit time points* are involved any more. It seems that the conventional secondary index would work now. Well, it returns a set of discrete states

$\{S_3, S_4, S_5, S_6\}$ (since these states have values greater than 38). Grouping these states into the time interval $[t_3, t_6]$ is not a trivial task, especially when the answer contains *several* intervals, or in the situation when the time sequence is large.

To conclude, the IP-index has the following advantages over a conventional secondary index:

1. The IP-index supports not only explicit values but also implicit values. This is achieved by the concept of the anchor-state sequences, A(v').

2. The IP-index keeps the *ordering* semantics of the original time sequence. The S_is in the A(v') are ordered by time as they are in the original time sequence. A conventional secondary index destroys the ordering of the original TS.

3. For range queries such as $F^{-1}(v>v')$, the IP-index is essential for efficiency even when the TS is viewed as *discrete*.

3.6 Related Indexes

Related indexes include temporal indexes, spatial indexes, and computational indexes. This section provides an overview of relevant indexing techniques and compares them with the IP-index.

3.6.1 Temporal Indexes

In the area of temporal databases [134], several indexing methods have been proposed to speed up temporal queries (a short overview of temporal databases can be found in Section 9.4). The common aspect of temporal indexes and our IP-index is that they all deal with indexing of temporal data. But there is a major difference, i.e., temporal indexes aim at indexing the *time domain* of temporal data, while the IP-index aims at indexing the *value domain*. Nevertheless, it is highly interesting to compare temporal indexes with our IP-index.

Elmasri et al. [47] propose an index structure termed the *Time Index*. The Time Index supports efficient retrieval of temporal data based on (valid) timestamps (the concept of *valid timestamps* can be found in [67]). A set of indexing points is created based on the starting and ending points of valid time intervals and these points are used to build an indexing structure. At each indexing point, all object versions that are valid during that point can be retrieved via a bucket of pointers. The Time Index is implemented by a B^+-tree. The differences between the Time Index and a regular B^+-tree is that the Time Index is based on objects whose search values are *intervals* rather than points.

- The similarity of the Time Index and our IP-index is that they both view temporal operations as *interval intersection* problems. However, in the Time Index the interval intersection is on the time domain while in the IP-index the interval intersection is on the value domain.

In [47], another index named the *Monotonic B^+-tree* is proposed. The Monotonic B^+-tree differs from the Time Index in the sense that it assumes time grows monotonically (it deals with transaction time), which allows for better space utilization and better search performance.

In [69] some indexing methods for temporal aggregates are proposed. For unordered relations, the *Aggregation Tree* was introduced to build a binary tree for the constant intervals to support efficient aggregate operations. For k-ordered relations, the k-ordered Aggregation Tree was introduced as a variation of the Aggregation Tree with the ability of garbage collection of tree nodes.

- The similarity of the Aggregation Tree compared to our IP-index is that they both transform temporal queries into *interval search* problems. However, the Aggregation Tree deals with interval aggregation on the time domain while the IP-index deals with interval range search on the value domain.

Gunadhi and Segev [58] present the *AP-Tree* which is a hybrid of an ISAM index and a B^+-tree. An AP-tree aims at indexing interval timestamps for append-only databases. It supports event-join optimization and temporal queries. An AP-tree is different from a regular B^+-tree in several respects: 1) If the tree is of degree d, then there is no constraint that a node must have at least $\lceil d/2 \rceil$ children, 2) there is no node splitting when a node gets full, and 3) the online maintenance of the tree is performed by accessing the right-most leafs.

The difference between the AP-tree and the Monotonic B^+-tree is that the Monotonic B^+-tree also takes care of migration of data (migration to optical disks). Nonetheless, they are very similar in design.

- The similarity between the AP-tree and the IP-index is that both support *append-only* databases. However, the AP-tree indexes on the time domain while the IP-index indexes on the value domain.

Shen et al. [113] introduce the *TP-index* (Time Polygon index) to support temporal operations. The TP-index maps the temporal data into a two-dimensional temporal space where the data can be clustered based on time.

3.6.2 Spatial Indexes

Efficient search of spatial data is required in geo-data applications or computer-

aided design. Traditional indexing methods are not well-suited for data objects located in multi-dimensional spaces. For example, structures based on exact matching of values, such as hash tables, are not useful in spatial search problems since a range search is required. Structures using one-dimensional ordering of key values, such as B-trees and ISAM indexes, do not work for spatial data since the search space is multi-dimensional.

Guttman [61] proposed a dynamic indexing structure named the *R-tree* to support efficient range search of spatial data. The main assumption of an R-tree is that the objects to be indexed can be modelled by means of the smallest rectangles, called Minimum Bounding Rectangles (MBRs), that contain them. R-trees are multi-dimensional generalizations of B-trees. They are paginated and balanced. The leaf nodes point to the actual data records. Non-leaf nodes either point to leaf nodes or represent a super-MBR that encompasses other super-MBRs or MBRs. To search for an MBR that overlaps (intersects with) a reference MBR in the R-tree, one starts from the root traversing each sub-tree that intersects with the reference MBR until the leaf node (possibly several) is reached. Each entry is then compared to the reference MBR and is returned as part of the response if it intersects with the reference MBR.

The R^+-Tree [108] and R*-Tree [18] are both variants of the original R-tree. They are both superior to the original R-tree structure. The R^+-Tree "clips" the MBR in such a way that no super-MBR has any overlap with any other super-MBR in the internal nodes of the structure. This enhances query-processing time (i.e., the number of tree nodes accessed) significantly. The R*-tree takes a different approach. The main idea of the R*-Tree is: whenever a node split is to occur, delete some of the nodes about to split and re-insert them. This will avoid splitting while ensuring good properties of the R-tree, hence improving the performance considerably.

In [72] a new indexing technique termed *SR-Tree* (Segment R-Tree) was proposed. A SR-tree is a combination of a Segment Tree [19] and an R-tree. The SR-Tree is used to index spatial data composed of multi-dimensional intervals that have non-uniform length distributions. It was shown that the SR-Tree improves the performance over conventional indexing techniques for both rectangle and line segment data.

- The similarity of the SR-Tree and the IP-index is that they both deal with *spatial search of multi-dimensional intervals*.

- We do not adapt an R-tree or a SR-Tree directly in the problem of value queries since the segments in time sequences (those Sg_i in Fig. 3.3) have the special property that the end point of Sg_i is the starting point of Sg_{i+1}. This property makes our index algorithm *much simpler* than that of the R-tree or SR-Tree. In other words, our indexing method can be implemented on top of any regular *one-dimensional* ordered index such as a B-tree, while an R-tree

requires a complicated algorithm for handling boundary conditions between regions.

3.6.3 Indexes in Computational Geometry

In the area of computational geometry, there are several data structures for indexing interval data. They are all based on variations of binary search trees. Examples are the *Segment Tree* [19], the *Interval Tree* [43], the *Priority Search Tree* [91], and the *Persistent Search Tree* [100]. Most of these data structures are designed with the assumption that the entire structure is contained in main memory.

- We do not adapt the above indexing structures in the problem of value queries since the segments in time sequences (the Sg_i in Fig. 3.3) have the special property that the end point of Sg_i is the starting point of Sg_{i+1} (i.e. S_{i+1}). This property can be used to make our index algorithm simpler than that of the Segment Tree or Interval Tree.

- Furthermore, the Segment Tree and Interval Tree are both just for main-memory implementation, while our IP-index can be implemented in disk-resident DBMSs as well.

3.6.4 SIQ-Index for Value Queries

Interestingly enough, inspired by our work of the IP-index [82], Nanopoulos and Manolopoulosa [93] propose a similar approach to deal with value queries. In [93], a time sequence is divided into sub-sequences and an R*-tree is used to index the MBRs (minimum boundary regions) of each sub-sequences. In this way the number of index entries can be made smaller than that of the IP-index. The index in [93] is named the *SIQ-index*.

The direct advantage of the SIQ-index [93] is that the space usage will be smaller than of the IP-index because there are fewer index entries. This also leads to shorter index insertion time [93]. On the other hand, the disadvantage is that the index search time will be slower. There are two reasons for this: 1) There might be more than one leaf node in the R*-tree need to be searched; 2) The sub-sequences found by the SIQ-index need to be scanned to check where the intersecting segments really occur [93]. If the sub-sequence is long, this may lead to more than one disk access to read in the entire sub-sequence, not to say the time spent in scanning in main-memory. On the other hand, by using the IP-index, we guarantee one disk access for one result because every anchor-state records one intersecting segment (Section 3.2.1).

Another advantage of the IP-index over the approach of [93] is that the IP-

index is based on the B^+-tree structure. B^+-trees are available in most commercial database systems. This means one can implement the IP-index on available database systems without the need to modify the system. On the contrary, R*-trees are not available in most current database systems. This leads to complexity in implementing the SIQ-index.

3.7 Generalized IP-index

Note that the idea of the IP-index can be generalized to include any forms of pre-processing in the time sequences (such as dividing the sequence into sub-sequences or limiting the precision of v_is) and index on the transformed time sequences. We can also keep the original precision of the values in the time sequences and use lower precision of values as keys in the IP-index. In this way the index size can be made smaller while not affecting the original precision of values in the TS.

3.8 Summary

This chapter introduces the idea of the *IP-index* based on the *time sequence* data model. The central concept is the *anchor-state sequence*, which records the intersecting segments for the queried value v'. The insertion and search algorithms of the IP-index were presented. The important relationship between the performance of the IP-index and the precision of values in the TS were investigated. It was shown that the insertion and search time of the IP-index can be made very fast regardless of the growing of the TS. We also introduced the generalized form of the IP-index which includes any forms of preprocessing of the TS (such as dividing the sequence into sub-sequences or limiting the precision of v_is) and index on the transformed TS.

The IP-index was compared to conventional secondary indexes to show why conventional secondary indexes cannot deal with the problem of value queries. The IP-index was also compared to related indexes in the area of temporal databases, spatial databases and computational geometry.

Chapter 4

Insertion/Search Time and Space Usage

As pointed out in Chapter 1, a good index is expected to have small insertion/search time, and space efficiency. In this chapter, we measure how the insertion and search time of the IP-index grows with the cardinality of time sequences. The measurements are made in both a main-memory and a disk-resident database system using both synthetic and real-life time sequences. The space usage of the IP-index is also investigated to show that it is practical to build the IP-index for large time sequences.

4.1 Performance in a Main-Memory Database System

This section presents the insertion and search time of the IP-index on a main-memory database system AMOS [49].

4.1.1 Implementation Notes

To evaluate how the IP-index performs in a main-memory database system, we have implemented the IP-index in an object-relational main-memory database system AMOS [49]. A time sequence TS was implemented as an array *ts*, where ts[i] = (t_i, v_i). The IP-index was implemented on top of an *AVL-tree* [2]. The reason why we chose the AVL-tree is that it has small re-balancing time [2]. The reason why we need to consider re-balancing time is that the keys v_i (Section 3.3) do not arrive in order, which means that the tree needs to be re-bal-

anced constantly during insertion. Therefore, each *index entry* in the insertion algorithm (Fig. 3.5) corresponds to a *node* in the AVL-tree. The anchor-state sequence was implemented as a *linked list* of integers (with a pointer to the end of the list in order to achieve fast appending). These integers denote indices of the array *ts*. Fig. 4.1 illustrates the implementation of the IP-index for the example TS in Fig. 3.3, i.e.,

$$anchors(k_1)=<S_1, S_2>$$
$$anchors(k_2)=<S_2>$$
$$anchors(k_3)=<S_2, S_3>$$
$$anchors(k_4)=nil$$

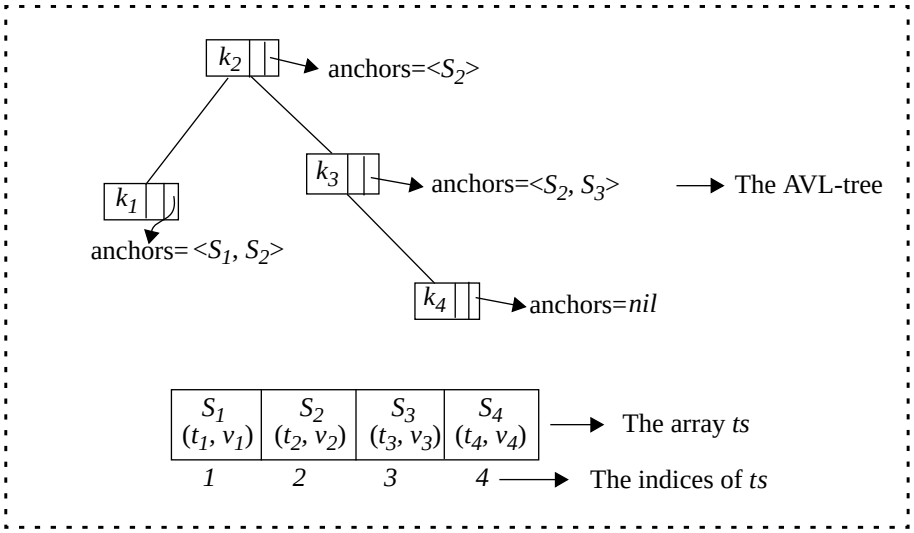


Fig. 4.1: The AVL-tree implementation of the IP-index in Fig. 3.3

4.1.2 Time Sequences Used in the Measurements

We measured[1] the insertion and search time of the IP-index (using the AVL-tree implementation) in AMOS [49]. The following three time sequences (cardinality = 10K) were used in the measurements.

1. A simulated periodic sequence, sin(t/100) (t = 1, 2...10K), plotted in Fig. 4.2.

1. All measurements were made on an HP9000/710 with 32M main memory and running HP/UX.

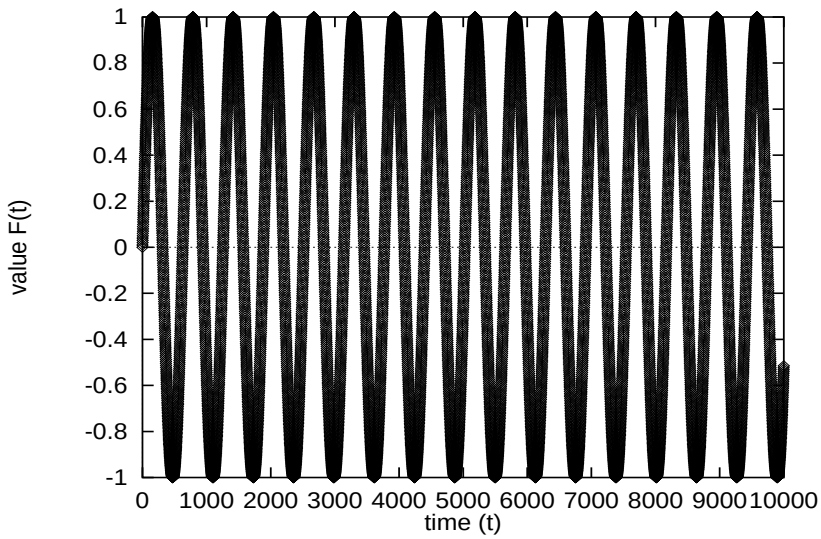


Fig. 4.2: Sine Data

The reason why we chose this time sequence is that it represents strictly periodic time sequences. Since most application time sequences are periodic, we chose to test the IP-index on a strictly periodic time sequence to see how it performs.

2. A time sequence from a real-life application [68] (measurements of the pressure in a fluidized bed), plotted in Fig. 4.3.

   The reason for choosing this time sequence is to see how the IP-index performs for real-life time sequences.

3. A simulated time sequence with a largely monotonic trend (not *strictly* monotonic), plotted in Fig. 4.4.

   The reason for choosing this time sequence is to see how the IP-index performs for non-periodic time sequences.

4.1.3 Insertion Time

Fig. 4.5 and Fig. 4.6 show the insertion time of the IP-index for the two time sequences shown in Fig. 4.2 and Fig. 4.3 respectively. The insertion time is measured as the sequences grow.

- The curves labelled “Original Value Insert” show the insertion time of the IP-index. For the pressure data the value range is [-6, 10] and the precision is 10^{-6}. For the sine data the value range is [-1, 1] and the precision is 10^{-6}.

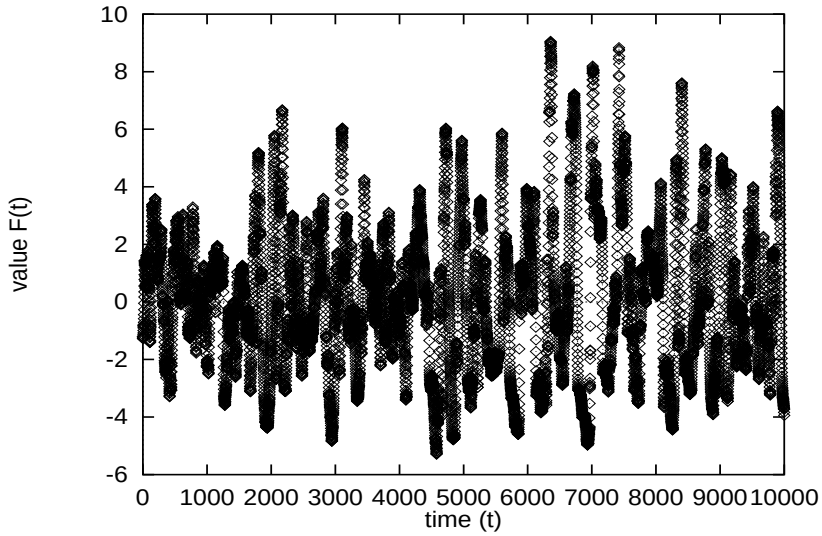


Fig. 4.3: Pressure Data

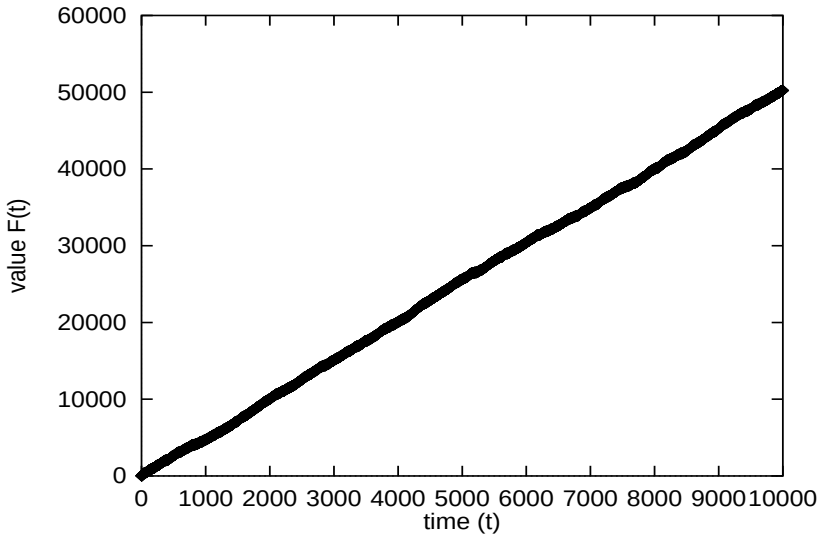


Fig. 4.4: Monotonic Trend Data

It can be seen that the insertion time increases linearly with the size (cardinality) of the sequence. This is because the parameters *M* and *m* (see Section 3.4) both grow with the size of the sequence.

- For the curves labelled “Limited Precision Insert” the precision of the val-

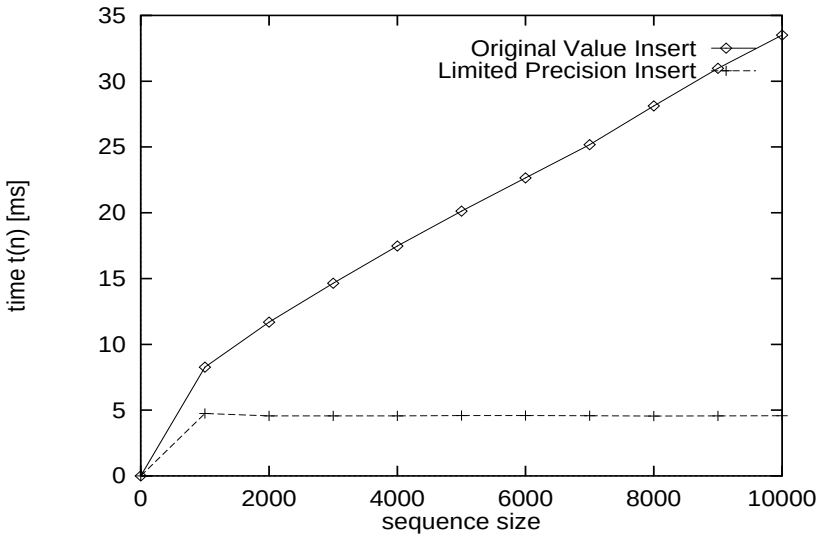


Fig. 4.5: Sine Data Insertion

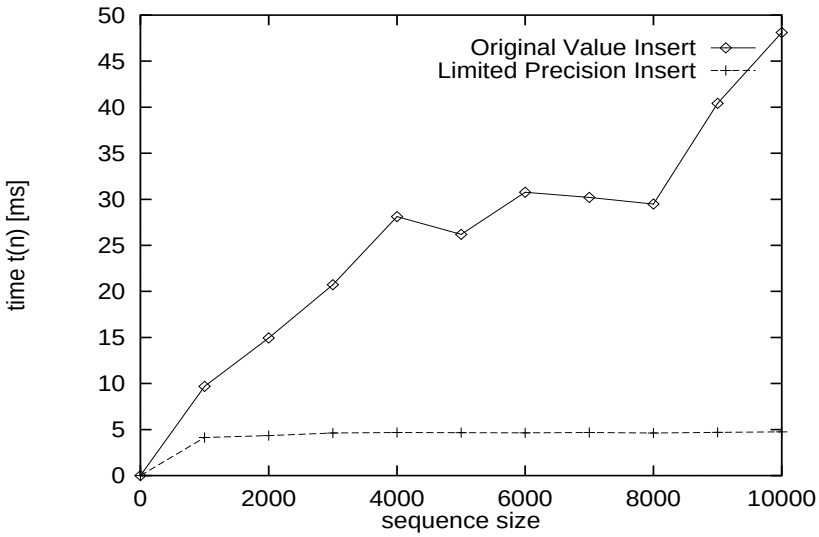


Fig. 4.6: Pressure Data Insertion

ues were limited to 10^{-3} for both TSs. It can be seen that the insertion time becomes *constant* after the total number of index entries has been inserted into the IP-index. This is because:

1) The number of nodes in the AVL-tree does not grow any more; only the

anchor-state sequence associated with each node grows with the time sequence.

2) We use a linked list structure to implement the anchor-state sequence, which makes the parameter *append-time* (see Section 3.4) constant.

3) The limited precision ensures that the *m* parameter (number of intervals spanned by the new segment, as discussed in Section 3.4) has an upper limit, which leads to an upper limit on the insertion time as well.

The conclusion is that for a periodic time sequence with a limited range and precision on the value domain, an upper bound on the IP-index insertion time can be achieved.

4.1.4 Search Time

In Fig. 4.7, we compare the approaches of using the IP-index or linear scanning TS to find the anchor-state sequence A(v') for some randomly generated value *v'*. The measurements were performed on the sine sequence as plotted in Fig. 4.2. The difference between the IP-index search time and the linear scanning time was measured as the sequence grows. The results show that difference between using the IP-index or not is dramatic. Note that the results are displayed in *logarithmic* scale since the difference is too great to display on a linear scaled axis. (Note that the curve labelled "IP-index Search" in Fig. 4.7 is the same as the one labelled "Original Value Search" in Fig. 4.8, they do not look the same because they are displayed on differently scaled axes.)

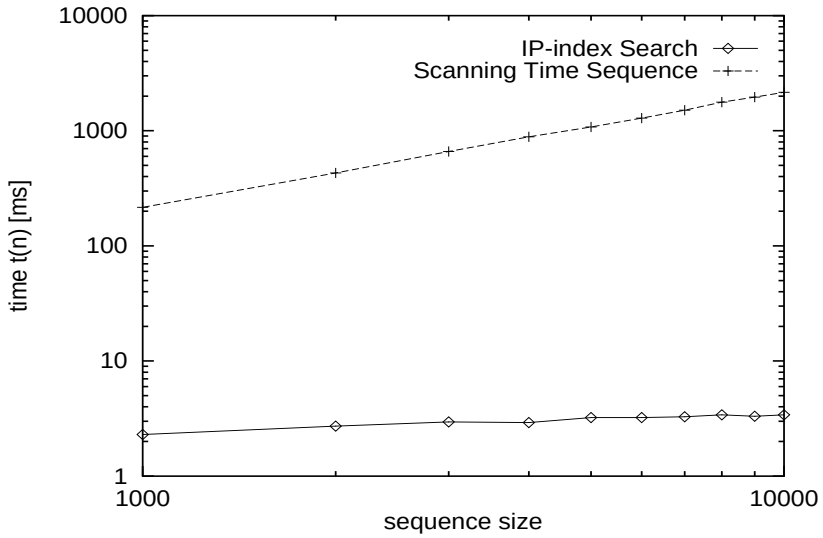


Fig. 4.7: Compare the IP-index with Linear Scanning

Fig. 4.8 and Fig. 4.9 show the IP-index search time for two periodic TSs. After every 1000 insertions, the IP-index search time for A(v') for some randomly generated value v' were measured. The results show that the search time is logarithmic due to the AVL-tree implementation (see the curves labelled "Original Value Search"). However, in the case of "limited range and precision", the IP-index search time is bounded regardless of the growing of TS (see the curve labelled "Limited Precision Search"). The reason is the same as in the insertion case: the number of nodes (the parameter *M*) of the AVL-tree does not increase after all index entries have been inserted, only the anchor-state sequences associated with each node grow, so the search time for A(v') stays constant at $\Theta(LogM)$ (where *M* stays constant).

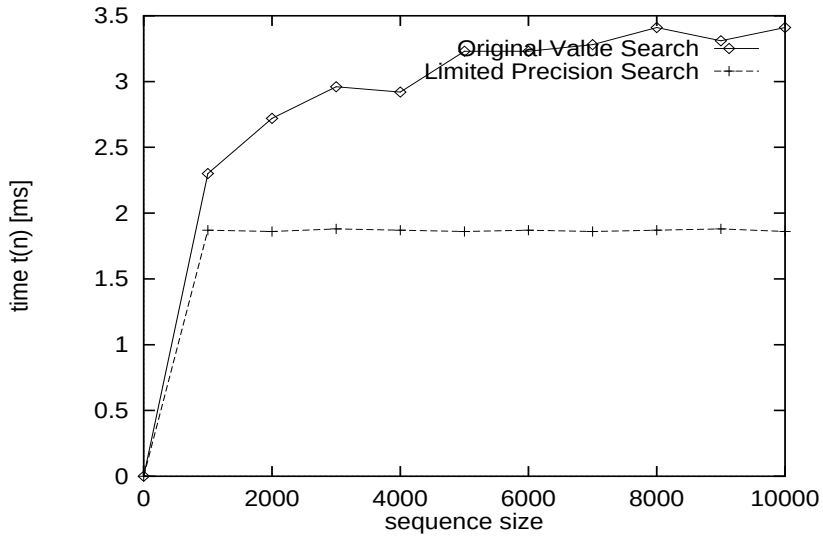


Fig. 4.8: Sine Data Search

4.1.5 Largely Monotonic Time Sequences

Fig. 4.10 shows the insertion and search time of the IP-index for the largely monotonic trend sequence plotted in Fig. 4.4. It can be seen that both the insertion and search time are approximately logarithmic to the cardinality of the TS. This is due to the AVL-tree implementation. In this case, since the value range cannot be limited (the value range grows with the TS), the "upper bound" on insertion and search time cannot be achieved.

Note that a *strictly* monotonic time sequence does not need an IP-index. This is because the value domain is then monotonic just as the time domain is, which means that conventional indexes on the time domain can be applied to the value

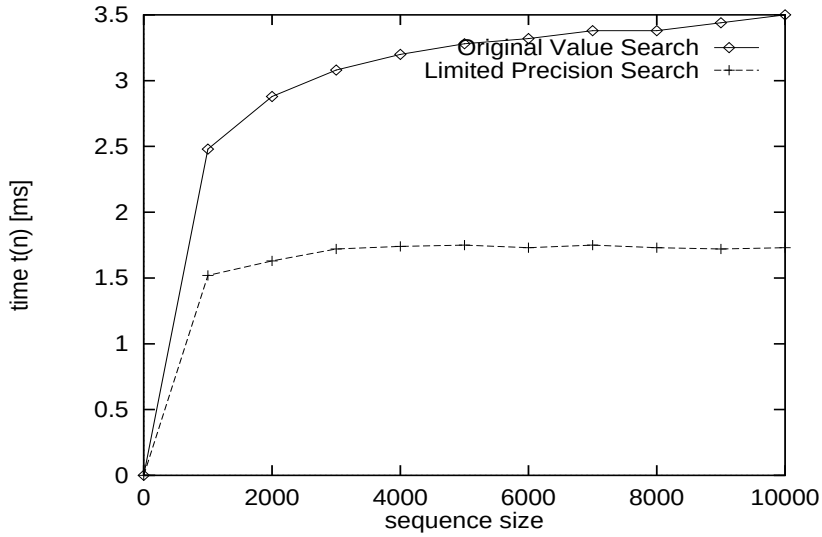


Fig. 4.9: Pressure Data Search

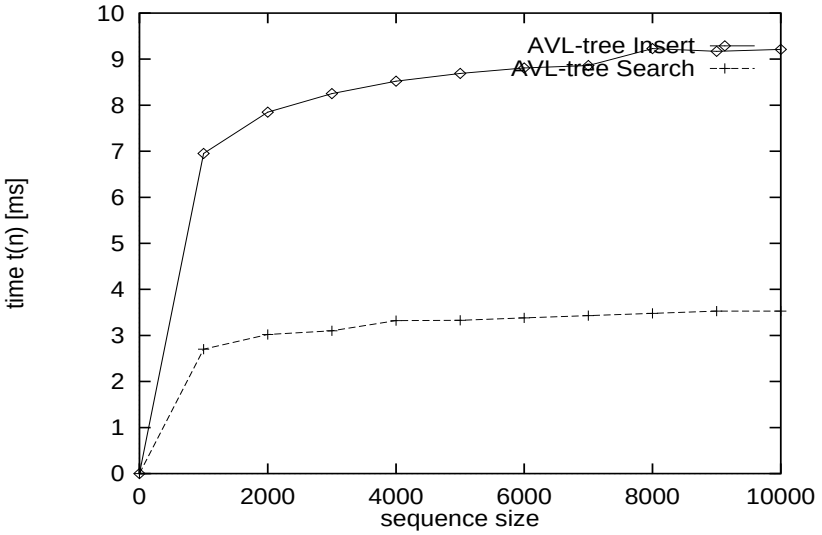


Fig. 4.10: Monotonic Trend Data

domain.

To conclude this section, we have shown that the IP-index exhibits good performance in a main-memory DBMS.

4.2 Performance in a Disk-Resident Database System

We also measured the performance of the IP-index in the disk-resident database system SHORE [22]. SHORE is an object-oriented database system. The reason why we did not choose a relational DBMS is that, as pointed out by Stonebraker [131], it is not a good choice to implement a time sequence as a relational table due to time and space inefficiency.

4.2.1 Implementation Notes

The reason why we chose SHORE is that recent work by Seshadri et al. [110] demonstrates that a SHORE array of records is a good choice for implementing sequential data. Therefore, we chose to implement a time sequence TS as an array of records (t_i, v_i) in SHORE. For simplicity (without affecting the performance) we use integers *i* (4-bytes) to store the time stamp t_i (instead of using the SQL timestamp value such as "1997/20/01"). The v_is are stored as 4-byte floating point numbers.

The IP-index was implemented as a B^+-tree in SHORE. The keys in the B^+-tree are the floating numbers v_is and each key is associated with a pointer to its anchor-state sequence. The anchor-state sequences were implemented as arrays of integers (not arrays of records (t_i, v_i)). For example, if A(v') = <S_1, S_6, S_{10}>, then <1, 6, 10> (an array of integers) is stored. There are two reasons for this: 1) We only store (t_i, v_i) in the original time sequence array. It will be redundant to store (t_i, v_i) in every A(v'). 2) The anchor-states only indicate the positions in the TS where to apply *ifn*. To apply *ifn*, all neighbour states need to be retrieved from TS (so it does not help if (t_i, v_i) is stored duplicated in A(v')).

Since anchor-state sequences are expected to be of dynamic length, these arrays were implemented as SHORE large objects which can grow arbitrarily large. For further details of implementations, please refer to the appendix — SHORE Implementation Notes. All measurements were made on a SPARC 20 machine with 64M main memory. The SHORE buffer pool size was set to 40 8K pages.

4.2.2 Time Sequences Used in the Measurements

Time sequences used in the measurements were the same as in the last section: 1) the sine sequence (Fig. 4.2): a simulated time sequence sin(t/100) (t = 1, 2,...10K) with the value range [-1, 1] and the precision of 10^{-6}; 2) the pressure sequence (Fig. 4.3): the time sequence from a real-life application representing the measurements of the pressure in a fluidized bed. The value range was [-6, 10] and the precision was 10^{-6}. The cardinality of both time sequences was 10K. The insertion and search time of the IP-index were measured as the

sequences grow.

In the “limited precision” measurement, both time sequences were rounded to a precision of 10^{-3}.

4.2.3 Insertion Time

Since all SHORE operations are performed inside *transactions*, we had to decide where to “commit” when building the IP-index for large time sequences. For the time sequences with size 10K, it is not possible to do the 10K insertions inside one transaction (the buffer pool is not big enough). We chose to divide each sequence into size = 100 sub-sequences (thus there will be 100 sub-sequences). The insertion of the sub-sequences [n*100+1, n*100+100] (n = 0, 1, 2...) was then done inside a transaction. The reason why we chose size = 100 as the size of sub-sequences (i.e., as the amount of work done in one transaction) is that: 1) For size < 100 it will be too slow to build the IP-index for the whole TS since we have to commit very frequently; 2) For size > 100 much work will be lost if the transaction is aborted. Also it needs more log space. (The sub-sequence size of 100 was chosen approximately to satisfy the above constraints. It does not have to be exactly 100, of course.)

The results are shown in Fig. 4.11 and Fig. 4.12.

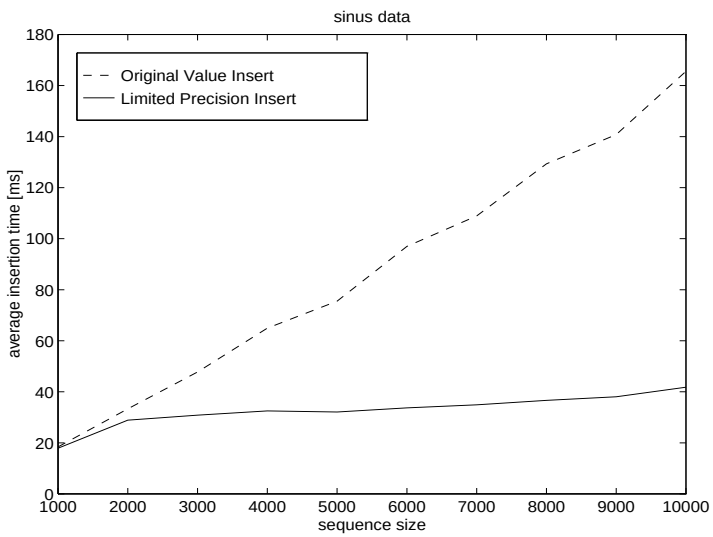


Fig. 4.11: Sine Data Insertion

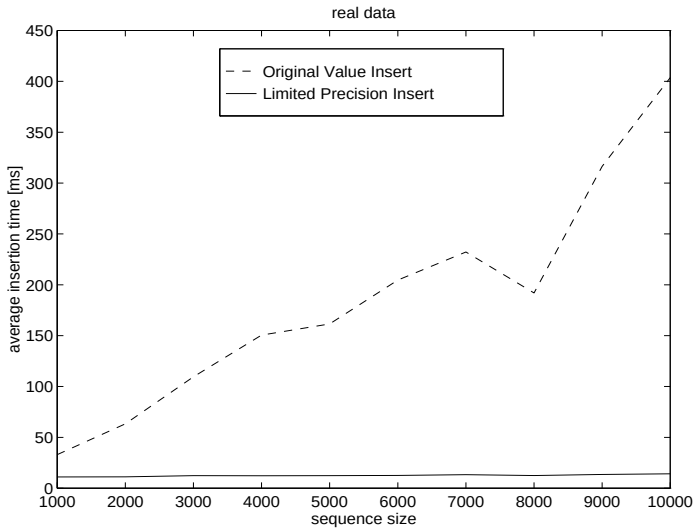


Fig. 4.12: Pressure Data Insertion

4.2.4 Search Time

To measure how the IP-index search time grows with the cardinality of the TS, we measured the average search time for A(v') for some randomly generated value *v'* after every [n*500+1, n*500+500] (n = 0, 1, 2...) sub-sequence was inserted into the IP-index. The results are shown in Fig. 4.13 and Fig. 4.14.

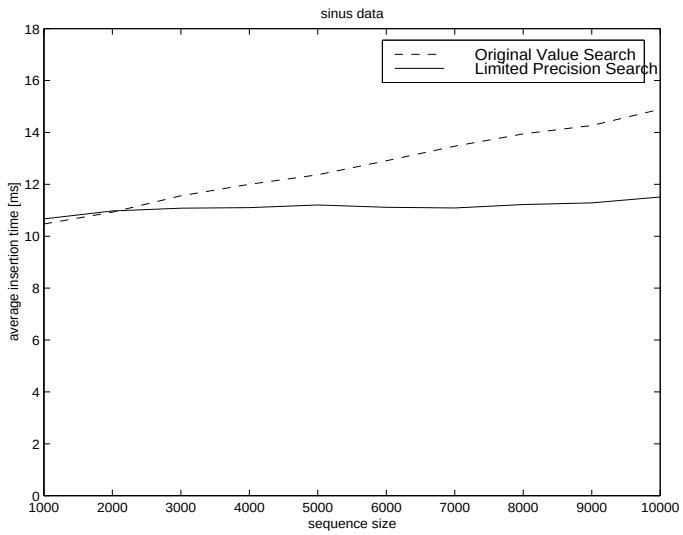


Fig. 4.13: Sine Data Search

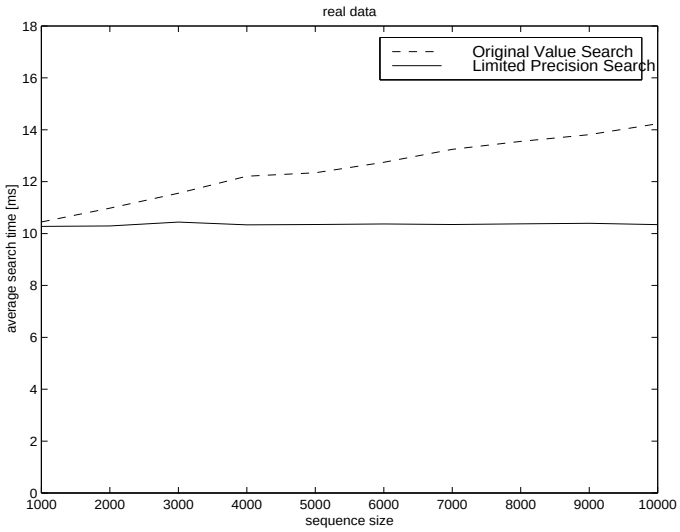


Fig. 4.14: Pressure Data Search

The measurement results show that the performance of the IP-index in SHORE is similar to the performance in the main-memory implementation (Section 4.1). That is: for original precision the insertion and search time grows with the size of the sequences; for limited precision the insertion and search time stays almost constant.

Note that several parameters affect the resulting measurement figures. Among these parameters there are the buffer pool size, the log space of the SHORE server, and the number of insertions in one transaction. Finally the operating system (I/O processing) and the different versions of SHORE release will also affect the measurement results.

4.3 Space Usage

After showing that the IP-index has small insertion and search time, we should investigate space usage of the IP-index, especially for *large* time sequences. Is it *practical* to build IP-indexes for large time sequences with regard to space usage and efficiency issues? Recall that the IP-index contains an index tree and many anchor-state sequences. We investigated how the size of the IP-index tree (the number of index entries) and the lengths of the anchor-state sequences, i.e., the cardinalities of A(v')s (denoted as card(A(v'))s), grow with the cardinality of the TS.

4.3.1 Time Sequences Used in the Experiments

The time sequence used in this experiment was the real-life pressure sequence in Fig. 1.4 (Chapter 1), with the cardinality of 100K and the value range (-0.5, 2.5).

The first 1K, 10K and 100K of the pressure sequence were used in the measurements in order to vary the cardinality of the TS. The precision of values (v_is) was varied from 10^{-1}, 10^{-2} to 10^{-3}. An IP-index was built for every combination of the above variations (e.g., the first 1K sequence with precision 10^{-1}, the first 10K sequence with precision 10^{-1}, etc.).

4.3.2 Experimental Results

The size of the IP-index (the index tree) with respect to the cardinality of the TS and the precision of v_is are plotted in Fig. 4.15. The cardinality of A(v') with respect to the cardinality of the TS and the precision of v_is are plotted in Fig. 4.16.

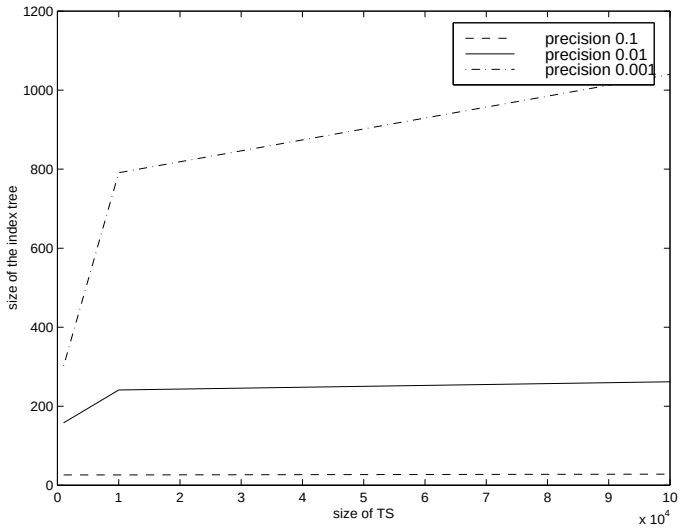


Fig. 4.15: How the size of the index tree grows with the cardinality of TS

Fig. 4.15 shows that: 1) the lower the precision is, the smaller the index tree will be; 2) for a specific value precision, the size of the IP-index tree (the number of index entries) does not grow much with the cardinality of the TS. (For the precision 0.1 and 0.01 the index tree size stays constantly small regardless of the growing of the time sequence.) The reason for the slow growing of

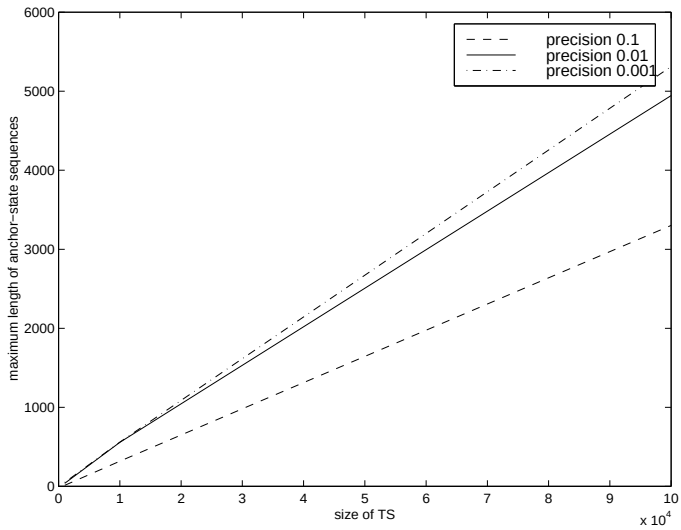


Fig. 4.16: How the maximum cardinality of A(v')s grows with the cardinality of TS

the index tree is that there are repeated values in a non-monotonic time sequence. For a specific precision and value range of v_is, there is a limited number of possible keys in the index tree (Section 3.4). This investigation shows that *it is practical to build IP-indexes for large time sequences with regard to space usage*. Meanwhile, since the index tree will generally be small, searching the IP-index to find A(v') will be very fast.

Fig. 4.16 shows how the card(A(v')) grows with the cardinality of TS. For every precision the maximum card(A(v')) was plotted as the worst case behaviour. Maximum card(A(v')) occurs when v' = -0.25 where the values are very noisy, as can be seen from Fig. 1.4. The card(A(-0.25)) is 4945 for the 100K pressure sequence, resulting in the ratio of 4945/100K = 5% (worst case). This only happens when the values are very noisy around v'. In most applications the time sequence will generally have much shorter A(v')s, especially in the case of monotonic trend time sequences such as stock prices.

Fig. 4.16 shows that: 1) the lower the precision is, the smaller the maximum card(A(v')) will be; 2) the maximum card(A(v')) grows linearly with the cardinality of the pressure sequence. This is again because of the periodic property of the pressure sequence. The longer the TS is, the more number of segments will probably cross the line v = v' (Section 3.2). This indicates that A(v') will normally grow with the size of TS for any value v'.

We also measured the *total* space usage of the IP-index (the index tree plus all

anchor-state sequences) for the pressure sequence in Fig. 1.4. The precision of values was set to 0.01. The results are: the total number of anchor-state sequences is 262 (i.e., there are 262 index entries in the IP-index), the sum of the cardinalities for these A(v')s is 69663, the average of card(A(v')) is 266 (with the maximum of card(A(v')) 4965). Therefore, the sum of card(A(v'))s is approximately 70% (69663/100,000) of the cardinality of the original TS. In other words, the total space used to store all A(v')s in the IP-index is 70% of the space used to store the TS. This indicates that the total space overhead of the IP-index is small (at least in this case). Of course the ratio '70%' is dependent on several factors such as the characteristics of the sequence (how the values go up and down in the sequence), and the precision of the values (0.01 in the above measurement). The total space overhead of the IP-index for different application data is an interesting topic for future work.

Some readers may wonder why the sum of card(A(v')) is even smaller than the cardinality of TS in this case. The reason is: limiting the precision of v_is in TS results in some v_i *equal* to v_{i+1} (where for the original precision $v_i \neq v_{i+1}$). Recall in Section 5.2.1 that horizontal segments ($v_i = v_{i+1}$) are not recorded in the IP-index. Therefore, limiting precision will make A(v') shorter (but it will not affect the cardinality of the original TS). That is why the sum of card(A(v')) is even smaller than the cardinality of TS in this case.

In the case of a long TS, the older part of the TS (i.e., the part of the TS that has time stamps $t < t'$) can be archived (or vacuumed [129]) to tape storage. The corresponding IP-index can be archived easily by copying the B^+-tree and archiving the parts of the A(v')s that are inside the time window $t < t'$.

The Case of Monotonic Trend Sequences

For monotonic trend time sequences such as stock prices, the size of the IP-index tree will be relatively large compared to a periodic time sequence due to the less number of repeated values. By contrast, all anchor-state sequences will then be much shorter than those of periodic time sequences. The overall effect, i.e., the total space usage (the index tree plus the anchor-state sequences) will be generally smaller than that of periodic time sequences.

Our experiments show that the claim in [117], i.e., "a secondary index over the data values is not needed in most applications — such an index can potentially be very expensive in terms of storage, because the number of entries for such an index is in the order of the number of data value" is not necessary true. We have shown that the size of the IP-index tree is generally small, and by limiting the precision of v_is the index tree can be made even smaller. Since value queries are very common in real-life applications, it is essential to have a secondary index on the value domain such as the IP-index to achieve a good performance.

4.4 Summary

This chapter presents experimental results on the insertion/search time of the IP-index and its space usage. Experiments were made on both a main-memory and a disk-resident database system using both synthetic and real-life time sequences. The experiments demonstrate the highlights of the IP-index: *fast insertion*, *fast search*, and *space efficiency*. These properties show that the IP-index is not only an elegant idea but also a practical solution for large sequence data. This chapter shows that an index on the *value* domain of a time sequence is not necessary expensive and impractical, as claimed by [117].

Chapter 5

Various Forms of Value Queries

This chapter shows how to compute various kinds of *value queries* using the IP-index. These queries include: *exact queries*, *approximate queries*, *range queries* and *time window queries*. In particular, we show the importance of the IP-index for range queries (i.e., sub-sequence extraction based on a value range). The importance of the IP-index for range queries also holds for *discrete* time sequences where interpolation is not required.

5.1 Exact Queries

An exact query asks when the value was equal to v' in a time sequence. Suppose the time sequence in Fig. 5.1 represents a patient's temperature reading in a hospital. An exact query could be:

- When did the patient have the temperature 38°C?

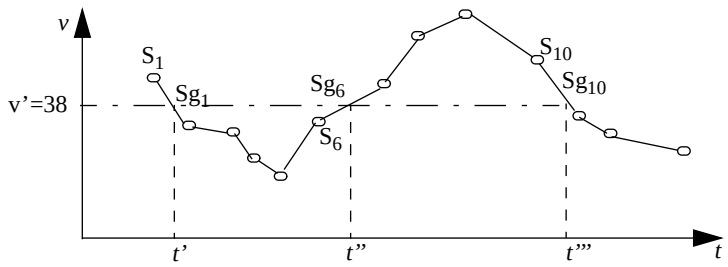


Fig. 5.1: Illustration of a value query

As pointed out in Section 3.1, this kind of query is denoted as $F^{-1}(v')$. It can be seen from Fig. 5.1 that there are no *explicit* (stored) time points when the values were equal to 38. To efficiently process this query, we would need the IP-index. Recall that the IP-index records the *anchor-state sequence* for any value v' (Section 3.2.1). By applying the interpolation function on each anchor-state of v', we can calculate all the time points when the values were equal to v' in the TS.

The anchor-state sequence of the value v' can be found by searching the IP-index to find the index entry N_L where

$$N_L.key = max\{N_i.key \mid N_i.key \leq v'\}$$

Then $N_L.anchors$ contains the anchor-state sequence for the value *v'*, denoted as A(v') (Section 3.2.1). The algorithm for computing $F^{-1}(v')$ using the IP-index is given in Fig. 5.2. It can be seen that this algorithm is similar to the IP-index search algorithm in Fig. 3.6, except that the interpolation function *ifn* is applied to every anchor-state. The notations used in the algorithm are the same as explained in Section 3.3.

The definition of *surrounding_states*(S_i) in Fig. 5.2 is determined by the interpolation function *ifn*. For example: a) If *ifn* is "linear interpolation", then *surrounding_states*(S_i) = {S_i, S_{i+1}}; b) If *ifn* is moving-average over three states, then *surrounding_states*(S_i) = {S_{i-1}, S_i, S_{i+1}} (or perhaps {S_i, S_{i+1}, S_{i+2}}). In the simplest case of the "step-wise constant" assumption, we have *surrounding_states*(S_i) = {S_i}.

5.2 Range Queries

In this section we show how to use the IP-index to compute range queries. A simple range query could be: "find the sub-sequences when the values inside those sub-sequences are greater than a threshold *v'*". The result of a range query is *sub-sequences*, denoted as *time intervals*.

5.2.1 Interpolated Range Queries

First we look at the cases when some interpolation function is assumed on the TS (the case of a discrete TS will be discussed later). Given the temperature sequence TS = <S_1, S_2,...S_8> as shown in Fig. 5.3 (assume linear interpolation), an example range query could be:

- During what time intervals were the values greater than v'?

This kind of query is denoted as $F^{-1}(v>v')$ (or $F^{-1}(v<v')$) [82]. It can be seen

Algorithm "Exact_query":

Computing the time points when the values were equal to v' for an interpolated time sequence.

Input:

ts — the time sequence (an array)

tree — the IP-index for *ts* (e.g., a B^+-tree or an AVL-tree)

v' — the queried value

ifn — the interpolation function assumed on *ts*

Output:

The sequence of time points *t** where the values at those points were equal to *v'* for *ts* under the interpolation assumption *ifn*.

Exact_query (*ts*, *tree*, *v'*, *ifn*):

$F^{-1}(v') = nil$

/* initialize the result */

find the index entry N_L where

$N_L.key = \max\{N_i.key \mid N_i.key \le v', 1 \le i \le size(tree)\}$

/* find the entry that stores the anchor-state sequence of v' */

for each state S_i in N_L.anchors

$F^{-1}(v') = F^{-1}(v') + ifn^{-1}(v', surrounding\_states(S_i))$

/* $surrounding\_states(S_i)$ can be retrieved from the ts array */

end for each

return $F^{-1}(v')$

Fig. 5.2: Computing $F^{-1}(v')$

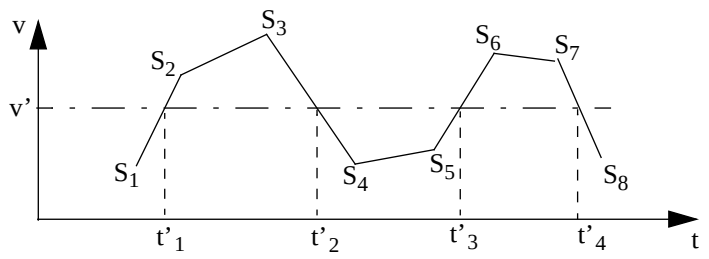


Fig. 5.3: Illustration of a range query

from Fig. 5.3 that:

- $F^{-1}(v>v') = <(t'_1, t'_2), (t'_3, t'_4)>$
- $F^{-1}(v<v') = <(t_1, t'_1), (t'_2, t'_3), (t'_4, t_8)>$

where $t_1 = S_1$.time (the first time point of TS, denoted as t_s) and $t_8 = S_8$.time (the last time point of TS, denoted as t_e). The observations are:

1. $F^{-1}(v>v')$ (or $F^{-1}(v<v')$) returns a sequence of time intervals.
2. Each time interval of $F^{-1}(v>v')$ (or $F^{-1}(v<v')$) is composed only of those time points returned by $F^{-1}(v')$ (plus t_s and t_e).

Now let us see how to compose the time intervals of $F^{-1}(v>v')$ (or $F^{-1}(v<v')$) using the time points returned from $F^{-1}(v')$. First, let us define the "direction" of an implicit time point *t'* as the following:

- *direction(t')* = '+' if S_{i+1}.value > S_i.value
- *direction(t')* = '-' if S_{i+1}.value < S_i.value

This is illustrated in Fig. 5.4. Notice that we do not store segments with S_{i+1}.value = S_i.value in the IP-index since we only record intersecting (non-horizontal) segments Sg_i.

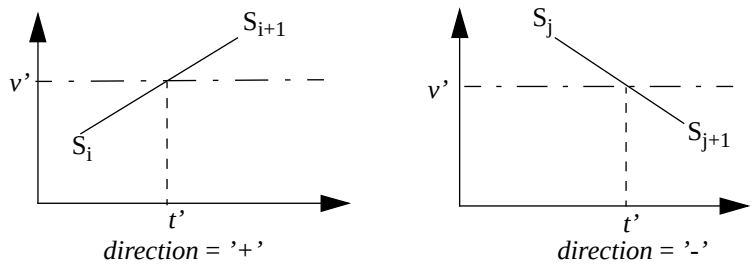


Fig. 5.4: The "direction" of interpolated time points

It can be seen from Fig. 5.3 that:

- $F^{-1}(v>v')=(t'_i, t'_{i+1})$* where *direction*$(t'_i)$ ='+'
- $F^{-1}(v<v')=(t'_i, t'_{i+1})$* where *direction*$(t'_i)$='-'

(The time intervals concerning t_s and t_e have to be treated specially by comparing S_1.value and S_8.value with *v'*.) Therefore, the algorithm of $F^{-1}(v>v')$ is shown in Fig. 5.5. The function *next(t', seq)* returns the time point in *seq* (a sequence of time points) that follows *t'*.

The algorithm of $F^{-1}(v<v')$ is similar to the one shown in Fig. 5.5 (simply replace "*direction(t')* = '+'" with "*direction(t')* = '-'").

5.2.2 Discrete Range Queries

The IP-index was originally designed for time sequences with interpolation assumptions. Interestingly enough, it turns out that the IP-index is essential for *discrete* time sequences as well. An example is given below.

Seshadri et al. [110] gave an example of calculating the monetary value of Stock1 traded in each hour when the low price fell below 50. The query was expressed as:

```
SELECT[1] ((A.high + A.low)/2) * A.volume
FROM Stock1 A
WHERE A.low < 50
```

It was argued in [110] that selection push-down (A.low<50) should be applied here so that the calculation of "((A.high+A.low)/2)*A.volume" only needs to be done for those states whose low values are below 50. But, without an index, the whole time sequence has to be *scanned* to find these states. One may argue that a conventional secondary index on the "low" value will help. Unfortunately it does not work, as explained in Section 3.5.

By viewing the time sequence as continuous (by applying linear interpolation function), and posing the query $F^{-1}(v<50)$ (see Fig. 5.6), the time points *t'* and *t''* (Fig. 5.6) can be calculated efficiently. Then, the calculation of "((A.high+A.low)/2)*A.volume" can be applied to only those states S_is that are inside the range *(t', t'')*. In this way the IP-index plays an important role on range queries on *discrete* time sequences.

In some applications (e.g., the terrain-aided navigation in Chapter 10), it is desirable to return the "state intervals" instead of the time intervals for the range query $F^{-1}(v>v')$ (or $F^{-1}(v<v')$). For example, in Fig. 5.3 the state intervals returned from $F^{-1}(v>v')$ are $[S_2, S_3]$ and $[S_6, S_7]$. This is trivial given that we can calculate $F^{-1}(v>v')$ (or $F^{-1}(v<v')$). Because, these "state intervals" can be found by rounding[2] the time intervals. For example, rounding (t'_1, t'_2) in

1. In [110] the term 'PROJECT' was used instead of the SQL keyword SELECT.

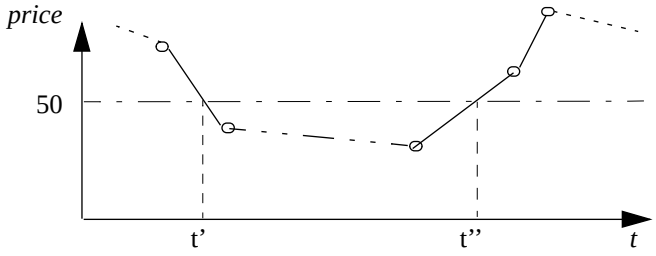


Fig. 5.6: A stock price sequence

Fig. 5.3 results in $[S_2, S_3]$.

5.2.3 Approximate Queries

Since the values in time sequences are often sampled with errors and uncertainty in measurements, many applications do not require to know when the values were *exactly* equal to *v'*, instead, it is more interesting to know when the values were *approximately* equal to *v'*. For example, given the temperature sequence in Fig. 5.1, if we are interested in "when did the patient have a temperature 38", we would pose the query:

- When did the patient have a temperature *around* 38°C?

This kind of query is termed an *approximate query* and denoted as $F^{-1}(v'-e<v<v'+e)$. Approximate queries can be processed easily once we can process range queries. This is because

$$F^{-1}(v'<v<v'') = F^{-1}(v>v') \cap F^{-1}(v<v'')$$

where '$\cap$' means "interval intersection".

For example, in Fig. 5.7, $F^{-1}(v>v')=<(t'_1, t'_2), (t'_3, t'_4)>$, $F^{-1}(v<v'')=<(t_s, t''_1), (t''_2, t''_3), (t''_4, t_e)>$. There we see that $F^{-1}(v'<v<v'')=<(t'_1, t''_1), (t''_2, t'_2), (t'_3, t''_3), (t''_4, t'_4)>$, which is the interval intersection of $F^{-1}(v>v')$ and $F^{-1}(v<v'')$.

Therefore, the calculation of $F^{-1}(v'<v<v'')$ is performed by the following:

1. Calculate $F^{-1}(v >v')$.

2. Here "rounding" means finding the largest state interval that is inside the time interval.

Algorithm "Range_query":

Computing the time intervals when the values were greater than v' for an interpolated time sequence.

Input:

ts -- the time sequence (an array)

tree -- the IP-index built for ts (a B^+-tree or an AVL-tree)

v' -- the queried value

ifn -- the interpolation function assumed on ts

Output:

The sequence of time intervals (*t', t''*)* where the values inside those intervals were greater than v' for ts under the interpolation assumption *ifn*.

```
Range_query (ts, tree, v', '>', ifn):
    seq = nil                    /* initialize a sequence seq */
    find the entry N_L where
        N_L.key = max{N_i.key | N_i.key ≤ v', 1 ≤ i ≤ size(tree)}
        /* find the entry that stores the anchor-state sequence of v' */
    for each state S_i in N_L.anchors
        t' = ifn^-1(v', surrounding_states(S_i))
            /* surrounding_states(S_i) can be retrieved from the ts array */
        if S_{i+1}.value > S_i.value
            direction(t') = '+'
        else
            direction(t') = '-'
        endif
        seq = seq + (t', direction(t'))
            /* seq stores the time points of F^-1(v') and their directions */
    end for each
    F^-1(v>v') = (t', next(t', seq))
            for those t' in seq where direction(t') = '+'
            /* combine the time points in seq into time intervals */
    return F^-1(v>v')
```

Fig. 5.5: Computing $F^{-1}(v>v')$

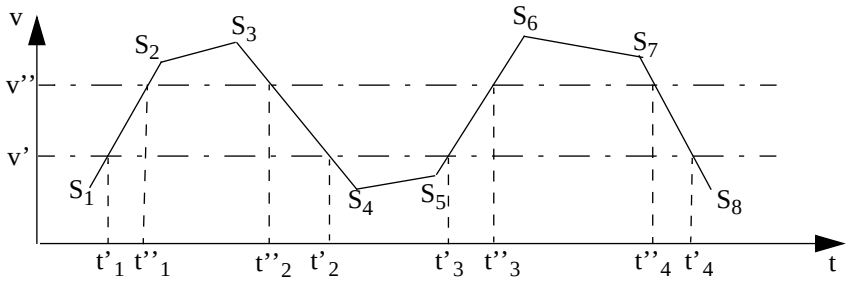


Fig. 5.7: Illustration of an approximate query

2. Calculate $F^{-1}(v < v'')$.

3. Apply interval intersection to the results returned from 1 and 2.

5.3 Time-Window Queries

Some value queries only concern a part of the time sequence, i.e., a *time window.* An example of a time window query could be:

- When did the patient have a fever *in the last few days* (denoted as t > t')?

This query can be denoted as $F^{-1}(v>38 \text{ AND } t>t')$. The answer to this query is marked by the two crosses in Fig. 5.8. A naive way to process this query is to first calculate $F^{-1}(v>38)$ and then check for each resulting time point t if t > t' holds. An optimized way is to retrieve those states S_i in A(38) where S_i.time > t', and apply the interpolation function to the surrounding states of S_i. Optimization of time window queries will be further illustrated in Chapter 8.

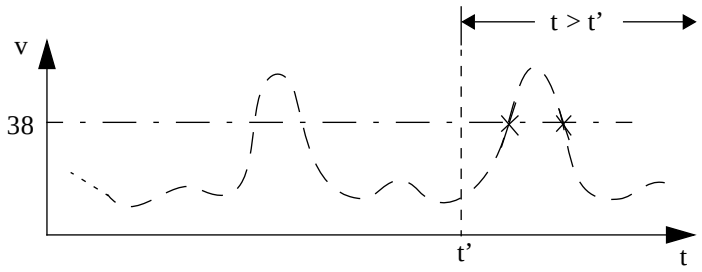


Fig. 5.8: A time windo w query

5.4 Amplitude-Sensitive Shape Queries

Some kind of shape queries, i.e., "amplitude-sensitive" shape queries, can be processed efficiently by using the IP-index.

In [112] it was mentioned that one of the symptoms of Hodkin's disease is a temperature pattern, known as "goalpost fever", that peaks exactly twice within 24 hours.

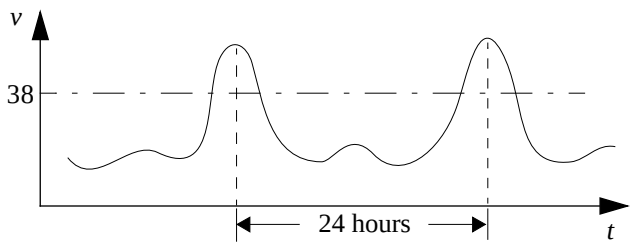


Fig. 5.9: The "goalpost fe ver" pattern

The IP-index can be used to find this temperature pattern in a time sequence. It contains two steps:

1. Compute $F^{-1}(v>38)$ (which are the periods of "fever").

   The result returned is a sequence of time intervals during which the patient has a fever. (This sequence is usually short, which means the query processing time of step 2 will be fast.)

2. Check if there exist two time intervals in the "fever" periods that have the distance *d* of *24 hours*.

   The distance between two time intervals can be defined either as the distance between the starting points of both intervals or as the distance between the mid-points of both intervals.

5.5 Summary

In this chapter we have shown how to solve various kinds of *value queries* efficiently by using the IP-index. In particular, we demonstrated the importance of the IP-index for *range queries* (i.e., sub-sequence extraction based on a value range). Other queries that benefit from the IP-index include time window queries and amplitude-sensitive shape queries.

In a survey by Chomicki on temporal query languages [29], it is argued that the densed temporal domain is very useful in many applications but is difficult to implement efficiently since the set of time instances is very large. The IP-index provides the ability to *derive* the densed instances from the original discrete sequence, saving both storage space and query processing time. The actual number of time instances (termed “states” in this paper) needed to be stored are determined by the range and precision of the values in the sequence. Also the sampling frequency can change during different periods, higher frequency can be used for interesting value ranges and lower frequency can be used for uninteresting ranges. Different interpolation functions can also be applied to different sub-sequences.

Chapter 6

The σ Operator*

This chapter introduces an extended SELECT operator, σ*, which retrieves *sub-sequences* (time intervals) in a time sequence TS where the values inside those sub-sequences satisfy some conditions. The σ* operator supports user-defined interpolation functions on TS.

In this chapter, the implementations of the σ* operator for various selection conditions are presented. The efficiency of the σ* operator is demonstrated by experiments made on SHORE [22]. Related work is studied to compare the σ* operator with other relevant operators.

6.1 Formal Definitions

In this section, the formal definitions of a time sequences TS and its interpolated (derived) time sequence $\overline{TS}$ under an interpolation function *ifn* are given. Then two SELECT operators, σ* and $\overline{\sigma}$, which work on TS and $\overline{TS}$ respectively, are introduced.

6.1.1 The Definition of TS and $\overline{TS}$

As defined in Section 2.1, a *time sequence* is a sequence of values ordered by time. Formally, a time sequence can be defined as the following:

Definition 6.1: A *time sequence* TS is a sequence of states where each state has a time stamp and a value, i.e.,

TS = <S_1, S_2,...S_n> where $S_i = (t_i, v_i)$ (i = 1, 2...n), and $t_i < t_{i+1}$.

□

An example time sequence is shown in the left side of Fig. 6.1, which represents the temperature reading of a patient in a hospital.

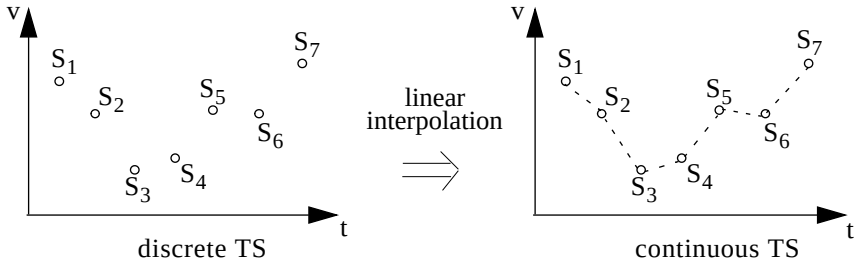


Fig. 6.1: Time sequences - from discrete to continuous

Many applications require a discrete time sequence to be seen as *continuous* where implicit values can be derived from explicit values by applying some user-defined interpolation functions. For example, a patient's temperature reading can be seen as a continuous curve by applying linear interpolation on the discrete TS, as shown in the right side of Fig. 6.1.

In order to formally define a continuous time sequence, let us first define a continuous time interval $[t_1, t_2]$.

Definition 6.2: A *closed interval* $[t_1, t_2]$ is defined as the infinite set of all real number time points between and including t_1 and t_2, i.e., $[t_1...t_2] = \{t \mid t \in R \text{ and } t_1 \leq t \leq t_2\}$.

□

(Notice that the appropriately modified definitions for $[t_1, t_2)$, $(t_1, t_2]$ and (t_1, t_2) are assumed, and the general term 'interval' will sometimes be used to refer to any of these.)

A continuous time sequence can be formally defined accordingly. For a time sequence TS, the notation $\overline{TS}$ is used to denote its *derived (interpolated)* time sequence under an y user-defined interpolation function *ifn*. Intuiti vely, $\overline{TS}$ defines the *infinite set of all states* defined o ver the time interv al $[t_1, t_2]$ (just as $[t_1, t_2]$ defines the infinite set of all time points between and including t_1 and t_2).

The precise definition of $\overline{TS}$ is the following:

Definition 6.3: Given the discrete time sequence $TS = <S_1, S_2, ...S_n>$ where $S_i = (t_i, v_i)$ $(i = 1, 2...n)$. $\overline{TS}$ denotes the *interpolated* time sequence defined over the closed interval $[t_1, t_n]$ by applying an interpolation function *ifn* on TS, i.e.,

$$\overline{TS}^{1} = <(t_1, f(t_1))...(t_n, f(t_n))> = \{(t, f(t)) \mid t \in R \text{ and } t_1 \le t \le t_n, \text{ for any } t \text{ where } S_i.time \le t < S_{i+1}.time, f(t) = ifn(t, surrounding\_states(S_i))\}.$$

□

The definition of *surrounding_states*(S_i) is determined by the interpolation function *ifn*. For example: a) If *ifn* is "linear interpolation", then *surrounding_states*(S_i) = {S_i, S_{i+1}}; b) If *ifn* is moving-average over three states, then *surrounding_states*(S_i) = {S_{i-1}, S_i, S_{i+1}} (or perhaps {S_i, S_{i+1}, S_{i+2}}). In the simplest case of the "step-wise constant" assumption, we have *surrounding_states*(S_i) = {S_i}.

An informal but intuitive notation would be

$$\overline{TS} = ifn(TS)$$

denoting $\overline{TS}$ is the *interpolated* TS.

Continuous and Non-Continuous Interpolation Functions

We shall point out that some interpolation functions are *continuous* while others are not. An example of a continuous interpolation function is linear interpolation, as illustrated in the left part of Fig. 6.2. An example of a non-continuous interpolation function is "step-wise constant" interpolation, as illustrated in the right part of Fig. 6.2. The reason why it is not continuous is that there is a "jump" in every state S_i when $S_i.value \neq S_{i+1}.value$.

In this thesis, when we claim that "$\overline{TS}$ denotes the *continuous* time sequence by applying some interpolation function *ifn*", the implication of "continuous" here does not mean a "continuous function". Instead, it denotes that $\overline{TS}$ is defined over the densed interval $[t_1, t_n]$ (see Definition 6.3). This "continuous" notion comes from [33] "Formal Semantics of Time in Databases". In [33], two concepts concerning interpolation on a historical database are defined, i.e., the *comprehension principle* and the *continuity assumption*.

1. A precise notation should be $\overline{TS}_{ifn}$ where *ifn* is the interpolation function used to interpolate TS. We omit *ifn* for the sake of clarity. We assume that a system-defined (default) interpolation function (e.g. linear interpolation) is used.

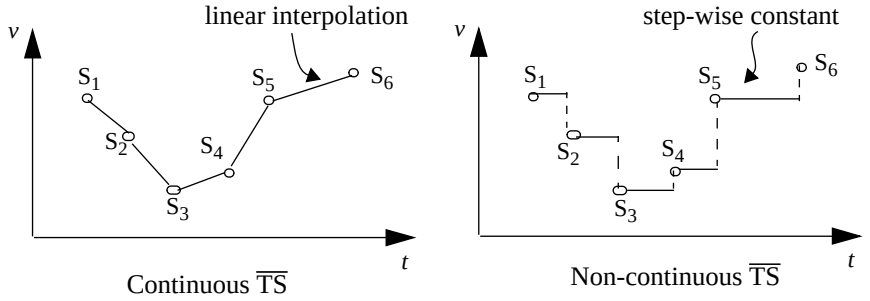


Fig. 6.2: Examples of continuous and not continuous $\overline{TS}$

1. The *comprehension principle*: under any reasonable interpretation a historical database defined over a sequence of states $<S_1, S_2,...S_n>$ should be considered as modelling an enterprise completely over the entire closed interval $[S_1, S_n]$. All information about the objects of interest to the enterprise can be assumed to be contained in or implied by the historical database for the entire interval $[S_1, S_n]$.

2. The *continuity assumption*: any assumption which extends a mapping from a finite set of moments $\{S_1, S_2,...S_n\}$ (ordered as in the sequence $<S_1, S_2,...S_n>$) into a set of individuals, into a mapping from all moments in the closed, dense interval $[S_1, S_n]$, into that set of individuals, will in general be called a *continuity assumption*. Although it was pointed out in [33] that there are many possible ways to interpolate the states inside the interval $[S_1, S_n]$, only "step-wise constant" was assumed in [33] for simplicity.

Therefore, by "continuous" we mean $\overline{TS}$ is defined o ver the dense interv al $[t_1, t_n]$ (i.e., assuming some interpolation functions). $\overline{TS}$ does not ha ve to be a "continuous function".

Some readers might w onder wh y we do not simply define $\overline{TS}$ as a function, i.e., $\overline{TS}$ is denoted by a function $v = f(t)$. There are se veral reasons for this. 1) In many real-life applications, dif ferent interpolation functions can be assumed on different parts of a time sequence, depending on the sampling frequenc y, value distrib ution, etc. Therefore, it is not appropriate to define $\overline{TS}$ as a *single* function $v = f(t)$; 2) Man y real-life time sequences are usually v ery long, it is not feasible to calculate the function definition *f*; 3) F or the same time sequence TS, there might be dif ferent kinds of interpolation functions assumed on it, depending on the application requirement, the resources a vailable, etc. Therefore, we define $\overline{TS}$ as an infinite set of states (Definition 6.3) where each implicit state is calculated by applying the interpolation function (required by

the application) on the neighboring explicit states. This is a more reasonable and flexible approach.

6.1.2 The Definition of $\overline{\sigma}$

Traditional SELECT operator σ [118] (in the relational algebra) retrieves tuples that are *explicitly* stored in a relational table. For time sequence applications, retrieving explicit (stored) values is far from adequate. As mentioned earlier, a discrete time sequence TS is often interpreted as the continuous sequence $\overline{TS}$ by assuming some user-defined interpolation functions. Therefore, selection on a time sequence should be defined over $\overline{TS}$ instead of TS.

Therefore, we introduce a new SELECT operator, $\overline{\sigma}$, to denote selection on the continuous sequence $\overline{TS}$ (i.e., supporting interpolation on TS). Unlike the traditional σ operator which returns a subset from a discrete set (i.e., a relational table), $\overline{\sigma}$ returns *sub-sequences* (denoted by time intervals) of the continuous sequence $\overline{TS}$. A sub-sequence of $\overline{TS}$ defined over the time interval (t', t'')[1] is denoted as $\overline{TS} \mid (t', t'')$.

To formally define the $\overline{\sigma}$ operator, let us first define a *selection condition*. Since $\overline{TS}$ has two dimensions, the time dimension *t* and the value dimension *v*, a selection condition, denoted by *cond*, is a conjunction of the terms $t \Theta C$ or $v \Theta C$ (C denotes a constant), where the operator Θ contains the following relational comparison operators, $\{=, >, <, \geq, \leq\}$. Examples of *cond* are:

- t = t' (here t' is a constant and t is a variable, the same holds for the following examples)
- t < t'
- t' < t < t''
- v = v'
- v < v'
- v' < v < v''

A state S = (t, v) in $\overline{TS}$ is said to satisfy the condition *cond* if the time stamp t and the value v satisfy *cond*, denoted as $P_{cond}(S)$ = TRUE. For example, a state S = (1, 2) satisfies *conds* such as t = 1 or v = 2. A sub-sequence $\overline{TS} \mid (t', t'')$ is said to satisfy *cond iff* for any state S where $S \in \overline{TS} \mid (t', t'')$, $P_{cond}(S)$ = True.

---

1. Here we use *open* intervals to denote sub-sequences. A sub-sequence that includes end points is accordingly denoted as $\overline{TS} \mid [t', t']$.

The formal definition of the $\overline{\sigma}$ operator is the following:

Definition 6.4: A $\overline{\sigma}$ operator on $\overline{TS}$ retrieves the sub-sequences that satisfy the condition *cond*, i.e,:

$$\overline{\sigma}_{cond}(\overline{TS}) = \overline{TS} \mid (t', t'')^*, \textit{ iff for any state } S \in \overline{TS} \mid (t', t''), \; P_{cond}(S) = \text{TRUE}.$$

□

For example, in Fig. 6.3, we have $\overline{\sigma}_{v>v'}(\overline{TS}) = \overline{TS} \mid <(t_1, t'_1), (t'_2, t'_3)>$, which are those sub-sequences of $\overline{TS}$ that have values greater than v'.

In the degenerated case, the $\overline{\sigma}$ operator returns *states* instead of sub-sequences. For example, in Fig. 6.3, $\overline{\sigma}_{t=t'1}(\overline{TS}) = (t'_1, v')$. A state S = (t', v') can be seen as a degenerated case of a sub-sequence, i.e., $S = \overline{TS}|[t', t']$.

6.1.3 The Definition of σ*

To directly support $\overline{\sigma}(\overline{TS})$ is not feasible since $\overline{TS}$ represents a function, not a discrete set. In other words, it is impossible to generate all states S's in $\overline{TS}$ and store them in the database. Therefore, we introduce an operator σ*, which works on the discrete TS, to implement the $\overline{\sigma}$ operator. First we start with informal discussion to show the relationship between the σ* and $\overline{\sigma}$ operator. Then give the formal definition of the σ* operator.

Recall that $\overline{TS}$ can be informally rewritten as $\overline{TS} = ifn(TS)$, therefore we have (informally)

$$\overline{\sigma}(\overline{TS}) = \overline{\sigma}(ifn(TS)) = \overline{\sigma} \circ ifn(TS)$$

This indicates that the implementation of the $\overline{\sigma}$ operator can be accomplished by a new operator ($\overline{\sigma} \circ ifn$) (i.e., the composition of $\overline{\sigma}$ and *ifn*) which works on the discrete TS. Thus, we introduce a new operator σ* on the discrete TS where $\sigma^* = \overline{\sigma} \circ ifn$, denoting that the semantics of σ* is first applying *ifn* on TS, then performing the selection $\overline{\sigma}$ (on the interpolated TS). The formal definition of σ* is the following:

Definition 6.5: A σ* operator, when applied to TS, generates the same result as applying $\overline{\sigma}$ on the corresponding $\overline{TS}$, i.e.,

$$\sigma^*_{cond}(TS)^1 = \overline{\sigma}_{cond}(\overline{TS}).$$

□

The *cond* clause specifies the selection condition, as in Definition 6.4.

The σ* operator can be efficiently implemented by using the IP-index. In the next section we will show how the σ* operator is implemented for various selection conditions.

6.2 Implementations of σ*

In this section we discuss how the σ* operator is implemented for selections such as $\sigma^*_{t=t'}(TS)$ and $\sigma^*_{v=v'}(TS)$.

6.2.1 $\sigma^*_{t=t'}(TS)$

Intuitively, $\sigma^*_{t=t'}(TS)$ returns the part of $\overline{TS}$ where the time stamp is equal to t'. Now let us see ho w to calculate $\sigma^*_{t=t'}(TS)$.

1. According to Definition 6.5, $\sigma^*_{t=t'}(TS) = \overline{\sigma}_{t=t'}(\overline{TS})$.
2. According to Definition 6.4, $\overline{\sigma}_{t=t'}(\overline{TS}) = \overline{TS}|[t', t']$.
3. $\overline{TS}|[t', t']$ is a de generated case of a sub-sequence, i.e., $\overline{TS}|[t', t'] = S' = (t' f(t'))$, where $f(t') = ifn(t', surrounding\_states\ (S_i))$ (Definition 6.3).

To further calculate $f(t')$, we need to locate the state S_i in TS. This can be done by linearly searching TS to find the state S_i where $S_i.time \le t' < S_{i+1}.time$. More efficient location methods take advantage of the physical organization of TS and available indexes. For example, If TS is implemented by an array [84], then a binary search will do.

The efficiency of computing $f(t')$ is determined by the interpolation function, of course. For most applications, simple interpolation functions such as step-wise constant or linear interpolation will do. In these cases, computing $f(t')$ is very fast. Some applications might require higher order interpolation functions such as least square.

In the simplest case of the "step-wise constant" interpolation, there is no need to apply *ifn*. We have $f(t') = S_i.value = v_i$.

---

1. A precise notation should be $\sigma^*_{t=t'}(TS, ifn)$ where *ifn* is the user-defined interpolation function. We omit the argument *ifn* assuming that a system-defined (default) interpolation function (e.g., linear interpolation) is used.

6.2.2 $\sigma^*_{v=v'}$(TS)

Intuitively $\sigma^*_{v=v'}$(TS) returns the part of $\overline{TS}$ where the v alue is equal to v'. It corresponds to the follo wing query:

- When w as the v alue equal to v'?

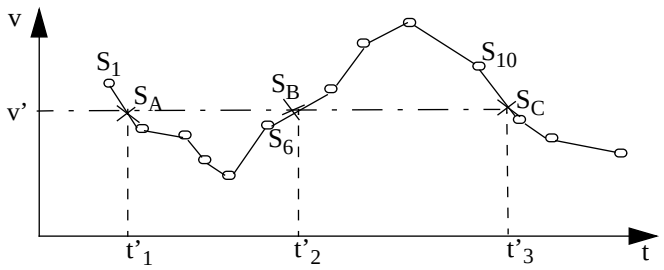


Fig. 6.3: Illustration of value queries on a TS

Normally the result of $\sigma^*_{v=v'}$(TS) is a sequence of states. For example, in Fig. 6.3, $\sigma^*_{v=v'}$(TS) = <(t'$_1$, v'), (t'$_2$, v'), (t'$_3$, v')>, which corresponds to the three implicit states S_A, S_B, and S_C in Fig. 6.3. The time points of these states, t'$_i$ (i = 1, 2, 3), are calculated by the algorithm of F^{-1}(v') in Fig. 5.2 (in Section 5.1). Recall that the computation of F^{-1}(v') is composed of two steps:

1. Find the *anchor-state sequence* of v' in the IP-index.

2. Applying ifn^{-1} over the neighbor-states of every anchor-state.

It can be seen that the first step is independent of the interpolation function *ifn*. Therefore, we define it as an operator termed the *IP operator.*

IP Operator

The *IP oper ator*, $IP_{v=v'}$(TS), returns the anchor-state sequence of v' (i.e., A(v')). Intuitively, the IP operator returns the *nearest neighbor states* of the value v' in order to apply the interpolation function ifn^{-1}.

Therefore, $\sigma^*_{t=t'}$(TS) is implemented by the sequential execution of the $IP_{v=v'}$(TS) and ifn^{-1}, as illustrated as Fig. 6.4.

A naive way to implement $IP_{v=v'}$(TS) is to linearly scan TS to find the state S_i where S_i.value ≤ v'< S_{i+1}.value. Since the IP-index stores A(v') (Section 3.2), the $IP_{v=v'}$(TS) operator can be implemented efficiently by searching the IP-index to find the key k_i where $k_i \le v' < k_{i+1}$ and return A(k_i).

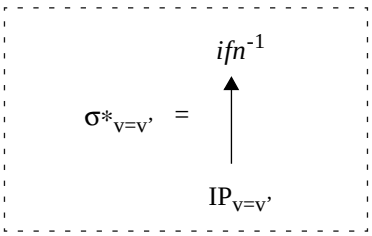


Fig. 6.4: The relationship between the σ^* operator and the IP operator

Since $A(k_i)$ is an ordered sequence of state_ids (see Section 3.2.1), $IP_{v=v'}(TS)$ can be implemented as a *stream* where the *next* element of $IP_{v=v'}(TS)$ is implemented by retrieving the *next* state in $A(k_i)$. Therefore, the $\sigma^*_{v=v'}(TS)$ can be implemented as a stream as well: the *next* state of the $\sigma^*_{v=v'}(TS)$ is generated by applying ifn^{-1} over the *next* state returned from $IP_{v=v'}(TS)$ (Fig. 6.4). More on stream processing can be found in Section 8.1 where query optimization of sequence data is discussed.

Get the First F ew Answers Quickly

By implementing $\sigma^*_{v=v'}(TS)$ as a stream we can generate the *first few answers* [16] quickly. This is especially important when card(A(v')) (i.e., the cardinality of A(v'), see Section 3.2.1) is large. To generate the first few answers, the interpolation function ifn^{-1} is applied to only the first few states in A(v'). In particular, *the first answer* of $\sigma^*_{v=v'}(TS)$ can be generated quickly since the first state_id in A(v') denotes the position in the TS where to apply ifn^{-1}.

By contrast, linearly scanning TS will take a very long time to get the first answer when the first answer appears late in the TS. This is demonstrated by experiments shown in Section 6.3.2.

The Exceptional Cases

In some e xceptional cases $\sigma^*_{v=v'}(TS)$ will return sub-sequences (time intervals). For example, if S_i.value = S_{i+1}.value, then applying linear interpolation we will have $\sigma^*_{v=v'}(TS) = [t_i, t_{i+1}]$, see the left side of Fig. 6.5. This case can be detected easily when we calculate $F^{-1}(v')$. According to the definition of linear interpolation, we have

$$t' = t_i + (v'-v_i) * (t_{i+1}-t_i) / (v_{i+1}-v_i)$$

This expression will trigger the "divided by zero" error if $v_{i+1} - v_i = 0$. Whenever this error is caught, we return the sub-sequence $\overline{TS} \mid [t_i, t_{i+1}]$ as the result

instead.

Another case when $\sigma^*_{v=v'}(TS)$ returns sub-sequences is when *ifn* is the "step-wise constant" assumption, see the right side of Fig. 6.5. In this case, $\sigma^*_{v=v'}(TS)$ return sub-sequences (intervals) for all $v' = v_i$; $\sigma^*_{v=v'}(TS)$ return *nil* for all $v' \neq v_i$.

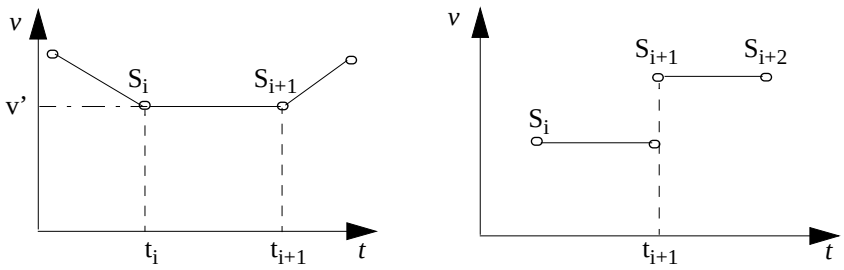


Fig. 6.5: Exceptional cases of $\sigma^*_{v=v'}(\overline{TS})$

Actually if *ifn* is the "step-wise constant" assumption, we would not need the IP-index to calculate $\sigma^*_{v=v'}(TS)$. This is because $F^{-1}(v')$ can be calculated easily by using a conventional secondary index (see Section 3.5). However, this does not lead to the conclusion that the IP-index is not useful in the case of the "step-wise constant" assumption. The reason is that *range queries* on time sequences would require the IP-index for the sake of efficiency, see Section 5.2. There we show the IP-index is essential for range queries, no matter what kind of interpolation is assumed. This also holds for discrete time sequences.

6.2.3 $\sigma^*_{t>t'}(TS)$

Intuitively, $\sigma^*_{t>t'}(TS)$ returns the sub-sequences in $\overline{TS}$ where the time stamps are greater than t'. According to Definition 6.5, $\sigma^*_{t>t'}(TS) = \overline{\sigma}_{t>t'}(\overline{TS}) = \overline{TS}|(t', t_n)$, where $t_n = S_n$.time (the last explicit time stamp in TS). For example, in Fig. 6.6, we have $\sigma^*_{t>t'}(TS) = \overline{TS}|(t', t_{10})$.

Inside the time interval (t', t_n), the value v for any time point t can be calculated by the definition of $\overline{TS}$ (see Definition 6.3), i.e., $f(t) = ifn(t, surrounding\_states(S_i))$.

Note that the values v inside the sub-sequence $\overline{TS}|(t', t_n)$ are normally non-monotonic, which means they normally do not reside inside the value range $(f(t'), f(t_n))$. Calculating the value range of $\overline{TS}|(t', t_n)$ is not trivial, since the maximum and minimum points inside the interpolated sub-sequence $\overline{TS}|(t', t_n)$

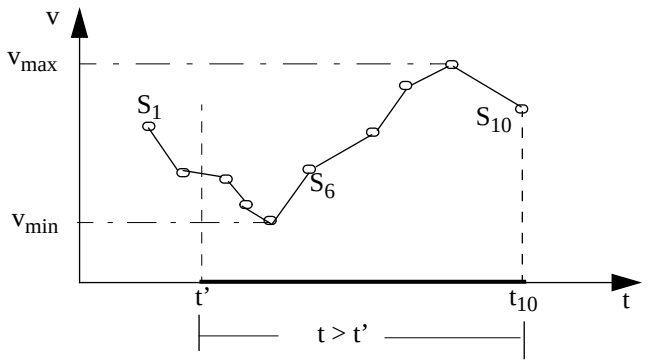


Fig. 6.6: Illustration of $\sigma^*_{t>t'}(TS)$

have to be found, see Fig. 6.6.

6.2.4 $\sigma^*_{v>v'}(TS)$

Intuitively, $\sigma^*_{v>v'}(TS)$ returns the sub-sequences in $\overline{TS}$ where the v alues are greater than v'. It corresponds to the follo wing query:

- When w as the v alue greater than v'?

In Section 5.2, we sho wed ho w to calculate range queries $F^{-1}(v>v')$ (Fig. 5.5). Therefore, $\sigma^*_{v>v'}(TS)$ can be processed easily. For example, in Fig. 6.7, since $F^{-1}(v>v') = <(t_1, t'_1), (t'_2, t'_3)>$, so we have $\sigma^*_{v>v'}(TS) = \overline{TS}|<(t_1, t'_1), (t'_2, t'_3)>$.

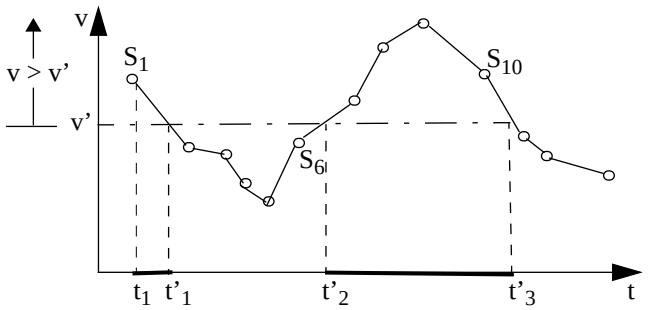


Fig. 6.7: Illustration of $\sigma^*_{v>v'}(TS)$

Discrete Range Selection

Recall from Section 5.2.2 that range queries can also be posed on *discrete* time sequences. For example, in Fig. 6.7, if we assume *no* interpolation function, then $\sigma^*_{v>v'}$(TS) will return a set of states $\{S_1, S_7, S_8, S_9, S_{10}\}$. How can we calculate these states? There are three steps involved:

1. Assume linear interpolation on TS. Pose the range query $\sigma^*_{v>v'}$(TS), the time intervals (t1, t'_1) and (t'_2, t'_3) will be returned.

2. By rounding these time intervals we get the state intervals $[S_1, S_1]$ and $[S_7, S_{10}]$ (see Section 5.2.2).

3. Return all states inside these state intervals. That is: $\{S_1, S_7, S_8, S_9, S_{10}\}$.

Therefore, the IP-index is essential for range queries on time sequences, even for discrete sequences.

6.3 Performance Measurements on SHORE

To measure the performance of the σ^* operator, we performed substantial experiments on SHORE [22]. The implementation notes (on how the IP-index and time sequences were implemented in SHORE) can be found in Section 4.2. In brief, the IP-index was implemented on top of a B^+-tree in SHORE. Anchor-state sequences were implemented as SHORE large objects which can grow arbitrarily large. Time sequences were implemented as an array of records (t_i, v_i). All measurements were made on a SPARC 20 machine with 64M main memory. The SHORE buffer pool size was set to 40 8K pages.

6.3.1 $\sigma^*_{v=v'}$(TS) — Using the IP-index or Scanning the TS?

As pointed out in Section 5.1, the alternate way to calculate $\sigma^*_{v=v'}$(TS) without the IP-index is to linearly scan the TS. To demonstrate the efficiency of $\sigma^*_{v=v'}$(TS) using the IP-index, we compared the time difference between using the IP-index and linear scanning.

Recall that the operator $\sigma^*_{v=v'}$(TS) is achieved by $IP_{v=v'}$(TS) and ifn^{-1} (Section 6.2.2). To exclude the time spent in ifn^{-1}, we assume t' = S_i.time (step 2 in Section 6.2.2) where S_i is returned by the IP operator in step 1. In this case the execution time of $\sigma^*_{v=v'}$(TS) will exclude the time spent in interpolation, both for using the IP-index and for linear scanning.

A detail is that S_i.time is not stored in A(v'); it has to be read from the time sequence array by using the state_id S_i (the state_id S_i is stored in A(v'), see Section 4.2.1).

Constructing the Synthetic T ime Sequence

In order to be able to control the properties of the time sequence used in the experiments, we generated a synthetic time sequence

```
v(i) = m(i) * sin(k * i) (i = 1, 2...10K)
```

which is a periodic time sequence with *growing* amplitude, see Fig. 6.8. The function *m(i)* is used to control the v_is so that 1) all v_is are inside a limited value range (it was [-10, 10] in the measurement) and 2) value ranges behave in the "step-wise constant" pattern as shown in Fig. 6.8. The reason for a limited value range is to make the size of the B^+-tree limited since we showed in Section 4.3 that most real time sequences result in limited size of the IP-index tree. The reason for the "step-wise constant" pattern of value ranges is that it makes it easy to construct different cardinalities of A(v')s by specifying the value of v'. For example, in Fig. 6.8 we have A(1.25) = 2*11 since 11 periods of sine data intersect with the line v = 1.25. The smaller the value v' is (v' > 0), the longer the A(v') will be. The maximum card(A(v')) occurs when v' = 0. The card(A(0)) was tuned to 2000 in the experiments by the parameter *k (*by tuning the frequency of the TS). Compared to the cardinality of the whole sequence, 10K, it results in the ratio of 2K/10K = 20%, which is sufficient to model the worst case behaviour. The reason is that we showed in Section 4.3 that the worst-case of card(A(v')) for the pressure sequence was only 5% of the cardinality of TS, although values are very noisy around v' = -0.25.

Experimental Results

We expect that the e xecution time of $\sigma^*_{v=v'}$(TS) using the IP-index will be linear to card(A(v')) since card(A(v')) is the number of states needed to be visited to get the results. By contrast, the execution time of $\sigma^*_{v=v'}$(TS) using linearly scanning TS will be linear to the cardinality of the whole TS since every state in the TS needs to be visited.

The selected v's and their corresponding cardinalities used in the measurements are listed in Table 6.1. The execution times of $\sigma^*_{v=v'}$(TS) with regard to card(A(v'))s are shown in Fig. 6.9. It verifies our "linear" speculation (above). It shows that the execution time of $\sigma^*_{v=v'}$(TS) by linearly scanning TS is the same for any value v', no matter how long the A(v') is. By contrast, the execution time of $\sigma^*_{v=v'}$(TS) by using the IP-index is linear to card(A(v')). Thus, *the smaller the card(A(v')) is, the more we gain by using the IP-index compared to linearly scanning TS.* Note that in most real life applications the submitted queries $\sigma^*_{v=v'}$(TS) are normally for

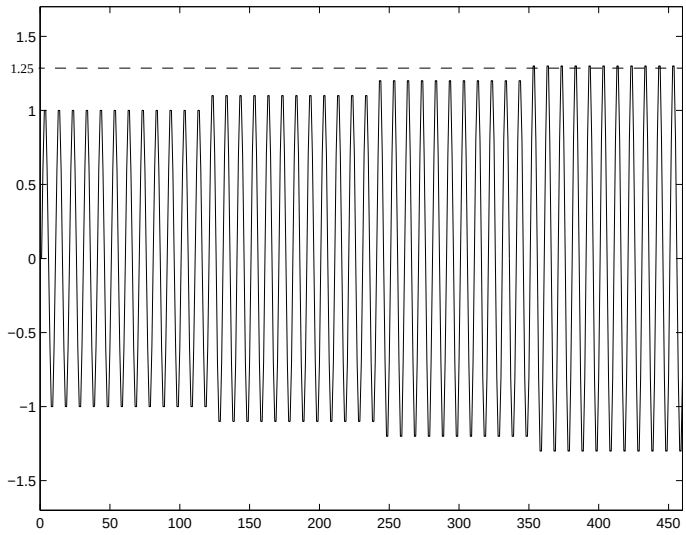


Fig. 6.8: The synthetic sine sequence

short A(v')s. For example, in Fig. 1.4, we are interested in those peaks where v > 1.5. Since $\sigma^*_{v>1.5}$(TS) is processed by $\sigma^*_{v=1.5}$(TS) (Section 6.2.4), the execution time is determined by the cardinality of A(1.5), which is then only 80 for the 100K time sequence, resulting in the factor of 80/100K = 0.08%. In this case the time difference between using the IP-index or not is dramatic.

Table 6.1: Selected v's and the cardinalities of A(v')s

v'	9.4	9.2	9	8.4	7.3	4.9	3.0	0
cardinality	14	60	106	246	504	1064	1508	2000

Another interesting observation is that for the card(A(v')) = 2000 (i.e., v' = 0), the query processing time of $\sigma^*_{v=v'}$(TS) by using the IP-index is approximately the same as linearly scanning TS — we do not gain anything any more. The reason is that to retrieve those S_is whose state_ids are in A(0), all disk pages storing the TS have to be visited since those S_is are evenly distributed in the disk pages that store the TS (page divisions for the TS are illustrated in Fig. 6.10). The cardinality of the anchor-state sequence is then 20% (2000/10K) of the cardinality of the original TS. The threshold of 20% is dependent on the page size, of course. The bigger the page size is, the smaller the threshold will be.

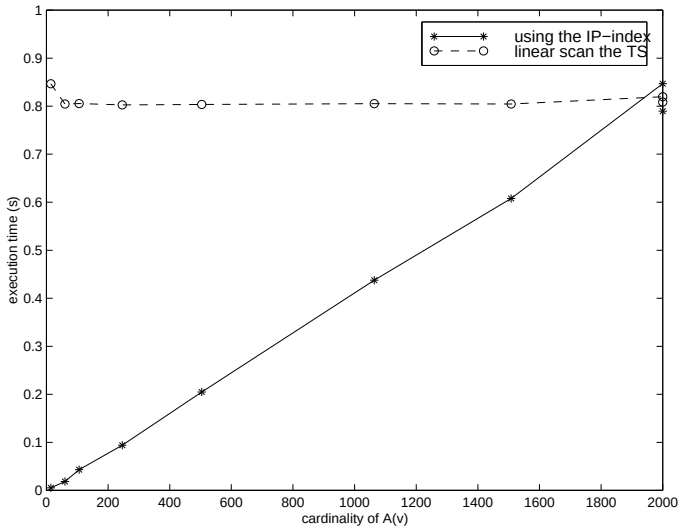


Fig. 6.9: The execution times of $\sigma^*_{v=v'}$(TS)

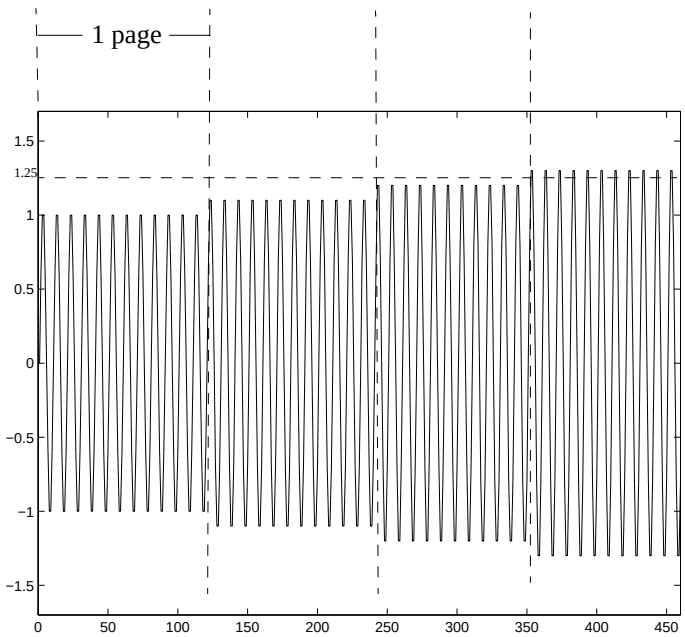


Fig. 6.10: The page division of a portion of the sine sequence

6.3.2 Getting the First Answer

We also measured the time to get the first answer of $\sigma^*_{v=v'}(TS)$ by using the IP-index, compared to linearly scanning TS. As mentioned in Section 6.2.2, it is important to get the first answer *quickly* in real-time query processing.

Constructing the Experimental Data

By using the synthetic sine sequence it is easy to simulate the situation when the first answer appears in different positions in the time sequence. The selected v's and the positions where they first appear in the TS (i.e., the state_id of the first state in A(v')) are listed in Table 6.2.

Table 6.2: Selected v's and the positions where they first appear in the TS

v's	1.0	2.9	5.1	7.3	8.5
first appears in position	122	2342	4912	7482	8882

Experimental Results

The execution times of getting the first answer to $\sigma^*_{v=v'}(TS)$ with regard to the position where the first answer appears in the TS are shown in Fig. 6.11. It shows that by using the IP-index the time to get the first answer is constant regardless of the position of the first anchor-state (because the first state_id in A(v') indicates where to retrieve the state S_i in TS). By contrast, the time for linear scanning to get the first answer can be very slow when the first anchor-state appears late in the TS.

The conclusion is that it is essential to have the IP-index in real-time query processing.

6.4 Related Work

In this section, we present the work related to the σ^* operator.

6.4.1 The Original σ* Operator

First of all, we would like to point out that our notation of σ^* is actually "borrowed" from an early paper on temporal databases — "Formal Semantics of Time in Databases" [33]. In [33], the operator σ^* was defined as "a historical database select", denoting selection of *implicit* states from $<S_1, S_2, ..., S_n>$

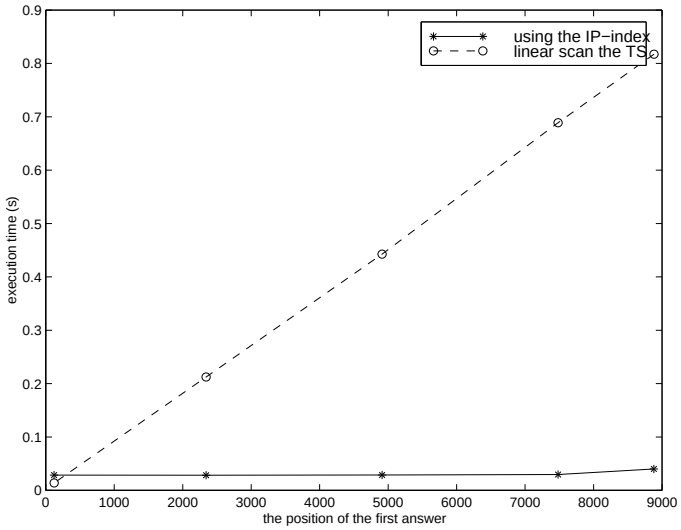


Fig. 6.11: The execution times of getting the first answer to $\sigma^*_{v=v'}(TS)$

(note that a state S_i in [33] actually means the time stamp t_i, not the pair (t_i, v_i) as we mean in this thesis). Since the *"step-wise constant"* assumption was assumed, [33] has the following formula for calculating $\sigma^*_{STATE = S}$:

$$\sigma^*_{STATE = s,\ A = x}(r_i) = \sigma_{STATE = [S],\ A = x}(r_i)$$

Here r_i denotes a relation and A denotes an attribute domain of this relation. The above formula basically says that: since "step-wise constant" is assumed on the relation r_i, the attribute at an implicit state S is equal to the attribute at the explicit state [S], where

$$[S] = max(S_i),\ \text{where}\ S_1 < S_i < S_n\ \text{and}\ S_i \leq S$$

However, in [33], only the formal semantics of the σ^* operator were given, no implementation issue was discussed. Also the σ^* in [33] only supports the "step-wise constant" assumption.

In this thesis we extend the σ^* operator in [33] to support arbitrary user-defined interpolation functions. The strategy is to separate the σ^* operator from the interpolation function ifn^{-1} (by introducing the IP operator, see Section 6.2.2). In this way different kinds of interpolation functions can be supported. In contrast, in the above formula $\sigma^*_{STATE = S}$, the interpolation function ("step-wise constant") is "hard coded" into the definition of the σ^* operator so that only the

pre-defined interpolation function is supported (also note that for more sophisticated interpolation functions, it will be difficult if not impossible to hard code them into the definition of the σ* operator).

6.4.2 The "System Query" Q'

Bettini et al. [21] address a problem in temporal databases [134] which is similar to the problem of value queries, i.e., deriving implicit information from explicitly stored information in DBMSs. In [21] it is pointed out that "when querying a temporal database, a user often makes certain semantic assumptions on stored temporal data". Two types of semantic assumptions are formalized in [21]: *point-based* and *interval-based*. The *point-based* assumptions are "those semantic assumptions that can be used to derive information at certain ticks of time based on the information explicitly given at different ticks of the same temporal type (i.e., temporal granularities)". The *interval-based* assumptions include those that involve different temporal types (time granularities). Each assumption is viewed as a way to derive certain implicit data from the explicit data stored in the database.

In [21], the approach to evaluate queries concerning implicit data is the following: By assuming an interpolation assumption, a database DB can be seen as a larger database $\overline{DB}$ that contains both e xplicit and implicit information. A user query Q is then translated into a system query Q' such that the answer to Q' over the e xplicit data is the same as the answer to Q o ver the e xplicit *and* the implicit data. The central point of this approach is that the interpolation assumption ("step-wise constant") is hidden inside the system query Q'.

By contrast, our approach is to associate the interpolation assumption *ifn* with the SELECT operator σ instead, resulting in the σ* operator. In this way there is no need to transform the database DB to $\overline{DB}$ or the user query Q to Q'.

The main contrib ution of [21] is on *formalization* of semantic assumptions and how to transform a user query into a system query . Query e valuation, on the other hand, is listed as a "future w ork" in [21]. By contrast, our w ork deal with *implementation* issues such as query e valuation and ho w to achie ve ef ficienc y (in addition to the formal definitions of $\overline{TS}$ and the σ* operator). Another difference is that we support more sophisticated interpolation functions such as linear interpolation or mo ving a verage, while in [21], only "step-wise constant" or the a verage of tw o neighbor points are supported.

6.4.3 Relevant Operators in T emporal Databases

In the research literature on temporal databases, we ha ve found se veral opera-

tors (e.g., [32]) that have semantics very similar to the σ* operator.

To introduce these operators, we have to explain the data model in [32] first. In [32], the relational data model is extended by allowing an attribute to be a *time-varying attribute* (TVA) (an attribute whose values vary over time). For example, in Fig. 6.12, the attribute HEAD in the temporal relation STUDIOS is a TVA. A TVA is represented as *functions* from the time domain to the attribute domain, such as "1924 —> Mayer". The same holds for the relation LAWYERS in Fig. 6.13.

STUDIOS		
(STUDIO	HEAD	NUM_FILMS
MCM	1924 —> Mayer	1924 —> 6
	1948 —> Schary	1925 —> 10
	1956 —> NULL	1970 —> 15
	1970 —> Aubrey	
	1974 —> NULL	
Paramount	1919 —> Cukor	1919 —> 2
	1925 —> Schulberg	1925 --> 12
	1935 —> NULL	1936 —> 10
RKO	1945 —> Schary	1945 —> 10
	1948 —> Hughes	1946 —> 11
	1957 —> NULL	1947 —> 12
Warner Br.	1923 —> J. Warner	
	1969 —> Ashley	NULL
	1927 —> NULL	
Universal	1912 —> Laemmle	1930 —> 6
	1936 —> Blumberg	1937 —> 9
	1946 —> Spitz	1965 —> 11
	1952 —> Rackmil	
	1955 —> Hunter	
	1965 —> Wasserman	

Fig. 6.12: The STUDIOS relation

The σ Operator

Since an attrib ute in [32] can be of a structured domain (i.e., functions from TIME to a simple domain), the definition of the traditional σ operator is signif-

LAWYERS (LAWYER	STUDIO	SALARY
Howell	1924 --> MGM	1924 --> 30K
	1930 --> Paramount	1925 --> 35K
	1937 --> MGM	1937 --> 40K
	1940 --> NULL	1940 --> NULL
Rosen	1912 --> Universal	1945 --> 70K
	1923 --> Warner Br.	1953 --> NULL
	1930 --> NULL	
	1945 --> RKO	
	1953 --> NULL	
McManus	1923--> Warner Br.	1923 --> 35K
	1930 --> NULL	1926 --> 40K
		1930 --> NULL

Fig. 6.13: The LAYWERS relation

icantly affected. Consider the following example:

$\sigma_{(Studio(1925) = MGM)}(LAWYERS)=$

(Lawyer	Studio	Salary)
Howell	1924 —> MGM	1924 —> 30K
	1930 —> Paramont	1925 —> 35K
	1937 —> MGM	1937 —> 40K
	1940 —> NULL	1940 —> NULL

This example highlights the need for an interpolation function for TVAs. As Clifford and Tansel [32] point out: “Users must be able to query the database at will with respect to time points or periods, and yet the database cannot possibly store values for every attribute at every point at time. Thus, each attribute must have an associated interpolation function, so that the database system can reconstruct an entire time series over the lifespan of each object from the partial specification stored.” In [32] “step-wise constant” was assumed on the TVA `Studio`, hence the selection

$$\sigma_{(Studio(1925) = MGM)}(LAWYERS)$$

yields the first tuple (in the relation LAWYERS) as shown above.

The τ Operator

Another relevant operator in [32] is the *Time Slice* (τ) operator. A *Time Slice* operator retrieves the *snapshot* state of a relation R at a certain time *t* (refer to [3][102] for more general discussions about the Time Slice operator). In [32] it is stated that: "In a sense, the τ operator is a kind of σ, in which a value from the domain of a TVA is given. However, it is more general in that it allows the selection across all of the attributes in the relation". Consider the following example:

```
τ(1925)(STUDIOS) =
               (Studio     Head                Num_Films)
                MGM        1925 --> Mayer      1925 --> 10
                Paramont   1925 --> Schulberg  1925 --> 12
                Warner Br. 1925 --> J. Warner  NULL
                Universal  1925 --> Laemmel    NULL
```

This example shows that there are different ways of modelling data over time. The attribute `Head` is interpolatable, and uses a simple step-wise constant interpolation. However, the attribute `Num_Films` is inherently non-interpolatable. From the value of `Num_Films` at a given time point, we cannot infer anything about its value at any other time point. Therefore the retrieved values of `Num_Films` are `NULL` (unknown) for certain tuples, as shown above.

The τ operator is similar to the $\sigma^*_{t=t'}$ or $\sigma^*_{t>t'}$ operator. They both retrieve the values (attributes) at a certain time point or time intervals. However, in [32], only the semantics of the τ operator are addressed, no implementation (query evaluation) is addressed. The purpose of [32] is on extending the relational data model to incorporate the temporal dimension. By contrast, our work on the $\sigma^*_{t=t'}$ and $\sigma^*_{t>t'}$ operators address not only formal definitions but also implementation issues. Another difference is that the τ operator only supports "step-wise constant" interpolation or the non-interpolatable assumption (the returned `NULL` values in the above example), while the $\sigma^*_{t=t'}$ or $\sigma^*_{t>t'}$ operators support arbitrary user-defined interpolation functions.

The Ω Operator

In [32], there is a time-related operator *WHEN* (Ω), which provides a mechanism for naming time values not simply with constants (like 1983) but with expressions (like WHEN A = v in the relation R). This unary operator on relations, unlike the other relational operators, yields as a result a set of *times* rather than a relation. It is used to form temporal expressions which can serve as components of a τ or σ operator in [32]. The result of an Ω is a set of time intervals. Consider the following example:

$$\Omega_{(\texttt{Studio = Paramount, Head = Cukor})}(\texttt{STUDIOS}) = [1919, 1925]$$

The Ω operator is similar to the $\sigma^*_{V=V'}$ operator in the sense that they both retrieve the time stamps when the values ("attributes" in [32]) satisfy some conditions. The main difference is that the Ω operator only retrieves the explicit (stored) time stamps (more specifically, valid timestamps) while the $\sigma^*_{V=V'}$ operator can also retrieve the implicit (interpolated) time stamps according to the user-defined interpolation function.

6.5 Proposing New Functions for the ADT of Time Sequences

As Stonebraker [131] points out, modelling time series in relational databases has drawbacks on both query processing time and space usage. Here is an example. The Wall Street financial center manages closing price of stocks for over 5000 securities. The traditional way of constructing this application is to form a table for each security, such as this table for IBM:

```
create table IBM (
               date date,
               price float);
```

Then, if we wish to find the difference between IBM's five-day moving average and its 200-day moving average on July 15, 1995, we would have to write the following program:

```
main ()
{
find July 16th IBM record;
until 5 records seen
   {
   read previous IBM record;
   update 5 day average;
   update 200 day average;
   }
until 195 records seen
   {
   update 200 day average;
   }
return (5 day average - 200 day average);
}
```

This application requires a custom program as well as a sequential scan of 200 records. In addition, suppose we wanted to perform the calculation for all stocks. In this case, we must put the above logic inside an outer loop that iterates over 5000 securities. Now, there are many records examined in 5000 dif-

ferent tables.

Therefore, Stonebraker [131] suggests that a time series should be modelled as an *abstract data type* in object-relational database systems. At the time this thesis is written, time series have already been implemented in several commercial database systems as a new data type, such as Informix's time-series DataBlade [65], Oracle's time-series DataCartrige [95], and IBM's time-series DataExtender (although *value queries* are not supported in any of these systems, and some issues such as physical organization of time sequences and query optimizations might need further investigation). For example, the time series data type in Informix consists of the following information:

- calendar obeyed by the time series
- starting time of the time series
- stride between values (for example, daily or monthly)
- data types of elements (for example, float or polygon)
- legal time series values in order

With this data type, the following stock table can replace the 5000 tables discussed earlier:

```
create table stock {
                name    varchar(30);
                prices  time_series of floats);
```

Here we have one table, not 5000 tables, and one record per stock, not one record per stock per date.

In Informix, there are 40 or so legal operations on time series. These include constructing a moving average, extracting a subset of the time series, and aggregating the time series to coarser granularity. With these operations, the above query can be formulated as follows:

```
select moving_avg (prices, 5, '1995-07-15') -
       moving_avg (prices, 200, '1995-07-15')
from stock
where name = 'IBM';
```

This introduces two main advantages over the previous representation. First, it can be completely expressed in SQL, making it easier for the user to code the functionality. Second, it runs much faster than the previous representation because only one row of one table needs be examined. Moreover, the code that

walks down the time series is very efficient and has a low overhead relative to the code that examines records in a relational system.

Therefore, time sequences should be modelled as an abstract data type instead of as relational tables. In addition to the above functions defined for time series, we propose the following functions which support interpolation assumptions on TS:

```
get_time_stamps(TS, '=', v')                                    (1)
    // assume default interpolation assumption
```

or:

```
get_time_stamps(TS, '=', v', ifn)
    // assume user-defined interpolation assumption ifn
```

The above functions return the time points when the value is equal to v' for a continuous TS. This function is translated to the $\sigma^*_{v=v'}$(TS) operator and it is efficiently supported by the IP-index.

To support range queries on continuous time sequences, we propose the function:

```
get_time_intervals(TS, '>', v')
    // assume default interpolation assumption
```

or:

```
get_time_intervals(TS, '>', v', ifn))
    // assume user-defined interpolation assumption ifn
```

to return those time intervals when the values are greater than v'. This function is translated to the $\sigma^*_{v>v'}$(TS) operator and is efficiently supported by the IP-index.

The data type of time sequences is just one example of an extension to base types in DBMSs. There are other application data which can be modelled as abstract data types as well. Examples are spatial objects (points, lines, polygons, etc.) and multimedia data such as images or videos. As Silberschatz et al. [118] point out, the object-relational DBMS allows complex types, nested relations, and object-oriented features. Standardizing queries on complex types in SQL3 is under way.

6.6 Summary

In this chapter we have presented the extended SELECT operator, $\sigma*$, which retrieves *sub-sequences* (time intervals) in a time sequence TS where the values inside those sub-sequences satisfy some conditions. The $\sigma*$ operator supports arbitrary user-defined interpolation functions on TS. The implementations of the $\sigma*$ operator for various selection conditions were presented. The $\sigma*$ operator is applicable to any 1-D sequence data.

We have performed extensive experiments on SHORE [22] using both synthetic and real-life time sequences. The experiments show that the $\sigma*$ operator (supported by the IP-index) dramatically improves the performance of value queries on time sequences. The performance gain is even more dramatic for large sequences with *small answer sets*, while most submitted value queries in real-life applications *are* for small answer sets. Another promising observation is that the performance of $\sigma*$ for *the first few answers* is stable, regardless of the *positions* where the first few answers appear in the time sequence. This shows that the IP-index is essential in the situations when the time sequence is long and the query processing time is limited.

Related work to the $\sigma*$ operator was discussed. The $\sigma*$ operator was compared to other proposed operators in temporal databases. The main advantages of the $\sigma*$ operator is in twofold: 1) the implementation issues are fully investigated; 2) it supports arbitrary user-defined interpolation functions (while most other operators only support “step-wise constant” interpolation).

We also proposed some new functions for the abstract data type of time sequences. The unique feature of these functions is that they support user-defined or system-defined interpolation assumptions on time sequences.

Chapter 7

Physical Organization

*P*hysical organization [137] determines the efficiency of a database system. Physical organization addresses many issues in DBMSs such as data structures, index design, file format, data transfer between main-memory and disks, buffer management, etc. In this chapter we are particularly interested in the following issues:

- Physical organization of time sequences.

  Time sequences are usually very large in volume, and many of them are dynamically growing. Designing a good data structure for large, dynamic time sequences is challenging since one needs to minimize the amount of storage used while maintaining reasonable access time. In Section 7.2.3, we present a persistent data structure which scales up gracefully with the growing of the time sequence and supports fast random access in the time domain.

- Physical organization of secondary indexes:

  What distinguishes the IP-index from conventional secondary indexes is the anchor-state sequences, A(v')s. For one particular time sequence, A(v')s are dynamically growing, and vary much in length for different values of v'. Thus it is important to have a good data structure for A(v'). The design goal is not to waste space for small A(v')s and to support fast random access for large A(v')s.

- Physical organization of large objects:

  Many application data result in large objects in DBMSs, such as time series, image, and video data. In Section 7.4, we provide an overview of how large

objects are managed in various systems, including relational DBMSs and object-oriented DBMSs.

- The impact of main-memory or disk resident DBMSs on physical data organization and index design.

  Main-memory resident DBMSs and disk-resident DBMSs have different properties that affect almost *every* aspect of system design and implementation. In Section 7.5, we investigate how these different properties affect physical implementation issues in DBMSs such as *index design* and *data structures*, especially for sequence data.

7.1 Database Access Time

This section provides a background to *database access time*.

Conventional database systems are *disk-resident* (DRDBs), i.e., data is stored permanently on disks. Data are moved into main-memory for processing and moved back to disks when they are no longer needed. Space on a disk is allocated in the unit of *blocks* (or *pages*), whose size ranges from 512 bytes to several kilobytes (depending on how the disk is configured). A key concept is that the transfer of data between disks and main-memory is in the unit of *blocks*, not in the exact size of the data that is needed. For example, if we wish to retrieve a 4-byte integer from a disk where the block size is 4K, then the whole block (size 4K) where this integer resides will be brought into main memory.

Currently, one disk I/O takes approximately 10ms, and one main-memory access takes approximately 0.1μs. Therefore, disk I/O is in orders of magnitude slower than the access of data in main-memory, which makes disk I/O the bottleneck of the database access time in a DBMS. In estimating database access time, we normally ignore operations in main-memory, count only the time spent on disk I/O. Since disk I/O is in the unit of blocks, this makes the cost of physical database access determined by the *number of blocks accessed*.

Therefore, the goal of physical database design in disk-resident database systems is to minimize the number of disk blocks accessed.

As main-memory becomes cheaper and the capacity of main-memory becomes larger, *memory-resident database systems* (MMDBs) are becoming more and more popular nowadays. The design of a main-memory DBMS is significantly different than that of a disk-resident DBMS [56]. We will discuss the differences in Section 7.5. For now, we assume that all of our discussions are based on conventional, disk-resident DBMSs.

7.2 Physical Organization of Time Sequences

Since time sequences are usually very large in volume, storage efficiency is the key to the practicality of time sequence support in DBMSs. To design a good data structure for time sequences, we have to understand their properties first.

7.2.1 Properties of Time Sequences

A time sequence normally has the following properties:

1. A time sequence is *ordered* by time.

   The order of values in a time sequence is important, and thus should be preserved in the physical structure.

2. Time sequences are usually very long [117].

   This indicates that the data structure should be aimed at disk storage (i.e., *persistent* data structure) instead at main memory storage.

3. Time sequences are mostly append only [117]. Updates or deletes are rare compared to insert.

   This indicates that a *non-updatable data structure* can be used in order to achieve better storage and access efficiency. In other words, deletion can be sacrificed in favour of insertion.

4. Time sequences can be dynamic (Section 2.1).

   This indicates that the data structure should *scale up* gracefully with the growing of the time sequence.

7.2.2 Arrays for Time Sequences

Considering the above properties, we suggest that an *array* structure is a good choice for storing a time sequence. An array structure is *compact* in storage, in the mean time it provides *fast* random access to any element. Let us see how arrays can be used to store different kinds of time sequences.

Regular/Irr egular T ime Sequences

Recall that a time sequence can be *regular/irr egular* (Section 2.1). This property af fects ph ysical or ganization substantially . For e xample, if we store a *regular* time sequence in an array , then we w ould not need to store the time stamps (see Fig. 7.1). This is because the time stamp t_i can be *computed* by $t_i = t_1 + (i-1)*\Delta t$ (Δt is a constant for a re gular time sequence). Also, because of the rela-

tionship between a time stamp t_i and the position of the record in the array, random access of any value given its time stamp t_i can be computed *without* the need to build an index on the time domain.

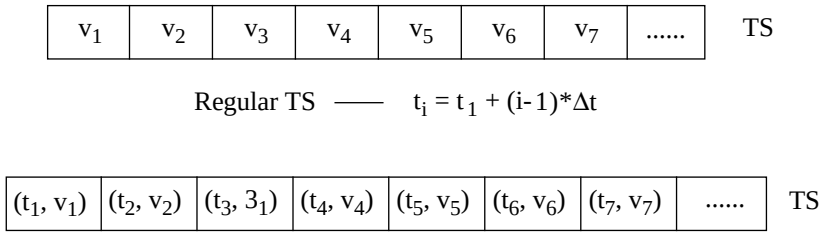


Fig. 7.1: Regular/Irregular TS stored in an array

Supporting *irregular* time sequences is more complex. Normally the (t_i, v_i) pair has to be stored in the array (see Fig. 7.1). There are two consequences of this approach. First. the ability to "factor out" time stamps (i.e., time stamps need not be stored) in regular time sequences is lost. Secondly, the simple indexing capability over the time domain that exists in the regular case is lost. This indicates that there is a need to build an *index* on the time domain for irregular time sequences in order to support fast random access.

Static/Dynamic T ime Sequences

The *static/dynamic* property (see Section 2.1) of a time sequence af fects ph ysical or ganization in the sense that a *dynamic* time sequence requires a data structure which is *dynamically gr owing*. Therefore, a simple, static *arr ay* structure will not do. An alternati ve is to allocate a small array first, then double the size whene ver the array becomes full. This requires cop ying the pre vious allocated array to the ne wly allocated array each time the array is e xpanded. Performance is apparently bad for lar ge sequences. Therefore, we propose a *multi-le vel dynamic arr ay* structure in the ne xt section to meet this challenge.

7.2.3 The Multi-Le vel Dynamic Array Structur e

The initial moti vation that led to this w ork came from our e xperiments in SHORE [86]. W e found in SHORE a persistent, dynamic data structure named sequence which seemed to be a perfect choice for implementing time sequences. Sequence is a built-in data type in SHORE. It is basically a dynamic array that can grow arbitrarily large and supports operations (methods) such as

`insert`, `delete`, and `update`. Unfortunately, we found out that when one element of a `sequence` is accessed, the *entire* sequence is read into main-memory. This is certainly very inefficient when the time sequence is long (e.g., 100K). The reason for this surprising behaviour is that, as stated in one of SHORE's manuals [115], demand-paging for `sequence` is not implemented in the current version (version 1.1).

Therefore, we started to investigate a good data structure for large, dynamically growing time sequences. Our first idea was to partition the large time sequence into arrays (each array fits in one disk page) and use a B^+-tree [35] to index these arrays. In this case only one disk page needs to be read into main memory when one element is accessed. Another advantage of this data structure is that it supports fast random access through the B^+-tree.

Further investigations indicate that a B^+-tree is not really needed. The reason is that these arrays are allocated in order (because a time sequence is ordered), while a B^+-tree is normally needed for keys that do not arrive in order (that is why a B+-tree is a *dynamic* data structure). Taking into account that a time sequence is an ordered sequence, a simpler solution is to use *arrays* instead of a *tree* to index these arrays.

The Data Structure

Therefore, we propose a *multi-level dynamic array* structure (Fig. 7.2), which meets the challenge of supporting both *fast appending* and *efficient random access*.

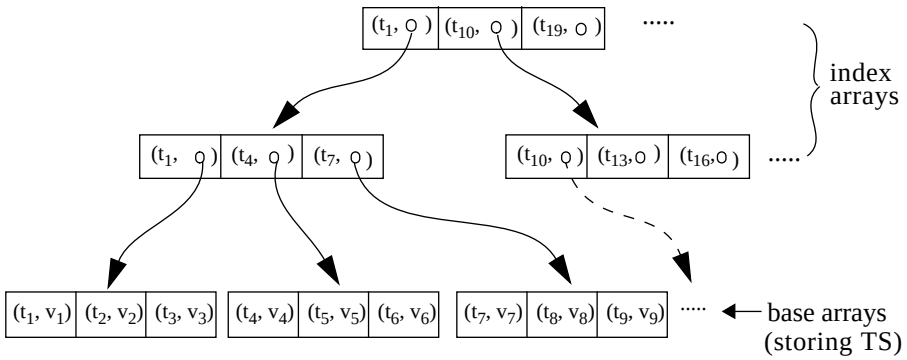


Fig. 7.2: The multi-level dynamic array structure for a dynamic, irregular TS

The multi-level dynamic array structure consists of *base arrays* and *index arrays*. Each base array or index array fits in one disk page. The base arrays are

used to store (t_i, v_i) pairs in the TS, see Fig. 7.2 (there we assume three (t_i, v_i) pairs fit in one page). The index arrays are for indexing base arrays. The first-level index arrays have the form (t_j, *pointer*) where *pointer* points to the base array with the starting time stamp t_j, see Fig. 7.2. The second-level index arrays have the form (t_j, *pointer*) where *pointer* points to the first-level index array with the starting time stamp t_j. The multi-level dynamic array structure grows from bottom to top as TS grows. Random access of any element of TS can be achieved easily by a search starting from the top-level index array and following the pointers down to the lower level index arrays until a base array is reached. Fast appending is assured because only the right-most arrays in each level needs to be accessed when a new (t_i, v_i) pair is inserted into TS.

The advantage of using an array as an index instead of a B^+-tree is that it is *more efficient in terms of both space and time*. The reason for space efficiency is that a node in a B^+-tree is not always full (> 50%), while our index arrays will be mostly full (except those belonging to the right-most chain). The reason for time efficiency is that we do not need to perform *node balancing* as in a B^+-tree when new keys are inserted. When a new array is allocated, only the pointers in the right-most arrays need to be adjusted.

Now let us see how this data structure scales up with the growing of the time sequence. Suppose that a pointer and a value v_i (a floating point number) takes 4-bytes each, a timestamp t_i takes 8-bytes [124], then a 4K page will hold approximately 333 ($\cong$ 4K/12) elements for either a *base array* or an *index array.* By using a 2-level dynamic array structure, we can store a TS with *cardinality* up to $333^2 \cong 111K$ (the size of this time sequence will then be 111K * 12 $\cong$ 1.3MB). By using a 3-level dynamic array structure, we can store a TS with cardinality up to $333^3 \cong 37M$ (the size of the time sequence will then be 37M * 12 $\cong$ 444 MB). Thus, the multi-level dynamic array structure scales up *gracefully* with the growth of the time sequence.

Notice that this estimation is pessimistic because most large time sequences are regular ones, i.e., *time series* (Section 2.1.4). For regular time sequences, only values need to be stored (time stamps are factored out, see Section 7.2.2). An example of the multi-level dynamic array structure for a regular time sequence is shown in Fig. 7.3. There, in base arrays, only values v_i are stored. In the first level index arrays, (*i, pointer*) pairs are stored where *pointer* points to the base array whose first element is v_i.

In this case a 4K page will hold 1K (4K/4) elements for a base array, and 500 (4K/8) elements for an index array. Then a 3-level dynamic array structure can hold a TS with cardinality of 500^2 * 1K = 250M. The size of the time sequence will then be 250M * 4 = 1 GB. See the “capacity” of the multi-level dynamic array structure in Table 7.1.

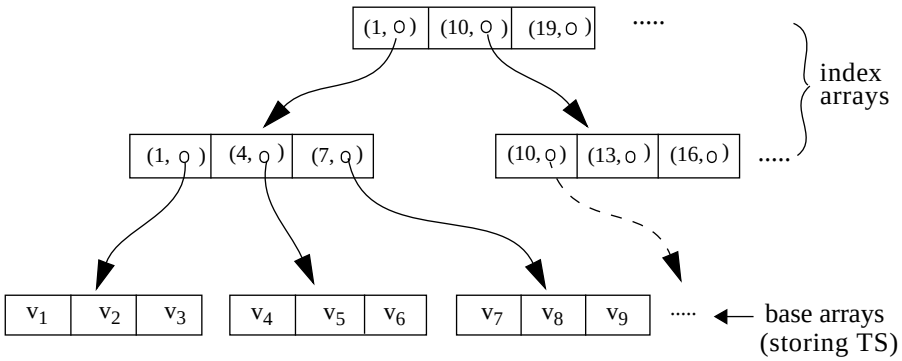


Fig. 7.3: The multi-level dynamic array structure for a dynamic, regular TS

Table 7.1: Capacity of the multi-level dynamic array (page size: 4K)

Levels	Regularity	TS Size	Index size	% index size / TS size
	Irregular	1.3MB	4KB	0.3
2	Regular	2MB	4KB	0.2
	Irregular	444MB	1.3MB	0.3
3	Regular	1GB	2MB	0.2

In Table 7.1 the space usage of the *index arrays* is also listed for each case. It can be of great interest to compare the size of the index arrays with the size of the TS. It can be seen that in the worst case (i.e., the case for irregular TSs), the space usage of the index arrays is 0.3% compared to base arrays. In the best case (regular TSs), the space usage of the index arrays is 0.2% compared to base arrays. Therefore, the space overhead of indexing a TS using the multi-level dynamic array structure is negligible.

Notice that for a small TS that fits in one page, we do not need the index arrays. One base array will do. This is a degenerate case of a multi-level dynamic array structure.

Insertion

Insertion into the multi-level dynamic array is very straightforward. We will illustrate the insertion process in Fig. 7.4 and Fig. 7.5. In our discussions we

assume one disk page holds 300 elements (≅ 333 above). However, for illustrative reasons, in Fig. 7.4 and Fig. 7.5, we assume 3 elements for each page.

Let us start with an empty TS. When the pair (t_1, v_1) arrives, we allocate a base array (let us call it base_array_1) and store (t_1, v_1) in the first element. In the meantime we allocate an index array (let us call it index_array_1) with the first element (t_1, *pointer*) where *pointer* points to the base_array_1, see Fig. 7.4. Suppose one disk page holds 300 pairs, then the insertion of the remaining (t_i, v_i) (i = 2, 3, 4...300) can be done easily by just storing them in base_array_1 without any further operation. When (t_{301}, v_{301}) arrives, we allocate a new base array (let us call it base_array_2) and fill in the second element in index_array_1 with (t_{301}, *pointer*) where *pointer* points to base_array_2. Repeating this procedure, we can store the time sequence with cardinality 300^2 without creating a new index array.

Now, when the No. 300^2+1 element in TS arrives, we have to allocate a new base array (illustrated as base_array_4 in Fig. 7.4) and a new index array (illustrated as index_array_2, see Fig. 7.4). The new base array is used to store the new (t_i, v_i) pair and the new index array is used to index this new base array and the old index arrays. The current structure is illustrated in Fig. 7.4. If you view it as a tree structure (with index arrays as internal nodes and base arrays as leaf nodes), then the tree is unbalanced because the *level* of the right sub-tree (which is 2) is less than that of the left sub-tree (which is 3).

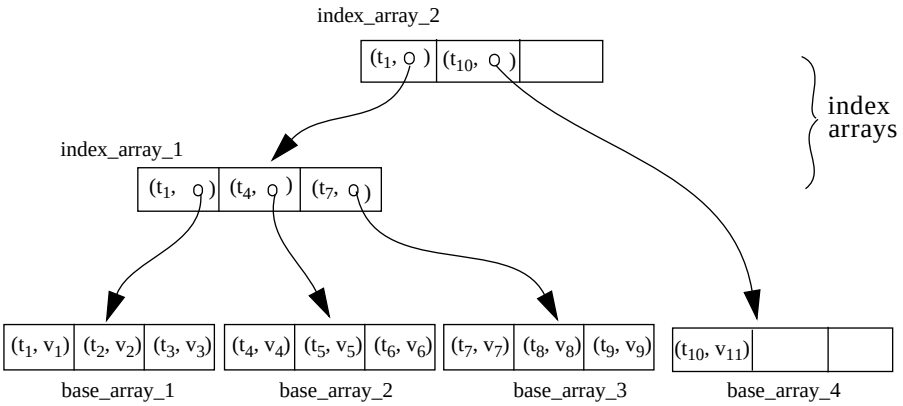


Fig. 7.4: Insertion in the multi-level dynamic array (unbalanced structure)

The insertions of the remaining 299 elements can be done easily by filling the base_array_4. When the 300^2+301 element in TS arrives, the multi-level dynamic array will appear as in Fig. 7.5. At this moment the tree structure is balanced.

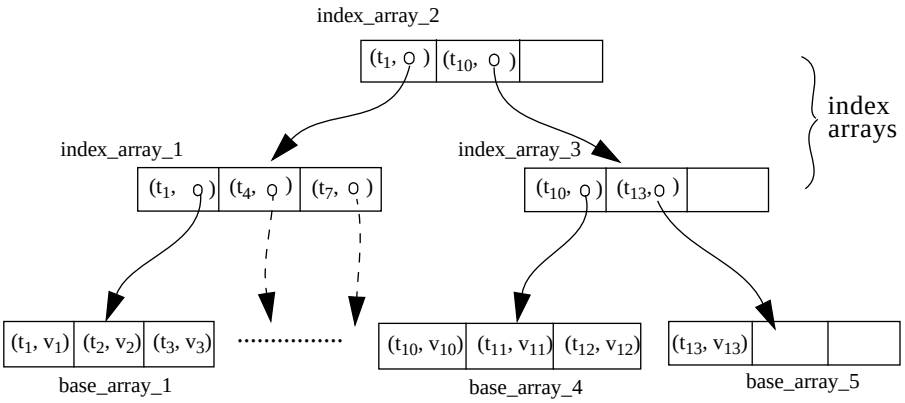


Fig. 7.5: Insertion in the multi-level dynamic array (balanced structure)

The insertion continues until the tree structure in Fig. 7.5 is full. Then a new, higher level index array will be needed. In this way the tree structure grows with the TS. Notice that, as we calculated in the last section, a 3-level dynamic array structure (as in Fig. 7.5) will be able to hold a TS with a considerable large cardinality. Therefore we normally do *not* need to allocate higher level index arrays.

Migration

When TS gro ws v ery long, old parts of the sequence can be migrated to tapes or other of f-line storage easily . For e xample, in Fig. 7.5, we could migrate the left sub-tree whose root is inde x_array_1 to tapes, and shift e very (t_i, *pointer*) pairs in inde x_array_2 one element left (or we can simply let the pointer (t_1 *pointer*) points to *nil*). This migration will not af fect the right sub-tree at all. Therefore, it can done easily without rebalancing the tree or adjusting other pointers in the sub-trees.

Sear ch

Search in the multi-le vel dynamic array can be done ef ficiently gi ven a time stamp t. Normally we w ould lik e to retrie ve the (t_i, v_i) pairs in TS where the t_is are *close* to the gi ven time stamp t (such as $t_i \le t < t_{i+1}$). This can be done by performing binary search in each relevant array, starting from the root array (index_array_2 in Fig. 7.5), and traversing down the tree structure until a leaf node is reached. In the leaf node (a base array), binary search is performed to find the positions where the "closest" (t_i, v_i) pairs reside.

The cost will be, of course, determined by the level of the structure (which is

determined by the cardinality of the TS). For a 3-level dynamic array, the cost of retrieving (t_i, v_i) pairs given a time stamp will be 2 disk I/Os since the root array is always kept in the main memory.

To facilitate linear scanning or sub-sequence retrieving, the leaf nodes (base arrays) can be linked together in allocating order. This is the same in most B^+-tree implementations.

7.2.4 Related Work

Although little work has been done in data structures of temporal data or time sequences, we found some related work on *indexing* temporal data. Some of this work is very close to the idea of our approach (such as the PLI-tree below). We compare work related to the multi-level dynamic array structure in this section.

Comparison with the PLI-tree and the AP-tree

The closest related work to our multi-level dynamic array structure is perhaps the I-tree [135], the PLI-tree [135] and the AP-tree [58]. These tree structures are all designed for efficient access of append-only temporal data. The I-tree and the PLI-tree are designed for indexing the transaction timestamps [67] of entries in a backlog [135] to efficiently support *differential computation* of *timeslices* [135] (a PLI-tree is an improved version of an I-tree [135] where *pointers* in each tree node are *computed* instead of stored). The AP-tree is designed for indexing *interval* timestamps of temporal relations to support *event-join* optimization [104]. A PLI-trees bares very similar structure to an AP-tree. They are both multiway search trees that are hybrid of an ISAM index and a B^+-tree. The main difference is that the PLI-tree favours insertion more than the AP-tree and completely sacrifices deletion. That is also the reason why pointers can be computed in a PLI-tree but not in an AP-tree. By saving the space for pointers, a PLI-tree node has more *fanout* than an AP-tree node. Therefore a PLI-tree takes less space than an AP-tree. (According to [135], a PLI-tree is approximately 33% smaller than an AP-tree when indexing 1 million pages.)

If we compare the I-tree (or PLI-tree) with our multi-level dynamic array structure, we see that the I-tree structure is very similar to the part of the *index arrays* in Fig. 7.2. Actually, if we replace the backlog in [135] with the base arrays in Fig. 7.2, then, an I-tree can be used to index the base arrays in the same way as they are used to index the backlog. The way the index arrays grow with the TS (Section 7.2.3) is also very similar to the way how an I-tree grows with the backlog. A minor difference is that the structure of a "node" is different between these two index structures. In the multi-level dynamic array struc-

ture, a node is an *array* structure so that binary search can be performed.

As we mentioned before, an I-tree is used to index a backlog [135] while a multi-level dynamic array structure is used to index time sequences (stored in base arrays). Let us look at the difference between a backlog and a TS. A *record* in a backlog [135] takes approximately 128 bytes [135], while a *record* in irregular TSs (i.e., (t_i, v_i)) takes 12 bytes (since t_i, takes 8 bytes and v_i takes 4 bytes). Therefore, for a backlog and a TS with the *same cardinality*, the number of pages used to hold the backlog will be 10 times more than the number of pages used to hold the irregular TS (the ratio will be even bigger if TS is regular). Therefore, the size of an I-tree (i.e., the number of tree nodes) for the backlog will be much bigger than the number of the index arrays needed for the TS. This is the reason why [135] uses computed pointers to increase the fanout of each node of the tree (thereby reducing the size of the tree). In our case, as we calculated before, a 3-level dynamic array will be able to hold a regular TS with size of 1G (assume 4K page size). Therefore, we do not need to investigate a "pointless" version of the multi-level dynamic array.

Another difference between the PLI-tree and the multi-level dynamic array is that the PLI-tree is more suitable for indexing *long* backlogs, while the multi-level dynamic array structure is suitable for both *long* and *short* time sequences. The reason is that in a PLI-tree, disk *extents* are allocated to store the nodes of the tree. To decide the size of the extent is tricky: too big a size will waste space for small trees (in the situation of short backlogs), too small a size will make the size of the array [135] that contains starting addresses of all extents (this array needs to be kept in main memory, as stated in [135]) rather large. Therefore, the PLI-tree is not suitable in the situation when both long and short backlogs need to be supported (unless the size of a disk extent can be made *dynamic* for different sizes of backlogs and this dynamic information is maintained somewhere as meta-data).

Linked List

Shoshani and Kawagoe [117] suggested a simple approach to store dynamic, irregular time sequences. That is, allocate pages (blocks) in order for the growing time sequence and use an ordered list of (*page_number start_time*) to index these pages. Given a time stamp t_i, such an index can be searched to find the page that holds the corresponding value v_i. In this case the access of an element requires two steps: one to determine the appropriate page, and one to locate the position of the value in that page.

This approach, compared to our multi-level dynamic array structure, has the drawback that it has to do linear scanning on the list (*page_number*, *start_time*) to find the corresponding page (and linear scan the page to find v_i). This will be slow when TS is long.

Arrays versus Relational Tables

Arrays are used to implement *time series* (i.e., *regular* time sequences, see Section 2.1.4) in some commercial DBMSs and special-purpose management systems. Examples are the object-relational DBMS Informix [65] and the special purpose management system FAME [52]. Compared to our multi-level dynamic array, arrays in these systems are simple, *static* structures, with no support for dynamic TSs. Neither do they support page faulting for large arrays. As Dreyer et al. point out in [41]: FAME "has many useful features, but search and retrieval facilities are very poor". In a paper on managing temporal financial data in extensible database systems [28], it is pointed out that: "storage methods for temporal objects encountered in trading applications is an open problem. Although there have been several proposals in the literature for efficient storage and retrieval of temporal and multi-dimensional data, it is not clear which proposal is the best or whether a completely new approach is required."

Due to the limitation of the current array implementations, some systems choose to implement time series as *relational tables* instead. An example is Oracle's TimeSeries DataCartrige [95]. The reasons for their choice are, as Lory claims in his tutorial "Managing Financial Time Series: Object-Relational and Object Database Systems" [140]: 1) Storing time series as relational tables provides relational access to time series data. On the other hand, arrays are opaque in SQL. 2) Tables support fast incremental loading (this implies that arrays do not support incremental loading well).

We have shown that by developing a dynamic array structure such as the multi-level dynamic array, fast incremental loading is well supported. Also, using the object-relational technology [131], corresponding access methods can be associated with the dynamic array structure. This will make the array access transparent to SQL [131].

On Access Patterns

We found [117] probably the earliest work addressing the issue of data models and physical organization of *time sequences*. Since physical organization of any data is heavily dependent on the expected access patterns, the following assumptions on access patterns on time sequences were made in [117]:

1. "The order of values in a time sequence is important, and thus should be preserved in the physical structure. One needs to minimize the number of disk pages (blocks) read from secondary storage for range queries in the time domain."

2. "We wish to have random access in the time domain. While in some applications one can envision accessing entire time sequences, we believe that efficient

cient access to parts of the sequences is necessary. Thus, some indexing methods on the time domain is necessary."

3. "A secondary index over the data values is not needed in most applications. Such an index can potentially be very expensive in terms of storage, because the number of entries for such an index is in the order of the number of data values. In any case, such an index provides a marginal benefit in situations where the typical access to the data involves restrictions on the time domains. We will assume that such indexes (if absolutely necessary) would use conventional indexing methods."

Our multi-level dynamic array structure satisfies the above requirements 1 and 2 well, i.e., the order of values in the time sequence is physically preserved in every base array, and the index on the time domain is achieved by the index arrays. Meanwhile, we argue that the third assumption is *not* true. The reasons are that, as stated in Section 1.2.2, a secondary index is definitely needed in the value domain for time sequences, and a conventional secondary index is not capable of dealing with value queries (Section 3.5). We have shown that it is possible to develop indexes such as the IP-index where the number of entries in the index can be small even for large time sequences (see Section 3.4 and Section 4.3).

By developing the IP-index and the multi-level dynamic array structure, we have shown that management of time sequences (or time series) can be supported well in a DBMS. Efficient search based on time stamps (or time intervals) is supported by the multi-level dynamic array structure, and efficient search based on value constraints is supported by the IP-index. Dynamic growing of time sequences is also supported by the multi-level dynamic array structure.

7.3 IP-index

In this section we discuss the physical organization of the IP-index. First of all, we have to make it clear the two properties of the IP-index: 1) the IP-index is an ordered index; 2) the IP-index is a secondary index.

To understand why the IP-index is an ordered index is easy. As defined by [137], an ordered index is an index based on a sorted ordering of the values. The IP-index is based on the sorted ordering of the v_is in the TS, thus it is an ordered index.

Why the IP-index is considered as a secondary index is not that obvious. We shall start with the definition of a primary index and a secondary index.

7.3.1 Primary Indexes and Secondary Indexes

There are basically two kinds of ordered indexes. An index that determines the location of the records in a file is called a *primary index* [118]. Generally, a primary index is based on a key for the file, but not necessarily the *primary key* [118]. Thus, a primary index should not be interpreted as an index based on a primary key.

Primary indexes are also called *clustering indexes*, because the key in the primary index determines the clustering of a file. An example of a primary index is shown in Fig. 7.6. There we have a relational table *employee* (*name, age, salary*) stored as a file, clustered on the key filed *name*. Therefore the index based on the field *name* is a primary index.

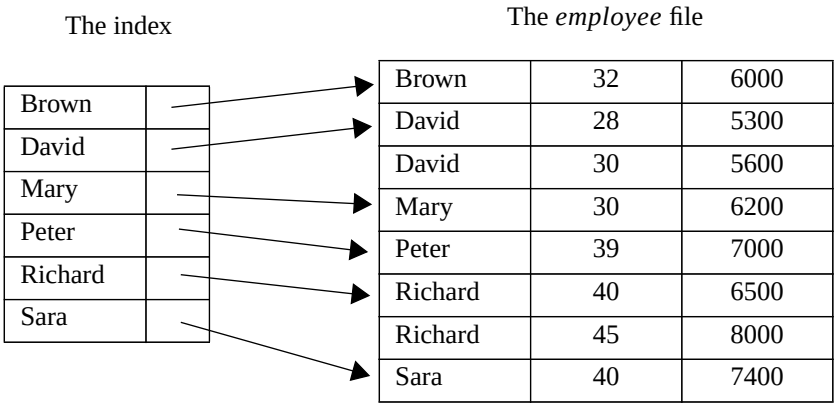


Fig. 7.6: A primary index on *employee* file, on the key field *name*

Given the value *v* of a field other than the key that determines the clustering of the file, to find all records that have value *v* in that field, we would need a *secondary index*. A secondary index is an index that does not determine the location of records in a file, also called a *nonclustering index*. An example of secondary indexes is shown in Fig. 7.7, where the secondary index is built on the field *age* (the file is clustered on the field *name*).

Now we shall see why the IP-index is a secondary index. Since a time sequence TS = (t_i, v_i) is normally clustered by the time stamps t_is, all indexes that are based on the value domain v_is are considered *secondary indexes* (nonclustering indexes) for a time sequence. So is the IP-index.

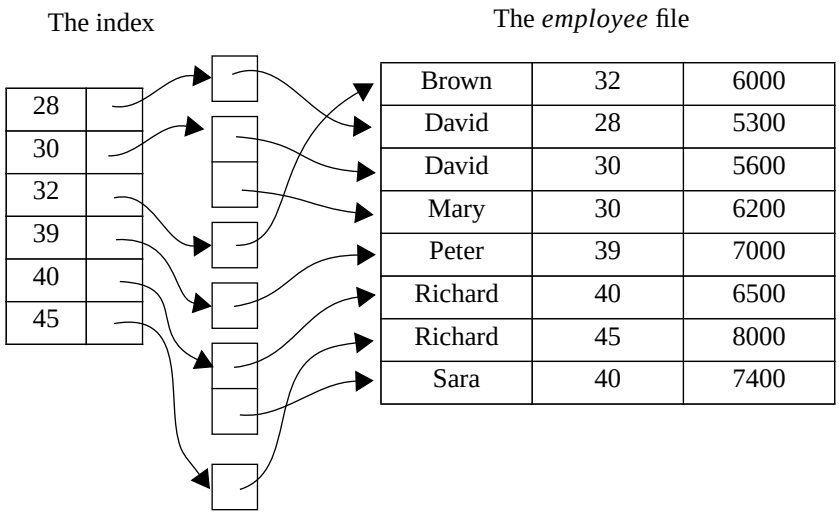


Fig. 7.7: A secondary index on *employee* file, on the field *age*

7.3.2 IP-index as a Secondary Index

A primary index and a secondary index have different structures. As shown in Fig. 7.6, a primary index consists of pairs

(<*key_value*>, <*block_address*>)

to indicate where the (first) record with the key equals to *key_value* resides. A secondary index on a field *F*, on the other hand, consists of

(*v, reference**)

to indicate where the record (or records) whose values in the field *F* equal to *v* reside. The *reference* can be implemented in one of the following two ways:

1. A pointer to the record in question.

2. The key value of the record in question.

In Fig. 7.7 we use the first approach, i.e., *reference** are pointers to the records in question. This approach has the advantage that getting to the intended records is faster than the second approach, i.e., using *key* values. With a key value we have to use the primary index structure to get the record. On the other hand, using key values rather than pointers prevents the records from becoming pinned (A pinned record cannot move freely around the storage space, see

[137]).

In the case of the IP-index, we chose to use the *key* value (i.e., the state_id) of the record as the reference to the record. For example, if A(v') = <S_1, S_6, S_{10}>, then we store the integers 1, 6 and 10 in the IP-index (i.e., we have A(v') = <1, 6, 10>). An alternative could be to store the pointers to the blocks where S_1, S_6 and S_{10} reside. The reason we chose to store the state_ids instead of block addresses is that we would like to have the IP-index *independent* of the physical organization of the time sequence.

How to Implement the Anchor -State Sequences?

Having shown that an anchor-state sequence A(v') in an IP-index is a sequence of integers, let us see how to implement this integer sequence. This is actually the most tricky part of the IP-index implementation. The challenge comes from the fact that for a time sequence, the anchor-state sequences A(v')s can vary much in length for different values v' (see Fig. 10.4 in Section 10.3) The design goal is in two-fold: 1) not to waste space for small A(v')s; 2) to achieve fast random access on large A(v')s (random access of A(v') is needed in the case of time window queries, as shown in Section 5.3). Suppose we have an IP-index as follows:

k_1, 1, 2, 6, k_2, 1, 5, k_3, 6, 7, k_4, 6

The first approach is to *pack* the (k_i, A(k_i)) pairs into blocks in order and then use some index structure (such as a B^+-tree) to index these blocks, as shown in Fig. 7.8. There we assume that six elements of any type fit in one block.

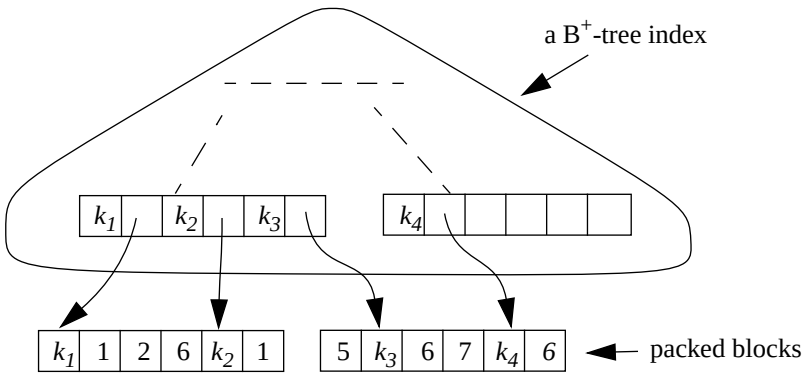


Fig. 7.8: Packed blocks for secondary indexes

This approach has the advantage that the number of blocks used to store the A(k_i)s are minimized (because all A(k_i)s are packed together.) But it is not suitable in the case of dynamic time sequences. For a dynamic time sequence, A(k_i)s will grow dynamically with TS. Therefore it is impossible to pack all A(k_i)s together as in Fig. 7.8.

The second approach to implement the (k_i, A(k_i)) pairs is to use separate storage for every A(k_i), as shown in Fig. 7.9. Every A(k_i) is implemented as a *chain of blocks*. In Fig. 7.9, each of the A(k_i) fits on one block. In general, an A(k_i) could cover many blocks.

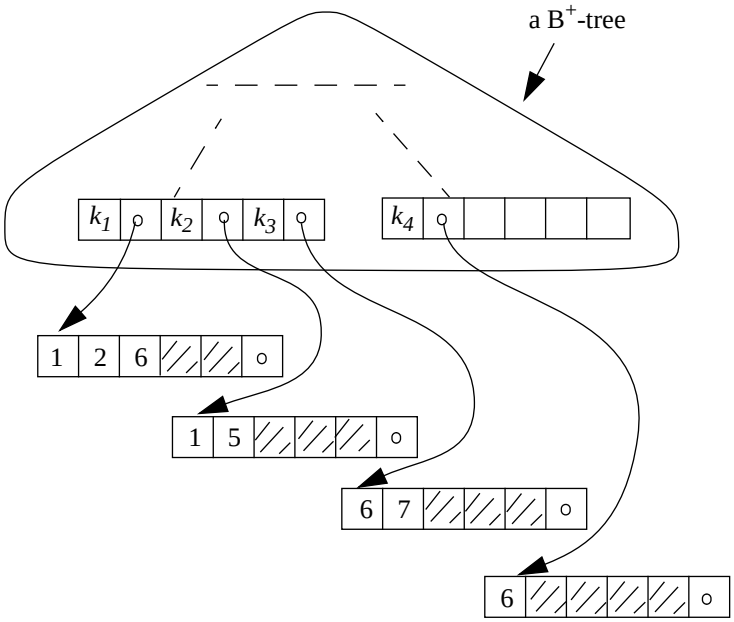


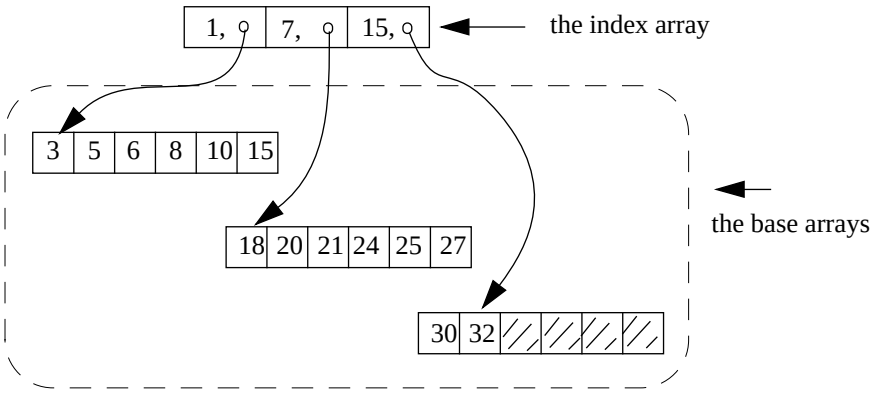
Fig. 7.9: Separate-storage structure for secondary indexes

Recall that we would like to have *fast random access* in A(v') in order to support time window queries (Section 5.3). Therefore we need to build some kind of *index* on these chained blocks. This leads to the idea of using the *multi-level dynamic array structure* (Section 7.2.3) to store A(v') instead. Suppose we have an anchor-state sequence as follows (the integers in the anchor-state sequence represents state_ids:

A(v') = <3, 5, 6, 8, 10, 15, 18, 20, 21, 24, 25, 27, 30, 32>

This anchor-state sequence will be organized as a dynamic array as illustrated

in Fig. 7.10 (suppose six integers fit in one page). There we store the state_ids in the base arrays and use an index array to index the base arrays by recording the starting state_id of every base array.



A(v') = <3, 5, 6, 8, 10, 15, 18, 20, 21, 24, 25, 27, 30, 32>

Fig. 7.10: A two-level dynamic array for an anchor -state sequence A(v')

It can be seen that implementing anchor-state sequences as multi-level dynamic arrays has two advantages: 1) It does not waste space for small A(v')s; 2) It guarantees fast random access for large A(v')s.

7.4 Storage Management for Large Objects

The reason why we investigate storage management for *large objects* is that many application data result in large objects in DBMSs, such as time series, image and video data. Among them, we are particularly interested in large *sequence data*. Most commercial database systems allow a sequence to be represented as a 'BLOB' (binary large object) [131]. Since a BLOB is treated as an uninterpreted sequence of bytes, no data structure is supported. For example, there is no support for operations such as `get_element(i), get_next()` as that for the `sequence` structure in SHORE (see the Appendix). In other words, it is up to the application program to interpret the structure of the BLOB. Some object-oriented systems like O2 [14] provide array and list constructs but do not support query languages over them.

In this section, we will give an overview of how large objects are managed in various systems, including relational DBMSs and object-oriented DBMSs.

7.4.1 In Relational DBMSs

Physically, a database is a collection of records stored in a file. For example, in relational DBMSs, a *tuple* in a relational table is stored as a *record*, and an *attribute* in the tuple is stored as a *field* of the corresponding record.

In relational DBMSs, records are composed of fields which are normally constrained to 255 bytes. Larger fields present problems to the database record manager, so they must be managed separately. Usually the work is done by the *long field manager* (the term l*ong fields* is used to refer to *large objects* in the relational world).

The first SQL relational database system, System R [12], managed long fields as a *linked list* of records, each 255 bytes in length. Operations were restricted to reading and writing *entire* long fields; partial reads or updates were not supported. The maximum length of a long field in System R was restricted to 32,767 bytes.

Later, an extension to SQL was proposed that provided operators for manipulating long fields [62]. A new interface, the long field *cursor*, provided the ability for partial reading and updating of long fields. A storage mechanism was proposed that stored long fields as a sequence of 4K data pages (rather than the previous scheme of a linked list of 255 byte records). The maximum length of a long field in extended SQL was about 2 gigabytes. This approach, compared to the one in system R, has the improvement that partial read/write is supported for long fields (although it is slow).

The Wisconsin Storage System (Wiss) [30] used a similar mechanism for storing long fields. A Wiss long field was split into 4K pages, called *slices*. To reduce internal fragmentation, a *crumb*, a partially filled slice managed similarly to a database record, was used to hold the last segment of a long field if it did not occupy a full slice. A long field was represented by a directory of slices, plus a crumb. Wiss long fields had a size limit of 1.6 megabytes. Compared to the previous approach, it has the advantage that space utilization is better due to the introduction of *crumbs*. Partial read/write is supported by a mechanism similar to a cursor, therefore it is still slow.

Starburst [103] is an extensible relational database system developed at IBM Almaden Research Center that supports extensions of data types and procedures. The Starburst long field manager [74] aims at managing those large objects that appear in modern applications such as voice, image and video data. The size of large objects is expected to be in the order of 100 megabytes. The design goal is to *minimize disk seeks* in disk I/O, and optimize time and space in *allocating* and *deallocating* a long field. The approach chosen was the *buddy system* [70] that was taken from some file systems (such as the Dartmouth Time

Sharing System [71]). It is characterized by the use of continuous disk space (termed disk *extents*, which can be much larger than the size of a disk page) to allocate large objects. In this way *disk seeks* are minimized when read/write a large object. This approach has the limitation that it only supports trimming/appending *at the end of* a large object, update in the *middle* of an object is difficult to support (due to the use of disk extents). However, Schwarz et al. [103] claim that applications such as those involving voice, image, sound, or video, will normally require read/write the *entire* long fields, and partial updates are normally only needed at the end of the long field (i.e., *trim* or *append*).

7.4.2 In Object-Oriented DBMSs

In object-oriented DBMSs, large objects are managed more gracefully than in relational DBMSs. This is perhaps due to the different data models of these two kinds of systems. For example, in OODBMSs, there is no such limit as the length of a *field* as in the relational DBMS case.

In the OODBMS world, the most successful system in dealing with large objects is probably EXODUS [25], which supports fast insertion, deletion, and retrieval *at any position* for a large object. EXODUS later evolved into the distributed object system SHORE [22]. In what follows we describe the data structure for large objects in this system.

Conceptually, a large object in EXODUS is an uninterpreted sequence of bytes; physically, it is represented on disk as a B^+-tree index on byte positions within the object, plus a collection of leaf (data) blocks. Fig. 7.11 shows an example of a large object in EXODUS. The root of the tree (the object header) contains a number of (count, page #) pairs, one for each child of the root. The count value associated with each child pointer gives the maximum byte number stored in the subtree rooted at that child; the count for the rightmost child pointer is therefore also the size of the object. Internal nodes are similar, being recursively defined as the root of another object contained within its parent node. Thus, an absolute byte offset within a child translates to a relative offset within its parent node. The left child of the root in Fig. 7.11 contains bytes 1-421, and the right child contains the rest of the object (bytes 422-786). The rightmost leaf node in the figure contains 173 bytes of data. Byte 100 within this leaf node is byte 192 + 100 = 292 within the right child of the root, and it is byte 421 + 292 = 713 within the object as a whole.

The leaf blocks in a large storage object contain pure data — no control information is required since the parent of a leaf contains the byte counts for each of its children. The size of a leaf block is a parameter of the data structure, and it is an integral number of contiguous disk pages. For often-updated objects, leaf blocks can consist of several contiguous pages to lower the I/O cost of scanning

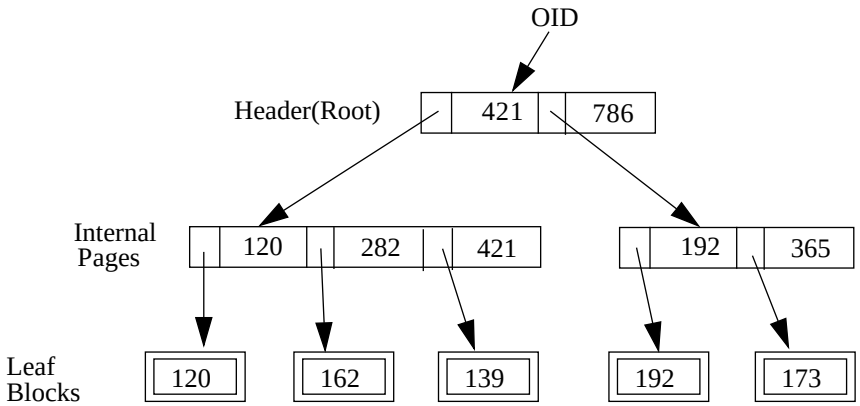


Fig. 7.11: An example of a large object on disk

long sequences of bytes within such objects. As in B^+-trees, leaf blocks are allowed to vary from half full to completely full.

Associated with this large object storage structure are algorithms to *search* for a range of bytes, to *insert* a sequence of bytes at a given position in the object, to *append* a sequence of bytes at the end of the object, and to *delete* a sequence of bytes from a given position in the object. These algorithms [23] are designed to achieve best-case storage utilization. According to [23], leaf blocks in this storage structure are at least 80% full, and internal nodes are at least 50% full. (To be more precise, the algorithms guarantee all but the last two leaf blocks to be completely full. This is important for space utilization since large objects are often created dynamically.) The reason why they pay more attention to the utilization of leaf nodes than internal nodes is that internal node utilization is not as critical as leaf node utilization because of the large fanout of internal nodes.

Table 7.2 shows examples of the approximate object size ranges that can be supported by trees of height two and three, assuming two different leaf block sizes. The table assumes 4k-byte disk pages, 4-byte pointers, and 4-byte counts, so the internal pages contain between 255 and 511 (count, pointer) pairs. It can be seen that two or three levels will suffice for most large objects.

Compare the EXODUS large object storage structure with our multi-level dynamic array structure: the main difference is that the EXODUS structure supports update in the *middle* of a large object, while our structure assumes append-only (sequentially growing) data. The flexibility of the EXODUS data structure is obtained by paying the price for a more complex insertion algo-

Table 7.2: Some examples of object sizes

No. of Tree Levels	Leaf Block Size	Object Size Range
	1	8KB - 2MB
2	4	32KB - 8MB
	1	2MB - 1GB
3	4	8MB - 4GB

rithm (taking care of node balancing, etc.).

The conclusion from the above discussions on large object management is that the design of data structures for large objects should be based on the expected operation patterns such as random/sequential access, delete/update frequency, the granularity of read/write, etc. Trade-offs exist between space utilization and complexity of insertion, search algorithms.

7.5 Main Memory DBMSs versus Disk-Resident DBMSs

The implementation of the IP-index in both a main memory database system (Section 4.1) and a disk-resident database system (Section 4.2) made it interesting to compare how these two kinds of database systems differ. In this section we investigate how these differences affect physical implementation issues such as *index design* and *data structures*, especially for sequence data.

7.5.1 Background

As pointed out in Chapter 7.1, a *main memory database system* (MMDB) is a database system where data reside *permanently* in main memory; while a *disk-resident database system* (DRDB) is a database system where data reside permanently on disk. Most commercial DBMSs are disk-resident since disks are more stable and have more capacity than main memory. However, as main memory becomes more stable (and cheaper also) and the capacity of main-memory becomes larger, MMDBs are becoming more and more popular nowadays. Examples of MMDBs are OBE [10], MM-DBMS [76] and AMOS [49].

Why it is so crucial to distinguish between MMDBs and DRDBs? The reason is that main memory and disks have different properties that have profound implications on the design and performance of a database system. According to [56],

the most crucial differences are:

- Retrieving disk-based data is often block-based. On the contrary, main-memory access is record-based.
- The access time of main-memory is in orders of magnitude less than that for disk storage.
- The layout of data on a disk is much more critical than the layout of data in main memory, since *sequential* access of a disk is much faster than random access. In other words, sequential access (clustering) is a critical issue for disk-resident data, but not for main-memory resident data.
- Main memory is normally volatile, while disk storage is more stable.

In the following sections we shall discuss the impact of these differences on physical organization issues such as index design and data structures.

7.5.2 Impact on Index Design

Index structures designed for main memory are different from those designed for disk-based systems. The primary goals of a disk-oriented index structure are to minimize the number of *disk accesses* and disk storage. In contrast, the primary goals of a main memory index structure are to minimize *processing time* (CPU time) and main memory space usage.

Main-Memory Index Structures

A wide variety of index structures have been proposed for main memory databases [37][75][138]. These include arrays, various forms of hashing and trees (such as AVL-trees, B-trees and T-trees). The main limitation of hashing, compared to trees, is that it does not support range queries. We will not discuss hashing here since it is not relevant to our IP-index.

Arrays are used as index structures in IBM's OBE project [10]. The advantage is that they use minimum space, providing that the size is known in advance. The drawback is that data movement is *O(N)* for each update, so it appears to be only useful for a read-only or non-update (e.g., time sequences) environment. Using arrays to store time sequences TS = (t_i, v_i) indirectly provides an index on the time domain since binary search can be performed on the array to find any element given the time stamp t.

AVL-trees [2] (Fig. 7.12) were used as index structures in the AT&T Bell Laboratory's Silicon Database Machine [77]. The AVL-tree was designed as an internal memory data structure because it uses a binary search structure.

Searching an AVL-tree is very fast in main memory since the binary search is intrinsic to the tree structure (no arithmetic calculations are needed, as in the array case). Updates always affect a leaf node and may result in an unbalanced tree, so the tree is kept balanced by rotation operations. One disadvantage of the AVL-tree is its poor space utilization, because each tree node holds only *one* data item but *two* pointers (the left child pointer and the right child pointer). The ratio of pointer to data is large compared to multi-way tree structures.

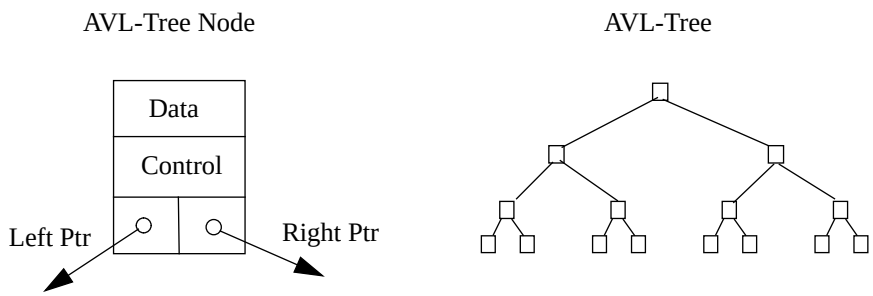


Fig. 7.12: AVL-tree index structure

The **B-tree** and its variations [35] were originally designed for disk-resident database systems, because the primary goal of this index structure is to minimize the number of blocks (pages) accessed. However, the structure of a B-tree can also be used in main memory databases to index data (see Fig. 7.13). A B-tree is a multi-way balanced tree. Compared to an AVL-tree, a main-memory B-tree is better in storage utilization because the pointer to data ratio is small, as leaf nodes hold only data items and they comprise a large percentage of the tree. However, searching and update will not be as efficient as an AVL-tree [75].

The **T-tree** [75] (Fig. 7.14) was introduced in 1986 as a data structure evolved from AVL-trees and B-trees. The T tree is a binary tree with many elements in a node. Since the T-tree is a binary tree, it retains the intrinsic binary search nature of the AVL-tree. Also, because a node in the T-tree contains many elements, the T-tree has the storage efficiency of the B-tree. Data movement is required for insertion and deletion, but it is usually needed only within a single node. Rebalancing is done using rotations similar to those of the AVL-tree, but it is done much less often than in an AVL-tree due to the possibility of intra-node data movement. According to [75], the T-tree over-performs both the B-tree and AVL-tree.

In our main-memory implementation of the IP-index, we chose to use the AVL-tree since the AVL-tree has a small rebalancing time.

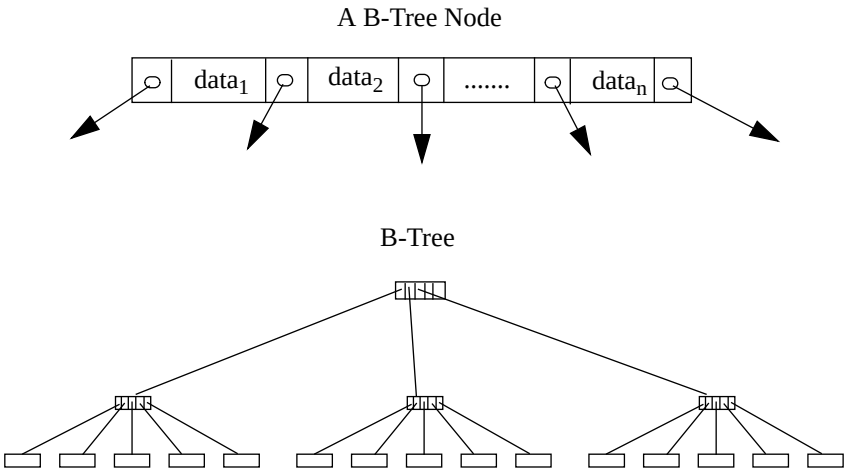


Fig. 7.13: B-tree index structure

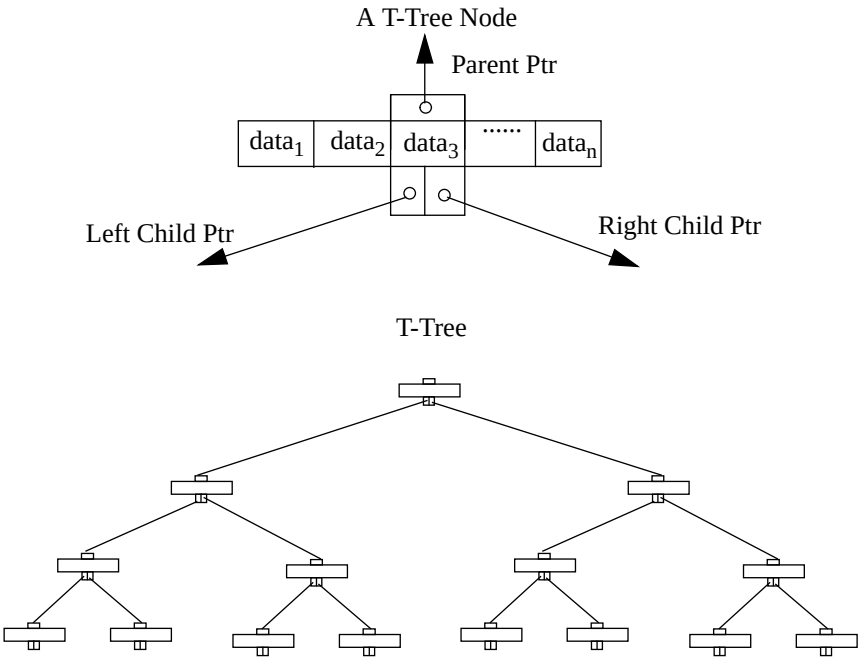


Fig. 7.14: T-Tree index structure

Disk-Based Index Structures

Disk-based index structures are optimized to minimize the number of disk blocks accessed and disk space utilization. Compared to main memory tree structures such as AVL-trees, they are characterized as *short, bushy* structures and require *few* node accesses to retrieve a value.

The most popular tree structure in a disk-resident database system is the **B-tree** and its variants [35]. (Actually the original publication on B-trees is [17], although it is not frequently referenced). The structure of the B-tree looks the same as the main-memory version in Fig. 7.13, except that every node is allocated in a separate disk page (block). Therefore, the *pointers* in the B-tree nodes are *block addresses* (logical or physical) instead of main memory addresses as in Fig. 7.13.

The reason why the B-tree is well suited for disk-based database systems is that the *depth* (the maximum levels of nodes) of a B-tree is much smaller than that of an AVL-tree or a T-tree for the same set of data to be indexed. This indicates that there are less disk I/Os when a B-tree is used to access data.

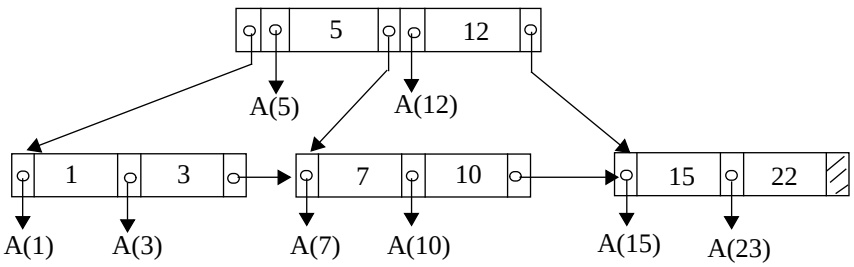


Fig. 7.15: B-tree implementation of the example IP-index

Given an example IP-index with keys $\{k_i \mid i = 1, 2...8\}$ = <1, 3, 5, 7, 10, 12, 15, 22> and the corresponding A(k_i)s, the B-tree implementation of this IP-index is shown in Fig. 7.15 (where we assume the *fanout* - the maximum number of keys in each node is 2). Notice that the implementation of A(k_i)s is not shown in Fig. 7.15 (it was discussed in Section 7.3.2), we are only concerned with the index tree structure here. It can be seen from Fig. 7.15 that the characteristics of a B-tree are: 1) both internal nodes and leaf nodes store pointers to A(k_i)s; 2) every key value k_i appears only *once* in the B-tree. (Note that normally the nodes in a B-tree are only more than 50% full, space utilization is not as good as shown in the figure.)

Most database systems use a variant of the B tree, the **B^+-tree** instead. Com-

pared to a B-tree, a B^+-tree keeps all of the actual data in the leaves of the tree. Thus internal nodes are only used to *index* these data. For example, the B^+-tree implementation of the same IP-index is shown in Fig. 7.16, where only leaf nodes store pointers to the anchor-state sequences. In this way internal nodes can store more elements than the B-tree implementation (Fig. 7.15). Since all pointers to A(v')s are stored in leaf nodes, rebalancing the tree (which is needed in the process of *insertion* or *deletion*) is easier than that of a B-tree. (Note that normally B^+-tree nodes are only more than 50% full, space utilization is not as good as shown in the figure.)

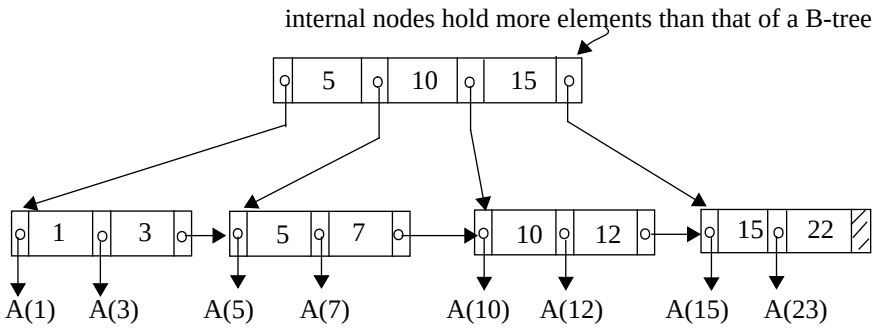


Fig. 7.16: B^+-tree implementation of the example IP-index

The advantage of the B^+-tree structure over the B-tree is that the B^+-tree is easy to maintain (i.e., the insertions and deletions are more efficient). Moreover, since the fanout of a B^+-tree node is larger than that of a B-tree node, the depth of a B^+-tree is smaller than that of a B-tree. This leads to better search time in a B^+-tree than a B-tree. The slight disadvantage of the B^+-tree is its space inefficiency, because in a B-tree every key value appears only once, while in a B^+-tree some key values appear in both non-leaf nodes and leaf nodes. But the space advantage of B-trees is marginal for large indexes, and usually does not overweigh the disadvantages that we have discussed. Thus, the structural simplicity of a B^+-tree is preferred by many database system implementors.

In the case of main memory DBMSs, however, the B-tree is preferable to the B^+-tree. This is because, in main memory, there is no advantage in keeping all of the data pointers in leaves — it only wastes space.

In our disk implementation of the IP-index, we chose to use a B^+-tree instead of a B-tree for the above reasons.

7.5.3 Impact on Data Structures

Data structures designed for main memory DBMSs are different from those designed for disk-based DBMSs. A data structure that exhibits good performance in main memory may turn out to be very inefficient when implemented on disks. For example, an array is a good data structure for storing sequence data in main memory, because random access is very efficient. For an array of size *n*, the access of any element takes *O(1)* time. This is the reason why we chose to implement a time sequence as an array in our main-memory implementation (see Section 4.1). However, when a large array is stored on disk, the situation is different. A large array stored on disk normally has to be partitioned into several disk pages. To access one element, we have to find the right page that contains the element, and read it in main memory. If *chained* pages are used to store the array (such as the "linked list" approach in Section 7.2.4), then a scan of this chain is needed in order to find the right page. The access time will be *O(N)*, where *N* is the number of pages allocated to store the array. If we build some *index* for the pages such as the *index arrays* for the multi-level dynamic array structure, then the access time will be determined by the level of the index arrays. In any case the access time will not be *O(1)* any more. In our disk implementation of time sequences, we proposed the multi-level dynamic array structure. We believe that this data structure is an optimal solution for storing dynamic, irregular time sequences on disk.

Another complication is *pointers* which affect data structures such as a *list*, or *references* between objects. In our IP-index structure, pointers (or references) play an important role. There are pointers in the IP-index which associate each key k_i in the index tree with the corresponding anchor-state sequence A(k_i), and each element in A(k_i) is a *reference* to a state S_i in the TS (see Section 7.3.2). Implementation of these pointers (references) is thus an important issue. A pointer in a main-memory structure is easier — it is simply a main-memory address. A pointer in a disk-based structure is more complicated. It can be a *logical* or *physical* [137] block address. Using the physical address on disk will make the object pinned [137], but is faster in access compared to the logical address. In our disk implementation of A(k_i)s (see Section 7.3.2), we chose to use the logical address (state_id) instead of physical address of S_i. The reason is that we would like to have the implementations of the IP-index and time sequences independent of each other.

Another relevant issue is *swizzling* [26]. When a disk data structure is brought into main-memory, disk addresses have to be transformed into main-memory addresses.

7.6 Is the IP-index Practical for Large Time Sequences?

After the discussion on physical organization in this chapter, we are able to answer several questions which concern the practical application of the IP-index.

Question 1:

Since TS can be very long, the IP-index can grow very large. Is it practical to build the IP-index for *large* time sequences?

Answer:

We have proved in Chapter 4 that it is not necessarily true that a large TS results in a large IP-index tree. The reason is that most real-life TSs appear periodic and there is a range *R* and precision *P* in the value dimension, which result in the total number of keys k_is in the IP-index being less than *R/P*. By lowering the precision we can get an even smaller index. Therefore, it is practical to build the IP-index for large time sequences.

Question 2:

The anchor-state sequence can grow very long which makes the IP-index take lots of space. Does it make the IP-index impractical?

Answer:

It is true that the anchor-state sequence can grow very long. But this does not imply that the IP-index is not practical. There are two reasons for this: 1) A long anchor-state sequence just indicates that the size of the answer set of $F^{-1}(v')$ is large (Section 5.1). There is nothing we can do about it. However, techniques such as stream processing (Section 6.2.2) can be used to prevent materializing the whole answer set. 2) The space to store the anchor-state sequences is independent of the space to store the index tree, which means the index tree can still be small and fetching A(v')s can still be fast. (Notice that only in the case of time window queries do we need to search A(v').) 3) We have shown in Section 4.3 that the total space overhead of the IP-index for the pressure sequence (Fig. 1.4) is rather small.

Question 3:

In the case of *many* time sequences, is it true that building an IP-index for *every* time sequence will take too much space?

Answer:

It is true that it takes space to build an IP-index for each TS. But, as in the case of any kind of indexes, having an index or not is always a trade-off between time and space. Consider a database that contains many relational tables, building an index for every table will take much space. One may argue that a relational table does not grow as quickly as a TS. The point is: the size of the IP-index tree is not proportional to the size of the TS, instead it is proportional to the number of distinct v_is, which is *R/P* as explained in the answer to Question 1 above. This indicates that a long TS does not necessarily result in a large IP-index tree. Furthermore, the space to store A(v')s is very compact as explained in Section 7.3.2. Therefore, it is still practical to build IP-indexes for many TSs.

Question 4:

In the case of *many* time sequences, is it too time consuming to build an IP-index for *every* time sequence?

Answer:

We have measured the insertion time of the IP-index in both main-memory and disk implementations (see Chapter 4). The experiments show that to build an IP-index is very fast even for large time sequences. For limited precision of v_is in TS, the insertion time can stay almost constant regardless of the growing of the TS. Therefore, it is practical to build the IP-indexes for many time sequences.

7.7 Summary

In this chapter we investigated physical organization issues. We are mostly interested in physical data structures for *dynamic, irregular* time sequences. The challenge is to support both *fast appending* and *efficient random access*. We developed a multi-level dynamic array structure which meets this challenge. Related work was compared with this data structure.

Physical structures of secondary indexes were also discussed. In the case of the IP-index (which is a secondary index), the challenge is how to store the *anchor-state sequences*, because they are dynamic and vary widely in length. The multi-level dynamic array structure is also a good choice for storing anchor-state sequences. It does not waste space for short A(v')s and guarantees fast random access for large A(v')s.

Other relevant issues such as storage management of large objects, the impact of main-memory or disk-resident DBMSs on data structures were also investigated.

Chapter 8

Query Optimization

Query optimization for sequence data is an important issue since sequences are usually very long. In this chapter, several optimization techniques for *value queries* are discussed. First, we show that the technique of *stream processing* can be used in processing the operator $\sigma^*_{v=v'}(TS)$ when card(A(v')) is large. Secondly, we investigate the *cost models* and *selectivity factors* [107] for selections such as $\sigma^*_{v=v'}(TS)$ and $\sigma^*_{v>v'}(TS)$. These information can be used to optimize complex value queries (sequence joins) by choosing a good join order. Optimizations of *time window queries* are also investigated and verified by experiments.

8.1 Stream Processing

Stream processing is an important technique in sequence query optimization since sequences are usually very long. Stream processing is used in SEQ [110] to optimize sequence queries: "each sequence is read in a single continuous pipelined stream without materializing it." This is accomplished by associating buffers with each operator, to cache some relevant portion of the most recent data from its inputs. In this section we shall see how stream processing can be used to process the operator $\sigma^*_{v=v'}(TS)$.

As pointed out in Section 6.2.2, $\sigma^*_{v=v'}(TS)$ is implemented by the sequential execution of the two operators: 1) $IP_{v=v'}(TS)$, and 2) ifn^{-1} (Fig. 8.1). The $IP_{v=v'}(TS)$ operator returns the anchor-state sequence of v', A(v'), and the inverse interpolation function ifn^{-1} is applied to each state in A(v'). Since the cardinality of A(v') can be large, to materialize the result of the operator

$IP_{v=v'}(TS)$ (i.e., to materialize the sequence A(v')) can be time consuming. Instead, we propose to implement the operator $IP_{v=v'}(TS)$ as a *stream* where the *next* element of $IP_{v=v'}(TS)$ is implemented by retrieving the *next* state in A(v'). Therefore, the $\sigma^*_{v=v'}(TS)$ can be implemented as a stream as well: the *next* state of the $\sigma^*_{v=v'}(TS)$ is generated by applying ifn^{-1} over the *next* state returned from $IP_{v=v'}(TS)$.

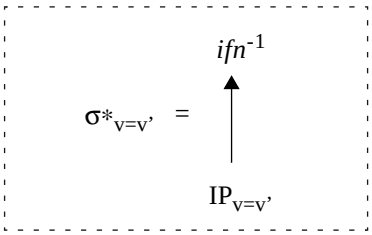


Fig. 8.1: Streaming processing of the σ^* operator and the IP operator

Notice that the stream of $\sigma^*_{v=v'}(TS)$ and $IP_{v=v'}(TS)$ can be generated in the *reverse* order as well, i.e., the states with newer time stamps come out first. This is useful in many applications since newer states are usually more interesting than older ones.

Another benefit of stream processing is that we are able to generate the *first few answers* [16] quickly. To generate the first few answers, the interpolation function ifn^{-1} is applied to only the first few states in A(v'). In particular, *the first answer* of $\sigma^*_{v=v'}(TS)$ can be generated quickly since the first state_id in A(v') denotes the position in the TS to apply ifn^{-1}. The benefit of this approach has been demonstrated by experiments in Section 6.3.2.

8.2 The Cost Model of $\sigma^*_{v=v'}(TS)$

Since time sequences can be very long, it is important to be able to estimate the cost of a value query $\sigma^*_{v=v'}(TS)$. As pointed out in the last section, $\sigma^*_{v=v'}(TS)$ is implemented by stream processing of the two operators: 1) $IP_{v=v'}(TS)$, and 2) ifn^{-1} (Fig. 8.1) — the $IP_{v=v'}(TS)$ operator returns A(v'), and ifn^{-1} is applied to every state in A(v'). One could predict that the cost of $\sigma^*_{v=v'}(TS)$ will be determined by the cardinality of A(v'). The longer A(v') is, the more time it will take to process $\sigma^*_{v=v'}(TS)$ since the answer set will be larger. Is this a valid prediction? Let us see the experimental results.

We have performed experiments on SHORE to investigate the cost of $\sigma^*_{v=v'}(TS)$ for different time sequences and different values of v'. The IP-index was implemented on top of a B^+-tree. Anchor-state sequences were implemented as

SHORE large objects which can grow arbitrary large. Time sequences were implemented as arrays of records (t_i, v_i). All measurements were done on a SPARC 20 machine with 64M main memory. The SHORE buffer pool size was set to 40 8K pages.

Both a synthetic and real-life time sequence were used in the measurements. The synthetic time sequence was v(i) = *m*(i) * *sin*(*k* * i) (i = 1, 2...10K) (introduced in Section 6.3.1) as shown in Fig. 8.2. The reason for constructing this sequence was to easily control the length of A(v') by varying the values of v', as explained in Section 6.3.1. The real-life time sequence was the pressure sequence introduced in Section 1.2.2, as shown in Fig. 8.3.

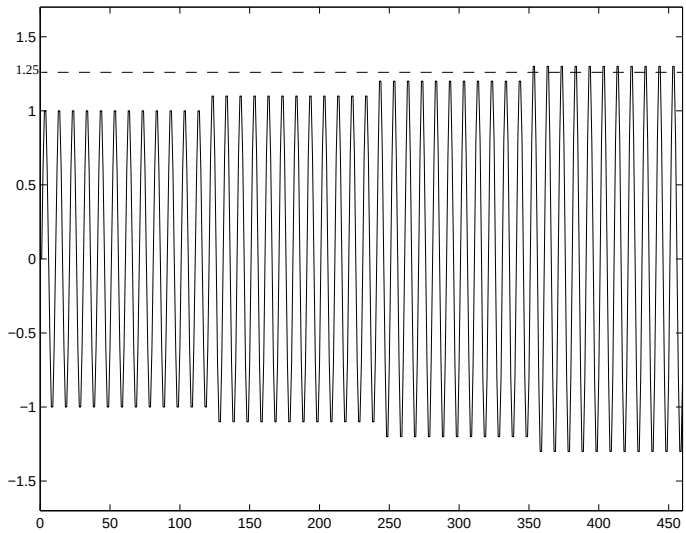


Fig. 8.2: The synthetic sine sequence

8.2.1 The Linear Case

To measure the relationship between the execution time of $\sigma*_{v=v'}(TS)$ and the cardinality of A(v'), we chose different values of v' in the synthetic sine sequence (Fig. 8.2) with varying card(A(v'))s. The selected v's and the corresponding card(A(v'))s are listed in Table 8.1. The execution times of $\sigma*_{v=v'}(TS)$ with regard to card(A(v'))s are shown in Fig. 8.4. It verifies our "linear" speculation above. It shows that the execution time of $\sigma*_{v=v'}(TS)$ by using the IP-index is linear to card(A(v')).

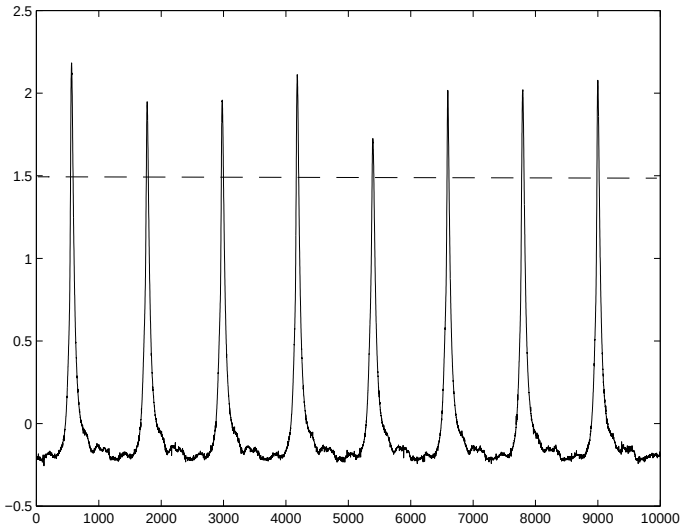


Fig. 8.3: The pressure sequence

Table 8.1: Selected v's and the cardinalities of A(v')s

v'	9.4	9.2	9	8.4	7.3	4.9	3.0	0
cardinality	14	60	106	246	504	1064	1508	2000

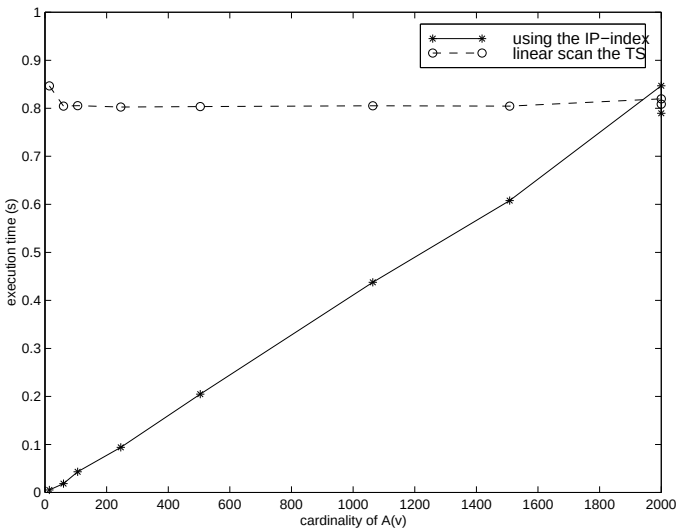


Fig. 8.4: The execution times of $\sigma^*_{V=V'}(TS)$

8.2.2 The Non-Linear Case

We also performed experiments on the real-life time sequence as shown in Fig. 8.3. The cardinality of the sequence as 100K and the precision of values was 10^{-2}. The selected v's and the corresponding cardinalities of A(v')s are listed partly in Table 8.2 and partly in Table 8.3.

Table 8.2: Selected v's and the cardinalities of A(v')s (part 1)

v'	-0.26	-0.25	-0.24	0.03	-0.23	-0.13
cardinality	4	30	92	284	504	758

Table 8.3: Selected v's and the cardinalities of A(v')s (part 2)

v'	-0.14	-0.18	-0.22	-0.21	-0.2
cardinality	1590	2081	2939	4357	4945

The execution times of $\sigma^*_{v=v'}(TS)$ with regard to card(A(v'))s are shown in Fig. 8.5. It is not a clean "linear" curve any more. What is surprising is that the execution time of $\sigma^*_{v=v'}(TS)$ for shorter A(v')s can be bigger than the execution time of $\sigma^*_{v=v'}(TS)$ for longer A(v')s. This leads to the *cost model* of $\sigma^*_{v=v'}(TS)$ in the next section.

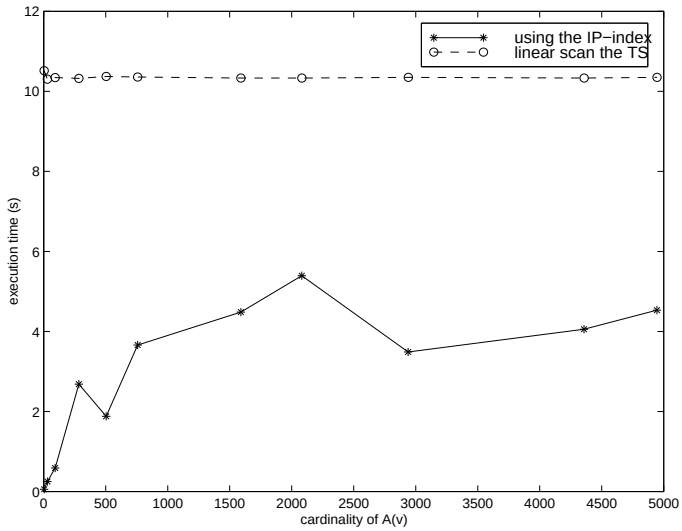


Fig. 8.5: The execution times of $\sigma^*_{v=v'}(TS)$ for the pressure sequence

8.2.3 Cost Model

After investigating the reasons why Fig. 8.4 and Fig. 8.5 look so different, we came to the conclusion that the nice "linear" property in Fig. 8.4 is only valid when the states in A(v') tend to reside in the same page as the sine sequence does. Fig. 8.6 illustrates this. Suppose that the portion of the sine sequence in Fig. 8.6 (defined over the time interval [0, 460]) occupies 4 pages, then all the states in A(1.25) will reside in the same page (the last page). And all states in A(1.20) will reside in two pages. In this case the number of pages visited is linear to the cardinality of A(v'). In reality most time sequences do not have this nice property. States in A(v') are "scattered" in different pages instead of clustered together. For example, states in A(1.5) in the pressure sequence (Fig. 8.3) are scattered instead of clustered. In this case the execution time of $\sigma^*_{v=v'}(TS)$ using the IP-index will not be linear to the cardinality of A(v'), instead it will be *linear to the number of disk pages visited.*

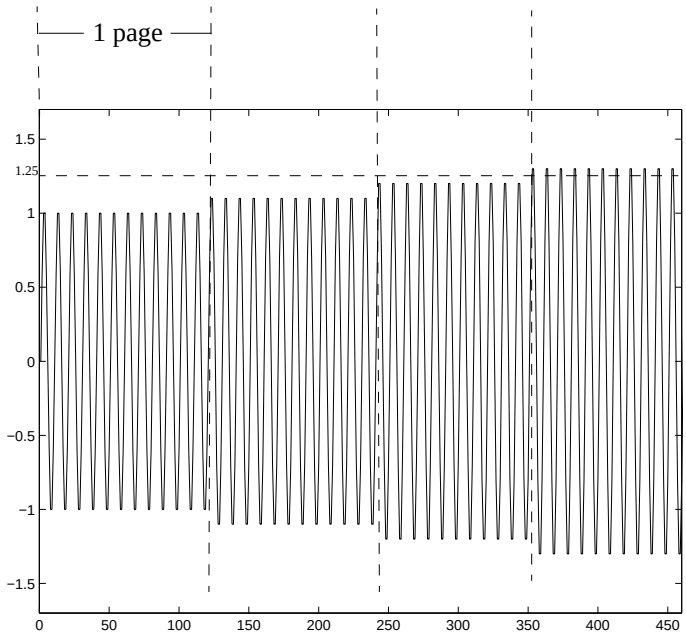


Fig. 8.6: The page division of a portion of the sine sequence

This indicates that to estimate the cost of $\sigma^*_{v=v'}(TS)$ using the IP-index, we need to have knowledge of the distributions of those S_is in A(v') in addition to the cardinality of A(v'). The statistics on distributions of S_is in A(v') can be maintained as meta-data in DBMSs (the cardinalities of A(v')s are stored in the IP-index). In the worst case we have to assume every S_i in A(v') resides in a different disk page.

8.3 Cardinalities of Range Queries

The *selectivity factor* [107] plays an important role in traditional query optimization. In the classical paper "Access Path Selection in a Relational Database Management System" [107], the cost of an execution plan is estimated as a weighted sum of I/O pages fetched and CPU time. In the case of using an index scan [107], the selectivity factor of a predicate can be used to estimate the number of I/O pages fetched. A selectivity factor of a predicate, *F(pred)*, is defined as "the expected fraction of tuples which will satisfy the predicate". For example, if we have a query such as "find the employees whose names are 'John'", then, using a clustered index on the attribute `name` on the `employee` relation, the cost of executing this query would be

```
F(pred) * (NINDX(I) + TCARD) + W * RSICARD
```

where `F(pred)` is the selectivity factor of the predicate `name = 'John'`, `(NINDX(I) + TCARD)` is the total number of index pages and data pages that hold all tuples in the relation, and `RSICARD` is the estimated CPU time. The number of index pages and data pages for the `employee` relation is stored as meta-data in the DBMS. The selectivity factor, `F(pred)`, can therefore be calculated by using those statistics such as the number of distinct keys in the index I and the number of index pages [107].

Therefore, selectivity factors play an important role in estimating the cost of a query when index scan is used. This is because the selectivity factor indicates how many disk pages (including index pages and data pages) will need to be fetched (see the above formula). (In the case where no index is available, the cost of scanning the entire relation will be the same as the number `TCARD` above, no *selectivity factor* will be involved any more.)

In the case of a 2-way join (a join involves two relations), the cost of the join is estimated by the cost of scans on each relation (could be *index scans* or *segment scans*, see [107]) and the cardinalities of the results of scans. For example, if a *nested loop* join method is used, then the cost is estimated by

```
C(path1, path2) = C_outer(path1) + N * C_inner(path2)
```

where `C_outer(path1)` is the cost of scanning the outer relation [107] via path 1, `C-inner(path2)` is the cost of scanning the inner relation [107] via path 2, and `N` is the estimated cardinality of the tuples in the outer relation that satisfy the join predicate [107]. The number `N` is calculated by the formula `F(pred) * NCARD(T)` (where `NCARD(T)` is the cardinality of the outer relation, stored as statistics in the DBMS). Therefore, aside from affecting the cost of an index scan, the selectivity factor `F(pred)` is an important factor in choosing a good

join order as well.

It is interesting to consider the selectivity factors for value queries. For example, what would the cardinalities of the results of $\sigma^*_{v=v'}$(TS) and $\sigma^*_{v>v'}$(TS) be? In the case of $\sigma^*_{v=v'}$(TS) on time sequences, the estimated cardinality of the answer set will be the same as the cardinality of A(v') (recall that card(A(v')) is stored in the IP-index, see Section 3.2.1). The reason is that every answer of $\sigma^*_{v=v'}$(TS) is produced by applying ifn^{-1} on an anchor-state of v', see Fig. 8.1. However, for a range query $\sigma^*_{v>v'}$(TS), there are two kinds of cardinalities of the result set. When the sequence TS is viewed as continuous, the cardinality of $\sigma^*_{v>v'}$(TS) is the number of time intervals returned. When TS is viewed as discrete, the cardinality of $\sigma^*_{v>v'}$(TS) is the number of states inside those time intervals. These will be further illustrated in the following paragraphs.

For a continuous TS, the results of $\sigma^*_{v>v'}$(TS) will be a sequence of time intervals. These time intervals (when the values are greater than v') can be computed by the algorithm of $F^{-1}(v>v')$ in Fig. 5.5. Since the *time intervals* of $F^{-1}(v>v')$ is composed by grouping the *time points* of $F^{-1}(v')$ into corresponding intervals, the cardinality of $\sigma^*_{v>v'}$(TS), i.e., the number of time intervals returned, is determined by the cardinality of A(v'). For example, in the following time sequence which represents the temperature reading of a patient in a hospital, we pose the query:

- When did the patient have a temperature higher than 38?

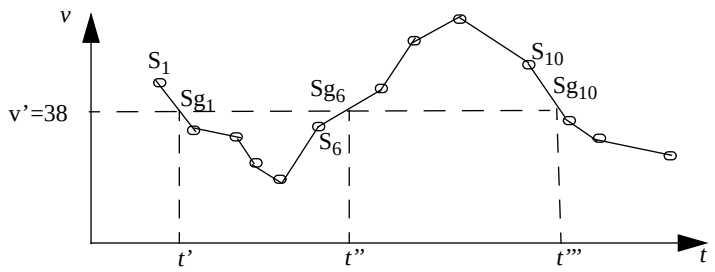


Fig. 8.7: Illustration of a value query

The cardinality of this query will be 2, because there are two time intervals (i.e., (t_1, t') and (t'', t''')) returned. On the other hand, if we view the time sequence as *discrete*, then the query:

- At what time points did the patient have the temperature higher than 38?

would yield cardinality of 5, because there are 5 explicit time points {t_1, t_7, t_8, t_9, t_{10}} where the temperature was higher than 38.

Therefore, there are two cardinalities of the range query $\sigma*_{v>v'}(TS)$, one is the number of sub-sequences returned, the other one is the number of (explicit) states inside those sub-sequences. Which one to use in the query optimization depends on whether the time sequence is viewed as discrete or continuous.

8.4 The Cost Model of $\sigma*_{v>v'}(TS)$

A relevant issue is the *cost model* of $\sigma*_{v>v'}(TS)$. As we mentioned above, $\sigma*_{v>v'}(TS)$ is computed by $F^{-1}(v>v')$. Fig. 5.5 shows that the cost of computing $F^{-1}(v>v')$ is nearly the same as the cost of computing $F^{-1}(v')$, because the *time intervals* of $F^{-1}(v>v')$ are composed by grouping the *time points* of $F^{-1}(v')$ into corresponding intervals. Therefore, the cost of $\sigma*_{v>v'}(TS)$ is nearly the same as the cost of $\sigma*_{v=v'}(TS)$.

This is surprising because we tend to think that the cost of a range query is proportional to the size of the range. The reason why the cost of $\sigma*_{v>v'}(TS)$ is not linear to the number of states in the range $v > v'$ is that the semantics of the $\sigma*$ operator is different from the semantics of the conventional σ operator. The $\sigma*_{v>v'}(TS)$ retrieves the *sub-sequences* that have values greater than v'. The cost is linear to the number of the sub-sequences returned, which has no direct relationship with the number of (explicit) states inside those sub-sequences.

The property that the cost of $\sigma*_{v>v'}(TS)$ is nearly the same as the cost of $\sigma*_{v=v'}(TS)$ makes the IP-index especially suitable for large sequences, because the cost of range queries does not grow with the number of states inside those ranges.

8.5 Time Window Queries

This section discusses time window queries. Time window queries are those value queries that only concern a *part* of the time sequence, i.e., a time window. An example of a time window query could be:

- When did the patient have a fever *in the last few days* (denoted as $t > t'$)?

Using the new functions proposed for the data type of time sequence (Section 6.5), this query can be expressed as the following:

```
SELECT t
FROM Temperature_seq TS
WHERE t IN get_time_stamps(TS, '='1, 38)
      AND t > t'
```

The answer of this query is marked by the two crosses in Fig. 8.8. This query can be processed in two steps: 1) $\sigma^*_{v=38}$(TS); 2) $\sigma^*_{t>t'}$(TS). The first step, $\sigma^*_{v=38}$(TS), generates all (explicit or implicit) states S' where S'.value = 38. Then, in the second step, every state S' from $\sigma^*_{v=38}$(TS) can be checked to see if it is in the time interval (t', now). If it is, then S' is returned as a result.

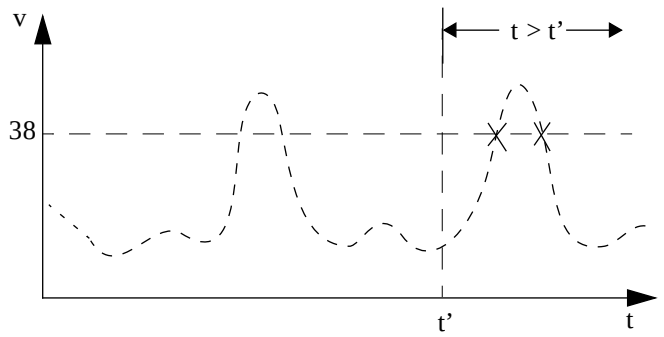


Fig. 8.8: A time windo w query

8.5.1 Optimization of Time Window Queries

When many states are returned from $\sigma^*_{v=38}$ and the resulting states are very few (the time window is small), it might be a waste to calculate all S's and check the condition later. Recall that the operator $\sigma^*_{v=38}$(TS) is accomplished by $IP_{v=38}$(TS) and *ifn* (illustrated as (a) in Fig. 8.9), the selection $\sigma^*_{t>t'}$(TS) can be "pushed down" to the $IP_{v=38}$(TS) operator, resulting in the $IP_{v=38\ AND\ t>t'}$ operator (see (b) in Fig. 8.9). The new operator, $IP_{v=38\ AND\ t>t'}$, can be processed efficiently by searching A(38) to find those states S_is where S_i.time > t'. In this way only a part of the anchor-state sequence A(38) (the part that is inside the time window) is involved in query processing.

Time window queries for the window condition t < t' can be optimized similarly by pushing the condition t < t' down to the IP operator as shown in Fig. 8.9. However, no search on A(v') is needed here since the starting position is the *first* state in A(v'). The stream output of the IP operator is terminated when the condition S_i.time < v' does not hold any more.

Another possible optimization strategy for "$\sigma^*_{v=38\ AND\ t>t'}$(TS)" is to generate $\sigma^*_{v=38}$(TS) as a *reverse* stream (as described in Section 8.1) and terminate when t > t' does not hold any more. This strategy is optimal when the time window is

1. In reality we should use '>' instead of '=' since a fever means body temperature > 38°C. All the discussions will hold.

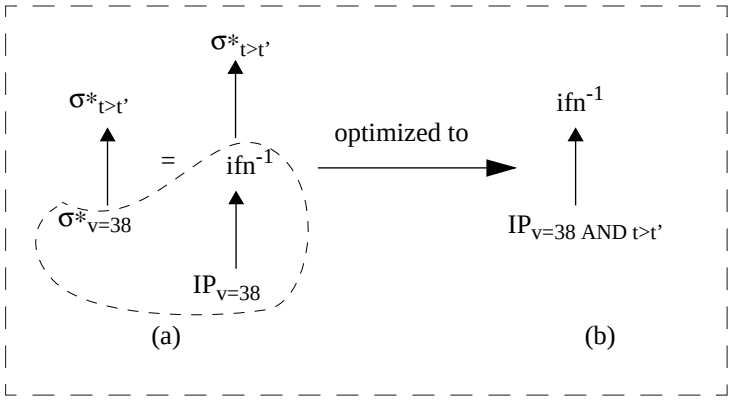


Fig. 8.9: Optimization of time window queries

t > t' (i.e., the window is with a *lower* bound), but it does not help when the time window is t < t' (i.e., the window is with an *upper* bound).

If we compare the strategy of generating a *reverse stream* with the strategy illustrated in Fig. 8.9, we see that generating a reverse stream has the limitation that the output stream is not in the same (time) order as the input stream. Therefore, this strategy (i.e., generating a reverse stream) cannot be used in methods such as sort-merge joins. Also note that a *general* time window query t' < t < t'' (i.e., a time window with both upper and lower borders) would normally require search on A(v') to efficiently find the positions of t' and t''.

In the next section we present a performance comparison of these different strategies with experiments on SHORE.

8.5.2 Experiments

We measured the three different strategies above for optimization of time window queries. The three strategies are: 1) Scanning A(v') to calculate all t's and check the condition later; 2) binary searching A(v'); 3) reversely scanning A(v').

The time sequences used in the experiments were the synthetic sine sequence in Fig. 8.2 (cardinality = 10K) and the real-life pressure sequences Fig. 8.3 (cardinality = 100K).

The time window was defined as t > t'. The window size was varied from 100, 500, 1K, 5K to 10K for the sine sequence and 1K, 5K, 10K, 50K to 100K for

the pressure sequence. Every window size results in a different number of anchor-states visited (Section 8.5.1). These numbers are plotted as the x-axis in Fig. 8.10 and Fig. 8.11. The experimental results are shown in Fig. 8.10 and Fig. 8.11.

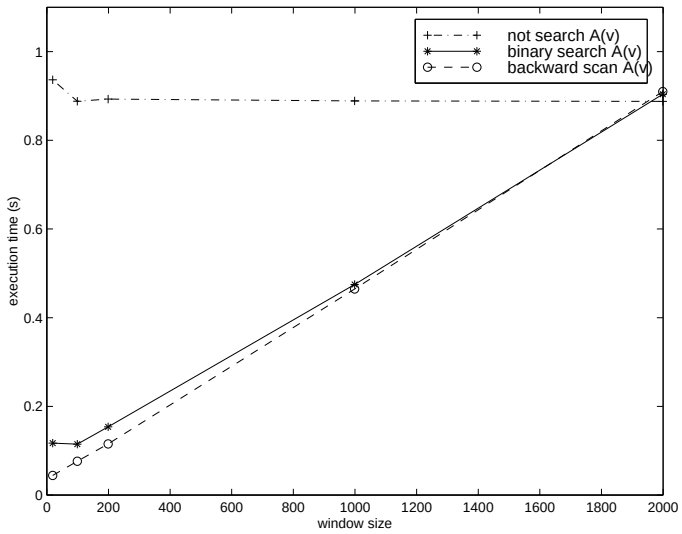


Fig. 8.10: The time window query for the sine sequence (10K)

The measurements show that: 1) reverse scanning of A(v') is the most efficient strategy since no extra overhead is needed; 2) binary searching A(v') (to get close to the position of t > t') performs almost as efficiently as reverse scanning; 3) when the time window is small, the difference between not searching A(v') and binary search A(v') is *substantial*.

The conclusion is that it is very important to optimize time window queries by pushing the condition t > t' into the IP operator (Section 8.5) when the window is small.

Some readers may wonder why we do not simply perform binay search on the original TS to find the sub-sequence that is inside the time window t > t' and perform $\sigma^*_{v=v'}$(TS) on this sub-sequence. The reason is that this strategy will always be slower than the strategy 2 above, i.e., binary searching A(v') (or strategy 3 above, i.e., reverse scanning A(v')), because A(v') is normally much *shorter* than the whole TS.

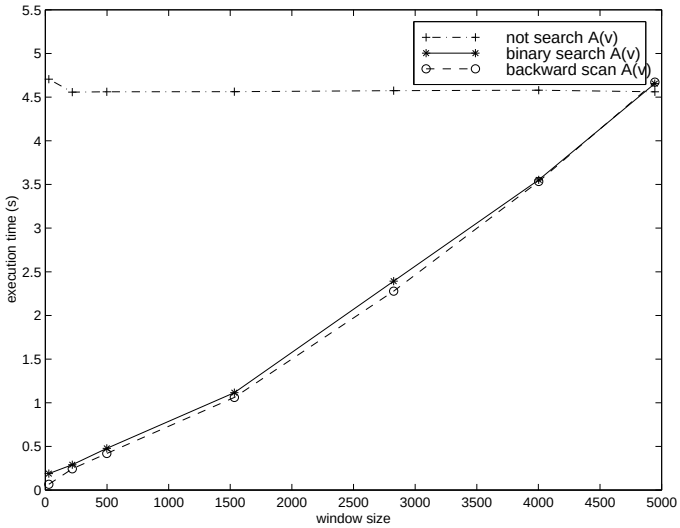


Fig. 8.11: The time window query for the pressure sequence (100K)

8.6 Complex Queries

This section shows possible ways to optimize complex value queries. A complex value query contains more than one $\sigma*$ operator.

Suppose that for the pressure sequence (see Fig. 8.3) we are interested in finding the pattern where "a small peak follows a big peak". Suppose a small peak means v=1.5, a big peak means v=2.2, and "follows" means the time difference is around 30 seconds (i.e., one period of this time sequence, see Section 1.2.2). Then this query can be expressed as Fig. 8.12.

```
SELECT t1, t2
FROM Pressure_seq S
WHERE t1 IN get_time_stamps(S, 1.5)
    AND t2 IN get_time_stamps(S, 2.2)
    AND 30 - e < t1 - t2 < 30 + e
```

Fig. 8.12: An example complex query

The variable *e* is a small value that denotes the *variation* of a period (around 30 seconds). Since $\sigma*_{V=V'}(TS)$ results in a stream of (explicit or implicit) states,

the function *get_time_stamps*(TS, v') results in a stream of time points. Therefore, there are two streams of time points generated from this query: *t1** generated from *get_time_stamps*(TS, 1.5), and *t2** generated from *get_time_stamps*(TS, 2.2). These two streams will be joined by the condition 30 - e < t1 - t2 < 30 + e. There are three possible ways to join these two streams:

- Query Plan1: Stream $\sigma^*_{v=1.5}$(TS) and $\sigma^*_{v=2.2}$(TS), performing join on these two streams using the predicate "30 - e < t1 - t2 < 30 + e" in lock step (similar to a sorted-merge join). This query plan is illustrated in Fig. 8.13.

  This plan has to go through *every* element in both stream. It might be inefficient when any of the streams is long.

- Query Plan 2: Stream $\sigma^*_{v=1.5}$(TS). For every output t1, using (t1-30-e, t1-30+e) as the window to pose the time window query "$\sigma^*_{v=2.2 \text{ AND } t1-30-e<t<t1-30+e}$(TS)". This plan is illustrated in Fig. 8.14.

- Query Plan 3: (Symmetric to Plan 2) Stream $\sigma^*_{v=2.2}$(TS). For every output t2, using (t2+30-e, t2+30+e) as the window to pose the time window query "$\sigma^*_{v=1.5 \text{ AND } t2+30-e<t<t2+30+e}$(TS)". This plan is illustrated in Fig. 8.15.

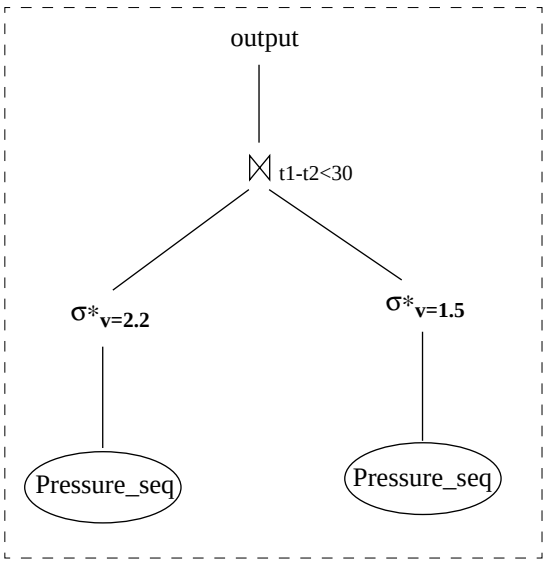


Fig. 8.13: Query Plan 1

Which plan is the best? It depends on several factors. For example, if the estimated cost of $\sigma^*_{v=1.5}$(TS) and $\sigma^*_{v=2..2}$(TS) are both small (the cost model can be found in Section 8.2), then Plan 1 will a good choice since the condition 30 - e < t1 - t2 < 30 + e can be checked easily while the two streams $\sigma^*_{v=1.5}$(TS) and

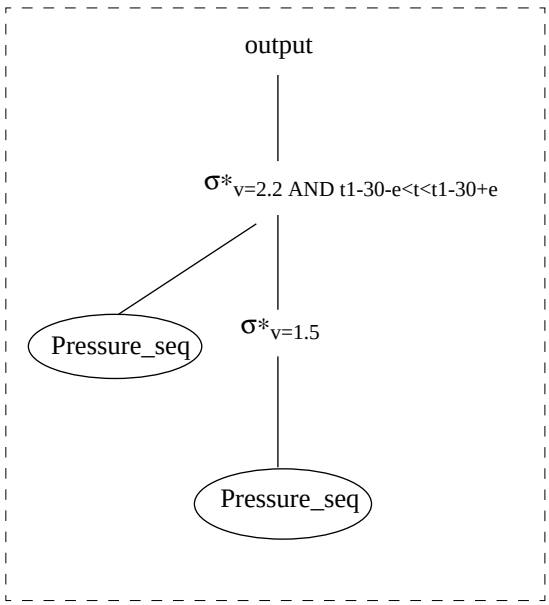


Fig. 8.14: Query Plan 2

$\sigma^*_{v=2..2}$(TS) can be generated quickly. If the estimated cost of $\sigma^*_{v=2..2}$(TS) is small but the estimated cost of $\sigma^*_{v=1.5}$(TS) is big, then Plan 3 is a better choice since the expensive operator $\sigma^*_{v=1.5}$(TS) will be replaced by a time window query "$\sigma^*_{v=1.5 \text{ AND } t2+30-e<t<t2+30+e}$(TS)", which only concerns a (hopefully small) part of TS. However, this will only be a good choice when the window size *e* is small, and the cardinality of $\sigma^*_{v=2..2}$(TS) (the number of time points t2 returned) is small as well. The reason is that: if the number of the time points t2 returned from $\sigma^*_{v=2..2}$(TS) is large, then the time window query $\sigma^*_{v=1.5 \text{ AND } t2+30-e<t<t2+30+e}$(TS) will be posed on TS many times, resulting in an expensive query plan.

Plan 1 can be seen as analogous to the *sorted-merge join* [107] strategy in relational query optimization, while Plan 2 and Plan 3 can be seen as analogous to the two possible choices of outer and inner relations [107] in a relational *nested-loop join*. Which plan is better depends on the cost and the cardinalities of the result sets of the σ^* operators. The cardinality of the result set of $\sigma^*_{v=v'}$(TS) is the same as the cardinality of A(v'), which is stored in the IP-index (Section 3.2.1). Note that the cardinality of different anchor-state sequences in a TS can vary very much. For example, in Fig. 8.3, the cardinality of A(1.5) is less than 10 and the cardinality of A(-0.2) is several thousand. So it is very important to optimize sequence queries using the cardinality information.

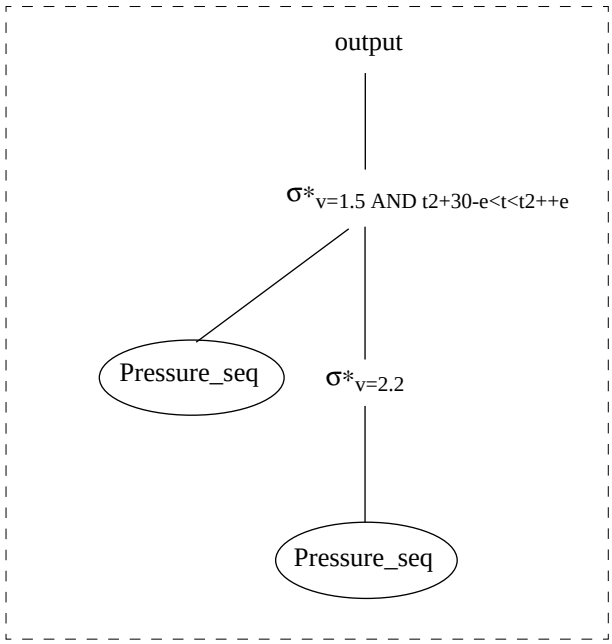


Fig. 8.15: Query Plan 3

Even the above seemingly simple query is not trivial when it comes to query optimization. The reason is that optimization of complex value queries depends on several factors such as the cost of $\sigma^*_{v=v'}$(TS), the cardinality of the answer set, and the time window size, etc. Optimization of complex value queries is an interesting topic for future work.

8.7 Summary

Query optimization is an important issue in managing sequence data. In this chapter we have discussed several possible optimization techniques for *value queries*. First we showed that *stream processing* can be utilized in processing the $\sigma^*_{v=v'}$(TS) operator when the cardinality of A(v') is large (which results in a large answer set). Secondly, a cost model of $\sigma^*_{v=v'}$(TS) was developed. It shows that the cost of $\sigma^*_{v=v'}$(TS) is determined by two factors: 1) the *cardinality* of A(v'); 2) how the anchor-states in A(v') are *distributed* in disk pages. This indicates that some *statistics on the value distribution* of the sequence are needed in order to estimate the cost of the σ^* operator. Another interesting issue is the *cardinality* of range queries. In contrast to the traditional case, the cardinality of range queries on a continuous sequence is *not* linear to the *size* of the value range or the size of the sub-sequences retrieved. Instead, it is linear to

the *number* of sub-sequences returned. This indicates that processing range queries is very efficient using the IP-index, especially for large sequences. We also investigated optimizations of time window queries and complex value queries. Time window queries can be optimized by pushing the *time window* into the IP operator (a component of the $\sigma*$ operator), reducing the number of anchor-states retrieved before applying ifn^{-1}. Complex value queries (sequence joins) can be optimized by choosing a good join order according to the cardinalities and the cost of the $\sigma*$ operators involved. Experiments were performed to verify the above optimization strategies.

Chapter 9

Related Work

This chapter discusses related work in general. More closely related work, such as work related to the IP-index, or work related to the σ* operator, can be found in corresponding chapters.

The most relevant related work is a database system named SEQ, which addresses the design and implementation issues of supporting *sequence data* in DBMSs. We shall provide an overview of the SEQ system, including its data model, query language, and query optimization techniques, and point out what is relevant to our work.

Other closely related work includes *similarity search* on time series, i.e., finding similar patterns in different time series. Research in this area has been very active in the recent years. Various methods have been proposed, such as those using the Discrete Fourier Transformation (DFT) to compare time series in the frequency domain, or transforming time series into some feature-preserved functions. We shall point out pros and cons of each approach and compare them with our work.

Since time series in trading or financial business have special requirements that cannot be met by conventional DBMSs, most time series data are managed by *special-purpose management systems*. In this chapter, we shall provide an overview of time series management systems such as FAME [52] and CALANDA [42].

Since a time series is a special case of temporal data, research on *temporal databases* is discussed, including temporal data models, temporal query lan-

guages, and temporal indexes.

9.1 SEQ — A Sequence DBMS

As pointed out in the introduction, sequence data appear in many application domains such as stock prices, product sales, scientific measurements, medical data, and event processing. Traditional database systems are based on the model of *sets*, not *sequences*. Consequently, expressing sequence queries is very tedious in SQL and query execution is very inefficient [109]. For this reason, Seshadri et al. [110] designed a sequence database system SEQ, which address the special issue of supporting sequence data in DBMSs.

SEQ [110] is based on a sequence data model SEQ [109]. This data model aims at capturing the *ordering* semantics of sequence data and specifies common operators on them. A sequence query language, *SEQUIN*, was proposed. Query optimization and physical organization were also addressed in [110].

The SEQ system later evolved into PREDATOR [111] to support other types of non-traditional data types such as image, audio, and spatial data. In this section we shall provide an overview of the SEQ system and point out what is relevant to our work.

9.1.1 The SEQ Data Model

The data model which the SEQ system is based on is also named SEQ [109]. This data model consists of an *ordering domain* and a *record domain* (see Fig. 9.1). The *ordering domain* can be composed of any kind of ordered data such as integers, time stamps, etc. Each element in the *ordering domain* is called a *position*. Records in the *record domain* can be of any data type such as floating values, strings, or complex data types. The relationship between the *ordering domain* and the *record domain* is “many to one”. That is, different positions in the ordering domain can be mapped to the same record, but every record can only be mapped to one position.

There can be “holes” in the ordering domain, which results in *sparse* sequences. Sparse sequences correspond to real-life sequences where there are missing values in measurements.

In SEQ, operations over sequences include the following:

- *transform*

  — apply a function *fn* on each record in the sequence

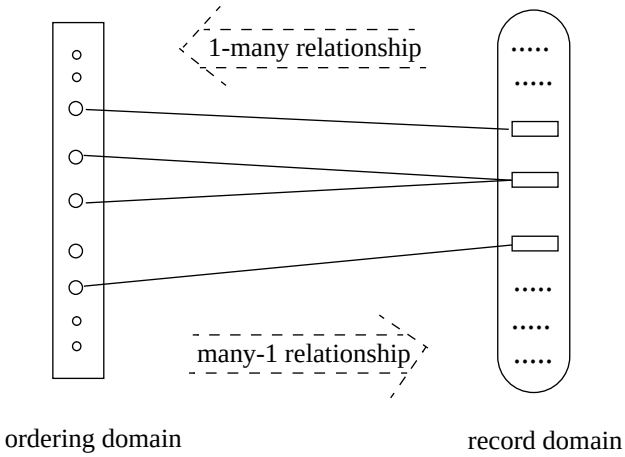


Fig. 9.1: The SEQ data model

- *binary*

  — e.g., two sequences join

- *offset*

  — e.g., shift in the ordering or record domain

- *aggregate*

  — e.g., *moving average, max, min*

- *zooming*

  — transform sequences according to different granularity in the ordering domain. In the SEQ data model [109], a *collapse* operation is defined to transform from a coarse domain to a finer domain (e.g., from weeks to days), and an *expansion* operation as the inverse. Certain well-known collapses and expansions on the ordering domain are pre-defined.

Based on the SEQ data model, Seshadri et al. [110] designed a sequence database system SEQ. In SEQ, sequence data are supported by extending DBMSs with *abstract data types* (ADTs). ADTs in extensible DBMSs are discussed in the next section.

9.1.2 Abstract Data Type (ADT)

Traditional DBMSs have limited support for data types, i.e., only simple types

such as *scalar* or *strings* are supported. To meet the requirements of modern applications, new generation DBMSs support type extension — new data types can be added to the system without changes to the existing codes.

The basic technology used is that of *Abstract Data Types* (ADTs). The concept of ADTs was adapted from programming languages (such as [9][60][87]) in the 1980s in the database system Postgres [129] (see also [128][132]). In this approach, the DBMS maintains a table of ADTs, and new ADTs may be added by a database developer or user. Each ADT provides methods that implement a common internal interface through which the system can access values of that type. The internal interface includes methods for the storage and indexed retrieval of values. Each ADT can also declare primitive operations for manipulating or querying values of that type. For example, an ADT for images might provide operations such as *Rotate(I, Angle)*, *Clip(I, Region)*, and *Oerlay(I1, I2)*. Libraries of primitive operations for each ADT are sometimes called "datablades" [65], "data cartridges" [95], or "data extenders" [27].

In SEQ, sequence data are modelled as a new data type named *sequence*. Methods defined on *sequences* are: *OpenScan(Cursor)*, *GetNext(Cursor)*, and *CloseScan(Cursor)*, which provide a scan of the sequence in the forward order of the ordering domain. Any positions in the domain which are not mapped to a record are ignored in the scan. *GetElem(Pos)* is used to find the record at the specified position in the sequence (or fails if none exists at that position).

In SEQ, each ordering domain is modelled as a data type associated with some additional methods. Methods defined for ordering domains are: *LessThan(Pos1, Pos2)*, *Equal(Pos1, Pos2)*, and *GreaterThan(Pos1, Pos2)*, which allow comparisons to be made among positions. *Collapse* is an operator used to transform sequences between different granularity in the ordering domain. In addition, all ordering domains can be organized into a hierarchical relationship to model the relationship between them. Fig. 9.2 shows one set of hierarchical relationships between common temporal ordering domains. A *collapse* operation is defined to map a position in one ordering domain to a position or set of positions in another domain.

9.1.3 Physical Organization of Sequences

As mentioned in Chapter 7, physical organization determines the efficiency of a database system. Let us take a look at how sequences are physically implemented in SEQ.

In SEQ [110], a sequence was implemented as an *array* of records. The array was implemented using a single SHORE large object, which can grow arbitrarily large and supports insertion or deletion in the middle of the object. Since

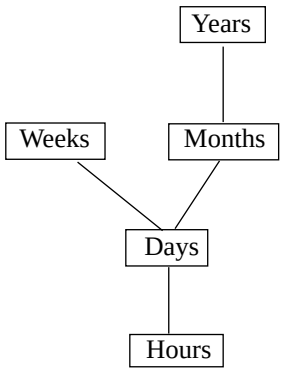


Fig. 9.2: Sample ordering hierarchy on times in SEQ

sequences can be irregular (i.e., have empty positions), a *compressed* array representation was used to reduce space used for empty positions. This compression makes some operations within a sequence (such as position lookup, insert and delete) more expensive to implement.

Compared to our approach of using the multi-level dynamic array to store sequence data (Section 7.2), this compressed array implementation has the following drawbacks: 1) In the current SHORE version, page faulting for large objects is not supported. This implies that when one record is needed, the whole sequence has to be brought into main-memory. This leads to very slow retrieval for large sequences. By contrast, in our approach, only one disk page (one base array) is read into main-memory when one record is needed. 2) Even if page faulting is supported in SHORE, two kinds of indexes need to be maintained for fast retrieval of a record: an index that records the mapping between the *record_ids* and the *page_ids*, and an index that records the mapping between the values in the *ordering domain* and *page_ids*. By using our multi-level dynamic array structure, these two kinds of indexes are naturally supported by the *index arrays*.

Therefore, our approach is a superior one compared to the compressed array described in [110]. Our multi-level dynamic array structure supports both fast random access and page-faulting.

9.1.4 *SEQUIN* Query Language

The query language in SEQ is named *SEQUIN* [110]. *SEQUIN* is similar in flavour to SQL. It is a declarative language for sequence queries. The result of a query in *SEQUIN* is always of the type *sequence*.

The following examples give a flavour of *SEQUIN*. Two stock price sequences Stock1 and Stock2 are used in the examples. Both sequences have the same schema: {*time*: Hour, *high*: Double, *low*: Double, *volume*: Integer}.

- Estimate the monetary value of Stock1 traded in each hour when the low price fell below 50.

```
Project[1] ((A.high + A.low) / 2) * A.volumn
From Stock1 A
Where A.low < 50     .                                    (1)
```

- Find the 24-hour moving average of the difference between the prices of the two stocks.

```
Project avg(A.high - B.high)
From Stock1 a, Stock2 B
Over $P - 23 TO $P                                        (2)
```

- Zoom:

```
Project min(A.volume)
From Stock1 A
Zoom days                                                 (3)
```

Example (1) examines the sub-sequences of Stock1 where the low prices are below 50. Example (2) applies a 24-hour moving average over the sequence. Example (3) demonstrates the *zooming* operation (zooms from hours to days).

9.1.5 Query Optimization

Several query optimization techniques were discussed in [110]. The most important ones are listed below.

Propagating Ranges of Interests

This class of optimization deals with the use of information that limits the range of positions of interest in the query answer. It is similar to the "selection push-down" technique in the relational DBMS. Selection push-down for sequence queries can be applied to either the *ordering domain* or the *record domain*. Selection based on the *ordering domain* can be illustrated by the following example:

1. In [110] 'project' is used instead of the standard SQL syntax 'select'.

```
Project count(*)
From Stock S
Where S.time > "<timestamp>"
Zoom All;
```

This query can be optimized by pushing the selection predicate (S.time > "<timestamp>") into the scan of the sequence (for performing the "count" operation). In SEQ, this is achieved by performing a *weighted binary search* in the *compressed array* (the array which stores the stock sequence, see Section 9.1.3) to get close to the starting position of the range (S.time > "<timestamp>"). Since this array is implemented as a SHORE large object [110] and page-faulting is not supported in the current SHORE version, the *entire* compressed array has to be read into main-memory in order to do binary search. This is inefficient, especially in the situation when the sequence is long but the relevant range is small (for example, the stock sequence consists of data accumulated for *several years* while this query is only interested in the data of the *last few weeks*).

This query demonstrates the importance of an *index* on the *ordering domain*. Our multi-level dynamic array structure (Section 7.2) is a better choice in this situation. This is because locating the starting position of the range S.time > "<timestamp>" can be done efficiently by searching the index arrays instead of bringing the entire sequence into main-memory.

Selection push-down on the *record domain* is illustrated by the following example:

```
Project[1] ((A.high + A.low) / 2) * A.volume
From Stock1 A
Where A.low < 50     .
```

In [110] it is claimed that selection push-down (A.low<50) should be applied here so that the calculation of "((A.high+A.low)/2)*A.volume" only needs to be done for those states whose low values are below 50. But, without an index, the whole time sequence has to be *scanned* to find these states. By contrast, using the IP-index, we can easily calculated the time intervals (t, t'')* where the prices inside those time intervals are below 50 (Section 5.2.2). Therefore, the calculation of "((A.high+A.low)/2)*A.volume" only needs to be applied to the states inside those time intervals.

Therefore, we conclude that our work on 1) the IP-index; and 2) the multi-level dynamic array structure, contributes to the current research field on managing *sequence data* in DBMSs.

---

1. In [110] 'project' is used instead of the standard SQL syntax 'select'.

Incremental Computation of Aggregate Operators

The second strategy on query optimization in [110] is *incremental* computation of aggregate operators. For example, consider the 3-position moving average of a sequence 1, 2, 3, 4, 5. Once the sum 1 + 2 + 3 has been computed as 6, this computation can be used to reduce the work done for the next aggregate. Instead of adding 2 + 3 + 4, one could instead computing 6 - 1 + 4. Due to the small aggregation window in this example, there is little benefit. However, when the window becomes larger and the operations are more expensive, there can be significant improvements due to this approach. Importantly, the time required for aggregation is independent of the size of the window.

Other query optimization techniques include detecting *common sub-expressions* (and evaluating them once), and *operator pipeline*. Operator pipeline (stream access [109]) is crucial for sequence processing. In this technique, each sequence is read in a single continuous pipelined stream without materializing it. This is accomplished by associating buffers with each operator, to cache some relevant portion of the most recent data from its inputs. We use a similar approach in processing $\sigma^*_{v=V}$(TS) operator in Section 6.2.2.

9.1.6 Comparison With Illustra

Seshadri et al. [110] compared SEQ with Illustra [64] (Illustra has now been acquired by Informix [65]). Illustra supports time series as an ADT and defines common functions on it. It is claimed in [110] that queries based on functional composition (as in the case of Illustra) have a *procedural* semantics instead of *declarative* semantics (as in the case of *SEQUIN*). Seshadri et al. [110] also claim that in Illustra, little or no *inter-function optimization* is performed. For example, the following optimizations are not supported by Illustra:

- pipeline operations on two functions
- identifying common sub-expressions in function compositions
- selection push down

According to [110], query processing in SEQ results in overall performance improvements of approximately two orders of magnitude compared to Illustra.

9.1.7 Conclusions

To conclude, we have provided an overview of the design and implementation of the sequence database system SEQ. The main contributions of SEQ are: 1) it proposed a general data model for sequence data; 2) it designed a sequence

query language *SEQUIN*; 3) it addressed query optimization techniques for sequence data.

The weakness of SEQ is in physical organization. The data structure proposed in [110] is inefficient in *lookup* and *retrieving* sequence data, especially when the sequence is large. Another limitation is that SEQ does not provide index access on the *record domain*.

Our multi-level dynamic array structure and the IP-index complement the work in SEQ by providing a good physical structure of 1-D sequence data and an indexing method for value queries.

9.2 Similarity Search on Time Sequences

Another closely related research field is *similarity search* on time sequences, i.e., finding similar patterns in different time series. Similarity search is essential in discovering and predicting the risk, causality, and trend associated with a specific pattern. More examples can be found in identifying companies with similar growth patterns, products with similar selling patterns, stocks with similar price movement, images with similar weather patterns, etc.

Several approaches have been suggested to deal with similarity search. In what follows we present an overview of this work and discuss pros and cons of each approach and see what is relevant to our work.

9.2.1 Using the Discrete Fourier Transform

It seems that the work by Agrawal et al. [4] is the first one addressing similarity search on time sequences. In [4], the similarity of two sequences is measured by the *Euclidean distance* between them. A sequence is considered to be similar to a query pattern if the Euclidean distance between them is less than the user-specified error δ, see Fig. 9.3. To measure the Euclidean distance between two time sequences, the Discrete Fourier Transform (DFT) is applied to each time sequence, and the first few coefficients are used to map each time sequence into a *point* in a multi-dimensional space. Then, an R*-tree is used to index these points and similarity of sequences is measured by the Euclidean distance between these points. This approach is based on two assumptions: 1) low frequencies constitute data, and high frequencies are noise; 2) DFT preserves the Euclidean distance between sequences in the time or frequency domain. Apparently, this work only deals with a preliminary notion of *similarity* because: 1) It does not deal with sub-sequence matching (all sequences have to be the same length); 2) It does not deal with amplitude shift or time shift (see Fig. 9.5).

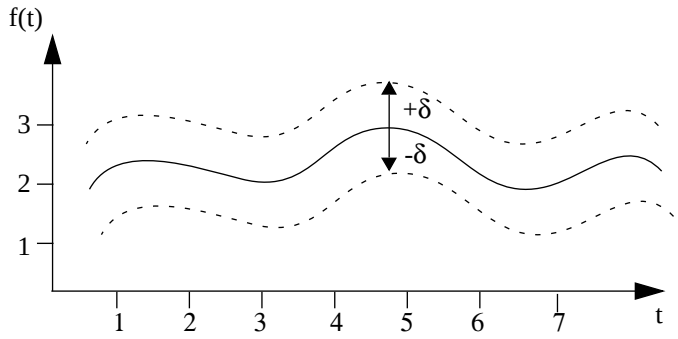


Fig. 9.3: A notion of similarity between sequences

Faloutsos et al. [51] extend the work of [4] by addressing the issue of *sub-sequence matching*. The goal is to locate 1-D sub-sequences within a collection of sequences such that the sub-sequences match a given (queried) pattern within a specified tolerance. Based on the work of [4], Faloutsos et al. [51] use a sliding window over the sequence and generate a *trail* instead of a point (as in [4]) in a multi-dimensional space. To save storage space and speed up indexing, not every point of the trail is stored in the database. Instead, the trail is divided into sub-trails and approximated by their *minimum bounding rectangles (MBR)*. These MBRs are organized in a R^+-tree and similarity search is performed by searching intersecting MBRs. This method may introduce *false matches* because it may happen that some sub-trails do not intersect with the queried region while their MBRs do. This is dealt with in a post-processing process.

The work of [4] and [51] has the limitation that they cannot detect similarity under transformations in the *frequency* domain, such as dilation (frequency reduction) or contraction (frequency increase). This is because the DFT is sensitive to the frequency. Also, the problem of amplitude scaling and offset translation are not addressed.

Li et al. [80] extend the work of [51] in sub-sequence matching. In [80], the Discrete Fourier Transform is also used to get the first few coefficients to represent the feature of the sequence. However, instead of using Euclidean distance of these coefficients to measure the similarity of two sequences, Li et al. [80] use *correlations* of these coefficients. Unfortunately, no comparison with related work such as [51] was mentioned in [80]. Only comparison to linear scanning was performed.

In [5][1] a different notion of *similarity* for time sequences is suggested. There, two time sequences are similar if they have enough non-overlapping sub-

1. In the original paper the term “time series” was used to mean “time sequences”.

sequences that are similar. The main goal of [5] is to deal with outliers in sequences, and amplitude scaling and offset translation. The algorithm contains three major phases: 1) atomic matching; 2) window stiching; and 3) subsequence ordering. Although experiments were performed to show that the algorithms actually worked, no measurements on the *efficiency* of the approach were performed. Therefore it remains a question is this method is practical considering time sequences are usually very large in volume.

9.2.2 Function Approximation

Shatkey and Zdonik [112] suggest a different strategy to deal with similarity search. The example of the "goalpost fever" pattern was used in [112] (see Fig. 9.4). Shatkey and Zdonik [112] claim to support many feature-preserved transformations as shown in Fig. 9.5 such as *scaling* (1, 2, 3, 4), *contraction* (1, 2, 4), *dilation* (3), *shift in time* (2), and *shift in amplitude* (4). The idea is to break sequences into meaningful sub-sequences and represent them using some feature-preserving functions such as regression lines or linear interpolation. Queries are approximated by features and performed on the function representations of sequences. The major advance of this work, compared to the older ones, is to suggest a strategy to handle approximate queries which is both *amplitude*- and *frequency-independent*. Also, using function approximations to represent a sequence saves storage space, compared to the normal approach where all the points in a sequence are stored. However, this approach has the main drawback that a large amount of work needs to be done in the pre-processing process where sequences are broken into meaningful sub-sequences and approximated by functions (although this can be done off-line). Also this approach is sensitive to the functions used (linear regression and linear interpolation were used in [112]), and the mount of work will be substantial when higher degree functions (such as polynomials) are used.

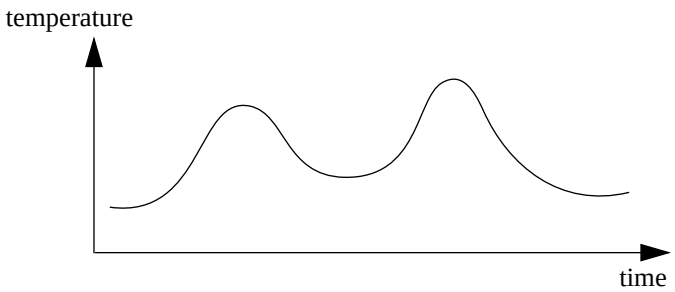


Fig. 9.4: A pattern of "2-peaks"

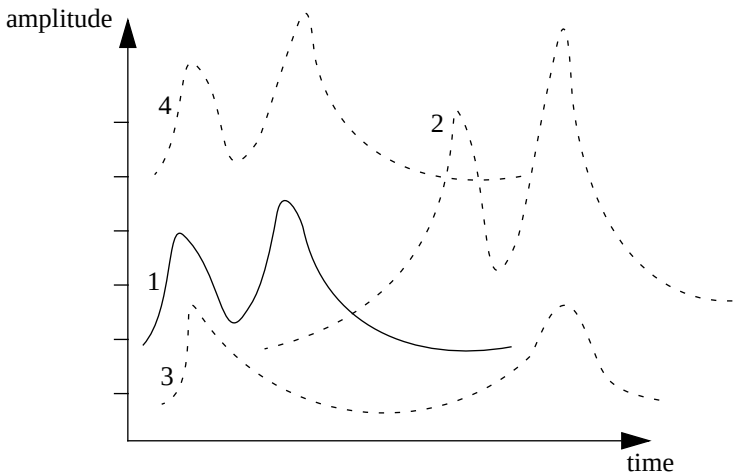


Fig. 9.5: Various 2-peaked sequences under transformations

Relevance to Our A pproach

An interesting observ ation is that the e xample used in [112] — finding the "goalpost fever" pattern in a temperature sequence, can be dealt with more naturally by the $\sigma*$ operator and the IP-index (see Section 5.4). Compared to the approach in [112], our approach covers all the cases listed in [112] easily, such as *scaling*, *contraction*, *dilation*, *shift in time*, and *shift in amplitude*.

9.2.3 Shape Languages

Agrawal et al. [6][1] use a different approach which was taken from text string matching technique. There, a sequence is transformed into a sequence of *symbols* from a pre-defined alphabet (such as "up", "down", "stable", etc.). This alphabet describes the shapes of the sequence. Sequences are indexed by the symbols and the positions where they appear in the sequences. Sub-sequence matching is performed by searching this index. A shape query language, SDL, has been defined in [6]. SDL is equivalent in expressive power to the regular expressions. We believe that this approach is very sensitive to the alphabet used. The definition of the alphabet is dependent on the application (such as when to consider a slope as an "up" and when to consider it as a "stable" state). This makes one alphabet set (and the correspondingly built index) only useful for one kind of application. Also no index insertion algorithm or performance measurement were mentioned in [6].

1. In the original paper the term "history" is used to mean "time sequences".

9.2.4 Conclusions

Research on similarity search on time series complements our work nicely. Similarity search is based on the general *shapes* (features) of a sequence, while our work is based on individual *values* of a sequence. These two aspects of support for sequence data are both highly needed in real-life applications.

9.3 Time Series Management Systems

In Section 2.1.4, we mentioned that a time series is a "regular" time sequence. Actually, a time series in the trading business has more complicated structure than the time sequence data model TS = (t_i, v_i). An example time series is shown in Fig. 9.6 [101], which represents the stock exchange history for the Union Bank of Switzerland (UBS) registered.

Name:	UBS registered
Security_number:	136 102
Start_date:	11.10.93
End_date:	23.12.93
Calendar:	Business week

Date	Low	High	Ticks
11.10.1993	78	85	79, 78, 77, 80, 83, 85,...
12.10.1993	80	84	84, 82, 80, 83,...
13.10.1993	82	86	84, 82, 83, 85, 86,...
.....	...	...	.....

Fig. 9.6: An example time series

In Fig. 9.6, we see that a *time series* consists of two parts: 1) a general description: such as the name, the starting and ending date of the time series, the calendar type used in the time series, etc.; 2) a chronologically ordered sequence of observations: such as low, high values, and values at every "tick" for every trading day. The general description is called the *header* [101], and the chronologically ordered observations are called *events* [101]. In what follows we discuss the special requirements on time series management in the trading business.

Multivariate Time Series

A time series is multivariate. It consists of a *header* and *events*, as shown in Fig. 9.6. The header consists of common attributes characterizing the whole time series. Events model data collected over successive points in time. Data fields in events can have scalar types (such as *integers* for the low/high values in Fig. 9.6) or structured types (such as the *array* for the sequence of trading prices at each tick).

An important aspect of time series analysis is the transformation of events between different periodicities (e.g., transforming daily data to monthly data and vice versa). Different kinds of values require different periodicity transformations. For example, for the *high* price, the monthly value is the *maximum* of all daily values. For the *closing* price, the monthly value will be the value of the *last day*. For the cash flow, the monthly value will be the *sum* of all daily values.

Since time series are usually subject to statistical evaluations in which matrix algebra plays a central role, time series management systems should provide the data type of *arrays* (vectors, matrices, and even arrays of higher dimensions) and support operations on them. Another important capability is the derivation of new time series from existing ones, e.g., by computing the difference of two time series, calculating a moving average or aggregating events for a coarser granularity.

Groups

In databases with a large number of time series, detecting the data relevant to the interests of a user is an important issue. Normally one needs to partition the set of time series into categories or groups according to various criteria (e.g., a set of share price series might be categorized along branches, country and/or size of company). For this, a time series management system must support a flexible, powerful *grouping* mechanism. It is desirable that a group can recursively contain other groups, that elements can belong to more than one group and that participation in a group is by enumeration or by condition.

Calendars

Each time series is associated with a *calendar* that expresses the mapping between the events and their corresponding points in time. A time series management system must support a variety of calendars, taking into account various *base calendars* (like the Gregorian calendar, Islamic calendar, etc.), different *granularities* (like daily, weekly, or quarterly calendars), business and non-business calendars and calendars with local holidays. For non-periodic

time series, concepts like an ordinal calendar (just representing time units by natural numbers) and enumerated calendars (enumerating irregular sequences of dates) should be supported. Calendar-related functionality must include operations to define all these calendars, to transform time units between calendars, to scan calendars sequentially, to compare and do arithmetic calculations involving dates and time spans.

Because of the above special requirements on time series management, conventional DBMSs cannot be used to manage time series data in the trading business. For this reason, *special-purpose management systems* have been developed to manage time series in finance and trading. We discuss the most important time series management systems below.

9.3.1 FAME

FAME [52] is currently the most mature commercial system specialized for time series management. The name "FAME" stands for "Forecasting, Analysis and Modeling Environment". It was developed by FAME Software Corporation, a subsidiary of the Citybank in Switzerland.

Data objects available in FAME are scalars, univariate time series, and computed time series described by formulas. Time series may be defined with various frequencies from seconds up to multiples of a year. Each time series is described by a standard header with information like frequency, first and last date recorded, and aggregation type (summed, averaged, etc.). FAME offers powerful specialized functionality for statistical evaluation and forecasting (such as linear regression, Box-Jenkins, moving average, Monte Carlo Analysis, and many others). There are also a number of functions to produce graphical output and reports in various formats. Import from and export to a variety of file formats is supported. FAME is a very comprehensive system as to calculations on and presentation of time series. Query capacities consist in finding data objects by name (including a wild card facility) and retrieving events by time stamp. Time series can be grouped by so-called name lists (e.g., all time series related to earnings could be named "*.earnings", where '*' is a wild card for the name of each company).

Operations on time series in FAME can be classified as follows: 1) data preparation: interpolation of missing values and time scale conversion; 2) queries: moving averages, cumulative sums, discretizing (e.g., rank the revenues by whether they are in the top third, the middle third, or the bottom third), statistical functions (e.g., correlation between two series), etc.; 3) forecasting: statistical or data mining-based extrapolation.

To interpret missing values in FAME, all values are divided into two categories:

1) *Level values*: these kinds of values stay the same from on period to the next in the absence of activity. For example, inventory is a level value, because inventory stays the same if you neither buy or sell. 2) *Flow values:* these kinds of values are zero in the absence of activity. For example, expenses go to zero if you buy nothing. Other interpolation methods such as cubic spline can also be used to derive missing values.

FAME has several limitations. According to Dreyer et al. [41], FAME has poor search and retrieval facilities, and no mechanisms for data quality management and data consistency control. The data model is not powerful enough: each event may have only one scalar field, and the group concept is too limited — group members can only be selected by pattern matching with simple wildcards, not by content. Finally, the 4GL requires special training and a lot of experience.

9.3.2 CALANDA

Another special-purpose management system for time series is CALANDA [42] developed at the Union Bank of Switzerland. CALANDA has an object-oriented data model with special root classes to model time series and groups of time series. All the usual object-oriented features like inheritance, method definition, overloading, etc., are supported. CALANDA supports multivariate time series with an arbitrary number of attributes per event. Besides simple attribute domains, multidimensional array types are an important modeling instrument for statistical applications. Each time series consists of a sequence of events and a header which is a record with a user-defined attribute structure. There are predefined operations for filtering time series, frequency transformation, array manipulation, etc. Of course, the user can extend this functionality by defining his/her own methods.

Groups are supported as a flexible instrument for categorizing and aggregating time series. The large set of time series can be structured by building up a directed acyclic graph which can be used for navigation, querying, and set operations. Grouping is not restricted to time series that describe one semantic entity, but can be applied to any set of time series that fulfill common criteria (e.g., all time series of securities which are currently above/below a market index, all time series of companies in a given country, and so on). CALANDA offers extensive calendar functionality providing data arithmetic, holidays, business weeks, calendar transformation, etc. CALANDA also offers a graphical user interface that resembles a spreadsheet or 4GL tools for relational DBMSs.

9.3.3 Informix TimeSeries DataBlade

The commercial system Informix (former Illustra) is an object-relational DBMS where Abstract Data Types (ADTs) (Section 9.1.2) are supported for various application domains. Among all ADTs supported by Informix, "TimeSeries DataBlade" offers functionalities for time series management.

In Informix's TimeSeries DataBlade, two new data types are supported, namely *time series* and *calendar*. A time series is modelled as an n-ary vector and associated with a set of additional information such as its frequency, life span, etc. Time series can be multivariate, i.e., consist of an arbitrary number of recorded attributes. Time series can have various granularities and be associated with different calendars. Access to data is based on an SQL extension, which allows combining time series and other relational data in one query. In addition to the general SQL facility, about 40 predefined functions are supplied, e.g., to aggregate time series, compute a time lag, clip a predefined interval, etc. Further analysis functions can be defined by the user by way of the abstract data type feature of the ORDBMS. The limitation of Informix's TimeSeries DataBlade is that there is no grouping mechanism to structure the set of time series.

9.3.4 Conclusions

Time series in financial and trading have special requirements that conventional DBMSs cannot meet. Most time series data are managed by special-purpose management systems such as FAME.

Object-relational DBMSs or object-oriented DBMSs provide a way to manage time series data by treating time series as a new data type and defining operations on them. An example is Informix's TimeSeries DataBlade. Our work contributes to this area by proposing a good physical structure for time series (Section 7.2) and providing an indexing method (the IP-index) for efficient processing of *value queries* on time series.

9.4 Temporal Databases

Conventional database management systems were designed to capture the most recent data which model the real world. As new data became available through updates, the existing data values were usually removed from the database or delegated to archival storage. This is because it was expensive or impractical to store and access large volume of temporal data on-line. Therefore, applications had to manage temporal information in an ad-hoc manner.

Since the 1980s, the cost of main memory and magnetic disks has been decreas-

ing, and new storage media such as optical disks are emerging. Therefore, managing temporal data in DBMSs has become feasible. As a consequence, research on temporal databases has been active for the last decade. According to the first book on temporal databases [134]: the general definition of a temporal database is the following:

> *A database that maintains past, present, and future data is called a temporal database.*

In what follows we shall provide an overview of research work done on temporal databases, and discuss the relationship between temporal databases and time series management.

9.4.1 Time Dimensions

After the research on temporal databases started (1985), it was quickly discovered that there are different notions of *time* associated with data values. According to [67], "A Glossary of Temporal Database Concepts", there are the following dimensions of time in temporal databases:

- *transaction time* —

  The time when the information is *stored* in the database (normally time points).

- *valid time* —

  The time when the information is *true* in the modelled reality (can be a set of time points or intervals).

- *User-defined time* —

  The *uninterpreted* attribute domain of data and time. It is maintained by application programs (not supported by DBMSs). In contrast, both *transaction time* and *valid time* are supported by DBMSs.

In [67] the term *temporal database* is defined as "a database that supports some aspect of time, not counting user-defined time". In other words, a temporal database supports either *transaction time* or *valid time*, or both. In contrast, the term *bitemporal database* [67] is used to refer to a database system that "supports exactly one *valid time* and one *transaction time*".

9.4.2 Research on Temporal Databases

Research on temporal databases has been active for more than one decade. Most of the effort was dedicated to proposing suitable data models and devel-

oping general-purpose, declarative temporal query languages. Relatively less work has been done on implementations, such as physical organization of temporal data, indexing, and query optimization.

Since relational DBMSs still dominate the database market, it is perhaps not surprising that — despite the well-known limitations of the relational data model — many of the proposals attempt to extend the *relational data model* with temporal support. As a consequence, relational query languages are extended with special syntax to deal with temporal semantics. These proposals culminate in the TSQL2 standard [125], which is the extension of SQL2 standard with temporal support. (Currently, there is also an on-going effort to extend the unfinished SQL3 standard with temporal features.)

Other proposals attempt to extend the *object-oriented* data model with temporal support. In what follows we compare these two different approaches by examples.

Extending the Relational Data Models

There are two basic approaches to extending a *relational* model with time information: *tuple time-stamping* [134] and *attribute time-stamping* [134]:

- *Tuple timestamping* —

  Timestamps are added to tuples to specify the time for which the tuples are defined.

  Example 9.1: *employee (name, salary, dept, time_stamp).* □

  The *time_stamp* in a tuple refers to the *valid time* or the *transaction time* of the tuple (i.e., the information of an employee). Note that a *timestamp* [67] can be either *time points* or *time intervals*.

- *Attribute timestamping* —

  Timestamps are added to attributes of a tuple to record the history of the changes of attributes.

  Example 9.2: *employee (name, (salary, time_stamp1), (dept, time_stamp2))* □

  Here *(salary, time_stamp1)* constitutes the salary history of this employee, and the *(dept, time_stamp2)* records the departments where this employee has been worked in.

Tuple timestamping keeps the relation in the first-normal-form (1NF) [46]. Thus it benefits from all the advantages of traditional database theory and tech-

nology. Attribute timestamping, on the other hand, requires non-first-normal-form (N1NF, nested) relations [46]. N1NF relations are naturally more complex and difficult to implement than 1NF relations. However, attribute timestamping provides more modelling power since each attribute can have its own timestamp which is independent of others.

Several temporal query languages have been suggested, based on either tuple timestamping or attribute timestamping. The basic approaches are the same, i.e., by introducing special-purpose operators to the query languages for querying temporal data. These new operators include "when" [32][55][121], "as-of" [121], "joins" [55], "slice" [32] and "shift" [55].

The example below is the temporal query language TQuel [121]. It is a superset of Quel, the query language of the INGRES system [127]. The first example shows a query with respect to valid time (using a *when* clause), and the second example shows a query with respect to the transaction time (using an *as-of* clause). Both examples are based on a temporal relation *f*.

Example 9.3: What was Mary's salary in October 1993? □

```
retrieve (f.salary)
where f.name="Mary"
when f overlap "October 1993"
```

Example 9.4: What was Mary's salary according to the information stored in the database in October 1993? □

```
retrieve (f.salary)
where f.name="Mary"
as of "October 1993"
```

The common issues in designing temporal query languages are: temporal selection and projection, definition of the Cartesian product operation, expressive power of the query language, homogeneity of tuples, and ease of use.

On implementation aspects of temporal, relational DBMS, several approaches have been suggested to improve the performance of temporal queries:

- Temporal partitioning

  Separate the *historical* data, which grow monotonically, from the *current* data, whose size is fairly stable and whose accesses are more frequent [89]. This separation was shown to significantly improve the performance of some queries [8]. This approach was later generalized to allow multiple cached states, which further improves performance [66].

- Temporal joins

For temporal databases, *join* operations become more complex. This is because the data model contains temporal semantics. New join algorithms proposed for temporal databases include *time-join*, *time-equijoin* (*TE-join*) [31], *event-join*, *TE-outerjoin* [59], *contain-join*, *contain-semijoin*, and *intersect-join* [78]. They are extensions of nested loop or merge joins that exploit sort orders or local workspace. In [79] a join algorithm for multi-processors has been proposed.

- Temporal indexes

  Since temporal data tend to be large in volume, designing suitable indexing methods is more crucial in temporal DBMSs than in conventional DBMSs. Many temporal indexes have been suggested to speed up query processing time. These include the Time Index [45][47], the Append-Only Tree [58], the Monotonic B^+-tree[44], the Time-Split B-tree[88], the Interval B-tree [11], the Time-Polygon index (TP-index)[113], the I-tree [135] and PLI-tree [135], and the Segment Index [72].

  Most of these indexes are designed for the *transaction time* (such as the Append-Only Tree, the Monotonic B^+-tree, the I-tree, and the Time-Split B-tree), others are designed for the *valid time* (such as the Time Index, the Segment Index, and the Time-Polygon index). Most of the indexes use only the timestamp (transaction time or valid time) as the key, others can include a non-temporal attribute as part of the key as well. For example, in the Time Index [45] a two-level index has been proposed where non-temporal keys are indexed in an upper level and temporal keys are indexed in a lower level.

- Query optimization techniques

  Leung and Muntz [78] suggest using the stream processing technique in temporal query evaluation and optimization. There, temporal join and temporal semi-join operations are carried out with a single pass over the input streams, and the amount of workspace required can be small.

Extending the Object-Oriented Data Model

This section shows an example of extending the object-oriented data model to incorporate temporal dimension. The example data model is OODAPLEX [36] that is based on the DAPLEX functional data model [114][120]. OODAPLEX uses the concept of *objects* to model real-world entities, and *functions* to model properties, relationships, and operations of objects; other object-oriented features, such as user-defined abstract data types, sub-typing and inheritance, polymorphism, and late bindings are also supported.

To extend the model in OODAPLEX to deal with temporal data, two generic object types, *time point* and *time point set*, are defined that carry the most general semantics of time. The special semantics for time required by specific

applications are then introduced through abstract data types that are subtypes of these generic time types. The time-varying behavior of an object is modelled by functions that relate time objects to the object.

Example 9.5: The *employee* type (corresponding to Example 9.2) is defined by the following: □

```
type employee is person
    function name (e:employee -> n:string)
    function salary (e:employee ->
                     f:(t: time -> s:money))
    function dept (e: employee ->
                     f:(t: time -> d:department))
```

In this example temporal information for *salary* and *department* are modelled as *functions* that take *time* as arguments.

An example of the query language based on this data model is given below. The function *extent(employee)* returns all the objects of type *employee*, the function *lifespan(e)* returns all the time points at which the object *e* is defined in the database.

Example 9.6: When did John get a salary raise? □

```
for the e in extent(employee)
    where name(e) = "John"
        for each t in lifespan(e)
            where salary(e)(t) > salary(e)(t-1)
        end
    name(e)
end
```

It can be seen that in the data model of OODAPLEX, no special time-oriented constructs are needed in the query language. This is because: 1) non-temporal and temporal data are treated in the same way (as objects and types); 2) all the properties and behavior of objects, including the time-varying aspects, are uniformly modelled by functions. Hence, the retrieval and manipulation of temporal and non-temporal information are *uniformly* expressed. Some queries that cannot be expressed in the extended relational languages can be expressed in this model. Therefore, this query language is rather general and powerful in expressing temporal queries. The price to pay is the difficulty in query processing.

9.4.3 Temporal Databases and Time Series Management

At first glance, one might think that time series are just a special case of temporal data which can be managed easily by temporal database management systems (TDBMSs). Unfortunately, it is not true. According to the research paper "Time Series, a Neglected Issue in Temporal Databases Research?" [101], it is argued that the current status of temporal database management systems does not satisfy the requirement of time series management. The reasons are as follows:

Structual aspects

The majority of temporal data models are straightforward extensions of the relational data model in that tuples are associated with some sort of time stamp (mostly intervals). Hence, these data models inherit the well-known limitations of the relational model, in particular the requirement of the first-normal-form (1NF). This is not good since an atomic single-valued attribute which varies over time could naturally be viewed as one complex multi-valued attribute instead of being scattered over a relation.

For example, Fig. 9.7 shows an approach to modelling a time series as a temporal relation:

t	att 1	att 2	...
1	14	"x3"	...
2	36	"y7"	...
3	23	"z3"	...
...	...	...	...

Fig. 9.7: Modelling a time series as a temporal relation

It can be seen that this is a simple and efficient organization. However, for a database containing many thousands of time series, this approach would end up with many thousands of relations. Current TDBMSs usually are not well suited for this since they are designed to work efficiently with hundreds to thousands of relations, but not with ten or a hundred thousand. Furthermore, for every deletion and instantiation of a time series, one would have to change the schema of the database, which is cumbersome.

Another approach would be to define one relation per time series type (see

Fig. 9.8). Because time series are usually read sequentially, it makes sense to sort such a relation by time series identifier and by date. Normally, time series data are rarely updated, with the exception that new events are appended at the end. However, to sort this relation makes appends rather expensive. Not to sort the relation in this way makes appends cheap but sequential access rather expensive. Therefore, neither organization is really satisfying. With a clever primary organization of the relation and with the addition of indexes, some of the drawbacks may be reduced. Unfortunately, usually only some operations benefit while others become more costly, and additional disk space is necessary.

t	ts_id	att 1	att 1	...
1	“TS_1”	14	“x3”	...
2	“TS_1”	36	“y7”	...
1	“TS_2”	234	“uip”	...
2	“TS_2”	327	“ytb”	...
...	...	...	...	...

Fig. 9.8: One temporal relation per time series type

Other possible approaches to modeling time series as temporal relations are discussed in [101]. None of the approaches is satisfactory. The reason is that when we assume that the attributes in a relation are only simple values, the *semantics* of a time series (each attribute is an atomic single-valued attribute which varies over time) in DBMSs is lost. Therefore, update, retrieval, or physical organization of time series are inefficient and cumbersome for end-users. The solution is to allow *arrays* as attributes. Unfortunately, current temporal DBMSs do not support array valued attributes. One could store arrays as BLOBs. However, in this case, no functionality is available to manipulate them.

Time Model

Another difficulty appears in the difference between the time models of temporal databases and time series. Temporal data models typically associate time intervals with the facts stored in the database, given that many values remain constant over long periods of time. Time series instead have the property that data values are collected at specific points in time, and the lifespan of a data value is normally very short. Actually, some data values even change continuously. Therefore, operations such as *temporal joins* are often meaningless in collections of time series because the notion of “concurrent events” is often dif-

ficult if not impossible to define. Furthermore, the notion of a *calendar* is a crucial abstraction in time series applications, because it defines the mapping from time points to positions (indices) within a time series. Although TSQL2 [125] supports the definition of calendars by the user, the proposed internal timestamp format is often too general for time series applications. A related point is that in time series applications, the lifespan of a data value may be very short, as for example, in some stock exchange applications. Therefore, the value of some quantify may be completely unknown except for the few sampling points represented in the database. Moreover, different time series may be associated with different, incompatible calendars.

Functional Aspect

According to [101], temporal DBMSs are not adequate in managing time series in the functional aspect. For example, statistical transformation methods such as moving average are often needed to be applied to time series; while temporal DBMSs (TDBMSs) do not provide the possibility to implement user-defined procedures or methods. TDBMSs rely on a declarative language for data manipulation, which is not suitable for formulating the necessary statistical transformations. Furthermore, *arrays* are not part of the data model of current TDBMSs, functions on array manipulations are missing as well. However, statistical functions are often array manipulations. Therefore, even to carry out basic transformations, the data have to be extracted into some application and stored back into the database afterwards.

With respect to interpolation, TDBMSs often assume the "step-wise constant" to interpret missing values. It is not possible to choose any other interpolation function (like linear or spline interpolation, etc.). Fig. 9.9 illustrates this discrepancy between TDBMSs and time series management systems.

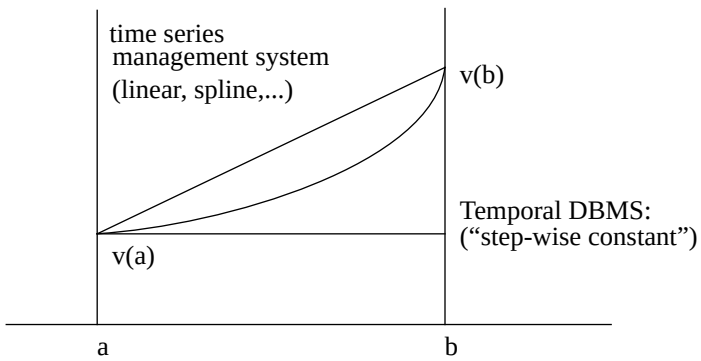


Fig. 9.9: Interpolation approaches in TDBMSs and time series management systems

To conclude, TDBMSs based on extensions of the *relational data model* do not satisfy the special requirements of time series applications: mapping time series into snapshot and temporal relations is intricate, performance is problematic, functionality is only partly adequate, and the capability to organize time series into groups is missing. It seems that *object-oriented DBMSs* or o*bject-relational DBMSs* are more suitable in management of time series information.

It remains an open question whether a DBMS (e.g., Informix's TimeSeries DataBlade) can replace special-purpose management systems (e.g., FAME) in time series applications.

9.5 Summary

In this chapter, we have discussed work related to this thesis in general.

The most relevant related work is the sequence database system SEQ. In SEQ, sequence data were modelled as an *abstract data type* (ADT), and supported by common operators such as subsequence extraction, aggregation, and composition. A sequence query language, *SEQUIN*, was developed to express sequence queries. Important issues such as query optimization were investigated. This system later evolved into PREDATOR [111] to support other types of non-traditional data types such as image, audio, and spatial data.

Another closely related field is *similarity search* on time series, i.e., finding similar patterns in different time series. We presented an overview of various approaches, such as those using the Discrete Fourier Transform (DFT), or transforming time series into some feature-preserved functions, or defining some shape languages to express feature queries. We have pointed out pros and cons of each approach and compare them with our work.

We also provided an overview of time series management systems such as Informix's TimeSeries DataBlade, FAME, and CLANADA. These systems demonstrate the special requirements on time series managements that are not met by conventional DBMSs.

This chapter also includes an overview of research on *temporal databases*, including temporal data models, temporal query languages, and temporal indexes. We have shown that temporal database management systems based on extensions on the *relational data model* are not adequate for managing time series data. Instead, the *object-oriented* and *object-relational* data models are more suitable for time series management.

Chapter 10

Application Study

This chapter presents a thorough application study on *terrain-aided navigation* [20] to show that the IP-index is applicable to other application domains. The IP-index improves the performance of a matching algorithm (the bayesian approach [20]) in terrain-aided navigation by efficiently filtering out sub-areas in a map where the terrain elevations inside these areas are inside some value range (h', h''). The efficiency of this approach is verified by experiments.

10.1 What is Terrain-Aided Navigation

Terrain-aided navigation is to use the terrain height over the mean sea level, the terrain elevation, to draw conclusions about the position of an aircraft. The idea is: A map with sampled terrain elevation measured in a uniformly spaced grid is stored onboard the aircraft. Flying over an area, the aircraft *altitude* over mean sea-level is measured with a barometric sensor and the *ground clearance* is measured with a radar. The difference between the altitude and the ground clearance is an estimate of the *terrain elevation*, which can be compared to the stored values in the map to determine the position of the aircraft, see Fig. 10.1.

Gathering samples as the aircraft flies over an area will produce a trajectory of measured elevations. The more samples gathered, the more likely it is that the trajectory is unique and that a good position estimate may be found when comparing the trajectory with the stored elevations in the map.

Errors and uncertainty exist in the measurements of the barometric sensor and radar. For example, flying through different local weather conditions will cause

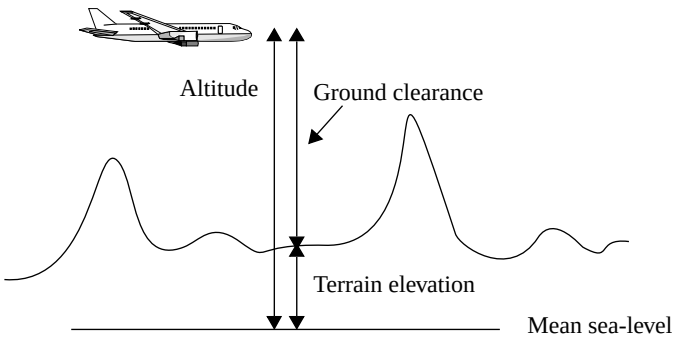


Fig. 10.1: Illustration of the terrain-aided navigation

the barometric sensor to produce biased errors. Thus, the measured terrain elevation is an approximation of the *real* terrain elevation. In order to locate the position of the aircraft on the map, a non-linear matching algorithm (the bayesian approach [20]) which takes into account the probability density function of the measured elevation has been developed [20]. This matching algorithm needs to be applied to every grid of the map and each new measured elevation in the trajectory needs to be processed recursively [20].

10.2 Using the IP-index in Terrain-Aided Navigation

Since the matching algorithm is expensive and has to be applied to every grid of the map, it becomes very inefficient when the number of grids in the map becomes large (see [20]). Therefore, we propose to use the IP-index. The idea is to filter out those sub-areas whose terrain elevations are in some range around the measured values h. We use a confidence interval (h-10, h+10) where the probability that the true elevation value is inside this interval is higher than 99.99% according to the probability density function. Then, the sub-areas whose terrain elevations are inside the interval (h-10, h+10) can be used as starting positions of the matching algorithm (this approach will be further illustrated in Section 10.2.1).

One may argue that a conventionally ordered secondary index is sufficient to efficiently find those sub-areas where the terrain elevations are inside the interval (h-10, h+10). Refer to Section 3.5 for detailed discussions on the differences between the IP-index and a conventional secondary index to see why a conventional secondary index does not work well here.

10.2.1 The Approach

Since the IP-index is designed for 1-D sequences, it cannot be directly applied to the two-dimensional map. The approach is: we view each row of the map as a time sequence (see Fig. 10.2). That is, each grid corresponds to a state in a time sequence, the position identifier *ij* corresponds to the timestamp *t*, and the elevation h_{ij} corresponds to the value *v*. For each time sequence, we pose the interval query $F^{-1}(h\text{-}10 < v < h\text{+}10)$ to get the position intervals (*ij_pos', ij_pos''*) where the values inside these intervals are inside the range (*h', h''*). By rounding these intervals (see Section 5.2) we can get corresponding column intervals [*ij', ij''*]. These column intervals are the sub-areas in this row whose terrain elevations are inside the interval (*h*-10, *h*+10).

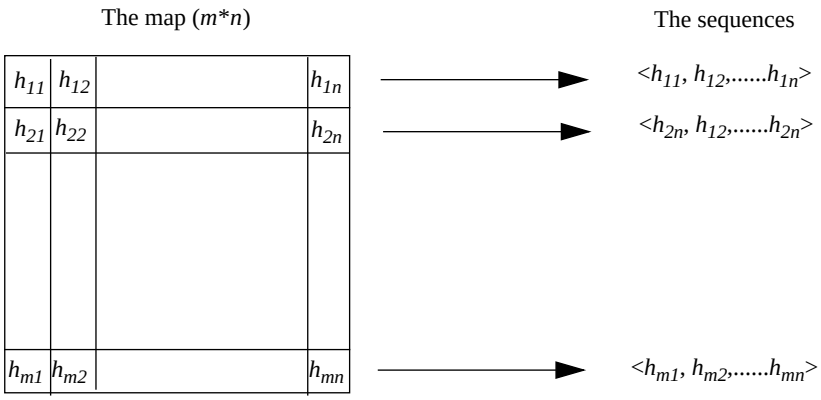


Fig. 10.2: Transformation between the map and the sequences

Now let us look at how the IP-index can improve the efficiency of the matching algorithm in terrain-aided navigation. As we mentioned earlier, the matching algorithm finds the true position of the aircraft by applying a non-linear algorithm to the whole map recursively for each measured elevation in the trajectory. Taking a trajectory of elevation measurements $<h_1, h_2,.....h_k>$, we can use the IP-index to find the sub-areas in the map whose terrain elevations are inside the interval (h_1-10, h_1+10) (this interval is based on the probability density function as explained earlier), thus providing starting positions for the matching algorithm. Since the matching algorithm does not have to be applied to the whole map, efficiency is improved.

10.2.2 Cardinality

We define the *cardinality* of an interval (h_i-10, h_i+10) as the number of grids in the map whose terrain elevations are inside this interval. In the measurements

we found that for a trajectory <h_1, h_2,.....h_k>, the cardinalities of (h_i-10, h_i+10) vary widely (see Fig. 10.4). Since the matching algorithm does not really have to start from the first measurement h_1, i.e., it can start from any sub-areas returned from the interval (h_i-10, h_i+10) ($i \leq k$) and apply the non-linear algorithm backwards (h_{i-1}, h_{i-2},.....h_1) and forwards (h_{i+1}, h_{i+2},.....h_k), we do not always have to take the first interval (h_1-10, h_1+10) to return the sub-areas. Instead we take an interval (h_i-10, h_i+10) in the trajectory which returns a small cardinality (this will be further illustrated in the next section). The matching algorithm starts from these small areas (and applies the non-linear algorithm backwards and forwards in the trajectory) will be much more efficient than starting from the whole map.

The cardinality of the interval (h_i-10, h_i+10) can be computed easily from the IP-index since the cardinality information is stored in the IP-index (Section 3.1). The cardinality of the interval (h'-10, h''+10) can be computed by adding together *cardinality*(k_j) for those keys k_j in the IP-index that satisfy the condition h'-10 < k_j < h' +10. This is efficient since the IP-index is an ordered index.

10.3 Measurements

We have performed performance measurements on a real map and simulated track files to see how efficient the above approach is. In this section we describe the experimental results.

10.3.1 The Real Map

To make the measurements as close to reality as possible, we use a real map over a part of Sweden (see Fig. 10.3) which consists of 101 by 101 samples in a uniformly spaced grid. (It is sampled with 50 m distance between each two sample points, yielding an area of 25 km^2 of terrain.) The elevation of each grid in the map is not the average elevation value over the grid but the measured terrain elevation rather exactly in the center of the grid. Some interpolation method (e.g., linear interpolation) could easily be applied to the map to produce any terrain elevation between the sampled points. As seen in Fig. 10.3, the terrain elevations have different characteristics in different parts of the map. For example, the flat area in the upper left corner of the map is a lake.

10.3.2 The Track Files

50 elevation track files were randomly generated in order to cover different parts of the map. The starting positions of these tracks were uniformly distributed along the leftmost line of the map. Likewise their final positions were uniformly distributed along the parallel finish line on the other side of the grid (see

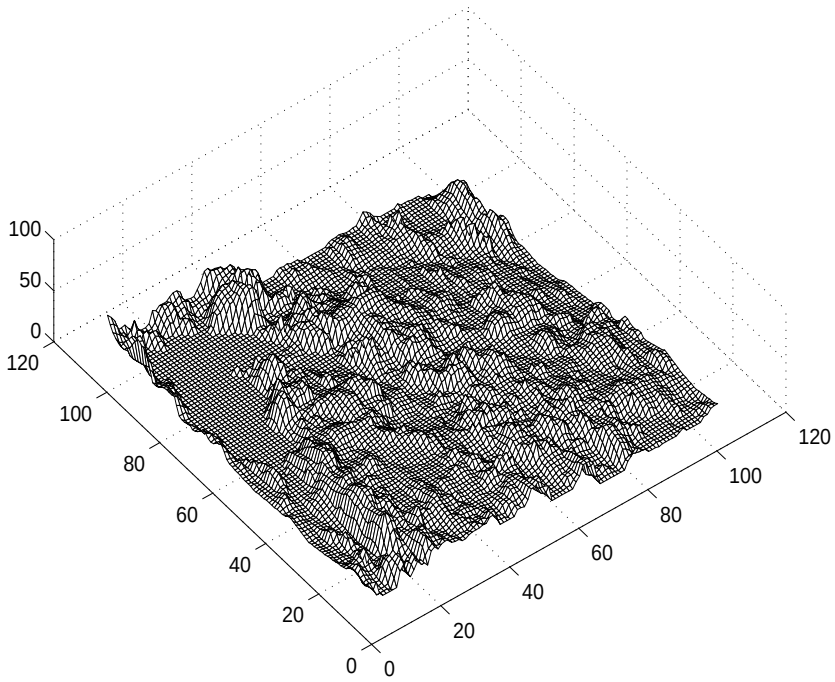


Fig. 10.3: The real map

Fig. 10.3). Each track file represents a trajectory of an aircraft and it contains 80 sampled terrain elevation measurements.

10.3.3 Cardinality

For a trajectory $<h_1, h_2, ...h_k>$, the cardinality of each interval (h_i-10, h_i+10) can vary widely, as shown in Fig. 10.4. As we mentioned in Section 10.2, we would like to take an interval (h_i-10, h_i+10) in the trajectory which returns a rather small cardinality. The more samples (in a trajectory) we take, the higher the probability will be to find a small cardinality. To measure the relationship between the minimum cardinality found and the size of the tracks, we calculate the cardinalities (see Section 10.2) for the first i * 10 (i = 1...8) samples for each track file. We recorded the minimum cardinality found and the corresponding size of the track. Thus, for each track we get a sequence of minimum cardinality min_i (i = 1...8) and the corresponding $size_i$ = i * 10 (i = 1...8). Fig. 10.5 shows the $avg(min_i)$ over the 50 tracks. It shows that: 1) The minimum cardinality decreases with the size of the tracks; 2) After approximate 30 samples, the average of the minimum cardinality reaches a stable value (i.e., a value around 609).

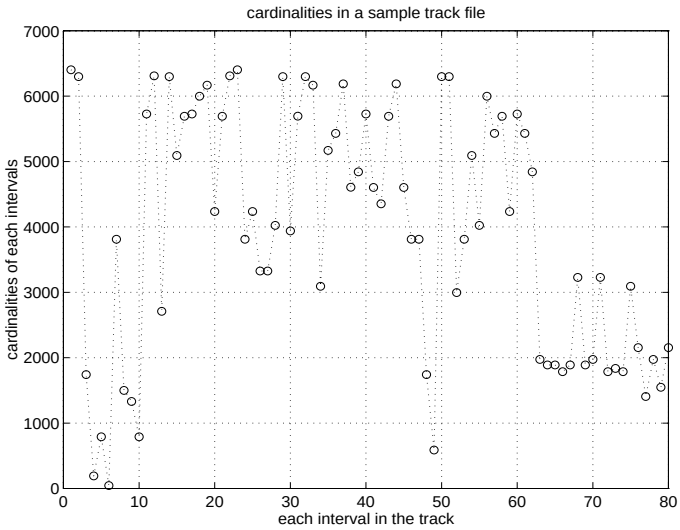


Fig. 10.4: Cardinality distribution for a sample track

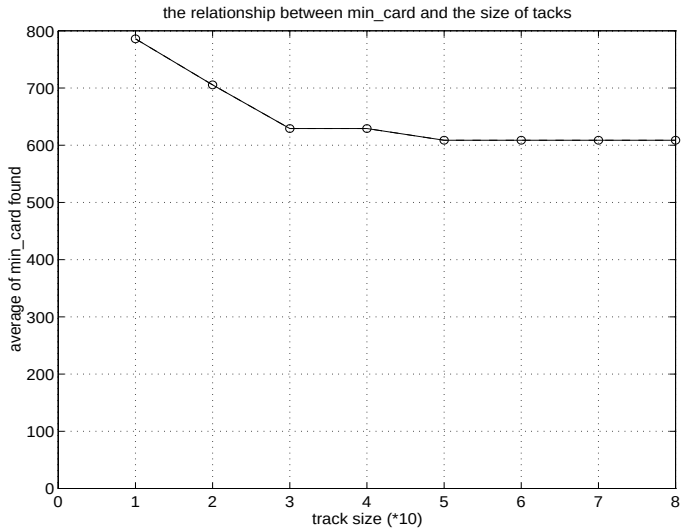


Fig. 10.5: The relationship between the number of samples taken and the average of the minimum cardinality found

10.3.4 The Settling Time of the IP-index

Thus we can use the value 609 as the "converge" threshold for the IP-index, i.e., we say the IP-index has settled when it finds a cardinality less than 609. Taken a trajectory of elevation measurements $<h_1, h_2,...h_k>$, it is interesting to see how fast the IP-index settles. We tested the IP-index on the 50 tracks and recorded the number of samples needed to reach the converge threshold 609. The histogram is shown in Fig. 10.6.

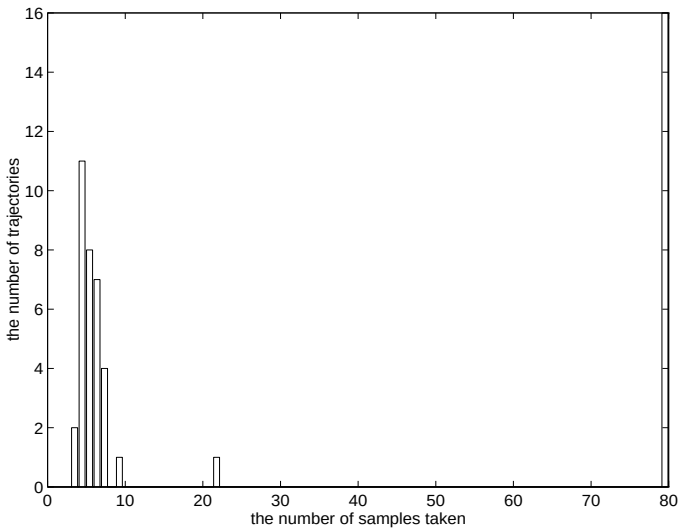


Fig. 10.6: The histogram of the settling time of the IP-index

Fig. 10.6 shows that the settling time of the IP-index is generally small. For most tracks less than 10 samples are needed. Notice that when the track does not cover areas with small cardinalities (for example, when the aircraft is flying over flat areas such as lakes), the cardinality threshold cannot be reached even if the whole track is checked. That is the reason why some tracks are located in the rightmost line (where the settling time is 80) in Fig. 10.6.

10.3.5 Conclusions

Note that the measurement results are affected by the application data. For example, the value of the "converge" threshold (609) used in these measurements is dependent on the elevation values in the terrain map and the position of the tracks (in the map). It should be tuned for different application data sets. This value in turn affects the settling time of the IP-index.

Nevertheless, these measurements show how the IP-index should be used in the navigation application to find good starting positions for the matching algorithm. The approach is: for any track file we calculate the cardinality for each sample h_i, and stop either when we find the first cardinality that is lower than the threshold (i.e. 609 in the example), or when the IP-index reaches its "converge threshold" (i.e. 30 samples in the example since the minimum cardinality will not continue to decrease based on statistics shown in Fig. 10.5). Suppose we stop at the sample h_s in the trajectory, then we pose the range query $F^{-1}(h_s\text{-}10<v<h_s+10)$ to find the sub-areas in the map whose terrain elevations are inside the interval (h_s-10, h_s+10). These sub-areas are returned to the matching algorithm to serve as the starting positions (recall in Section 10.2 that the matching algorithm can work backwards and forwards). Since the number of grids inside these areas is guaranteed to be small, the matching algorithm will always be *much* more efficient than starting from the whole map.

10.4 Summary

To conclude this chapter, we have shown that the IP-index is applicable to other applications domains such as *terrain-aided navigation* [20]. The IP-index improves the performance of the bayesian approach [20] in terrain-aided navigation by efficiently filtering out sub-areas in a map where the terrain elevations inside these areas are inside some range (*h'*, *h''*). The efficiency of this approach was verified by experiments.

Chapter 11

Conclusions and Future Work

*I*n this chapter we conclude this thesis and point out possible future work.

11.1 Concluding Remarks

To conclude this thesis, we have shown that: although considerable research has been dedicated to supporting *sequence data* in DBMSs in the last decade, some important requirements from applications are *neglected*, i.e., how to support sequence data viewed as *continuous* under arbitrary user-defined interpolation assumptions; and how to perform sub-sequence extraction efficiently based on the conditions on the *value* domain (instead of on the ordering domain). To address this challenging problem, we have performed extensive research which results in this thesis. Our main contributions consist of the following:

- We have developed an innovative index, the IP-index, which supports efficient calculation of implicit values of sequence data under arbitrary user-defined interpolation functions. The idea of the IP-index is general and it can be implemented on top of an ordered index such as a B^+-tree.

- We have implemented the IP-index in both a disk-resident database system and a main-memory database system. We have demonstrated by experiments that the insertion and search time of the IP-index remains small regardless of the growing of the time sequence. We also investigated the space usage of the IP-index to show that it is practical to build IP-indexes for large sequences.

- We introduced an extended SELECT operator, σ^*, for the abstract data type

of time sequences. The σ* operator, σ^*_{cond}(TS), retrieves *sub-sequences* (time intervals) where the values inside those intervals satisfy the condition *cond*. Experiments made on SHORE [22] using both synthetic and real-life time sequences showed that the σ* operator (supported by the IP-index) dramatically improves the performance of value queries.

- We developed a cost model for the σ* operator. We also showed that the cost of a range query $\sigma^*_{v>v'}$(TS) is nearly the *same* as the cost of the exact query $\sigma^*_{v=v'}$(TS). This indicates that processing range queries is very efficient using the IP-index, especially for large sequences.

- We investigated optimizations of time window queries and complex value queries (sequence joins). The optimization techniques were verified by experiments.

- We proposed a multi-level dynamic array structure for dynamic, irregular time sequences. The highlight of this data structure is that it is highly *space efficient* and supports both *efficient random access* and *fast appending*.

- We investigated issues such as management of large objects in DBMSs, physical organization of secondary indexes, and the impact of main-memory or disk-resident DBMSs on sequence data structures and indexes.

- We performed a thorough application study on "terrain-aided navigation" [83] to show that the IP-index is applicable to other application domains. Experiments on a real terrain map and simulated track files were performed to verify the efficiency of the approach.

We also performed an extensive study on related work to give the readers an overview of research work done in this area. Closely related work was studied in depth.

11.2 Future Work

For future work, we would like to further investigate optimizations for complex sequence queries when the σ* operator is involved. Query optimization for sequence data is an interesting and challenging issue, especially for *continuous* sequences when user-defined interpolation functions are supported. For example, optimization of sequence joins (see Section 8.6) would require the combined knowledge on the cost and selectivity factors of the σ* operators involved, and different join techniques.

Another interesting direction for future work would be to investigate how to extend the idea of the IP-index to multi-dimensional sequence data. This is certainly a highly challenging subject.

Appendix

SHORE Implementation Notes

This appendix describes how the IP-index and time sequences were implemented on SHORE (to complement Section 4.2.1).

SHORE (Scalable Heterogeneous Object REpository) [22] is a persistent object system developed at the University of Wisconsin. SHORE represents a merger of object-oriented databases and file system technology. SHORE evolved from an earlier object-oriented database system called EXODUS [25].

SHORE is a peer-to-peer distributed system (see Fig. 1). Each node where objects are stored or where an application program wishes to execute contains a SHORE server process that talks to other SHORE servers, interfaces to locally executing applications, and cashes data pages and locks in order to improve system performance. A SHORE server communicates with the local application through RPC and shared-memory.

To allow databases built by an application written in one language (e.g. C++) to be accessed and manipulated by applications written in other object-oriented languages (e.g., Smalltalk), SHORE defines its type system in SHORE Data Language, SDL. SDL is quite similar to ODL (the Object Definition Language) proposed from the ODMG consortium.

All persistent objects in SHORE are defined in SDL as instances of *interface types.* An interface definition is similar to a class definition in C++. Interface types can have attributes, methods, and relationships. The *attribute* of an interface type can be of one of the primitive types (e.g., integer, character, real), or they can be of constructed types. SHORE provides the usual type of constructors: *enumeration*, *structures*, *arrays*, and *references* (which are used to define relationships). In addition, SHORE provides a variety of *bulk types*, including *lists*, *sets*. and *sequences*, that enable a SHORE object to contain a collection of

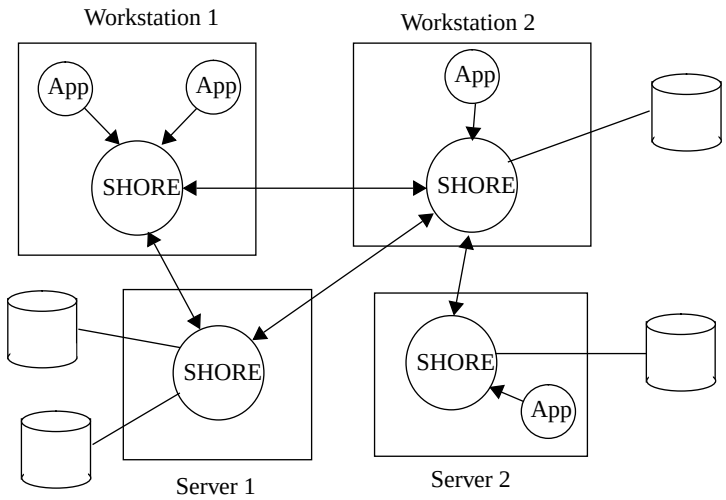


Fig. 1: The SHORE process architecture

references to other objects. In particular, a *sequence* is a homogeneous sequence of values of a base type. A sequence can grow arbitrarily large.

SHORE also provides the notion of *modules*, to enable related types to be grouped together for name scoping and type management purpose.

Interface Types in SDL

To implement the IP-index and the data type of *time sequence* in SHORE, the first thing is to define the *interface types* for them. The IP-index was implemented using the B^+-tree interface [116] provided by SHORE. In SHORE, indexes such as B^+-trees or R-trees are declared in SDL as `index<keytype, valtype> varname`. A B^+-tree in SHORE is initialized by the `init(index_type)` statement where the parameter `index_type` can be `"BTree"` or `"UniqueBTree"`. Scanning the index is accomplished by the template class `IndexScanIter(const Index<key, val> idx)`. This template opens a "cursor" to indicate the current (key, value) pair in the range and a `next` member function to move the cursor to the next pair. Range bound may be specified by `SetUB` and `SetLB` member function.

A time sequence was modelled as an array of records (t_i, v_i) (for reasons see Section 4.2.1). Since anchor-state sequences are dynamically growing, they cannot be modelled as static objects such as fixed-sized arrays or unordered objects such as sets or bags. Instead, they are implemented as a sequence data structure `Seq` [116] in SHORE. `Seq` is a dynamic array (variable-length array) stored on disk. It supports operations such as `append_elt, get_ele, get_size,`

`delete_elt`, etc.

Member functions that do not update the contents of an object are flagged as `const` in their SDL definition.

1. The `Time_sequence` interface:

   A time sequence is implemented as an array of state_ids where each state_id S_i is a pointer that points to where the pair (t_i, v_i) is stored. There are three methods defined on the time sequence interface type: *insert(v)*, *get_state_value(i)*, and *print_time_sequence()*.

```
interface Time_sequence {
   public:
       typedef ref<State> state_id;
       attribute state_id state_ids[100000];
           // an array of references to the "state" object!
       attribute int curr_length;
           // current length!
       int insert(in float v);
           // "in" means this function will not modify the argu-
              ment!
       void print_time_sequence() const;
       float get_state_value(in int i) const;
           // retrieves the value v_i given the state_id i!
};
```

2. The `Anchor_state_sequences` interface:

   An anchor-state sequence is stored using a sequence data structure in SHORE. This interface contains four methods: *initialize()*, *initialize(seq)*, *append(s)*, and *print_anchors()*. The method *initialize*() initializes a *nil* sequence and the method *initialize*(*seq*) initializes the anchor-state sequence with a known sequence *seq* (by copying that sequence).

```
interface Anchor_state_sequence {
   public:
       attribute sequence<int> anchors;
           // sequence of state_id!
       void initialize();
           // initialize with nil!
       void initialize(in sequence<int> seq);
           // initialize with sequence "seq"!
       void append(in int s);
       void print_anchors() const;
 };
```

3. The `IP_index` interface:

   The IP-index is stored as *index<float, ref<Anchor_state_sequence>>*. The

structure *index<float, ref<Anchor_state_sequence>>* indicates that this index has floating point numbers as keys and each key is associated with a pointer that points to an *Anchor_state_sequence*. The `IP_index` interface has the following methods: *initialize(), insert(i, v), search(v), print_ip_index(), modify_ip_index(v1, v2, s), get_left_entry(v),* and *simulate_back_scan(v).* The method *insert(i, v)* inserts a new state (t_i, v_i) into the IP-index where $t_i = i$ (for simplicity we use integers as time stamps, see Section 4.2.1). The method *search(v)* searches the IP-index to find the anchor-state sequence A(v) for the value *v.* The method *print_ip_index()* prints the IP-index by traversing the IP-index tree. The method *modify_ip_index(v1, v2, s)* appends the anchor-state sequences of those k_is where $v1 < k_i < v2$ with the state s. The method *get_left_entry(v)* returns the entry in the IP-index with the key value k_i where $k_i \le v < k_{i+1}$. The method *simulate_back_scan(v)* will be explained in the section "C++ binding" later. For now we can ignore it.

```
interface IP_index {
   public:
       attribute index<float, ref<Anchor_state_sequence> > ind;
       void insert(in int i, in float v);
       void search(in float v) const;
       leftmost get_left_entry(in float v) const;
       leftmost simulate_back_scan(in float v) const;
       void modify_ip_index(in float v1, in float v2, in int s);
       void initialize();
       void print_ip_index() const;
           // "const" means this function will not modify the
               object!
  };
```

The IP_index Module

There is only one module in this implementation. The module name is `IP_index`. The `IP_index` module contains the following interface types and data structures:

```
module ip_index {
   interface IP_index;
   interface Anchor_state_sequence;
   interface Time_sequence;

   struct leftmost {
       char flag;
       sequence<int> anchors;
  };
};
```

All interface types in this module have been explained in the previous section. The structure *leftmost* is mainly used to store the anchor-state sequence returned by the method *get_left_entry(v)*. (The *flag* field is used to mark some internal conditions in the program.) The `IP_index` module is stored in a file named *ip_index.sdl*. This file is compiled into a C++ header file *ip_index.h*, which can be included in C++ programs, see the next section.

C++ Binding

SHORE is designed to allow databases built by an application written in one language (e.g., C++) to be accessed and manipulated by applications written in other object-oriented languages as well (e.g., Smalltalk). This capability is important for large-scale applications, where different modules are probably written in different languages. In SHORE, the methods associated with SDL interfaces can therefore be written using any of the languages for which a SHORE language binding exists. Currently, only the C++ binding is supported. For the SDL file *ip_index.sdl* (the last section), part of the generated header file *ip_index.h* file is shown in Fig. 2.

Some of the data types in Fig. 2 correspond directly to SDL types, as C++ offers direct support for those simple types. For SDL types with no corresponding C++ types, like *sequences* and *references*, SHORE uses pre-defined, macro-based classes (similar to parameterized types) such as `Ref` and `Sequence`. For example, the SDL type `ref<Anchor_state_sequence>` is compiled into the class `Ref(Anchor_state_sequence>;` C++ overloading features make it behave like a pointer to a read-only instance of `Anchor_state_sequence`. The class `Sequence<long>` encapsulates a data structure containing a sequence of long integers and provides member functions that enable its contents to be accessed (such as `append_elt, get_ele, get_size,` and `delete_elt).`

Given the header file generated by the binder, the application program can implement the operations associated with each interface type. For example, the following C++ code implements the member function *print_time_sequence()* for the class `Time_sequence`:

```
  void Time_sequence::print_time_sequence() const {
     int i;
     float v;

     for (i = 0; i<curr_length; i++) {
     v = this->get_state_value(i);
     cout << v;
     cout << " ";
  }
}
```

```
class IP_index: {
public:
   Index<float, Ref<Anchor_state_sequence> >  ind;
   virtual void insert(long i, float v);
   virtual void search(float v) const;
   virtual struct leftmost get_left_entry(float v) const;
   virtual struct leftmost simulate_back_scan(float v) const;
   virtual void modify_ip_index(float v1, float v2, long s);
   virtual void initialize();
   virtual void print_ip_index() const;
};

class Anchor_state_sequence: {
public:
   Sequence<long> anchors;
   virtual void initialize(Sequence<long> seq);
   virtual void initialize();
   virtual void append(long s);
   virtual void print_anchors() const;
};

class Time_sequence: {
public:
   typedef Ref<State> state_id;
   Ref<State> state_ids[100000];
   long curr_length;
   virtual long insert(float v);
   virtual void print_time_sequence() const;
   virtual float get_state_value(long i) const;
};
```

Fig. 2: The *ip_index.h* file (generated from *ip_index.sdl*)

The above `const` flag for this function denotes that this function does not update the contents of the object. To modify an object, the C++ application must first call a special generated member function, `update()`, which returns a read-write reference. For example:

```
Ref<Time_sequence> ts;
ts.update()->insert(v);
```

The function `update()` coerces the type of `ts` from `Ref<Time_sequence>` to `(non const) Time_sequence *`. It also has the runtime effect of marking the referenced object as “dirty” so that changes will be transmitted to the server when the transaction commits.

The B$^+$-tree in SHORE

The IP-index is implemented on top of a B$^+$-tree in SHORE. A B$^+$-tree in SHORE is initialized by the `init(index_type)` statement where the parameter `index_type` can be `"BTree"` or `"UniqueBTree"`. Scanning the index is accomplished by the template class `IndexScanIter(const Index<key, val> idx)` which opens a "cursor" to indicate the current (key, value) pair in the range and a `next` member function to move the cursor to the next pair. Range bound may be specified by `SetUB` and `SetLB` member function.

An IP-index is initialized by the following code:

```
void IP_index::initialize() {
    SH_DO(ind.init(UniqueBTree));
}
```

The macro `SH_DO` [116] is used for calling functions that are not expected to fail. It evaluates its argument and verifies that the result is valid. If not, it prints an error message and aborts the program.

An index scan is accomplished by the following code:

```
IndexScanIter<float, Ref<Anchor_state_sequence> >
       iter(this->ind);
iter.SetLB(v);
iter.SetLowerCond(geOp);
rc = iter.next();
```

The above code sets the lower bound to *v* (no upper bound is specified). The condition for the lower bound is '>=' (the parameter "geOp" above). The limitation of scanning a B$^+$-tree in SHORE (given a key range) is that scanning *has* to start from the lower (key) bound and moves toward the upper bound. Backward scanning (scanning that starts from the upper bound and moves toward the lower bound) is not supported. Unfortunately, in the algorithm of the IP-index (see Section 3.3), we need backward scanning (see the function *get_lower*(*tree*, v_i) in Section 3.3) to find the first k_i where $k_i \leq v_i$ (of course we could start scanning from the minimum key value to find the *last* k_i where $k_i \leq v_i$, but this will certainly be very slow). In order to be able to get the right measurement figures, we had to simulate the situation when the backward scan was available. This is why we have the method *simulate_back_scan(v)* in the *IP_index* interface. The code for this method can be found in [84].

Conclusions

SHORE is a state-of-the-art OODBMS designed for distributed applications. Its separation between the SDL language and implementation languages conforms to the spirit of the ODMG standard for open systems. However, there are several limitations in the current version of SHORE. For example, page faulting for large objects is not supported, and index scan starting from the upper bound toward the lower bound is not available.

Bibliography

1. H. Abelson and G. J. Sussman, "*Structure and Interpretation of Computer Programs*." MIT Press, 1985.

2. G. M. Adelson-Velskii and E. M. Landis, "*Doklady Akademia Nauk SSSR"*, 146, 1962, pp. 263-266; English translation in *Soviet Math*, 3, pp. 1259-1263.

3. M. E. Adiba and B. G. Lindsay, "Database Snapshots," *Proceedings of 6th VLDB Conference*, pp. 86-91, 1980.

4. R. Agrawal, C. Faloutsos and A. Swami, "Efficient Similarity Search in Sequence Databases," in *Proceedings of the 4th International Conference on Foundations of Data Organization and Algorithms*, pp. 69-84, Chicago, Oct. 1993.

5. R. Agrawal, K. Lin, H. S. Sawhney, and K. Shim, "Fast Similarity Search in the Presence of Noise, Scaling, and Translation in Time-Series Databases," in *Proceedings of 21st VLDB Conference*, pp. 490-501, 1995.

6. R. Agrawal, G. Psaila, D. L. Wimmers and M. Zaït, "Querying Shapes of Histories," in *Proceedings of 21st VLDB Conference*, pp. 502-514, 1995.

7. I. Ahn and R. Snodgrass, "Performance Analysis of Temporal Queries," in *Information Science*, vol. 49, pp. 103-146, 1989.

8. I. Ahn and R. Snodgrass, "Partitioned Storage for Temporal Databases," in *Information Systems*, 13(4):369-391, 1988.

9. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, "*Data Structures and Algorithms*." Addition-Wesley, 1987.

10. A. Ammann, M. Hanrahan and R. Krishnamurthy, "Design of a Memory Resident DBMS," in *Proceedings of IEEE COMPCON*, San Francisco, February, 1985.

11. C-H. Ang and K-P. Tan, "The interval B-tree", in *Information Processing Letters*, 53(2): 85-89, Jan. 1995.

12. M. Astrahan et al., "System R: Relational Approach to Database Management," in *ACM TODS*, Vol. 1, No. 2, June 1976.

13. M. Atkinson, et al., "The Object-Oriented Database System Manifesto," in *Proceedings of the First International Conference on Deductive and Object-Oriented Databases*, Kyoto, Japan, Dec. 1989.

14. F. Bancilhon, C. Delobel, and P. Kanellakis (eds), "*Building an Object-Oriented Database System: The Story of O2*". Morgan Kaufmann Publishers, 1992.

15. D. S. Batory, T. Y. C. Leung, and T. E. Wise, "Implementation Concepts For an Extensible Data Model and Data Language", in *ACM Transactions on Database Systems*, 13(3):231-262, Sept. 1988.

16. R. J. Bayardo Jr., D. P. Miranker, "Processing Queries for First-Few Answers," in *Proceedings of 5th International Conference on Information and Knowledge Management*, pp 45-52, Maryland, USA, Nov. 1996.

17. R. Bayer and E. McCreight, "*Organization and Maintenance of Large Ordered Indices*," Technical Report No. 20, Boeing Scientific Research Laboratories, July, 1970.

18. N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger, "The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles," in *Proceedings of the 1990 ACM SIGMOD Conference*, pp. 322-331, June 1990.

19. J. L. Bently, "*Algorithms for Klee's Rectangle Problems*", Technical Report, Computer Science Department, Carnegie-Mellon University, Pittsburgh, 1972.

20. N. Bergman, "*A Bayesian Approach to Terrain-Aided Navigation*", Technical Report, LiTH-ISY-R-1903, Linköping University, Oct. 1996.

21. C. Bettini, X. S. Wang, E. Bertino and S. Jajoda, "Semantic Assumptions and Query Evaluation in Temporal Databases," in *Proceeding of the 1995 SIGMOD Conference on the Management of Data*, May 1995.

22. M. J. Carey, et. al, "Shoring Up Persistent Applications," in *Proceeding of the 1994 SIGMOD Conference on the Management of Data*, Minneapolis, MN, May 1994.

23. M. J. Carey, D. J. DeWitt, J. E. Richardson, and E. J. Shekita, "Storage Management for Objects in EXODUS," in "*Object-Oriented Concepts, Databases, and Applications*," by W. Kim and F. Lochovsky, eds., Addison-Wesley Publishing Co., 1989.

24. M. J. Carey, D. J. DeWitt, J. e. Richardson, and E. J. Shekita, "Object and File Management in the EXODUS Extensible Database System," in *Proceedings of the 12th VLDB Conference*, Kyoto, Japan, 1986.

25. M. Carey et. al, "The Architecture of the EXODUS Extensible DBMS," in *Proceedings of the International Workshop on Object-Oriented Database Systems*, Asilomar, Califonia, 1986.

26. R. G. G. Cattel, "*Object Data Management*". 2nd edition, Addison-Wesley Publishing Company, ISBN 0-201-54748-1, 1994.

27. D. Chanberlin, "*Using the New DB2*." Morgan Kaufmann, 1996.

28. R. Chandra and A. Segev, "Managing Temporal Financial Data in an Extensible Database," in *Proceedings of 19th VLDB Conference*, Dublin, 1993.

29. J. Chomicki, "Temporal Query Languages: a Survey," in *Proceedings of the International Conference on Temporal Logic*, Bonn, Germany, July 1994.

30. H-T Chou et. al, "Design and Implementation of the Wisconsin Storage System," in *Software Practice and Experience*, Vol. 15, No. 10, Oct. 1985.

31. J. Clifford and A. Croker, "The historical relational data model (HRDM) and algebra based on lifespans," in *Proceedings of the 3rd International Conference on Data Engineering*, pp. 528-537, Log Angeles, CA, Feb. 1987.

32. J. Clifford and A. U. Tansel, "On an algebra for historical relational databases: Two views," in *Proceedings of ACM SIGMOD Conference*, pp. 247-265, Austin, TX, May 1985.

33. J. Clifford and D. S. Warren, "Formal Semantics for Time in Databases," in *ACM Transactions on Database System*, Vol 8, No. 2, June 1983.

34. E.F Codd, "A Relational Model of Data for Large Shared Data Banks," in *Communications of the ACM*. 13(6):377-387, June 1970.

35. D. Comer, "The Ubiquitous B-Tree," in *ACM Computing Surveys*, vol. 11, No. 2, pp. 121-137, June 1979.

36. U. Dayal, "Queries and Views in an Object-Oriented Data Model," in *Proceedings of the 2nd Workshop on Database Programming languages*, 1989.

37. D. J. Dewitt et al., "Implementation techniques for Main Memory Database Systems," in *Proceedings of ACM SIGMOD Conference*, June, 1984.

38. D. J. Dewitt, N. Kabra, J. Luo, J. M. Patel and J. Yu, "Client-Server Paradise," in *Proceedings of VLDB Conference, Santiago*, Chile, 1994.

39. K. Dittrich, A. Kotz, and J. Mulle (editors), *Proceedings of the International Workshop on Object-Oriented Database Systems*, IEEE CS, Pacific Grove, California, September 1986.

40. W. Dreyer, A. K. Dittrich, and D. Schmidt, "An Object-Oriented Data Model for a Time Series Management System," in *Proceedings of International Conference on Scientific and Statistic Database Management*, Charlottesville, Virginia, USA, 1994.

41. W. Dreyer, A. K. Dittrich, and D. Schmidt., "Research Perspectives for Time Series Management Systems," in *SIGMOD Record* 23(1): 10-15 (1994).

42. W. Dreyer, A. K. Dittrich, and D. Schmidt., "Using the CALAND A Time Series Management System," in *Proceedings of ACM SIGMOD*, San Jose, CA, 1995.

43. H. Edelsbrunner, "*Dynamic Rectangle Intersection Searching*," Technical Report, Institute for Information Processing, Rept. 47, Technical University of Graz, Graz, Austria.

44. R. Elmasri, M. Jaseemuddin, and V. Kouramajian, "Partitioning of Time Index for Optical Disks". In *Proceedings of the 8th International Conference on Data Engineering*, Feb. 1992.

45. R. Elmasri, Y. J. Kim, and G. T. J. Wuu, "Efficient Implementation Techniques for the Time Index." In *Proceedings of the 7th International Conference on Data Engineering*, pp. 102-111, 1991.

46. R. Elmasri and S. B. Navathe, "*Fundamentals of Database Systems*." The Benjamin/Cummings Publishing Company, Inc. ISBN -201-53090-2. 2nd edition, 1994.

47. R Elmasri, G. T. J. Wuu and V. Kouramaijian, "The Time Index and the Monotonic B^+-tree," in [134], pp. 433-455.

48. Lars Eriksson and Lars Nielsen, "Ionization Current Interpretation for Ignition Control in Internal Combustion Engines," in *IFAC Control Engineering Practice*, Vol. 5, No. 8, August. 1997.

49. G. Fahl, T. Risch and M. Sköld, "An Architecture for Active Mediators," in *Proceedings of the International Workshop on Next Generation Information Technologies and Systems*, Haifa, Israel, 1993.

50. E. T. Falkenroth, "Computational Indexes for Time Series," in *Proceedings of 8th International Conference on Scientific and Statistical Database Management*, pp. 18-23, Stockholm, Sweden, June 1996.

51. C. Faloutsos, M. Ranganathan and Y. Manolopoulos, "Fast Subsequence Matching in Time-Series Databases," in *Proceedings of 1994 ACM SIGMOD*, Minneapolis, Minnesota, May, 1994.

52. FAME Software Corporation, "*User's Guide to FAME*," 1990.

53. R. A. Finkel and J. L. Bentley, "Quad trees: a data structure for retrieval on composite keys," in *Acta Informatica*, 4(1), 1-9.

54. D. H. Fishman et. al, "Overview of the Iris DBMS", in W. Kim, F. H. Lochovsky (eds.), "*Object-Oriented Concepts, Databases and Applications*," ACM Press, Addison-Wesley Publishing Co., 1989.

55. S.K. Gadia and C. S. Yeng, "A Generalized Model for a Relational Temporal Databases," in *Proceedings of 1988 ACM SIGMOD*, Chicago, IL, June 1988.

56. H. Garcia-Molina, and K. Salem, "Main Memory Database Systems: A Overview", in *IEEE Transactions of Knowledge and Data Engineering*, Vol. 4, No. 6, Dec. 1992.

57. N. H. Gehani, H. V. Jagadish, and O. Shmueli, "Composite Event Specification in Active Databases: Model and Implementation," in *Proceedings of the International Conference on Very Large Databases*, 1992.

58. H. Gunadhi and A. Segev, "Efficient Indexing Methods for Temporal Relations," in *Transactions of Knowledge and Data Engineering*, Vol. 5, No. 3, pp. 496-509, June 1993.

59. H. Gunadhi and A. Segev, "Query Processing Algorithms for Temporal Intersection Joins," in *Proceedings of the 7th International Conference on Data Engineering*, Kobe, Japan, 1991.

60. J. Guttag, "Abstract Data Types and the Development of Data Structures," in *Communications of the ACM*, June 1997.

61. A. Guttman, "R-Tree: A Dynamic Index Structure for Spatial Searching," in *Proceedings of ACM SIGMOD Conference*, Boston, MA, June 1984.

62. R. L. Haskin and R. A. Lorie, "On Extending the Functions of a Relational Database System," in *Proceedings of ACM SIGMOD*, June, 1982.

63. IBM Almaden's research group on data mining, "http://www.almaden.ibm.com/cs/quest/".

64. Illustra Information Technologies, "*Illustra User's Guide.*" June 1994.

65. Informix Software. *Informix Time Series DataBlade Module*. 1997.

66. C. S. Jensen, L. Mark, N. Roussopoulos, and T. K. Sellis, "Using Cashing, Cache Indexing, and Differential Techniques to Efficiently Support Transaction Time," in *VLDB Journal*, 1992.

67. C. S. Jensen, J. Clifford, S. K. Gadia, A. Segev, and R. T. Snodgrass, "A Glossary of Temporal Database Concepts," in *SIGMOD RECORD*, Vol 21, No. 3, pp. 35-43, Sept. 1992.

68. F. Johnsson, R. C. Zijerveld, C. M. van den Bleek, J. C. Schouten and B. Leckner, "*Characterization of Fluidization Regimes in Circulating Fluidized Beds - time series analysis of pressure fluctuations.*", Technical Report, Chalmers Institute of Technology, Sweden, 1996 (submitted for publication).

69. N. Kline and R. Snodgrass, "Computing Temporal Aggregates," in *Proceedings of Data Engineering Conference*, pp. 222-231, 1995.

70. D. E. Knuth, "*The Art of Computer Programming, Vol. 1, Fundamental Algorithms*", Addison-Wesley Publishing Co., 1969.

71. P. D. L. Koch, "Disk File Allocation Based on the Buddy System," in *ACM TOCS*, Vol. 5, No. 4, November 1987.

72. C. P. Kolovson and M. Stonebraker, "Segment Indexes: Dynamic Indexing Techniques for Multi-Dimensional Interval Data," in *Proceedings of ACM SIGMOD Conference*, pp. 138-148, 1991.

73. R. Laurini and D. Thompson, "*Fundamentals of Spatial Information Systems*." Academic Press, 1992.

74. T. J. Lehman and B. G. Lindsay, "The Starburst Long Field Manager," in *Proceedings of the 15th VLDB Conference*, Amsterdam, 1989.

75. T. J. Lehman and M. J. Carey "A Study of Index Structures for Main Memory Database Management Systems," in *Proceedings of the 12th International Conference on Very Large Data Bases*, Kyoto, Japan, August, 1986.

76. T. J. Lehman and M. J. Carey, "Query Processing in Main Memory Database Management Systems," in *Proceedings ACM SIGMOD Conference*, Washington DC, May, 1986.

77. M. Leland and W. Roome, "The Silicon Database Machine," in *Proceedings of 4th International Workshop on Database Machines*, Grand Bahama Island, March 1985.

78. T. Y. C. Leung and R. R. Muntz, "Generalized Data Stream Indexing and Temporal Query Processing," in *2nd International Workshop on Research Issues in Data Engineering: Transaction and Query Procossing*, Feb. 1992.

79. T. Y. C. Leung and R. R. Muntz, "Temporal Query Processing and Optimization in Multiprocessor Database Machines," in *Proceedings of the 1992 VLDB Conference*, Vancouver, Canada, 1992.

80. C. S. Li, P. S. Yu and V. Castelli, "HierachyScan: A Hierachical Similarity Search Algorithm for Databases of Long Sequences," in *Proceedings of Data Engineering Conference*, Feb. 1996.

81. L. Lin, T. Risch, and D. Badal, "*Indexing Interpolated Time Sequences*." Technical Report, LiTH-ID A-R-96-03, Linköping University, Jan. 1996.

82. L. Lin, T. Risch, M. Sköld, and D. Badal, "Indexing Values of Time Sequences," in *Proceedings of 5th International Conference on Information and Knowledge Management*, Rockville, USA, Nov. 1996.

83. L. Lin and T. Risch, "Using a Sequential Index in Terrain-aided Navigation," in *Proceedings of 6th International Conference on Information and Knowledge Management*, Las Vegas, USA, Nov. 1997.

84. L. Lin, "Implementing the IP-index in SHORE", in *Linköping Electronic Press*, "http://www.ep.liu.se/ea/cis/1997/017/", 1997.

85. L. Lin, “Study of Supporting Sequences in DBMSs — Data Model, Query Language, and Storage Management,” in *Linköping Electronic Press*, Vol. 3, Nr. 4, 1998.

86. L. Lin and T. Risch, “Querying Continuous Time Sequences,” in *Proceedings of 24th International Conference on Very Large Data Bases*, New York City, USA, August, 1998.

87. B. Liskov and S. Zilles, “Programming with Abstract Data Types,” in *SIGPLAN Notices*, April 1974.

88. D. B. Lomet and B. Salzberg, “The Performance of a Multiversion Access Method,” in *Proceedings of ACM SIGMOD Conference*, Atlantic City, NJ, May 1990.

89. V. Lum, P. Dadam, R. Erbe, J. Guenauer, P. Pistor, G. Walch, H. Werner, and J. Woodfill, “Designing DBMS Support for the Temporal Dimension,” in *Proceedings of ACM SIGMOD Conference*, Boston, MA, July 1984.

90. P. Lyngbaek et al., “*OSQL, A Language for Object Databases*.” Technical Report, HP Labs., HPL-DTD-91-4, Jan. 1991.

91. E. McCreight, “Priority Search Trees,” in *SIAM Journal of Computing*, 14(2):257-276, May 1985.

92. Z. Michalewics (ed.), “*Statistical and Scientific Databases*.” The Ellis Horwood Limited, ISBN 0-13-850652-3, 1991.

93. A. Nanopoulos and Y. Manolopoulos, “Indexing Time-Series Databases for Inverse Queries,” in 1998 *International Conference on Database and Expert System applications*, Vienna, Austria, 1998.

94. K. Ooi, B. McDonell and R. Sacks-Davis, “Spatial kd-tree: Indexing Mechanism for Spatial Database,” in *IEEE COMPSAC 87*, 1987.

95. Oracle Corporation, “*Oracle Time Series Cartridge User’s Guide*.” 1997.

96. D. S. Parker, “Stream Data Analysis in Prolog,” in *The Practice of Prolog*. MIT Press, Cambridge, MA, 1990.

97. D. S. Parker, R. R. Muntz, and H. L. Chau, “The Tangram Stream Query Processing System,” in *Proceedings of the International Conference on Data Engineering*, Los Angels, CA, Feb. 1989.

98. PREDATOR project web page, “http://simon.cs.cornell.edu/Info/Projects/PREDATOR”.

99. F. P. Preparata and M. I. Shamos. “*Computational Geometry.*” Springer-Verlag, ISBN 3-540-96131-3, 1985.

100. N. Sarnak and R. Tarjan, “Planar Point Location Using Persistent Search Trees,” in *Communications of ACM*, 29(7):669-679, 1986.

101. D. Schmidt, A. K. Dittrich, W. Dreyer, and R. Marti, "Time Series, a Neglected Issue in Temporal Database Research?" in *Proceedings of the International Workshop on Temporal Databases*, Zurich, Switzerland, Sept. 1995.

102. B. Schueler, "Update Reconsidered," in G.M. Nijssen (ed.), "*Architecture and Methods in Data Base Management Systems*." North Holland, 1977.

103. P. Schwarz et. al, "Extensibility in the Starburst Database System," in *Proceedings of the International Workshop on Object-Oriented Database Systems*, Asilomar, Califonia, 1986.

104. A. Segev and H. Gunadhi, "Event-join optimization in temporal relational databases," in *Proceedings International Conference Very Large Data Bases*, Sept., 1989.

105. A. Segev and A. Shoshani, "A Temporal Data Model Based on Time Sequences," in [134], pp. 248-269.

106. A. Segev and R. Chandra, "A Data Model for Time-Series Analysis," in Workshop on Current Issues in Databases and Applications, Rutgers Univ., Oct. 1992. Appear in: "*Advanced Database Systems*." Editors: N. Adam and B. Bargrava. Lectures Notes in Computer Science Series, Springer Verlag, 1993.

107. 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 *Proceedings of ACM SIGMOD Conference*, Boston, MA, May 1979.

108. T. Sellis, N. Roussopoulos, and C. Faloutsos, "The R^+-Tree: A Dynamic Index for Multi-Dimensional Objects," in *Proceedings of 1987 VLDB Conference*, Brighton, England, Sept. 1987.

109. P. Seshadri, M. Livny, and R. Ramakrishnan, "'Sequence Query Processing," in *Proceedings of ACM SIGMOD'94*, Minneapolis, MN, May 1994.

110. P. Seshadri, M. Livny, and R. Ramakrishnan, "The Design and Implementation of a Sequence Database System," in *Proceedings of the 22nd VLDB Conference*, Mumbai, India, 1996.

111. P. Seshadri, M. Livny, and R. Ramakrishnan, "The case for Enhanced Abstract Data Types," in *Proceedings of the 23rd VLDB Conference*, Athens, Greece, 1997.

112. H. Shatkay, S. B. Zdonik, "Approximate Queries and Representations for Large Data Sequences," in *Proceedings of 1996 Data Engineering Conference*, Feb. 1996.

113. H. Shen, B. C. Ooi, and H. Lu, "The TP-Index: A Dynamic and Efficient Indexing Mechanism for Temporal Databases," in *Proceedings of 1994 Data Engineering Conference*, 1994.

114. D. W. Shipman, "The Functional Data Model and the Data Language DAPLEX," in *ACM Transactions on Database Systems*. 6(1):140-173, March 1981.

115. SHORE project document, "*An Overview of SHORE*." Computer Science Department, University of Wisconsin-Madison, August, 1996.

116. SHORE project on-line information, "http://www.cs.wisc.edu/shore/".

117. A. Shoshani and K. Kawagoe, "Temporal Data Management," in *Proceedings of the 12th VLDB Conference*, Kyoto, Japan, Aug. 1986.

118. A. Silberschatz, H. F. Korth and S. Sudarshan, "*Database System Concepts*." The McGraw-Hill Companies, Inc. ISBN 0-07-044756-X, 1996.

119. J. M. Smith and P. Y. T. Chang, "Optimizing the Performance of a Relational Algebra Database Interface," in *Communications of ACM*, 18(10):568-579, Oct. 1975.

120. J. M. Smith, S. A. Fox, and T. A. Landers, "*DAPLEX: Rationale and Reference Manual*." Technical Report CCA-83-08, Computer Corporation of America, May 1983.

121. R. Snodgrass, "The Temporal Query Language TQuel," in *ACM Transactions on Database System*s, 12(2): 247-198, July 1987.

122. R. Snodgrass and I. Ahn, "Temporal Databases", in *IEEE Computer*, pp. 35-42, Sept. 1986.

123. R. Snodgrass and I. Ahn, "A Taxonomy of Time in Databases", in *Proceedings of ACM SIGMOD Conference*, Austin, TX, May 1985.

124. R. Snodgrass, "Temporal Databases," in A. U. Frank, I. Campari, and U. Formentini, eds., *Theories and Methods of Spatio-Temporal Reasoning In Geographic Space*. Spring-Verlag, Lecture Notes in Computer Science 639, pp. 22-64, 1992.

125. R. Snodgrass, I. Ahn, G. Ariav, D. Batory, J. Clifford, C. E. Dyreson, R. Elmasri, F. Grandi, C. S. Jensen, W. Käfer, N. Kline, K. Kulkarni, T. Y. C. Leung, N. Lorentzos, J. F. Roddick, A. Segev, M. D. Soo, and S. M. Sripada, "TSQL2 Language Specification," in *SIGMOD RECORD*, 23(1):65-86, March 1994.

126. S. M. Sripada, B. L. Rosser, J. M. Bedford and R. A. Kowalski, "Temporal Database Technology for Air Traffic Flow Management," in *Proceedings of the 1st International Conference on Applications of Databases*, Vadstena, Sweden, June 1994.

127. M. Stonebraker, E. Wong, P. Kreps, and G. Held, "The Design and Implementation of INGRES", in *ACM Transactions on Database Systems*, 1(3):189-222, Sept. 1976.

128. M. Stonebraker, "Inclusion of New Types in Relational Data Base Systems," in *Proceedings of 1986 Data Engineering*, 1986.

129. M. Stonebraker, "The Design of the POSTGRES Storage System," in *Proceedings of the 13rd VLDB Conference*, Sep. 1987.

130. M. Stonebraker, "The Implementation of POSTGRES," in *IEEE Transactions on Knowledge and Data Engineering*, March 1990.

131. M. Stonebraker, *"Object-Relational DBMSs."* The Morgan Kaufmann Publishers. ISBN 1-55860-397-2, 1996.

132. M. Stonebraker, B. Rubenstein, and A. Guttman, "Application of Abstract Data Types and Abstract Indices to CAD Data Bases.," in *Proceedings of the Engineering Applications Stream of Database Week*, San Jose, CA, May 1983.

133. M. Stonebraker, H. Stettner, N. Lynn, J. Kalash, and A. Guttman, "Document Processing in a Relational Database System," in *ACM Transactions on Office Information Systems*, vol. 1, No. 2, April 1983.

134. A. U. Tansel et al. (editors), *"Temporal Databases, Theory Design and Implementation."* The Benjamin/Cummings Publishing Company, Inc. ISBN 0-8053-2413-5, 1993.

135. K. Torp, L. Mark and C. S. Jensen, "Efficient Differential Timeslice Computation," in *IEEE Transactions on Knowledge and Data Engineering*, Vol. 10, No. 4, July 1998.

136. V. J. Tsotras and N. Kangelaris, "The Snapshot Index, and I/O Optimal Access Method for Timeslice Queries," in *Information Systems*, 3(20): 237-260, 1995.

137. J. D. Ullman, *"Principles of Database and Knowledge-Base Systems."* The Computer Science Press. ISBN 0-7167-8158-1, 1988.

138. K. Y. Wang and R. Krishnamurthy, "Query Optimization in a Memory Resident Domain Relational Calculus System," in *ACM Transactions on Database Systems*, Vol. 15, No. 1, pp. 67-95. March 1990.

139. G. Özsoyoglu and R. Snodgrass, "Temporal and Real-Time Databases: A Survey," in *IEEE Transactions on Knowledge and Data Engineering*, Vol. 7, No. 4, August 1995.

140. *VLDB'98 Tutorial Notes*, 24th International Conference on VLDB, New York City, USA, Aug. 1998.

Index

—S—

—T—

—U—

—V—

—W—