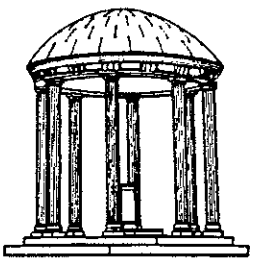
Performance Modeling and Access Methods for Temporal Database Management Systems

TR86-018

August, 1986

Ilsoo Ahn



The University of North Carolina at Chapel Hill
Department of Computer Science
Sitterson Hall, 083A
Chapel Hill, NC 27599-3175

UNC is an Equal Opportunity/Affirmative Action Institution.

Performance Modeling and Access Methods for Temporal Database Management Systems

by

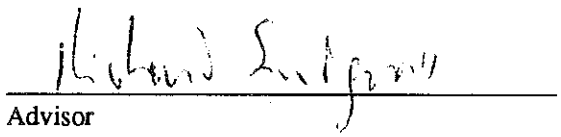
Ilsoo Ahn

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

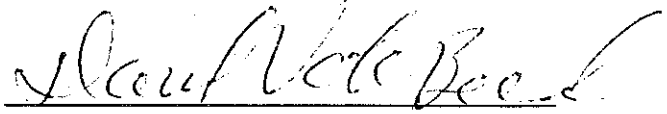
Chapel Hill

1986

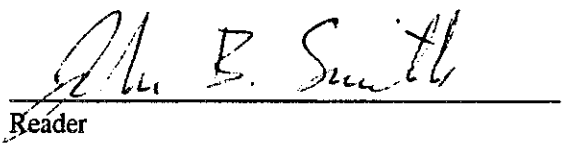
Approved by:



Advisor



Reader



Reader

© 1986

Ilsoo Ahn

ALL RIGHTS RESERVED

ILSOO AHN. Performance Modeling and Access Methods for Temporal Database Management Systems (Under the direction of Richard Snodgrass)

Conventional databases storing only the latest snapshot lack the capability to record and process time-varying aspects of the real world. The need for temporal support has been recognized for over ten years, and recent progress in secondary storage technology is making such support practically feasible.

There are three distinct kinds of time in databases: *transaction time*, *valid time*, and *user-defined time*. Depending on the capability to support either or both of transaction time and valid time, databases are classified into four types: *snapshot*, *rollback*, *historical*, and *temporal*. Each of the four types has different semantics and different implementation issues.

Database systems with temporal support maintain history data on line together with current data, which causes problems in terms of both space and performance. This research investigates the *temporally partitioned store* to provide fast response for various temporal queries without penalizing conventional non-temporal queries. The *current store* holds current data and possibly some history data, while the *history store* contains the rest. The two stores can utilize different storage formats, and even different storage media, depending on the individual data characteristics. Various issues on the temporally partitioned store were studied, and several formats for the history store were investigated.

To analyze the performance of TQuel queries on various access methods, four models forming a hierarchy were developed: one each for *algebraic expressions*, *database/relations*, *access paths*, and *storage devices*. The model of algebraic expressions maps the *algebraic expression* to the *file primitive expression*, based on information represented by the model of database/relations. The model of access paths maps the file primitive expression to the *access path expression*, which is converted to the *access path cost* by the model of storage devices.

As a test-bed to evaluate the access methods and the models, a prototype of a temporal DBMS was built by modifying a snapshot DBMS. The prototype was used to identify problems with conventional access methods and also to provide performance data to check the analysis results from the models. *Reverse chaining*, among the *temporally partitioned* storage structures, was incorporated in the prototype to enhance its performance.

Acknowledgements

I wish to express my sincere gratitude and appreciation to Professor R. Snodgrass, who has guided, encouraged, and enlightened me throughout this research.

I would also like to thank my committee, Professors J. Nievergelt, D. Stanat, J. Smith, and D. Beard, for their valuable suggestions on various aspects of this research.

Last, but not least, I thank my family, on both sides of this globe, for their love and support.

Table of Contents

PART I. Introduction 1

Chapter 1. Overview 3
1.1. Motivation 3
1.1.1. Terminology 4
1.1.2. Applications for Databases with Temporal Support 5
1.2. The Problem 6
1.2.1. Characteristics of Databases with Temporal Support 6
1.2.2. Conventional Access Methods 7
1.3. The Approach 9
1.3.1. Temporally Partitioned Store 9
1.3.2. Performance Models 10
1.3.3. Experiments 10
1.3.4. Summary 11
1.4. Structure of the dissertation 11

Chapter 2. Previous Work 13
2.1. Access Methods and Performance Analysis 13
2.1.1. Access Methods 13
2.1.2. Access Cost Estimation 14
2.1.3. Systems and Models 15
2.2. Databases with Temporal Support 17
2.3. TQuel 18

PART II. Temporal Database Management Systems 21

Chapter 3. Types of Databases 23
3.1. Snapshot Databases 23
3.2. Rollback Databases 25
3.3. Historical Databases 29
3.4. Temporal Databases 32
3.5. User-defined time 36
3.6. Summary 36

Chapter 4. Models and Performance Analysis 39
4.1. Models 39
4.1.1. Model of Algebraic Expressions 39
4.1.1.1. Algebraic Expressions 40
4.1.1.2. File Primitive Expressions 43
4.1.1.3. Model of Algebraic Expressions 45
4.1.2. Model of Database/Relations 50
4.1.3. Model of Access Paths 52

4.1.4. Model of Storage Devices 62
4.2. Performance Analysis 64
4.2.1. Examples 65
4.2.2. Performance Analyzer 69

Chapter 5. New Access Methods 71
5.1. Temporally Partitioned Store 71
5.1.1. Split Criteria 73
5.1.2. Update Procedures 74
5.1.3. Retroactive or Proactive Changes 80
5.1.4. Key Changes 81
5.1.5. Performance 82
5.2. Structures of the History Store 83
5.2.1. Reverse Chaining 83
5.2.2. Accession Lists 86
5.2.3. Indexing 88
5.2.4. Clustering 90
5.2.4.1. Variations 91
5.2.4.2. Nonlinear Hashing 94
5.2.5. Stacking 105
5.2.6. Cellular Chaining 106
5.3. Secondary Indexing 108
5.3.1. Types of Secondary Indices 108
5.3.2. Structures of Secondary Indices 109
5.4. Attribute Versioning 110
5.4.1. Conversion 111
5.4.2. Storage Requirements 114
5.4.3. Temporally Partitioned Store 115
5.5. Summary 117

PART III. Benchmarks 119

Chapter 6. Prototype with Conventional Access Methods 121
6.1. Prototype 121
6.2. Benchmarking the Prototype 127
6.2.1. A Benchmark 127
6.2.2. Performance Data 132
6.2.3. Analysis of Performance Data 135
6.2.4. Non-uniform Distribution 138
6.3. Analysis from Models 139
6.4. Summary 142

Chapter 7. Temporally Partitioned Store 143
7.1. Implementation of the Temporally Partitioned Store 144
7.2. Performance Analysis 147
7.2.1. Performance on a Rollback Database 148
7.2.2. Performance on a Temporal Database 151
7.3. Secondary Indexing 156

PART IV. Conclusions 163

Chapter 8. Conclusions and Future Work 165
8.1. Conclusions 165
8.2. Future Work 167

Bibliography 169

Appendix A. TQuel Syntax in BNF 177
Appendix B. Nonlinear Hashing 181
Appendix C. Benchmark Results 187
Appendix D. Performance Analysis (1) 193
Appendix E. Update Algorithms 205
Appendix F. Performance Analysis (2) 213

List of Figures

Figure 3-1: A Snapshot Relation 24
Figure 3-2: A Rollback Relation 25
Figure 3-3: A Rollback Relation 26
Figure 3-4: Historical Relation 29
Figure 3-5: A Historical Relation 30
Figure 3-6: A Temporal Relation 33
Figure 3-7: A Temporal Relation 33
Figure 3-8: A TRM Relation 35
Figure 3-9: Types of Databases 37
Figure 3-10: Time to be Supported by Databases 37
Figure 4-1: BNF Syntax for Algebraic Expressions 42
Figure 4-2: BNF Syntax for File Primitive Expressions 44
Figure 4-3: IDL Description for the Model of Database/Relations 51
Figure 4-4: BNF Syntax for File Path Expressions (Single File) 54
Figure 4-5: Structures for an Inverted File and a Multilist File ($n = 3$) 56
Figure 4-6: Access Paths with Three Files 60
Figure 4-7: BNF Syntax for Access Path Expressions (Multiple Files) 60
Figure 4-8: Access Path Graphs ($n = 3$) 61
Figure 4-9: Time (in *msec*) to Access a Block 63
Figure 4-10: Performance Analysis with the Four Models 64
Figure 4-11: Performance Analyzer for TQuel Queries 69
Figure 5-1: A Delete Statement 76
Figure 5-2: Base Interval *vs.* Update Interval for **`Delete`** 76
Figure 5-3: Base Interval *vs.* Update Interval for **`Replace`** 78
Figure 5-4: Reverse Chaining 84
Figure 5-5: Accession List 86
Figure 5-6: Indexing 89
Figure 5-7: Clustering 91
Figure 5-8: Insertions in Nonlinear Hashing 97
Figure 5-9: Insertions in Nonlinear Hashing 98
Figure 5-10: Insertions in Nonlinear Hashing 99
Figure 5-11: Deletions in Nonlinear Hashing 101
Figure 5-12: Deletions in Nonlinear Hashing 102
Figure 5-13: Stacking (depth $d = 3$) 105
Figure 5-14: Cellular Chaining (cell size $c = 3$) 107
Figure 5-15: Types of Secondary Indices for Each Type of Databases 109
Figure 5-16: A Relation in Tuple Versioning 111
Figure 5-17: A Relation in Attribute Versioning 112
Figure 5-18: Partial **`UNNEST`**'ing of A Relation with Attribute Versions 112
Figure 5-19: Full **`UNNEST`**'ing of the Relation in Figure 5-17 113
Figure 5-20: Attribute Versions 116
Figure 5-21: Structures for the History Store 117

Figure 6-1: Internal Structure of INGRES 122
Figure 6-2: A TQuel Query 123
Figure 6-3: A Syntax Tree 123
Figure 6-4: Creating a Temporal Database 127
Figure 6-5: Benchmark Queries 129
Figure 6-6: Space Requirements (in Blocks) 132
Figure 6-7: Input Costs for the Temporal Database with 100% Loading 133
Figure 6-8: Input Costs for Four Types of Databases 134
Figure 6-9: Graphs for Input Costs 134
Figure 6-10: Fixed Costs, Variable Costs, and Growth Rates 137
Figure 6-11: Analysis Results using Performance Models 140
Figure 6-12: Error Rates in the Analysis Results 141
Figure 6-13: Elapsed Time (in *sec*) 142
Figure 7-1: Space Requirements for the `Rollback_h` Relation 148
Figure 7-2: The Rollback Database with 100% Loading 149
Figure 7-3: Space Requirements for the `Temporal_h` Relation 152
Figure 7-4: The Temporal Database with 100% Loading 153
Figure 7-5: Fixed Costs, Variable Costs, and Growth Rates 155
Figure 7-6: Space Requirements for a Secondary Index 157
Figure 7-7: Secondary Indexing as Snapshot or Rollback 158
Figure 7-8: Secondary Indexing as Historical or Temporal 160
Figure C-1: I/O Cost for the Rollback DBMS with 100% Loading 187
Figure C-2: Space for the Rollback DBMS with 100% Loading 187
Figure C-3: I/O Cost for the Rollback DBMS with 50% Loading 188
Figure C-4: Space for the Rollback DBMS with 50% Loading 188
Figure C-5: I/O Cost for the Historical DBMS with 100% Loading 189
Figure C-6: Space for the Historical DBMS with 100% Loading 189
Figure C-7: I/O Cost for the Historical DBMS with 50% Loading 190
Figure C-8: Space for the Historical DBMS with 50% Loading 190
Figure C-9: I/O Cost for the Temporal DBMS with 100% Loading 191
Figure C-10: Space for the Temporal DBMS with 100% Loading 191
Figure C-11: I/O Cost for the Temporal DBMS with 50% Loading 192
Figure C-12: Space for the Temporal DBMS with 50% Loading 192

PART I

Introduction

Temporal databases with the capability to record and process time-dependent data expand the area of database applications, bringing a wide range of benefits. The thesis of this dissertation is that new access methods can be developed to provide temporal support in database management systems without penalizing conventional non-temporal queries and the performance of such systems can be analyzed by a set of models forming a hierarchy.

Part one consists of two chapters. The first chapter describes the background, motivation, and approach of this dissertation. The second chapter summarizes previous work in the area of this research and describes TQuel, a temporal query language used throughout this dissertation.

Chapter 1

Overview

This chapter describes the motivation for database management systems with temporal support and discusses the benefits and applications for such systems. It then identifies the problems involved in providing temporal support and presents the approach taken in this dissertation.

1.1. Motivation

Time is an essential part of information concerning the real world, which is constantly evolving. Facts or data need to be interpreted in the context of time. Causal relationships among events or entities are embedded in temporal information. Time is a universal attribute in most information management applications and deserves special treatment as such.

Databases are supposed to model reality, but conventional database management systems (DBMS's) lack the capability to record and process time-varying aspects of the real world. With increasing sophistication of DBMS applications, the lack of temporal support raises serious problems in many cases. For example, conventional DBMS's cannot support *temporal queries* about past states, nor can they perform *trend analysis* over a series of history data. There is no way to represent *retroactive* or *proactive* changes. Support for *error correction* or an *audit trail* necessitates costly maintenance of backups, checkpoints, or transaction logs to preserve past states. There is a growing interest in applying database methods for *version control* and *design management* in computer aided design, requiring capabilities to store and process time-dependent data. Without temporal support from the system, many applications have been forced to manage temporal information in an *ad hoc* manner.

The need for providing temporal support in database management systems has been recognized for at least a decade. A bibliographical survey contained about 70 articles relating time and information processing [Bolour et al. 1982]; at least 90 more articles have appeared in the literature since 1982 [MCKENZIE 1986]. In addition, the steady decrease of secondary storage cost, coupled with emergence

of promising new mass storage technologies such as optical disks [Hoagland 1985], have amplified interest in database management systems with temporal support or version management. G. Copeland asserted that

> ... as the price of hardware continues to plummet, thresholds are eventually reached at which these compromises [to achieve hardware efficiency] must be rebalanced in order to minimize the total cost of a system. ... If the deletion mechanism common to most database systems today is replaced by a non-deletion policy ..., then these systems will realize significant improvements in functionality, integrity, availability, and simplicity. [Copeland 1982]

G. Wiederhold also observed, in a review of the present state of database technology and its future, that

> The availability of ever greater and less expensive storage devices has removed the impediment that prevented keeping very detailed or extensive historical information in on-line databases. ... An immediate effect of these changes will be the retention of past data versions over long periods. [Wiederhold 1984]

As a result, numerous schemes have been proposed to provide temporal support in database management systems by incorporating one or more time attributes in recent years. However, there has been some confusion concerning terminology and definitions on several concepts in this area, and many issues remain to be investigated for implementing such systems with adequate performance.

1.1.1. Terminology

The first question concerning temporal databases is the definition of the term *temporal database* itself. The term in the generic sense, as used in the title of this dissertation, refers to databases with any degree of support for recording and processing *temporal* or *time-dependent* data. Databases in this category are, for example, an engineering database with a collection of design versions, a personnel database with a history of employee records, or a statistical database with time series data from scientific experiments.

If we look into the characteristics of *time* supported in these databases, we can identify three distinct kinds of time with different semantics, as will be discussed further in Chapter 3: *valid time*, *transaction time*, and *user-defined time* [Snodgrass & Ahn 1985, Snodgrass & Ahn 1986]. Valid time is the time when an event occurs in an enterprise. Transaction time is the time when a transaction occurs in a database to record the event. User-defined time is defined by a user, whose semantics depends on each application. This taxonomy of time naturally leads to the next question of *what kind* of time is to be supported. Depending on the capability to support either or both of valid time and transaction time, databases are classified into four types: *snapshot*, *rollback*, *historical*, and *temporal*. Rollback databases support

transaction time, recording the history of database activities. Historical databases support valid time, recording the history of a real world. Databases supporting both kinds of time are termed temporal databases in the narrower sense to emphasize the importance of both kinds of time in database management systems. In the remainder of this dissertation, the term *temporal databases* is used in this narrower sense, unless indicated otherwise.

1.1.2. Applications for Databases with Temporal Support

Providing temporal support in database management systems brings about many benefits and interesting applications. For example, it is possible to make *historical queries* to ask the status of an enterprise valid at a past or even future moment, or to perform *rollback operations* shifting the reference point back in time and inquiring the state of a database in the past [Snodgrass & Ahn 1985, Snodgrass & Ahn 1986]. These capabilities help in understanding the dynamic process of state evolution in an enterprise, and in identifying temporal or causal relationships among events or entities.

The capability for retrospective analysis is essential in decision support systems to evaluate planning models based on the frozen state of knowledge about the world at the time of planning [Ariav 1984]. It is possible to ask *what if* questions on the past events, to perform trend analysis over a series of data, to forecast the future based on the past and the current data, and to plan resources over time.

Temporal databases can record retroactive changes which occurred in the past, or proactive changes which will take effect in the future. Correct handling of time is important in modeling temporal constraints or writing complex rules such as those in legislation or high level system specifications [Jones & Mason 1980]. Maintaining history data without physical deletion facilitates error correction, audit trail, and accounting applications. The ability to control the configuration of a series of versions is useful for version management in engineering or textual databases [Katz & Lehman 1984].

Supporting time in database management systems not only adds to the functionality for various applications, but also benefits system operations. Temporal information or time stamps can be utilized for concurrency control of multiple transactions, recovery after systems crashes, and synchronization of distributed databases [Bernstein & Goodman 1980]. Enforcing the *no-update-in-place* paradigm increases reliability, facilitates error recovery, reduces burdens on backups, check-points, or transaction-logs, and

results in lower system cost [Copeland 1982, Schueler 1977]. Retention of history data is also attractive for utilizing low cost and large capacity write-once media such as optical disks.

1.2. The Problem

Despite the benefits of database systems with temporal support as described above, there are several problems to be overcome before implementing such systems with adequate performance. This section describes the characteristics of databases with temporal support and then discusses whether conventional access methods are appropriate for such databases.

1.2.1. Characteristics of Databases with Temporal Support

Database systems with temporal support follow the *non-deletion* policy in one way or another to preserve past information needed for historical queries or rollback operations. It means that no record will ever be deleted once it is inserted, except to correct errors in the case of historical databases. For each update operation, a new version is created without destroying or over-writing existing ones. This strategy solves many of the problems caused by the *update-in-place* practice common in conventional DBMS's [Schueler 1977], but also introduces several new problems.

An immediate concern is the large volume of data to be maintained on line. Storage requirements will increase monotonically, potentially to an enormous amount, no matter what data compression technique is utilized. This problem is one of the major reasons why databases with temporal support have not been put into practice even though their benefits have been long recognized. It is often impractical to store all the states of a database while it evolves over time. It is necessary to devise mechanisms dealing with the ever-growing storage size effectively, and to represent temporal versions into physical storage in such a way that past states of a database can be maintained with little redundancy.

The large amount of data to be maintained also causes performance problems. For example, the number of block accesses to get a record from an unordered file with m blocks is $O(m)$. Storing temporal data in such a file will require a large m, significantly degrading the performance. In addition, each update operation adds a new version, generating multiple versions for some tuples. Unless temporal information is utilized as a part of a key, there will be multiple records for a single key value. However, time attributes are in general not suitable to be used as a key for storing and accessing records. A time attribute alone

cannot be used as a key in most applications. Including time attributes in a key results in a multi-attribute key, which complicates the maintenance of the key. Even though time attributes are maintained as a part of a key, it is difficult to make a *point query* (*exact match query*), which requires a single point in time specified as a predicate, especially when the resolution of time values is fine. Thus, we should be able to support a *range query* on time attributes, which is not possible with many access methods, *e.g.* various forms of hashing. These issues present serious problems for most conventional access methods, as will be further discussed in the next section and in Chapter 6.

On the other hand, there are several interesting characteristics unique in databases with temporal support. There are two distinct types of data, the *current* and the *history*, which exhibit clear differences in their characteristics on many aspects. There is only one current version for each tuple at one time, yet multiple versions exist for some tuples in history data. Storage requirements for history data may be potentially enormous, while the size of current data is relatively static once it has stabilized. Unlike current data, history data need not be updated except when errors are corrected in the case of historical databases, which makes *write-once* optical disks attractive as the storage media.

There is also a correlation between the age of data and their access frequencies or access urgencies. Conventional databases store only the latest snapshot of an enterprise being modeled, which represents the current data. Hence all the conventional database applications deal with only the current data. Retaining history data for temporal support will encourage new applications to process history data together with current data, such as historical queries, rollback operations, and trend analysis. But in general, conventional applications dealing with current data are still expected to dominate new applications concerning history data. Therefore, history data are accessed less frequently than current data. Likewise, history data are needed less urgently than current data. Since databases with temporal support have these unique characteristics, not found in conventional databases, it is a challenge to exploit them in system implementation for better performance.

1.2.2. Conventional Access Methods

Access methods such as *sequential*, *hashing*, *indexing*, and *ISAM* are *static* in the sense that they do not accommodate growth of files without significant loss in performance. Accessing data in a sequential

file requires sequential scanning, which is often too expensive. Access methods such as hashing and ISAM also suffer from rapid degradation in performance due to ever-growing overflow chains caused not only by key collisions but also by the existence of multiple versions for a single key, as will be demonstrated in Chapter 6. Reorganization does not help to shorten overflow chains, because all versions of a tuple share the same key. Hence performance will deteriorate rapidly not only for temporal queries but also for non-temporal queries [Ahn & Snodgrass 1986].

There are *dynamic* access methods that adapt to dynamic growth better, such as *B-trees* [Bayer & McCreight 1972], *virtual hashing* [Litwin 1978], *linear hashing* [Litwin 1980], *dynamic hashing* [Larson 1978], *extendible hashing* [Fagin et al. 1979], *K-D-B trees* [Robinson 1981], or *grid files* [Nievergelt et al. 1984]. These methods maintain certain structures as records are added or deleted. But the performance is still dependent on the count of all versions, which is significantly higher than the count of current versions. Furthermore, a large number of versions for some tuples will require more than a bucket for a single key, causing similar problems to those exhibited in conventional hashing. It is also difficult to maintain secondary indices for these methods, because they often split a bucket and rearrange its records. Performance problems of conventional access methods in the environment of databases with temporal support will be further discussed in Chapter 6.

Secondary storage cost has been decreasing rapidly and consistently, and various new technologies are emerging in recent years. In particular, optical disks are becoming commercially available from several manufacturers at a reasonable cost [Fujitani 1984, Hoagland 1985]. A single disk provides storage capacity of up to 5 Gbytes, whose per byte cost is about four orders of magnitude lower than magnetic disks. Data can be accessed randomly, though about an order of magnitude slower, with data transfer rate comparable to magnetic disks. It takes about a minute to mount a new disk manually, but there is a system which houses 64 disks with the total capacity of 128 Gbytes and changes a disk in less than 5 sec [Ammon et al. 1985]. One limitation of optical disks is that they are currently *write-once*, not allowing reorganization or rewriting of data once they are stored. This peculiarity makes many of the conventional storage structures, especially the dynamic ones such as B-trees or dynamic hashing, unsuitable for optical disks, and requires new storage structures to utilize their potential benefits.

1.3. The Approach

These observations on the inadequacy of conventional access methods lead to the conclusion that new access methods need to be developed to provide fast access paths for a wide range of temporal queries without penalizing conventional non-temporal queries. Therefore, this dissertation investigates new access methods tailored to the particular characteristics of database management systems with temporal support, and also develops a set of models to analyze the performance of query processing in such systems.

For this research, *TQuel* (Temporal QUEry Language) [Snodgrass 1986] was chosen as the query language, because it is the only temporal language to support both historical queries and rollback operations. A description of TQuel will be given in Section 2.3.

1.3.1. Temporally Partitioned Store

The solution proposed in this dissertation to the problems discussed in Section 1.2 is the *temporally partitioned store* to divide current data and history data into two storage areas. The *current store* holds current data and possibly some history data, while the *history store* contains the remaining history data. This scheme to separate current data from the bulk of history data can minimize the overhead for conventional non-temporal queries, and at the same time provide a fast access path for temporal queries. The two stores can utilize different storage formats, and even different storage media, depending on individual data characteristics.

There are many issues to be investigated about the temporally partitioned storage structure. The main issues are the *split criteria* on how to divide data between the current and the history store, update procedures for each type of databases with temporal support, methods to handle retroactive changes, proactive changes, or key changes, and the performance with regard to the update count. This research addresses these issues in general, then concentrates on the details of various formats for the history store. It investigates various forms of the history store, studies their characteristics, analyzes their performance, and implements one of them to obtain performance data for comparison with analysis results. Relative advantages and disadvantages of the various formats will be evaluated in this process to determine the cost of supporting temporal queries. Issues on how to support *secondary indexing* and *attribute versioning* in the temporally partitioned storage structure are also studied.

1.3.2. Performance Models

Models of the various phases of query processing in database management systems can facilitate the process of investigating access methods by reducing the need to implement each method for performance evaluation. Though significant contributions have been made for models and systems to analyze the performance of file organizations and database management systems as will be described in Section 2.1, the general problem of evaluating the access cost given a query as an input has not been addressed adequately. Furthermore, particular characteristics of query processing and access methods considered in this research for database management systems with temporal support demand a new set of models different from those for conventional systems.

Therefore, this research develops four models, forming a hierarchy, to characterize the process of temporal query processing: one each for *algebraic expressions*, *database/relations*, *access paths*, and *storage devices*. The model of algebraic expressions maps the *algebraic expression* to the *file primitive expression*, based on information represented by the model of database/relations. The model of access paths maps the file primitive expression to the *access path expression*, which is converted to the *access path cost* by the model of storage devices.

These models combined can estimate the input and output cost for a collection of *TQuel* queries, and analyze various alternatives in the design of new access methods without the time consuming process of case by case implementation or simulation.

1.3.3. Experiments

As a test-bed to evaluate the access methods and the models, a prototype of a temporal DBMS was built by modifying a snapshot DBMS. Since TQuel is a superset of Quel, *INGRES* [Stonebraker et al. 1976] was a natural choice as the host system for this purpose.

The initial prototype uses the conventional access methods available in INGRES. Therefore, it can be used to identify problems with conventional access methods, and to suggest possible improvements. One of the *temporally partitioned storage structures* is actually implemented and incorporated in the prototype to enhance its performance. Performance data measured from the prototype will be compared in Part III with the analysis results from the models described above to check the accuracy of models.

1.3.4. Summary

This research investigates various forms of temporally partitioned storage structures for database management systems with temporal support, and develops models to analyze the performance of query processing in such systems. It also demonstrates the feasibility of providing temporal support in database management systems without penalizing conventional non-temporal queries. By investigating performance models and access methods for database management systems with temporal support, this research will contribute to expanding the capabilities and application areas of database management systems.

1.4. Structure of the dissertation

This chapter described the background, motivation, and the approach of this research. The second chapter summarizes previous work related with this research, and briefly describes TQuel, a temporal query language used throughout this dissertation.

Part II consists of three chapters. Chapter 3 defines the types of databases in terms of temporal support. Chapter 4 describes the models developed to analyze the performance of query processing in database systems with temporal support. Chapter 5 discusses various issues for the temporally partitioned storage structure, and investigates the formats of the history store.

Part III presents the benchmark results, measured from the prototype implementation. Chapter 6 is for the prototype with the conventional access methods, and Chapter 7 is for the the prototype with the temporally partitioned storage structure developed in Chapter 5.

Finally, Part IV presents the conclusions of this research and suggests areas of future work.

Chapter 2

Previous Work

This chapter reviews previous research in the area of access methods and performance analysis for conventional database systems, and in the area of database management systems with temporal support emphasizing the aspects of implementation.

2.1. Access Methods and Performance Analysis

Contributions in this area are described below in three categories, access methods, access cost estimation, and systems and models for performance analysis.

2.1.1. Access Methods

There has been a massive amount of research on the design and analysis of specific file structures with various characteristics. Some examples are ISAM files [Larson 1981], B-trees [Bayer & McCreight 1972, Comer 1979, Held 1978], prefix B-trees [Bayer & Unterauer 1977], and a performance comparison between ISAM and B-trees [Batory 1981].

Hashing schemes can be classified into *fixed size* and *variable size*, depending on the adaptability to the change of the file size. For fixed size hashing [Bloom 1970, Coffman & Eve 1970, Lum et al. 1971], schemes such as linear probing [Mendelson 1980] and coalesced hashing [Chen & Vitter 1984] were studied to handle overflow records. Perfect hashing attempts to eliminate overflow records for a given set of keys by selecting a perfect hash function [Cichelli 1980, Larson & Ramakrishna 1985, Sprugnoli 1977]. Various methods have been proposed to extend the hashing technique to maintain high performance even when the file size changes dynamically. Among those are virtual hashing [Litwin 1978], dynamic hashing [Larson 1978], extendible hashing [Fagin et al. 1979, Mendelson 1982], linear hashing [Litwin 1980], linear hashing with partial expansions [Larson 1982], and recursive linear hashing [Ramamohanarao & Sacks-Davis 1984].