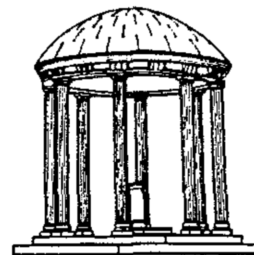
Performance Modeling and Access Methods for Temporal Database Management Systems

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

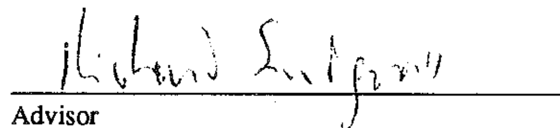
Ilsoo Ahn

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.

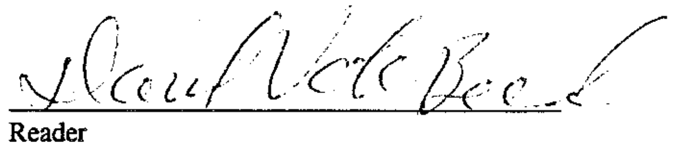
Chapel Hill

1986

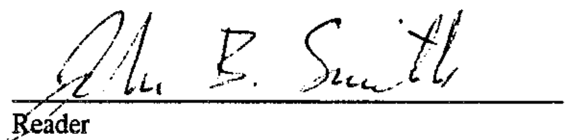
Approved by:



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].

Differential files [Aghili & Severance 1982, Gremillion 1982, Severance 1976] were proposed to increase data availability by localizing modifications to a separate file. Grid files [Nievergelt et al. 1984] and multi-dimensional K-D-B-trees [Robinson 1981] have been developed for random access through multiple keys. Many of these structures are applicable to the current store, and some variations may also be useful for the history store of the temporally partitioned storage structure, as will be discussed in Chapter 5.

2.1.2. Access Cost Estimation

There are two basic problems to be solved for evaluating the cost of a query. One is to determine the size of the response set which satisfies the query, and the other is to estimate the number of block accesses required to retrieve those records.

[Yu et al. 1978] studied the problem of estimating the number of records accessed for a given query from a clustered database. They compared empirical data with the estimations under the assumption of attribute independence, and improved the accuracy of the estimations by relaxing the assumption of independence. [Richard 1980] presented a probabilistic model for evaluating the size of derived relations from a query expressed in relational algebra, given the expected size of all projections of each relation in a database.

The problem to estimate the number of block accesses for retrieving k records out of n records stored in m blocks was first addressed by [Cardenas 1975]. [Yao 1977A] noted that the solution of [Cardenas 1975] was for the case where records might have duplicates, and gave a solution when all records were distinct. [Cheung 1982] presented a formula for the case where requested records might have duplicates but their ordering was immaterial. [Whang et al. 1983] derived a closed, noniterative formula for fast computation of this problem, and analyzed resulting errors. [Luk 1983] examined the case where the variables k, n, and m were stochastic and non-uniform. [Christodoulakis 1983] provided estimates of the number of sequential and random block accesses for retrieving a number of records from a file when the distribution of records was not uniform, and applied the result to estimating the size of the join operation.

[Christodoulakis 1984] noted that most performance analyses assumed uniformity and independence of attribute values, uniformity of queries, a fixed number of records per block, and random placement of

qualifying records. He showed that these assumptions predicted the upper bound of expected system cost, and led to the choice of worst-case strategies.

2.1.3. Systems and Models

There have been several systems to analyze the performance of file organizations based on a collection of individual models. [Cardenas 1973] designed and implemented a system to evaluate and select file organizations. The system estimated disk access time and storage requirements given a measure of query complexity for a single file retrieval. It contained file structure modules derived from analytical analysis for inverted files, multilists, and doubly chained tree files. [Siler 1976] implemented a stochastic model of data retrieval systems to analyze inverted list, threaded list, cellular list organization, and hybrid combinations under varying degrees of query complexity.

[Scheuermann 1977] presented a simulation model to compute a weighted cost function of storage and retrieval time for a hierarchical DBMS given descriptions of workload and storage structure. Data definition and query definition sublanguages described the workload, and a mapping sublanguage represented several levels of mappings to storage structures. [Satyanarayanan 1983] developed a methodology and a simulator for modeling storage systems with device modules and hierarchy descriptions.

[Hawthorn & Stonebraker 1979] measured the performance of INGRES with a benchmark query stream, rather than a performance modeling approach. They studied I/O reference and CPU usage patterns for each of data-intensive, overhead-intensive, and multi-relation query types. The result was used to discuss the effect of storing temporary relations in a cache, using multiple processors, and prefetching data blocks.

On the other hand, a series of generalized models have been proposed with varying complexity and descriptive power for the past 15 years. Hsiao and Harary proposed a formal model to analyze and evaluate generalized file organizations [Hsiao & Harary 1970]. The model represents the directory of a file with a set of sequences $(K_i, n_i, h_i; a_{i1}, a_{i2}, ..., a_{ih_i})$ for each keyword K_i, where n_i is the number of records containing the keyword, h_i the number of sublists holding such records, and a_{ih_i} the starting address of the h_i'th sublist. By varying the number and the length of sublists for each keyword, it can represent structures

such as multilist files, inverted files, indexed sequential files, and some combinations of those.

Severance noted that this one dimensional model is unable to represent files which are not strictly list oriented, so introduced a *two-dimensional* model [Severance 1975]. One dimension is whether the successor node is physically contiguous (*address sequential*), or connected through a pointer (*pointer sequential*). The other dimension is whether there is an index for the data (*data indirect*), or not (*data direct*). The four corners of this two-dimensional space represent sequential files, inverted files, list files, and pointer sequential inverted files.

Yao observed that Severance's model represents only a one-level index, imprecisely models indexed sequential files, and cannot model cellular list organizations [Yao 1977B]. Instead, he represented the process of searching a file by an access tree composed of hierarchical levels such as attributes, keywords, accession lists, and virtual records. Additional parameters to characterize the access path were the average number of records, overflow ratio, loading factor, and maximum overflow ratio for each level. Based on this access path model, generalized access algorithms and cost functions for search and retrieval were presented. He also presented a file retrieval algorithm and an associated cost function for a single file query in a disjunctive normal form. Some of the parameters for the query were the total number of attributes and the average number of conjuncts in a query. Since this model has the underlying structure of the tree shaped access path, it is suitable for directory based file organizations such as inverted files, but is less applicable to files with other structures.

Based on this generalized model of [Yao 1977B], a file design analyzer was built to evaluate storage structures and access methods such as sequential, direct, inverted, multilist, and network structures [Teorey & Das 1976]. It estimated I/O cost and storage requirements given a user workload expressed in terms of the number of retrievals and updates on a single record type. [Teorey & Fry 1980] presented a logical record access (LRA) approach as a practical stepwise database design methodology, and [Teorey & Fry 1982] used the physical block access (PBA) approach to estimate I/O performance of various file structures for a set of typical query types.

Yao also proposed a model for systematic synthesis of a large collection of access strategies for two relation queries [Yao 1979]. He identified 11 basic access operators such as restriction, join, record access, and projection, then presented without derivations cost equations for each operator measured in terms of

page accesses. Permuting these operators gave 7 classes of evaluation algorithms for each relation, and 339 different algorithms for two relation queries, whose cost could be computed from cost equations of each operator. He modeled the storage structures with parameters indicating the existence of clustering, parent, child or chain links among relations, and the existence of clustering or non-clustering index for each attribute of relations. However, the model of [Yao 1977B] was not used for this study.

[Yao & DeJong 1978] built the model of [Yao 1979] described above into a system which can calculate access path costs given parameters for the model and particular algorithms to be evaluated. Some examples of typical parameters were attributes per record, records per page, levels of index, fraction of file after projection, restriction selectivity, and join selectivity.

Batory and Gotlieb proposed a *unifying* model, which decomposes physical databases into *simple files* and *linksets* [Batory & Gotlieb 1982]. The model for simple files characterizes file structures with a set of parameters grouped as design parameters, file parameters, and cost parameters. The model for linksets describes relationships between records in two simple files with parameters such as parent, child, cell size, and implementation methods. Basic operations and associated cost functions were also defined for simple files and linksets. This model relies on a collection of parameters to describe various file organizations, rather than mapping their characteristics to an abstract structure. Batory augmented the unifying model later with *transformation model* which defines a set of *elementary transformations* [Batory 1985] to aid the process of decomposing physical databases into simple files and linksets.

As described above, significant contributions have been made for models and systems to evaluate the performance of file organizations and database management systems. [Yao 1977B], [Yao 1979] and [Batory & Gotlieb 1982] are particularly relevant to this research, but none of these actually addressed the whole problem of evaluating the access cost given relational queries as an input. Furthermore, particular characteristics of query processing and access methods considered in this research for the database management systems with temporal support are not adequately handled by any of the above models.

2.2. Databases with Temporal Support

There have been vigorous research activities in formulating the semantics of time at the conceptual level [Anderson 1982, Breutmann et al. 1979, Bubenko 1977, Hammer & McLeod 1981, Klopprogge

1981], developing models for time varying databases analogous to the conventional relational model [Clifford & Warren 1983, Codd 1979, Sernadas 1980], and the design of temporal query languages [Ariav & Morgan 1981, Ben-Zvi 1982, Jones & Mason 1980, Snodgrass 1986]. We will discuss these efforts in Chapter 3, grouping them into three types based on the capability for temporal support. However, there has been no major effort to investigate implementation aspects for either historical or temporal database systems, let alone performance analysis of such systems.

2.3. TQuel

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. TQuel supports two types of relations, *interval* relations and *event* relations. An interval relation, with two time attributes, consists of tuples representing a state valid during a time interval. An event relation, with a single time attribute, consists of tuples representing instantaneous occurrences.

TQuel extends several Quel [Held et al. 1975] statements to provide query, data definition, and data manipulation capabilities supporting all four types of databases. It expresses historical queries by augmenting the `retrieve` statement with the `when` predicate to specify temporal relationships among participating tuples, and the `valid` clause to specify how the implicit time attributes are computed for result tuples. The rollback operation is specified by the `as of` clause for the rollback or the temporal databases. These added constructs handle complex temporal relationships such as `precede`, `overlap`, `extend`, `begin of`, and `end of`. They are composed of a reserved word followed by an *event expression* or a *temporal expression*, whose syntax is derived from *path expressions* [Andler 1979].

The `append`, `delete`, and `replace` statements were augmented with the `valid` and the `when` clauses in a similar manner. Finally, the `create` statement was extended to specify the type of a relation, whether snapshot, rollback, historical or temporal, and to distinguish between an interval and an event relation if the relation is historical or temporal.

In addition, temporal aggregates for TQuel have been developed to provide a rich set of statistical functions that range over time [Snodgrass & Gomez 1986]. Aggregates are either *instantaneous* or

cumulative, are either *unique* or not, and may be *nested*. The formal semantics for the aggregates were defined in the tuple relational calculus. TQuel also defines how to handle *indeterminacy* or *incomplete* information, but this dissertation focuses on the core of the language without aggregates and indeterminacy. Since TQuel is a superset of Quel, both syntactically and semantically, all legal Quel statements are also valid TQuel statements. Statements have an identical semantics in Quel and TQuel when the time domain is fixed. The semantics of TQuel was formalized using tuple relational calculus and transformation rules [Snodgrass 1986], demonstrating that **when** and **valid** clauses are direct semantic analogues of Quel's **where** clause and target list. The complete syntax of TQuel is given in Appendix A.

PART II

Temporal Database Management Systems

Part two consists of three chapters, describing the conceptual aspects of this research. Chapter 3 defines the types of databases in terms of their capability for temporal support. Chapter 4 develops four models forming a hierarchy to analyze the process of query processing in database management systems with temporal support. Chapter 5 investigates various issues for the temporally partitioned storage structure which can provide temporal support for database management systems without penalizing conventional non-temporal queries.

Chapter 3

Types of Databases

As presented in [Snodgrass & Ahn 1985, Snodgrass & Ahn 1986], there are three distinct kinds of time with different semantics in databases: *valid time*, *transaction time*, and *user-defined time*. Valid time is the time when an event occurs in an enterprise. Transaction time is the time when a transaction to account for the event is executed in a database modeling the enterprise. 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 transaction time and valid time, databases are classified into four types: *snapshot*, *rollback*, *historical*, and *temporal*. This chapter, a summary of [Snodgrass & Ahn 1985], first discusses representational inadequacies of snapshot databases, and then compares three types of databases with temporal support. Though the following discussion is based on the relational model, analogous arguments readily apply to hierarchical or network models.

3.1. Snapshot Databases

Conventional databases model an enterprise, as it changes dynamically, by a snapshot at a particular point in time. A *state* or an *instance* of a database is its current contents, which does not necessarily reflect the current status of the enterprise. The state of a database is updated using data manipulation operations such as `append`, `delete` or `replace`, taking effect as soon as they are committed. In this process, past states of the database, representing those of the enterprise, are discarded. We term this type of database a *snapshot database*.

In the relational model, a database is a collection of *relations*. Each relation consists of a set of *tuples* with the same set of *attributes*, and is usually represented as a two dimensional table (see Figure 3-1). As changes occur in the enterprise, this table is *updated* appropriately.

Figure 3-1: A Snapshot Relation

For example, an instance of a relation 'Faculty', with two attributes `Name` and `Rank`, at a certain moment may be

Name	Rank
Merrie	Full
Tom	Associate

and a query in Quel, a tuple calculus based language for the INGRES database management system [Held et al. 1975], inquiring Merrie's rank,

```
range of f is Faculty
retrieve (f.Rank)
     where f.Name = "Merrie"
```

yields

Rank
Full

There are many situations where this snapshot database relying on snapshots is inadequate. For example, it cannot answer queries such as

What was Merrie's rank 2 years ago? (historical query)

How did the number of faculty change over the last 5 years? (trend analysis)

nor record facts like

Merrie was promoted to a full professor starting last month. (retroactive change)

James is joining the faculty next month. (proactive change)

Without system support in these respects, many applications have been forced to manage temporal information in an *ad hoc* manner. For instance, many personnel databases attempt to record the entire employment history of the company's employees. The facts that some of the attributes record time, and that only a subset of the employees actually work for the company at any particular point in time are not

the concerns of the DBMS itself. The DBMS provides no facility for interpreting or manipulating this information; such operations must be handled by specially-written application programs. The fact that data changes values over time is not application specific, but should be recognized as being universal. It is possible to identify the properties and the semantics of time common to all database applications, distinguish different kinds of time in databases, and provide the capability to handle each kind of time. These aspects should be supported in a general fashion by the database management systems, rather than by application programs.

3.2. Rollback Databases

One approach to resolve the above deficiencies is to store all past states, indexed by time, of the snapshot database as it evolves. Such an approach requires the support of *transaction time*, the time when the information is stored into the database. A relation under this approach can be illustrated conceptually in three dimensions with transaction time serving as the third axis (Figure 3-2). The relation can be regarded as a sequence of snapshot relations (termed *snapshot states*) indexed by time, and provides the capability to return to any previous state to execute a (snapshot) query. By moving along the time axis and selecting a particular snapshot state, it is possible to retrieve a snapshot of the relation as of some time in the past, and to make queries upon it. The operation of selecting a snapshot state is termed *rollback*, and a database supporting the operation is termed a *snapshot rollback database*, or simply a *rollback database*.

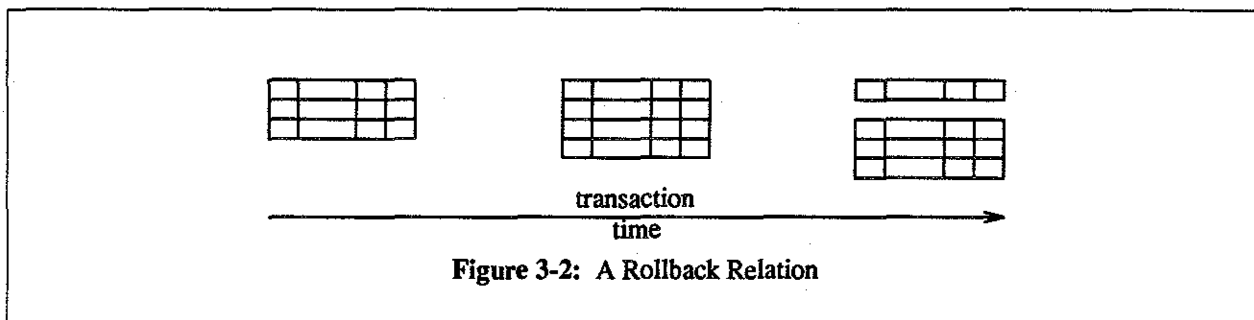


Figure 3-2: A Rollback Relation

Changes to a rollback database may be made only to the most recent snapshot state. The relation illustrated in Figure 3-2 had three transactions applied to it, starting from the null relation:

(1) three tuples were added,

(2) one tuple was added, and

(3) one tuple (entered in the first transaction) was deleted, and another tuple was added.

Each transaction results in a new snapshot state being appended to the front of the time axis. Once a transaction is committed, the snapshot states in a rollback relation may not be altered.

A typical relation in this approach looks like Figure 3-3. The double vertical bars separate the non-temporal attributes from the implicit time attributes *transaction start* and *transaction stop*. The latter attributes do not appear in the relation scheme, but may rather be considered as a part of the overhead associated with each tuple. Note the fact that Merrie was previously an associate professor, a fact which could not be expressed by a snapshot relation. The value '–' for the transaction stop attribute denotes 'on-going' or 'still true'.

Name	Rank	transaction time	
		(start)	(stop)
Merrie	Associate	08/25/77	12/15/82
Merrie	Full	12/15/82	–
Tom	Associate	12/07/82	–
Mike	Assistant	01/10/83	02/25/84

Figure 3-3: A Rollback Relation

Any query language may be extended to one for rollback databases by adding a clause effecting the rollback operation. TQuel (*T*emporal *QUE*ry *L*anguage) [Snodgrass 1986], an extension of Quel for temporal databases, augments the **retrieve** statement with an **as of** clause to specify the point of reference in time. The TQuel query

```
range of f is Faculty
retrieve (f.Rank)
      where f.Name = "Merrie"
      as of "12/10/82"
```

on a 'Faculty' relation shown in Figure 3-4 will find the rank of Merrie as of 12/10/82:

Rank
Associate

Note that the result of a query on a rollback database is a pure snapshot relation.

One limitation of supporting transaction time is that the history of database activities, rather than the history of the real world, is recorded. When a tuple is entered into a database, the transaction start time is set to the current time, making the tuple effective immediately as in a snapshot database. There is no way to record retroactive/proactive changes, nor to correct errors in past tuples. Errors can sometimes be overridden (if they are in the current state) but cannot be forgotten or corrected. For instance, if Merrie's promotion date was later found to be "12/01/82" instead of "12/15/82", this error could not be corrected in a rollback database.

There have been several systems which can be classified as rollback database systems. *MDM/DB* (Model Data Management/Database) presented the history and dynamics in the source data by maintaining cumulative, append-only, time ordered lists of transactions [Ariav & Morgan 1982]. Each transaction contains a time stamp and a pointer to the previous transaction related to the same entity. The status of an entity at any given moment is computed from the collection of transactions for the entity, which have been recorded prior to that moment.

In [Lum et al. 1984], current tuples are stored in a table carrying a time stamp and a pointer to history tuples in reverse time order. History tuples are of the same structure, but are stored in a separate table, and may be compacted to save space. A module to walk through the data and deliver appropriate tuples according to the specified time is created in the database system. To support access of random data, history information for the index is maintained with two trees, a *current index tree* and a *history index tree*. The former contains all the index values from current tuples, the latter for those that existed in the past but no longer in current tuples. These trees are of conventional structures, such as B-trees or B*-trees, with leaves containing pairs of an index value and a pointer. The pointer references a *pointer list* stored in a separate area having a similar structure to the data area. Each pointer list and its history chain correspond to only one index value from one of the two trees.

GemStone [Copeland & Maier 1984], an extension of *Smalltalk-80* [Goldberg & Robson 1983] for database management applications, uses transaction time as an index to map an element name to its

associated value. It supports navigation through history tuples using a notation

```
E!rank@"12/05/82"
```

which retrieves E's rank as of 12/05/82. It represents an object as a block of contiguous memory, which grows with time to retain history data. An object is broken into *elements*, each of which is represented as an element name and a table of *associations*. This table, composed of pairs of transaction times and object pointers, provides the mapping from arbitrary time to an element value. Each pair represents that the element acquired the object as its value at the moment shown in the transaction time. This system was implemented with special purpose hardware.

Version Storage is a component for a distributed data storage system called *SWALLOW*, rather than a database system, but it is mentioned here because it maintains the history of data objects and information necessary for concurrency control and crash recovery [Svobodova 1981]. Each time an object is updated, a tentative version called *token* is created, and eventually saved as a current version if committed. Each version carries a pointer to its immediate predecessor in the history, and a time attribute to specify its range of validity. The *start time* of a version is the time specified in the write request that created the token. A read operation selects a version that has the highest start time lower than the time specified in the read request. The *object header* contains pointers to the current version and a potential token together with information for synchronization and recovery.

[Katz & Lehman 1984] applied database methods to support versions and alternatives in computer aided design. It uses record level versioning to reduce the redundancy of stored records, where each logical record in a design file is identified by a system generated surrogate. A versioned file consists of a *history index* and two separate files. The history index is a B^+-tree with leaf nodes containing pointers *(version history)* to records stored in either of the two separate files. Though time domain addressing is not supported explicitly, it is possible to access all records within a version, or all versions of a certain logical record. But it is not clear how to handle inserted records which disrupt the logical ordering of surrogate values.

Compared to historical or temporal DBMS's, there have been more efforts for the actual implementation of rollback DBMS's. Some of the systems described above are being implemented, or have already been built. All of these support transaction time explicitly or implicitly for rollback

operations. However, there is no query language to express complex relationships among history data, which is only reasonable considering the simple semantics of rollback databases.

3.3. Historical Databases

Another alternative is the *historical database* which records the history of the real world by supporting *valid time*, the time when the relationship in the enterprise being modeled is valid. While a rollback database records a sequence of snapshot states, a *historical database* records a single *historical state* per relation. As errors are discovered, they are corrected by modifying the database. Previous states of the database itself are *not* retained, so it is not possible to view the database as it was at a past moment. No record is kept about errors that have been corrected. Historical databases are similar to snapshot databases in this respect.

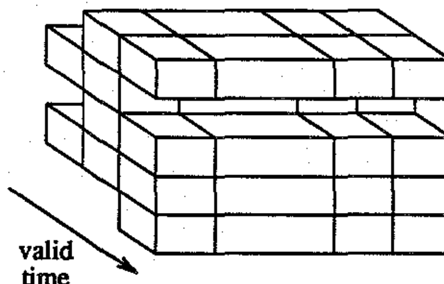


Figure 3-4: Historical Relation

Another distinction between historical and rollback databases is that historical databases support arbitrary modification, whereas rollback databases only allow snapshot states to be appended. The same sequence of transactions which led to the rollback relation in Figure 3-2 followed by a change of the valid from time will result in the historical relation in Figure 3-4, where the label of the time axis indicates the valid time. However, the historical relation can represent that a later transaction has changed the time when a tuple takes effect in the relation, which is not possible on a rollback relation. Rollback DBMS's can rollback to an incorrect previous snapshot relation; historical DBMS's can represent the current knowledge about the past.

As with rollback databases, implementing a historical relation directly as a sequence of snapshot states is impractical. Figure 3-5 illustrates an alternative: appending the implicit time attributes *valid from* and *valid to* to each tuple, indicating the period while the tuple was actually in effect. Like the transaction time attributes in rollback databases, the valid time attributes are not included in the relation scheme. Note that the relation in Figure 3-5 models an *interval*. A relation modeling an *event* needs only one attribute *valid from*. The value '∞' for the valid to attribute denotes 'forever', distinguished from the value '–' for the transaction stop attribute. Handling incomplete information is beyond the scope of this dissertation.

Name	Rank	valid time (from)	valid time (to)
Merrie	Associate	09/01/77	12/01/82
Merrie	Full	12/01/82	∞
Tom	Associate	12/05/82	∞
Mike	Assistant	01/01/83	03/01/84

Figure 3-5: A Historical Relation

The semantics of valid time is closely related to reality, hence more complex than the semantics of transaction time concerned with database activities. Therefore, historical databases need sophisticated operations to manipulate the complex semantics of valid time adequately. TQuel supports such queries (termed *historical queries*) by augmenting the `retrieve` statement with a `valid` clause to specify how the implicit temporal attributes are computed, and a `when` predicate to specify the temporal relationship of tuples participating in a derivation. These added constructs handle complex temporal relationships such as `precede`, `overlap`, `begin of`, and `end of`. The TQuel query requesting Merrie's rank when Tom arrived,

```
range of f1 is Faculty
range of f2 is Faculty
retrieve (f1.Rank)
     where f1.Name = "Merrie"  and  f2.Name = "Tom"
     when  f1 overlap begin of f2
```

on the historical relation 'Faculty' in Figure 3-5 yields

Rank	valid time	
	(from)	(to)
Full	12/01/82	∞

Note that the derived relation is also a historical relation, which may be used in further historical queries. While both this query and the example given for a rollback relation seem to ask Merrie's rank on 12/05/82, the answers are different. The reason is that Merrie was promoted on 12/01/82, but this information was recorded into the rollback database of Figure 3.3 two weeks later. The historical database of Figure 3.5 represents the correct information, but it is not possible to determine whether some error had ever been corrected.

Historical databases have been the subject of several research efforts, especially on the conceptual aspects such as formal semantics and the design of query languages. *LEGOL 2.0* [Jones & Mason 1980] was developed for writing complex rules such as those in legislation or high level system specifications where the correct handling of time is important. It augments each tuple with two time attributes, *start time* and *end time*, which delimit the period of existence for the associated member of the entity set. Its query language is based on the *relational algebra* with temporal operators such as *while, during, since, until, begin of*, and *end of*.

Clifford and Warren presented a formal semantics for time in databases [Clifford & Warren 1983] based on the *intensional logic* (IL_s), where a database is a collection of relations idealized as a cube fully specified over a set of *states*.

CSL (Conceptual Schema Language) is a high level data definition language to define conceptual schemas, not only for static but also for dynamic aspects of the database universe. It has the option of embedding database instances into the time axis based on an application specific calendar system [Breutmann et al. 1979].

TERM (Time-extended Entity Relationship Model) augments the entity-relationship model to include the semantics of temporal aspects into the database schema. It provides facilities for data definition and manipulation of problem dependent representation structures for time, values and histories [Klopprogge 1981].

[Findler & Chen 1971] built a question answering system which understands explicit or implicit temporal relations and causal relationships among time-dependent events based on information entered by a user. It used *AMPPL-II* (Associative Memory Parallel Language II), and stored data in list structures or as a sequence of content-addressable relations.

HTQuel (Homogeneous Temporal Query Language) is based on the representation of a historical database where the time intervals are associated with attributes [Gadia & Vaishnav 1985]. The language introduces the *temporal domain* which is finite unions of intervals, and is based on the *homogeneity requirement* that the temporal domains of all the attributes in a tuple should be the same. The semantics of temporal operators were defined using *snapshots*.

As described above, significant contributions have been made to the conceptual aspects, such as formal semantics and the design of query languages of the historical DBMS. But little work has been done towards the actual implementation, except that an earlier version of LEGOL 2.0 [Jones et al. 1979] was implemented.

3.4. Temporal Databases

Benefits of both approaches can be combined by supporting both kinds of time in a database. Such a database supporting both transaction time and valid time is termed a *temporal database* in the narrower sense to emphasize the need for both kinds of time in handling temporal information. The rollback database views stored tuples, whether valid or not, *as of* some moment in the past, and the historical database views tuples *valid* at some moment as of *now*. But the temporal database can view tuples *valid* at some moment seen *as of* some other moment, thereby completely capturing the history of retroactive and proactive changes. Users of a temporal DBMS can examine historical information from the viewpoint of a previous state of the database by specifying both kinds of time in a query.

Since there are two orthogonal time axes involved now, a temporal relation should be illustrated in four dimensions. Figure 3-6 shows a *single* temporal relation which may be regarded as a sequence of historical states, each of which is a complete historical relation.

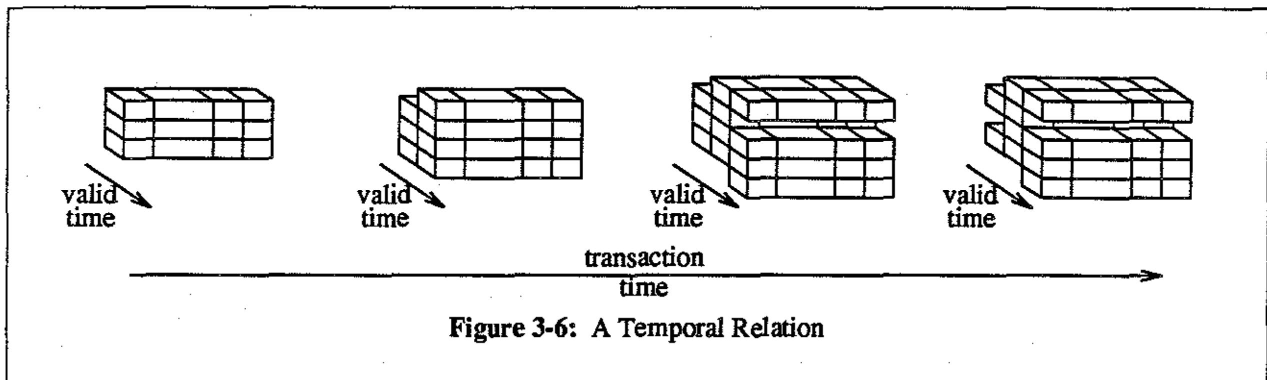


Figure 3-6: A Temporal Relation

The rollback operation on a temporal relation selects a particular historical state, on which a historical query may be executed. Each transaction causes a new historical state to be created. Thus, temporal relations are append-only. The temporal relation in Figure 3-6 is the result of four transactions, starting from a null relation:

(1) three tuples were added,

(2) one tuple was added,

(3) one tuple was added and an existing one deleted, and

(4) one tuple was modified so that it became effective at a later valid time.

Name	Rank	valid time		transaction time	
		(from)	(to)	(start)	(end)
Merrie	Associate	09/01/77	∞	08/25/77	12/15/82
Merrie	Associate	09/01/77	12/01/82	12/15/82	–
Merrie	Full	12/01/82	∞	12/15/82	–
Tom	Full	12/05/82	∞	12/01/82	12/07/82
Tom	Associate	12/05/82	∞	12/07/82	–
Mike	Assistant	01/01/83	∞	01/10/83	02/25/84
Mike	Assistant	01/01/83	03/01/84	02/25/84	–

Figure 3-7: A Temporal Relation

For example, the relation in Figure 3-7 combines information represented in Figures 3-3 and 3-5, supporting both valid time and transaction time. It has four implicit time attributes: *valid from*, *valid to*, *transaction start*, and *transaction stop*. It shows that Merrie started as an assistant professor on 09/01/77, which was recorded into the database on 08/25/77 as a proactive change. Then she was promoted on 12/01/82, but the fact was recorded retroactively on 12/15/82. Tom was entered into the database on

12/01/82, joining the faculty as a full professor on 12/05/82, but the fact that his rank was actually an associate professor was corrected on 12/07/82. Mike left the faculty effective on 03/01/84, which was recorded proactively on 02/25/84. Note all the details of history captured here, which were not expressible in other databases with less temporal support.

There are only three examples which can be cited as temporal databases. *TQuel* (Temporal QUEry Language) [Snodgrass 1986] is an extension of a relational calculus query language *Quel* [Held et al. 1975] for supporting temporal queries. TQuel uses the **as of** clause to perform rollback operation, and the **when** clause for specifying historical queries. Further details on TQuel were given in Section 2.3.

Since TQuel supports both historical queries and rollback operations, it can be used to query temporal databases. The TQuel query

```
range of f1 is Faculty
range of f2 is Faculty
retrieve (f1.Rank)
      where f1.Name = "Merrie"  and  f2.Name = "Tom"
      when  f1 overlap begin of f2
      as of "12/10/82"
```

on this relation retrieves Merrie's rank when Tom arrived, according to the state of the database as of 12/10/82. The result is

Rank	valid time		transaction time	
	(from)	(to)	(start)	(end)
Associate	09/01/77	∞	08/25/77	12/15/82

This derived relation is a temporal relation, so further temporal relations can be derived from it. If a similar query were made as of 12/20/82, the answer would be *Full* because the fact was recorded retroactively by that time.

TRM (Temporal Relational Model) [Ben-Zvi 1982] is another example of a temporal database, though it was not actually implemented. It maintains 5 time attributes:

Tes/Tee : effective-time-start/end
Trs/Tre : registration-time-start/end
Td : deletion time

in each tuple, where deletion time is used to correct erroneous data [Ben-Zvi 1982]. It extends *SQL* [IBM

1981] with the *time view* operator to search data effective at some moment seen as of some other point in time.

name	rank	Tes	Tee	Trs	Tre	Td
Merrie	Associate	09/01/77	12/01/82	08/25/77	12/15/82	-
Merrie	Full	12/01/82	-	12/15/82	-	
Tom	Full	12/05/82	-	12/01/82	-	12/07/82
Tom	Associate	12/05/82	-	12/07/82	-	-
Mike	Assistant	01/01/83	03/01/84	01/10/83	02/25/84	-

Figure 3-8: A TRM Relation

The relation in Figure 3-8 shows the same information contents as in Figure 3-7. Note that "Tom" was mistakenly entered as a full professor on 12/01/82 (which was a proactive entry), but corrected later using the *deletion time*. A query

```
TIME-VIEW   E-TIME=12/5/82    AS-OF=12/10/82
SELECT      RANK
FROM        FACULTY
WHERE       NAME = "Merrie"
```

on the relation in Figure 3-8 gives the answer *Assistant* since her promotion was not recorded until 12/15/82. If a similar query is made as of 12/20/82,

```
TIME-VIEW   E-TIME=12/5/82    AS-OF=12/20/82
SELECT      RANK
FROM        FACULTY
WHERE       NAME = "Merrie"
```

the answer is *Associate* because the fact was recorded retroactively by that time. However, this is not a true temporal query language, because it can derive only snapshot relations.

TODMS (Temporally Oriented Data Management System) is similar to TRM in that it supports both valid and transaction time, and its query language is an extension of SQL [Ariav 1984]. Unlike TRM, it is a true temporary query language, supporting both historical queries and rollback operations. The major limitation is that only one relation may be referenced in a query, and no implementation has been

attempted.

3.5. User-defined time

User-defined time is necessary when additional temporal information, not handled by transaction or valid time, is stored in the database. Such an attribute needs to be specified in the relation scheme. The values of user-defined temporal attributes are not interpreted by the DBMS, and thus are easy to support. The system only needs to provide definitions of external and internal representations, and input/output functions to convert one form to the other. Multiple representations with varying resolutions, each associated with input and output, are also useful. As an example of user-defined time, consider a 'Promotion' relation with three attributes: `Name`, `Rank`, and `Approval-Date`. `Approval-Date` is the user-defined time indicating when the promotion was approved. The valid time is the date when the promotion takes effect, and the transaction time is the date when the promotion was recorded into the database.

Supporting user-defined time is orthogonal to supporting rollback operations or historical queries. Hence the three kinds of time actually define eight different types of databases. However, we note that user-defined time is much closer to valid time than to transaction time, in that both valid time and user-defined time are concerned with *reality* itself, as opposed to transaction time which is concerned with the *representation* of reality (i.e., the database). Database management systems and their query languages purporting to provide full temporal support should handle all three kinds of time.

3.6. Summary

Four types of databases in terms of temporal support were defined and compared with one another. Snapshot databases provide no temporal support. Rollback databases provide rollback operations requiring the support of transaction time, which records the history of database activities. Historical databases provide historical queries requiring the support of valid time, which is associated with the history of the real world. Temporal databases provide both rollback operations and historical queries, supporting both transaction time and valid time. Figure 3-9 shows the four types of databases differentiated by the capability to support rollback operations and historical queries: snapshot, rollback, historical and temporal. Each of these types may or may not support user-defined time.

	No Rollback	Rollback
Snapshot Queries	Snapshot	Rollback
Historical Queries	Historical	Temporal

Figure 3-9: Types of Databases

Figure 3-10 summarizes the kinds of time to be supported in each type of database management systems.

	Transaction	Valid	User-defined
Snapshot			
Rollback	√		
Historical		√	√
Temporal	√	√	√

Figure 3-10: Time to be Supported by Databases

It is interesting to note that those concerned with the physical implementation leaned towards *transaction time*, while those more interested in conceptual aspects favored *valid time*. Most implementation oriented efforts have been on the version management systems [Katz & Lehman 1984, Svobodova 1981], or rollback DBMS's [Ariav & Morgan 1982, Copeland & Maier 1984, Lum et al. 1984]. To the author's knowledge, there has been no major effort to investigate implementation aspects for either the historical or the temporal DBMS, let alone the performance analysis of such systems.

Chapter 4

Models and Performance Analysis

As described in Section 2.1, there have been several models and systems attempting to analyze the performance of database management systems with various forms of access methods. However, none of those actually address the whole problem of evaluating the access cost given queries as input, nor can adequately handle the particular characteristics of query processing and access methods for databases with temporal support considered in this dissertation. Therefore, a set of new models to analyze the performance of database management systems with temporal support were developed. The first section of this chapter describes the models, and the second section discusses how these models can be combined together to estimate the I/O cost given one or more TQuel queries as input.

4.1. Models

Performance analysis of a database management system requires models, whose quality determines the effectiveness of the analysis. We want to analyze the input and output cost for temporal queries on a database with temporal support using various access methods. Thus we need models which can characterize various phases of query processing in database management systems with temporal support. For this purpose, four models forming a hierarchy were developed: one each for *algebraic expressions, database/relations, access paths,* and *storage devices.*

4.1.1. Model of Algebraic Expressions

TQuel is a language based on the tuple calculus, and hence is non-procedural. There are many different ways to evaluate a TQuel query and obtain the same answer, each exhibiting different I/O cost. This section first defines the *algebraic expression* to describe procedurally the process of evaluating TQuel queries. Next, the *file primitive expression* is defined to characterize the input and output activities involved in evaluating the algebraic expression. Finally, the *model of algebraic expressions* is constructed to represent the mapping between the algebraic expression and the file primitive expression.

4.1.1.1. Algebraic Expressions

An algebraic expression consists of *algebraic operators* and *connectives*. Algebraic operators are of three types: *snapshot*, *temporal*, and *auxiliary*.

Snapshot operators are the conventional relational operators such as `Select`, `Project`, `Join`, `Union` and `Difference`. `Select` has two parameters: a relation and a predicate to specify the constraint that result tuples must satisfy. `Project` takes as parameters a relation and a set of attributes to be extracted from the relation. `Join` is to perform θ-*join* of two relations given as the first two parameters. The third parameter, the *join method*, specifies how to perform the *join* operation, since there are many ways to perform the operation. The fourth parameter is the predicate specifying how to combine information from two relations. Both `Union` and `Difference` take two relations as parameters, performing set addition and set subtraction respectively.

Temporal operators are included for temporal query constructs in TQuel. `When` performs temporal selection on a relation according to a temporal predicate applied to the values of *valid time* attributes. `AsOf` also performs temporal selection on a relation, but takes two time constants as parameters to compare with the values of *transaction time* attributes. `Valid` performs temporal projection, determining the value of the attribute *valid from*, *valid to*, or *valid at*.

Auxiliary operators are introduced to account for miscellaneous operations which do not change the query result but affect the query cost significantly. `Temporary` is used to create and access a temporary relation for the result of the operation marked by the parameter `label`. `Sort` is used to sort tuples in the relation specified by the first parameter, using the remaining parameters as the key attributes for sorting. `Reformat` is used to change the structure of the relation specified by the first parameter to the form given by the second parameter, using the remaining parameters as the key attributes.

These algebraic operators can be combined together through *connectives* which specify information on ordering and grouping of the component operators. Two operators may be ordered in sequence, expressed as

```
Op1 ; Op2
```

when `Op1` should complete execution before `Op2` starts. Or they may be in parallel, denoted by

```
Op1 , Op2
```

when two operations can proceed concurrently. Grouping of operators to delimit a query is denoted by a pair of braces, '{' and '}', while a pair of square brackets, '[' and ']', represent a set of operators which can be evaluated simultaneously for each tuple. These connectives can characterize different strategies for evaluating a query expressed by a combination of algebraic operators.

An operator may have a *label* which can be referred to in other operators such as **`Temporary`**. By using labels, we can eliminate deeply nested parentheses common in algebraic descriptions of a query. Thus an *algebraic expression*, describing TQuel queries in a procedural form, is a combination of labels, algebraic operators with appropriate parameters, and connectives.

For example, an algebraic expression, to be referred to as AE-1,

```
{ L1:   Select    (h, h.id = 500);
        Project   (L1, h.id, h.seq) }
```

specifies that it is for a single query that selects tuples with `id` = 500 from the relation `h`, then extracts attributes `id` and `seq` from the result of the previous operation labeled as `L1`.

Another example is AE-2:

```
{[ L1:  Select    (h, h.id = 500);
        Project   (L1, h.id, h.seq) ]}
```

This is similar to AE-1, but specifies that **`Select`** and **`Project`** can be evaluated together for each tuple. Thus the need for a temporary file to store intermediate results between the two operations is explicitly eliminated.

Abbreviated BNF syntax for the algebraic expression is shown in Figure 4-1. In this description, `<temporal pred>` is a *temporal predicate* involving time attributes and *temporal predicate* operators such as **`precede`** and **`overlap`** in TQuel. `<event expr>` is an *event expression* involving time

attributes and *temporal constructor* operators such as **extend** and **overlap** in TQuel, which yields a time value as its result. Complete syntax for the temporal predicate and the event expression is given in Appendix A. `<stor spec>` specifies one of the storage structures such as `Heap, Hash, Isam, Btree`, *etc.*, or one of the new access methods to be developed in Chapter 5.

```
<alg exp>     ::=   <query>
              |     <alg exp>  <query>
<query>       ::=   {  <access>  }
<access>      ::=   <acc term>
              |     <access>  <acc term>
<acc term>    ::=   <term>
              |     [  <term>  ]

<term>        ::=   <l oper>
              |     <term>  <order>  <l oper>
<order>       ::=   ;      |      ,

<l oper>      ::=   <oper>
              |     <label>  :  <oper>
<label>       ::=   <id>

<oper>        ::=   <Snapshot>
              |     <Temporal>
              |     <Auxiliary>

<Snapshot>    ::=   Select       ( <rel> ,  <predicate> )
              |     Project      ( <rel> ,  <attr list> )
              |     Join         ( <rel> ,  <rel> ,  <join method>
                                         ,  <predicate> )
              |     Union        ( <rel> ,  <rel> )
              |     Difference   ( <rel> ,  <rel> )

<Temporal>    ::=   When     ( <rel> ,  <temporal pred> )
              |     AsOf     ( <rel> ,  <event expr> ,  <event expr> )
              |     Valid    ( <rel> ,  <FTA> ,  <event expr> )

<Auxiliary>   ::=   Temporary ( <label> )
              |     Sort      ( <rel> ,  <attr list> )
              |     Reformat  ( <rel> ,  <stor spec> ,  <attr list> )
<FTA>         ::=   From
              |     To
              |     At

<attr list>   ::=   <attribute>
              |     <attr list>  ,  <attribute>

<rel>         ::=   <rel id>   |   <label>
```

Figure 4-1: BNF Syntax for Algebraic Expressions

A more complex example is AE-3:

```
{  L1:  Join       (h, i, TS, h.id = i.amount & h overlap i);
   L2:  When       (L1, i overlap "now");
        Project    (L2, h.id, i.id, i.amount)     }
```

This specifies **Join** of two relations, h and i, followed by temporal selection **When**, followed by **Project**, all in sequence. Another example is AE-4:

```
{  L1:  When       (i, i overlap "now");
   L2:  Project    (L1, i.id, i.amount, i.valid_from, i.valid_to);
   L3:  Join       (h, L2, TS, h.id = i.amount & h overlap i);
        Project    (L3, h.id, i.id, i.amount)     }
```

This is functionally equivalent to AE-3, but differs in evaluation procedures. AE-4 specifies that the **When** operation is first executed to select tuples from the relation i whose valid to attribute is "now", then four attributes are extracted from the result tuples, then the result is joined with the relation h using *tuple substitution* (TS), and finally three attributes are extracted. However, AE-4 does not provide information on what operations can proceed together and whether a temporary relation is needed. Adding such information leads to AE-5:

```
{[ L1:  When       (i, i overlap "now");
   L2:  Project    (L1, i.id, i.amount, i.valid_from, i.valid_to)];
   L3:  Temporary  (L2);
 [ L4:  Join       (h, L3, TS, h.id = i.amount & h overlap i);
        Project    (L4, h.id, i.id, i.amount)     ]}
```

This is similar to the previous expression AE-4, but specifies that **When** and **Project** can be evaluated together on each tuple, the intermediate result is stored into a temporary relation, and **Join** and **Project** can also be performed together.

4.1.1.2. File Primitive Expressions

In this section, we define the *file primitive expression* which represents the process of accessing a file in terms of two file primitives: **Read** and **Write**. Both of the primitives take parameters such as the access method, the size of a file, or the length of the overflow chain. The access method may be one of

Heap, Hash, Isam, Btree, *etc.*, or one of the new access methods to be developed in the next chapter.

Primitives are combined to form an arithmetic expression, called the file primitive expression, to describe the situation when one or more primitives are repeated or executed together to perform an algebraic operation. Abbreviated BNF syntax for the file primitive expression is shown in Figure 4-2, where <expr> is evaluated to a constant, and <parm> is a constant to denote the size of a file, or the length of the overflow chain.

```
<fpe>         ::=   <term>
              |     <fpe>     <a op>      <term>
<term>        ::=   <primitive>
              |     <term>    <m op>    <primitive>

<primitive> ::=     <oper>    (    <acc method>    <parm list>    )
              |     (    <fpe>    )

<oper>        ::=   Read
              |     Write

<parm list> ::=     <parm>
              |     <parm list>    <parm>

<a op>        ::=   +      |      -

<m op>        ::=   *      |      /
```

Figure 4-2: BNF Syntax for File Primitive Expressions

For example, a file primitive expression may be as simple as FPE-1:

```
Read  (Hash, 0)
```

specifying one hashed access without any overflow records, or more complex like FPE-2:

```
  Read  (Heap, 128)            +
( Read  (Heap, 19) * 2 - 1     +
  Write (Heap, 19) * 3 - 1 )   +
  Read  (Heap, 19)             +
  Read  (Hash, 0) * 1024
```

specifying one *read* from the heap of 128 blocks, two *read*'s from the heap of 19 blocks, three *write*'s to the heap of 19 blocks, another *read* from the heap of 19 blocks, and finally a hashed access repeated 1024 times.

4.1.1.3. Model of Algebraic Expressions

Now that the algebraic expression and the file primitive expression have been defined, the *model of algebraic expressions* is constructed to represent how the algebraic expression can be evaluated in terms of the file primitive expression. For example, the algebraic expression AE-2 can be mapped to the file primitive expression FPE-1 shown earlier, assuming that the relation `h` is hashed on the attribute `id` with no overflow records.

There are a large number of valid combinations for algebraic expressions even for conventional snapshot databases. The problem gets more complicated with introducing historical queries and rollback operations for temporal databases. It is neither possible nor useful to list all the possible algebraic expressions and evaluate their costs one by one. Rather, we identify basic constructs occurring in snapshot and temporal queries, and map the subset of *algebraic expressions*, composed of such constructs, to *file primitive expressions*. The mapping is also dependent on the characteristics of data such as the structure and the size of each relation, selectivity and distribution of each attribute value, and the update count in case of a database with temporal support, as will be represented by the model of database/relations in the next section.

Algebraic operators involve either one relation or two relations. **`Select`**, **`Project`**, **`When`**, **`AsOf`**, **`Valid`**, **`Temporary`**, **`Sort`**, and **`Reformat`** operate on one relation, while **`Join`**, **`Union`**, and **`Difference`** operate on two relations. The characteristics of each operator is discussed one by one in terms of the file primitive expression.

- **`Select`** (*relation*, *predicate*)

  The first parameter *relation* is the base relation for the operation, and the second parameter *predicate* specifies constraints on the *relation* that result tuples must satisfy. Performance of **`Select`** depends on various factors such as the structure of the *relation*, the type of the *predicate*, and the characteristics of data stored in the *relation*.

(1) If the *predicate* fully specifies a key for a random access path existing for the *relation*, the file primitive expression is:

```
Read    (access path, n)
```

where the *access path* may be one of `Hashing`, `Isam`, `Btree`, or one of the new access methods to be developed in the next chapter. The second parameter n is the length of the overflow chain, which is determined from the model of database/relations.

(2) Otherwise, the file primitive expression is:

```
Read    (Heap, b)
```

where b is the size of the *relation* in blocks, meaning the relation is sequentially scanned.

- `Project` (*relation, attr list*)

  This operation scans the *relation* to extract a list of attributes, *attr list*, hence its file primitive expression is:

```
Read    (Heap, b)
```

  where b is the size of the *relation* in blocks.

- `Join` (*relation*$_1$, *relation*$_2$, *join method, predicate*)

  There are several methods to perform a join, such as `TS`, `BS`, and `SM`. Let

  t_1 : the number of tuples in *relation*$_1$
  t_2 : the number of tuples in *relation*$_2$
  b_1 : the size of *relation*$_1$ in blocks
  b_2 : the size of *relation*$_2$ in blocks

  Each method is briefly described with the corresponding file primitive expression.

  (1) `TS` : tuple substitution method

  Each tuple in the smaller relation is substituted to select tuples from the other relation satisfying the *predicate*.

```
Read   (Heap, b1) +
FPE2 * t1
```

assuming $t_1 < t_2$. FPE_2 is the file primitive expression for

```
Select   (relation2, predicate')
```

where *predicate'* is the predicate with the tuple variable for *relation*$_1$ replaced by each tuple in *relation*$_1$.

(2) BS : block substitution method

For each block in the smaller relation, the other relation is scanned. In this process, all tuples in one block of each relation are joined according to the *predicate*. It is faster than *tuple substitution* especially when there is no random access path to evaluate the *predicate*.

```
Read   (Heap, b1) +
Read   (Heap, b2)  * b1
```

where $b_1 < b_2$.

(3) SM : sort & merge method

Each relation is sorted first, then the resulting relations are scanned in parallel to merge tuples satisfying the *predicate*.

```
Read   (Heap, b1) +
Read   (Heap, b2) +
FPE  (Sort  (relation1,  attr list) )   +
FPE  (Sort  (relation2,  attr list))
```

where *FPE* (**Sort** (...)) is the file primitive expression for **Sort** to be described later, and *attr list* is the list of attributes participating in the *predicate*. If both relations are already in order, the file primitive expression is simply

```
Read  (Heap, b1) +
Read  (Heap, b2)
```

- **Union** (*relation*$_1$, *relation*$_2$)
- **Difference** (*relation*$_1$, *relation*$_2$)

Both operators need to scan two relations, so the file primitive expression is:

```
Read  (Heap, b1) +
Read  (Heap, b2)
```

where b_1 and b_2 are the sizes of *relation*$_1$ and *relation*$_2$, respectively, in blocks.

- **When** (*relation, temporal pred*)

When is similar to **Select**, where the temporal predicate, *temporal pred*, is restricted to a single variable predicate specifying the constraint on the *valid time* attributes that result tuples must satisfy. Hence the file primitive expression is, like **Select**:

```
Read  (access path, b)
```

or

```
Read  (Heap, b)
```

depending on the type of the *predicate*, and the existence of a random access path to satisfy the temporal predicate.

- **AsOf** (*relation*, t_1, t_2)

AsOf is similar to **Select** with the predicate of:

where $t_1 \leq$ transaction_stop and transaction_start $\leq t_2$

Hence the file primitive expression is similar to that for **Select**.

- **Valid** (*relation, From/To/At, temporal expr*)

Valid is similar to **Project**, where the temporal expression, *temporal expr*, is restricted to a

single variable expression with the domain of time values. The file primitive expression is

```
Read  (Heap, b)
```

where b is the size of the *relation* in blocks.

- **Temporary** (*label*)

  This operator, as shown in AE-5, is to create a temporary relation, and to store the intermediate result from the previous operation marked by the *label*. Its file primitive expression is in general:

  $$(\textbf{Read} \ (Heap,\ b) * k_r - l_r + \textbf{Write} \ (Heap,\ b) * k_w - l_w)$$

  where b is the number of blocks in the resulting relation, and k_r, l_r, k_w, l_w are implementation dependent constants. For the prototype to be used in Chapters 6 and 7, each block, except the last one, of a temporary relation is read twice and written three times, so $k_r = 2$, $k_w = 3$, and $l_r = l_w = 1$.

  ```
  ( Read  (Heap, b) * 2 - 1    +
    Write (Heap, b) * 3 - 1 )
  ```

- **Sort** (*relation, attr list*)

  This is used to sort the *relation* using a list of attributes, *attr list*, as key attributes for sorting. Since it takes $O\ (b \times \log_m b)$ block accesses to sort a file of b blocks using the m-way sort-merge, the file primitive expression is in general:

  $$\begin{aligned} &\textbf{Read} \ (Heap,\ b_1) * O\,(log_m b_1) + \\ &\textbf{Write} \ (Head,\ b_1) * O\,(log_m b_2) + \\ &\textbf{Read} \ (Heap,\ b_2) * O\,(log_m b_2) + \\ &\textbf{Write} \ (Head,\ b_2) * O\,(log_m b_2) \end{aligned}$$

- **Reformat** (*relation, stor spec, attr list*)

  This is to reformat the *relation* to the storage structure, *stor spec*, using a list of attributes, *attr list*, as key attributes. Its file primitive expression is in general:

```
( Read  (Heap, b)          +
  Write (Heap, b)          +
  FPE  (Sort  (relation, attr list) )
```

where *FPE* (Sort (...)) is the file primitive expression for Sort in case we need to sort the *relation* for reformating.

Thus far, each operator has been discussed in terms of file primitive expressions. An algebraic expression with multiple operators can be mapped to the file primitive expression which is the sum of the file primitive expressions for the component operators. An exception to this rule is the case when Project or Valid follows Select, Join, or When, and the two operations are grouped together by a pair of square brackets. In this case, the file primitive expression is simply that of the first operation. For example, an algebraic expression

```
{[ L1:  Select     (h, id = 500);
        Project    (L1, h.id, h.seq) ]}
```

is mapped to

```
Read  (Hash, 0)
```

performing Project effectively for free.

4.1.2. Model of Database/Relations

The second model in the hierarchy is the *model of database/relations* which characterizes information on the relations composing a database. Typical catalog relations in conventional DBMS's hold information for all relations such as relation names, temporal types, storage structures, attribute counts, attribute names, attribute formats, attribute lengths, key attributes, tuple lengths, and tuple counts.

Additional information on data contents is needed to provide data for the model of algebraic expressions so that the algebraic expression can be mapped to the file primitive expression. Examples are selectivity and distribution of attribute values, volatility of data, and the update count in case of a database with temporal support. Figure 4-3 shows an abbreviated *IDL* (Interface Description Language [Nestor et

al. 1982]) description of information to be represented by the model of database/relations.

```
Structure   DbRel Root  database    Is

database     =>    name              :  String,
                   relations         :  Set Of relation;

relation     =>    name              :  String,
                   temporalType      :  TemporalType,
                   attributes        :  Seq Of attribute,
                   tupleCount        :  Integer,
                   updateCount       :  Integer,
                   storageType       :  StorageType,
                   keys              :  Seq Of key,
                   loadingFactor     :  Rational,
                   blockSize         :  Integer;

TemporalType  ::= snapshot                 | rollback              |
                  historicalInterval       | historicalEvent       |
                  temporalInterval         | temporalEvent;
      snapshot =>;                         rollback =>;
      historicalInterval =>;         historicalEvent =>;
      temporalInterval =>;           temporalEvent =>;

key          =>    name              :  String,
                   attributes        :  Seq Of attribute;

attribute    =>    name              :  String,
                   type              :  ValueType,
                   length            :  Integer,
                   selectivity       :  Rational,
                   volatility        :  Rational;

ValueType    ::=   typeInteger       | typeRational     |
                   typeString        | typeBoolean      |
                   typeTime;
      typeInteger =>;                typeRational =>;
      typeString =>;                 typeBoolean =>;
      typeTime =>;
End
```

Figure 4-3: IDL Description for the Model of Database/Relations

In this description, a database consists of a name and a set of relations. Each relation consists of a name and various information on the relation. For example, `temporalType` specifies one of six possible temporal types: snapshot, rollback, historical interval, historical event, temporal interval, and temporal event. `storageType` specifies the storage structure of the relation, whether it is a heap, a hashed file, an ISAM file, or one of the structures to be discussed in Chapter 5. An example of a database

represented in IDL's ASCII external representation is found in Appendix D.

It is a difficult problem to estimate the response set of a query and the number of block accesses without actually examining stored data, though there has been significant research on the subject as summarized in section 2.1.2. This problem may account for a large portion of the discrepancy between the analysis result and the actual performance data.

4.1.3. Model of Access Paths

The third model in the hierarchy is the *model of access paths (MAP)* which represents the path taken through the storage structure to satisfy an access request represented by a file primitive expression. An access path is usually confined to a single file, but it may involve more than one file, which is the case with storage structures for temporal databases discussed in the next chapter. This section first describes how the model of access paths represents a *single file path*, and then extends it for a *multiple file path*.

The conceptual unit of an access in this model is a *node*, which consists of one or more *physically contiguous* records participating in the access. The node itself consists of one or more *records*, depending on the underlying storage structure.

A set of nodes are connected together to make up an access path either *directly* or *indirectly*. In simple cases, an access path is directly represented as a set of nodes. In other cases, it helps to conceptualize an access path as being composed of some components, each of which is itself a set of nodes. This process of *hierarchical decomposition* may proceed for as many levels as useful.

The process of decomposition is restricted to three levels, which is sufficient to describe the storage structures discussed in this dissertation. However, it is straightforward to extend it to incorporate more levels. In this three level hierarchy, a set of nodes are grouped to make up a *chain*, and a set of chains compose an access path. Therefore, an access path through a single file, or simply a *file path*, is represented as a set of chains, each of which is a set of nodes. As mentioned above, each node itself consists of one or more records.

The model of access paths identifies a fixed number of *modes*, specifying how components such as nodes, chains, or file paths are connected with one another. We can classify the modes as either *guided* or *searched*.

Guided : If a random access mechanism exists to locate the component

H : the address is computed by a *hash* function

P : the address is provided by a *pointer*

A : the component is physically *adjacent* to its predecessor

S : the component shares the *same* starting address with its higher level component

M : the component is in the *main memory*

Searched : If no random access mechanism exists

O : the file is *ordered*, so logarithmic search is possible

U : the file is *unordered*, so sequential search is necessary.

This process of hierarchical decomposition, decomposing an *access path* or a *file path* into *chains*, a chain into *nodes*, and a node into *records*, is all captured into a single expression called the *access path expression (APE)*. A canonical form for an access path expression, whose syntax is shown in Figure 4-4, is

$$(Mode\ \ count_1\ (Mode\ \ count_2\ (Mode\ \ count_3)^+)^+)$$

where

$count_1$ is the number of chains in the file path,

$count_2$ is the number of nodes in the chain, and

$count_3$ is the number of records in the node.

As described earlier, the components in the three level hierarchy are the access path, chains, and nodes. Each component is described by a '*mode-count*' pair, where the *mode* tells how to locate the component, and the *count* shows the number of subcomponents in it. Then the '*mode-count*' pair is followed by a list of descriptors for its subcomponents enclosed in parentheses. The level of a component in the hierarchy is determined by the depth of enclosing parentheses. The outermost parentheses represent the access path, while the innermost parentheses represent a node which is defined to consist of records.

```
<APE>        ::=   <FilePath>

<FilePath>   ::=   (   <desc>   <chains>   )
<desc>       ::=   <Mode>   <count>

<chains>     ::=   <chain>
             |     <chains>   <chain>
<chain>      ::=   (   <desc>   <nodes>   )

<nodes>      ::=   <node>
             |     <nodes>   <node>
<node>       ::=   (   <desc>   )

<Mode>       ::=   H
             |     P
             |     A
             |     S
             |     M
             |     O
             |     U
<count>      ::=   <integer>
```

Figure 4-4: BNF Syntax for File Path Expressions (Single File)

Each subcomponent is described one by one in sequence, but if all the successors of a certain subcomponent are the same, they need not be repeated. Therefore, if the number of descriptors is smaller than the specified count, the remaining subcomponents are assumed to have the same descriptor as the last one. When a component has only one subcomponent and the mode of the subcomponent is `S` (meaning the subcomponent shares the same starting location), the extra level of decomposition does not provide any further information, and may be omited.

In the access path expression, a set of file parameters are used to quantify physical properties of a file. Some of the parameters are:

f: number of records in a file

b: number of records in a block

r: number of bytes in a record, and

n: number of records to be accessed.

Some examples of access path expressions are described now for various access methods.

Example-1. Scanning a sequential file:

The access path can be considered as an unordered collection of f records. The access path

expression is:

```
(U f* records)
```

or simply:

```
(U f)
```

Since the head of the path expression is U, the path needs to be searched sequentially. The access path can also be regarded as consisting of a single node, which has *f* records. Then the expression becomes:

```
(U 1 (S f))
```

We can follow the three level hierarchy by introducing the level of *chain*. Then the access path has a single chain, which has one node. The node itself consists of *f* records.

```
(U 1 (S 1 (S f)))
```

Example-2. Accessing a hashed file without an overflow:

```
(H 1) ≡ (H 1 (S 1)) ≡ (H 1 (S 1 (S 1)))
```

This is similar to Example-1 except that the head of the access path is located through hashing, and that a node is of one record.

Example-3. Accessing an inverted file as shown in Figure 4-5 (a):

```
(P 3 (P 1 (S 1)) (P 1 (S 1)) (P 1 (S 1)))
```

The path, whose head is located through a pointer, contains a key value and three chains. Each chain is also located through a pointer, and each has one node. Each node shares the same address with the

chain, and is of one record. Since all the chains are identical, we need not repeat the descriptor for each chain. Then the expression is abbreviated to:

```
(P 3 (P 1 (S 1)))
```

In general, there will be n chains:

$$(\mathtt{P}\ n\ (\mathtt{P}\ 1\ (\mathtt{S}\ 1)))$$

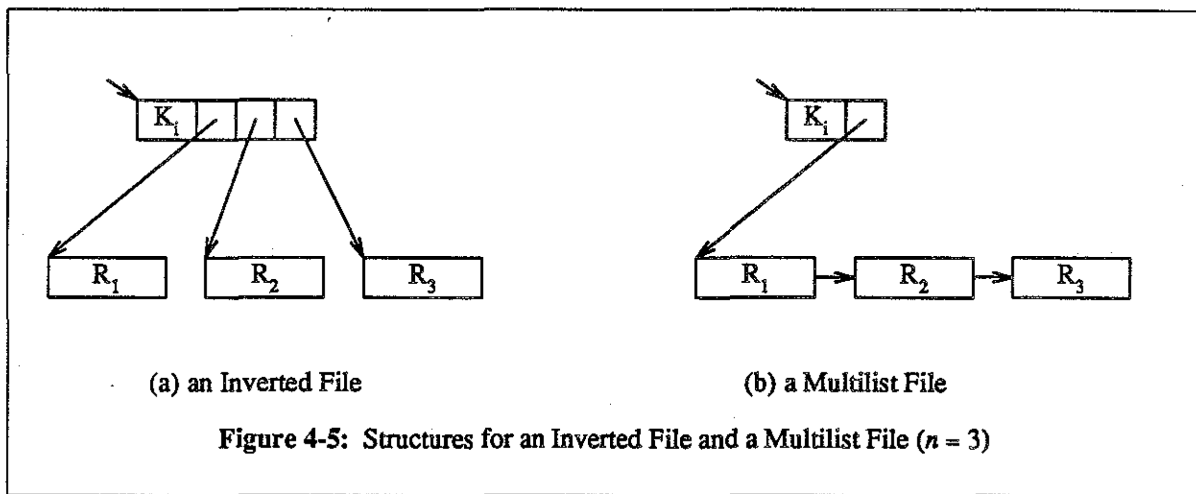


Figure 4-5: Structures for an Inverted File and a Multilist File (n = 3)

Example-4. Accessing a cellular inverted file, where each node is a cellular block of size b:

$$(\mathtt{P}\ \frac{n}{b}\ (\mathtt{P}\ 1\ (\mathtt{S}\ b)))$$

Similar to Example-3, but the path has $\frac{n}{b}$ chains. Each chain has one node, which consists of b records.

Example-5. Accessing a multilist file as shown in Figure 4-5 (b):

```
(P 1 (P 3 (S 1) (P 1) (P 1)))
```

The path, whose head is located through a pointer, has one chain. The chain is located through a

pointer, and has three nodes, each of which has one record. The first node shares the same address as the chain, and the subsequent nodes are located through pointers. Since the second node and the third node are identical, the expression can be abbreviated to:

```
(P 1 (P 3 (S 1) (P 1)))
```

In general, there will be a chain of n nodes:

$$\texttt{(P 1 (P } n \texttt{ (S 1) (P 1)))}$$

Note the difference from the expression for an inverted file in Example-3.

Example-6. Accessing a cellular multilist file, where each node is a cellular block of size b:

$$\texttt{(P 1 (P } \frac{n}{b} \texttt{ (S } b\texttt{) (P } b\texttt{)))}$$

Similar to Example-5, but the chain has $\frac{n}{b}$ nodes, each of which consists of b records. Note that we can repeat the descriptor for the second node, (P b), $\frac{n}{b} - 1$ times.

Example-7. Accessing an ISAM file with the master index in core:

```
(M 1 (P 1 (P 1)))
```

An entry in the master index, which resides in the main memory, points to the head of a single chain, corresponding to a directory entry. The chain consists of a node, which consists of a single record. The head of the node is located through a pointer. If the file has an overflow chain of n nodes, each of which is a single record, the access path expression is:

$$\texttt{(M 1 (P } n{+}1 \texttt{ (P 1)))}$$

Example-8. Accessing a hashed file with an overflow chain of *n* records:

```
(H 1 (P n (S 1) (P 1)))
```

The access path is located by hashing, and has a single chain. The chain has *n* nodes, each of which has a single record. The head of the chain is located through a pointer, and shares the same address with the head of the first node.

Thus far, we have discussed access paths involving only one file. When two or more files are involved in an access, the *composite* access path is represented by the combination of the individual file paths. There are two criteria to determine the relationship between two files. One is *ordering*, which determines whether two files are ordered or not. If *ordered*, they are accessed in *serial*, where one file path always precedes the other one. If *unordered*, there is no restriction on ordering, so two files may be accessed in *parallel*. The other criterion is whether only *one* file needs to be accessed, or *both* files should be accessed. Obviously, if both files should be accessed, the *ordering* information between the two files must be known. With this restriction, the two criteria lead to five possible combinations as follows.

(1) [$FilePath_1$; $FilePath_2$]

Two files are accessed in serial, like the temporally partitioned storage structure to be discussed in the next chapter.

(2) [$FilePath_1$, $FilePath_2$]

Both files need to be accessed, but there is no fixed ordering, like a horizontally partitioned relation [March & Severance 1977].

(3) [$FilePath_1$?; $FilePath_2$]

The first file is accessed. If it is unsuccessful, then the second file is accessed. An example is a differential file [Severance 1976].

(4) [$FilePath_1$?, $FilePath_2$]

Either of the two files is accessed. If it is unsuccessful, then the other file is accessed. An example is to retrieve a record from a vertically partitioned relation [Ceri & Pelagatti 1984].

(5) [$FilePath_1$? $FilePath_2$]

Only one of the two files needs to be accessed, and which one to access is known. An example is found inside the access path expression of the differential file with the Bloom filter in main memory [Gremillion 1982]:

[(M 1) ; [$FilePath_1$? $FilePath_2$]]

It is also possible to involve more than two files in various combinations.

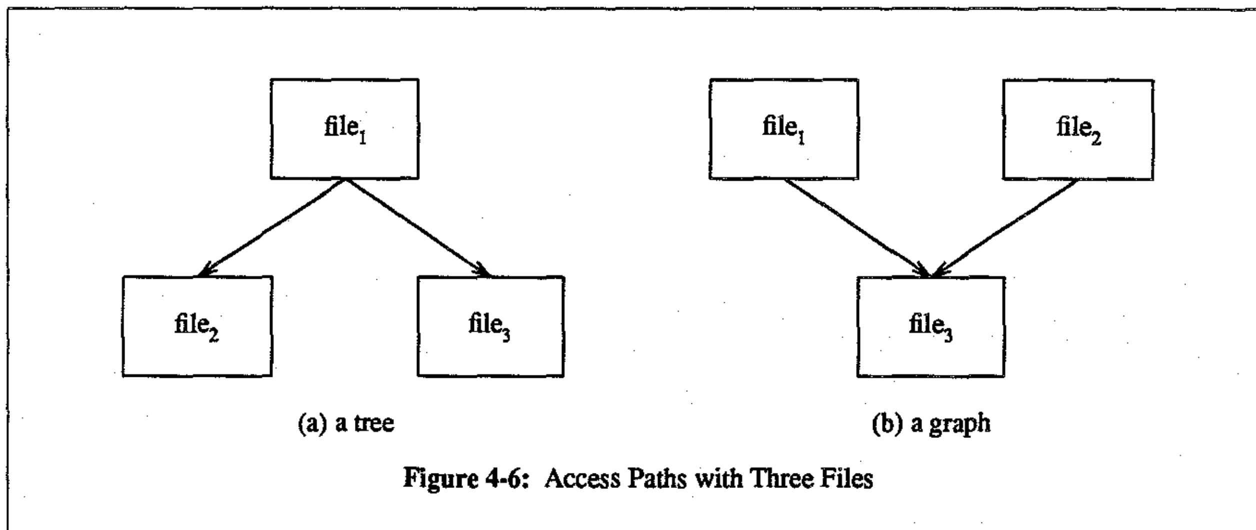


Figure 4-6: Access Paths with Three Files

Example-9. Accessing a path composed of three files:

If they are accessed in sequence like a three level store, the access path expression is:

[[$FilePath_1$; $FilePath_2$] ; $FilePath_3$]

If they are in the shape of a tree, as in Figure 4-6 (a), file 1 is accessed first, then the other two files are accessed in any order. The access path expression is:

[$FilePath_1$; [$FilePath_2$, $FilePath_3$]]

In Figure 4-6 (b), files 1 and 2 are accessed in any order, then then the third file is accessed. The access path expression is:

$[\ [\ FilePath_1\ ,\ FilePath_2\ ]\ ;\ FilePath_3\ ]$

BNF syntax for the access path expression involving multiple files is given in Figure 4-7, where `<FilePath>` was defined in Figure 4-4.

```
<APE>        ::=   <term>
             |     <APE>    <a op>    <term>

<term>       ::=   <AccPath>
             |     <term>    <m op>    <AccPath>

<a op>       ::=   +     |     -
<m op>       ::=   *     |     /

<AccPath>    ::=   <FilePath>
             |     [   <AccPath>   <one>   <ord>   <FilePath>   ]
             |     (   <APE>   )

<one>        ::=   ?
<ord>        ::=   ;     |     ,
```

Figure 4-7: BNF Syntax for Access Path Expressions (Multiple Files)

Given an access path expression, it is possible to parse the expression, and derive an *access path graph (APG)*. In the graph, each component is denoted as a vertex, while relationships among components are denoted as an edge marked with the associated mode. For an access path involving a single file, the graph results in a tree, with the vertex for *file path* as the root. Access path graphs for the access path expressions in Example-3 and Example-5 are shown in Figure 4-8. While there is a similarity between the physical structure in Figure 4-5 and the access path graph in Figure 4-8, this is not always the case. The access path graph is only conceptual, and not necessarily tied to the physical structure itself.

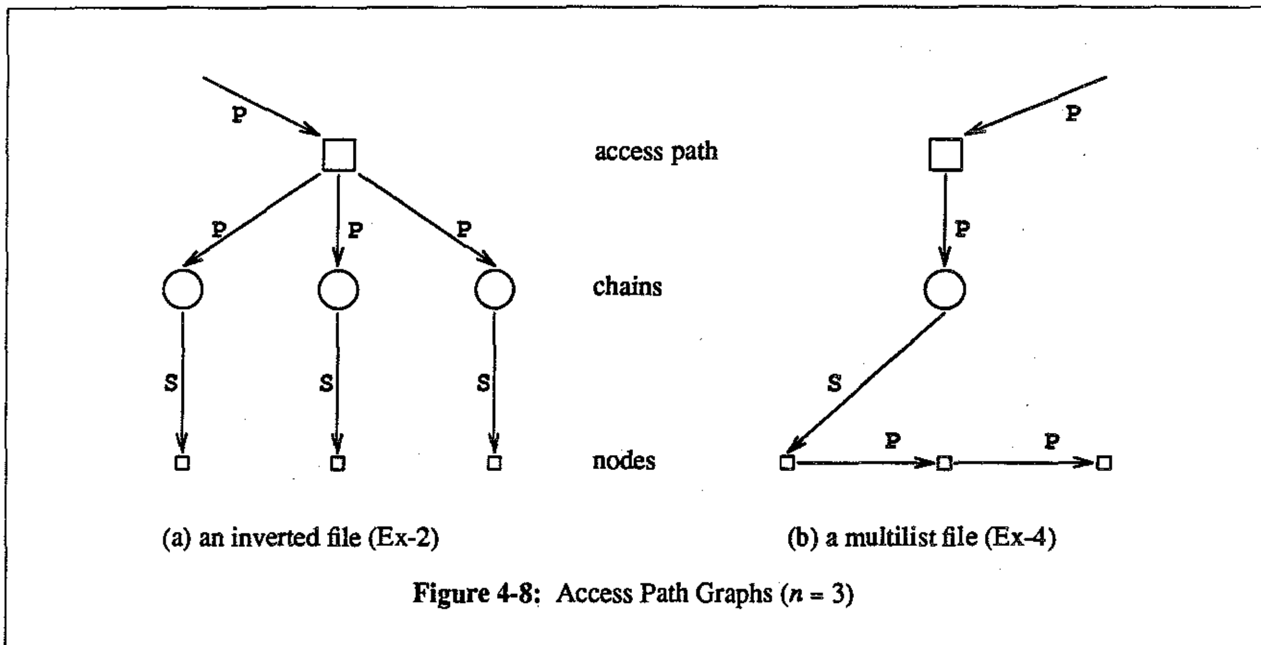


Figure 4-8: Access Path Graphs ($n = 3$)

The access path graph not only visualizes the process of accessing files, but also represents the cost incurred in traversing an access path by the length of the path. In fact, it is possible to estimate the *access path cost (APC)* from the access path expression, based on the modes to connect components. The rule to estimate the upper bound for the access path cost is:

H (Hashing) : $(1 + \alpha)$ random accesses
where α is determined by the overflow handling method

P (Pointed) : 1 random access

A (Adjacent) : 1 sequential access

S (Same-as-before) : no access cost

I (Main-memory) : no access cost

O (Ordered) : logarithmic search $(O\ (log\ \frac{f}{b}))$

U (Unordered) : sequential search $(\frac{f+b}{2b})$

In summary, the model of access paths (MAP) represents access paths, taken through a storage structure to satisfy a request represented by a file primitive expression, with the access path expression augmented with a set of file parameters. The access path expression is simple and well-defined, yet versatile in representing a variety of access methods which may involve more than one file. Given an access path expression, it is also possible to derive the associated access path graph and the access path

cost. The model of access paths is loosely based on four models described in Section 2.1.3. It captures the concepts of the sublist in [Hsiao & Harary 1970], data direct/indirect & address/pointer sequential in [Severance 1975], hierarchy of levels in [Yao & Merten 1975] and a set of parameters in [Batory & Gotlieb 1982], but extends them significantly in a systematic framework.

4.1.4. Model of Storage Devices

The last model in the hierarchy is the *model of storage devices* which represents physical characteristics of storage media. There are many parameters affecting the performance of storage devices, such as the medium type, fixed or moving heads, read/write or write-once, seek time, transfer rate, number of cylinders-tracks-sectors, sector size, *etc*. Though it is difficult to model exact behaviors of storage devices under typical time sharing environments, significant contributions have been made to analyze their characteristics [Satyanarayanan 1983]. For the purpose of this research, we adopt a model of storage devices characterizing the performance with two parameters. They are t_{ra}, time needed to access a block randomly, and t_{sa}, time needed to access a block sequentially. Given the count of random and sequential accesses, *e.g.* from the model of access paths, it is possible to calculate the time required to satisfy the request.

For a typical moving head disk, time needed to access a block randomly is the sum of seek time, rotational delay, and data transfer time.

$$t_{ra} = t_{seek} + t_{rd} + t_{tr}$$

The average seek time, t_{seek}, assuming uniform distribution of seek distances is [Wiederhold 1981]:

$$E\ (t_{seek}) = \sum_{i=1}^{c-1} t_i \times \frac{2(c-i)}{c^2 - c}$$

where t_i is the seek time for distance over i cylinders, and c is the total number of cylinders for the disk. The average rotational delay, t_{rd}, is the time for one revolution divided by two, and the data transfer time, t_{tr}, is the block size divided by the data transfer rate.

Ideally, accessing a block sequentially is free of any head movement and even the rotational delay.

$$t_{sa} = t_{tr}$$

However, a *logically* sequential block may not be *physically* adjacent under many operating systems, *e.g.* Unix, which may allocate a block to a file randomly from the pool of free pages [Stonebraker 1981]. Even when the block is physically adjacent, it is highly probable in a multi-process system that another process sharing the disk disrupts the sequentiality by moving the head to another sector or cylinder.

Another factor to be considered is the difference between the block size of the database management system and the page size of the operating system. Let b_{db} be the block size of the database management system, and let p_{os} be the page size of the operating system. If b_{db} is bigger than p_{os}, it takes extra disk accesses to retrieve one database block. In the opposite case, which is actually the case in the prototype to be described in Chapters 6 and 7, some sequential blocks are already in the main memory with the effect of *read-ahead*. If we let $n = \frac{p_{os}}{b_{db}}$, the average t_{sa} in a multi-process environment will be:

$$t_{sa} = \frac{1}{n} \left(t_{seek} + t_{rd} + n \times t_{tr} \right)$$

An experiment was run to measure the average t_{ra} and t_{sa} on a moving head disk connected to a Vax/780. Here, the file used for sequential access was in fact physically contiguous. The results were:

	Low Load	High Load	Average
Sequential	16.9	19.9	18.4
Random	24.8	37.8	31.3

Figure 4-9: Time (in *msec*) to Access a Block

Figures for the average time were used successfully in estimating the elapsed time to process sample queries, as will be described in Chapter 6.

4.2. Performance Analysis

With the four models described in the previous section, it is possible to analyze the input and output cost to process TQuel queries. Any complex query involving more than two relations can be decomposed into simpler queries of two or less relations [Wong & Youssefi 1976]. Hence a TQuel query can be

represented by an algebraic expression, which consists of algebraic operators involving one or two relations, reflecting the strategy used to process the query. The algebraic expression is then mapped into the file primitive expression according to the *model of algebraic expressions* and the *model of database/relations.*

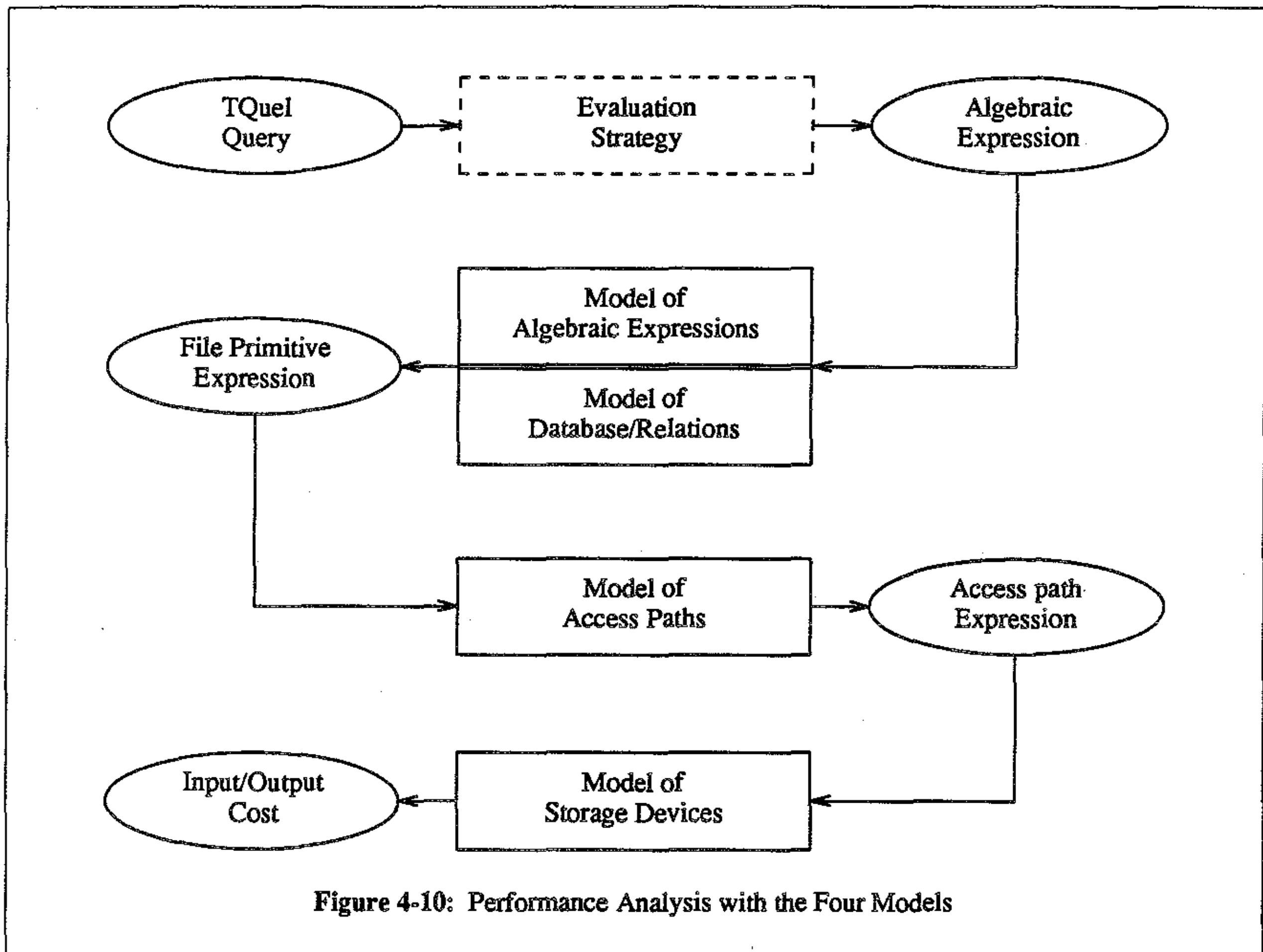


Figure 4-10: Performance Analysis with the Four Models

Next, the *model of access paths* maps the file primitive expression into the access path expression, and eventually to the access path cost in terms of the number of random and sequential block accesses. Finally, the access path cost is converted to the time required to satisfy the request according to the *model of storage devices.* These steps are illustrated in the Figure 4-10, where we show the *evaluation strategy* in a dotted box to denote that the evaluation strategy is not a part of the models.

4.2.1. Examples

This section describes how the performance of two sample queries in TQuel can be analyzed using the four models developed in Section 4.1. For example, both of the algebraic expressions AE-1 and AE-2, shown in Section 4.1.1.1, represent the TQuel query:

```
range of    h  is relation_h
retrieve    (h.id, h.seq)    where h.id = 500
```

Since AE-2 provides more information on how to process the query, let's evaluate the input and output cost for AE-2 using the four models. We first try the case where the model of database/relations shows that `relation_h` is a hashed file with no overflow records. Then from the model of algebraic expressions, we get the file primitive expression:

```
Read  (Hash, 0)
```

which is the same as FPE-1 shown in Section 4.1.1.2. This is converted, according to the model of access paths, to the access path expression:

```
(H 1)
```

whose access path cost is

$$APC = C\,(\mathrm{APE}) = C\,((\mathtt{H}\ 1)) = 1\ \textit{random access}$$

Now the average time to perform 1 random access is found to be about 31.3 *msec* according to the model of storage devices.

If the model of database/relations shows that `relation_h` is a hashed file with 14 overflow records, then its file primitive expression becomes:

```
Read (Hash, 14)
```

Now the corresponding access path expression is:

```
(H 1 (P 14 (S 1) (P 1)))
```

Its access path cost is

$$APC = C\,(\text{APE}) = C\,((\text{H } 1\ (\text{P } 14\ (\text{S } 1)\ (\text{P } 1)))) = 15\ \textit{random accesses}$$

which is equivalent to 470 *msec* according to the model of storage devices.

For another example, algebraic expressions AE-3, AE-4, and AE-5 can all be considered as representations of the TQuel query:

```
range of    h  is relation_h
range of    i  is relation_i
retrieve (h.id, i.id, i.amount)
      where h.id = i.amount
      when  h overlap i and   i overlap "now"
```

Let's evaluate the input and output cost for AE-5:

```
{[ L1:  When       (i, i overlap "now");
   L2:  Project    (L1, i.id, i.amount, i.valid_from, i.valid_to)];
   L3:  Temporary  (L2);
 [ L4:  Join       (h, L3, TS, h.id = i.amount & h overlap i);
        Project    (L4, h.id, i.id, i.amount)     ]}
```

First, the model of database/relations is assumed to show that `relation_h` is a hashed file and `relation_i` is an ISAM file, each without any overflow records. It is also assumed that the size of `relation_i` is 128 blocks, the size of the temporary relation is 19 blocks, and there are 1024 tuples in the temporary relation. Then the model of algebraic expressions maps AE-5 to the file primitive expression:

```
  Read  (Heap, 128)           +
( Read  (Heap, 19) * 2 - 1    +
  Write (Heap, 19) * 3 - 1 )  +
  Read  (Heap, 19)            +
  Read  (Hash, 0) * 1024)
```

which is in fact the same as FPE-2 shown earlier. The first **Read** primitive accounts for the **When** and the **Project** operations, the second **Read** and the **Write** primitives account for the **Temporary** operation, and the third and the fourth **Read** primitives account for the **Join** and the **Project** operations.

According to the model of access paths, the **Read** operations in the file primitive expression are mapped to the access path expression for input:

```
(U 128)              +
(U 19) * 2 - 1       +
(U 19)               +
(H 1) * 1024
```

Likewise, the **Write** operation in the file primitive expression is mapped to the access path expression for output:

```
(U 19) * 3 - 1
```

Now, the access path cost for input is:

$$\begin{aligned} APC_i &= C((\mathrm{U}\ 128)) + C((\mathrm{U}\ 19) * 2 - 1) + C((\mathrm{U}\ 19)) + C((\mathrm{H}\ 1)) * 1024 \\ &= 1028 \textit{ random accesses} + 180 \textit{ sequential accesses} \end{aligned}$$

and the access path cost for output is:

$$\begin{aligned} APC_o &= C((\mathrm{U}\ 19) * 3 - 1) \\ &= 3 \textit{ random accesses} + 53 \textit{ sequential accesses} \end{aligned}$$

Hence it takes 35.5 *sec* for input, and 1.07 *sec* for output according to the model of storage devices.

Let's consider the case where `relation_h` is a hashed file, and `relation_i` is an ISAM file, but both of them are temporal relations with the update count of 14 according to the model of database/relations. Then on the average, there are 28 overflow records for each tuple, since each **replace** operation inserts two versions into a temporal relation. We also assume that the size of `relation_i` is 3712 blocks, which is 128 blocks multiplied by 29, that the size of the temporary relation is 19 blocks, and that there are 1024 tuples in the temporary relation. Now the file primitive expression corresponding to the algebraic expression AE-5 becomes:

```
  Read  (Heap, 3712)          +
( Read  (Heap, 19) * 2 - 1    +
  Write (Heap, 19) * 3 - 1 )  +
  Read  (Heap, 19)            +
  Read  (Hash, 28) * 1024
```

As in the previous example, the first **Read** primitive accounts for the **When** and the **Project** operations, the second **Read** and the **Write** primitives account for the **Temporary** operation, and the third and the fourth **Read** primitives account for the **Join** and the **Project** operations. This is mapped to the access path expression for input:

```
(U 3712)                  +
(U 19) * 2 - 1            +
(U 19)                    +
(H 1 (P 28 (S 1) (P 1))) * 1024
```

and the access path expression for output:

```
(U 19) * 3 - 1
```

Then, the access path cost for input is:

$$
\begin{aligned}
APC_i &= C((\mathbf{U}\ 3712)) + C((\mathbf{U}\ 19) * 2 - 1) + C((\mathbf{U}\ 19)) \\
&\quad + C((\mathbf{H}\ 1\ (\mathbf{P}\ 28\ (\mathbf{S}\ 1)\ (\mathbf{P}\ 1)))) * 1024 \\
&= 29700\ \textit{random accesses} + 3764\ \textit{sequential accesses}
\end{aligned}
$$

and the access path cost for output is:

$$APC_o = C((\mho\ 19) * 3 - 1)$$
$$= 3\ \textit{random accesses} + 53\ \textit{sequential accesses}$$

which is equivalent to 999 *sec* for input, and 1.07 *sec* for output according to the model of storage devices.

In fact, these queries, among others, were run on the prototype temporal database management system, which was built by extending a snapshot DBMS INGRES. Measuring input and output cost for sample queries on the prototype provided performance figures, which were quite close to the analysis results obtained by using the four models as discussed in this section. Further descriptions and the results of the benchmark will be presented in Chapter 6.

4.2.2. Performance Analyzer

Based on the four models forming a hierarchy, it is possible to construct the *Performance Analyzer for TQuel Queries* (PATQ), which can automate computation of the input and output cost given a collection of TQuel queries as input. The internal structure of the PATQ is shown in Figure 4-11.

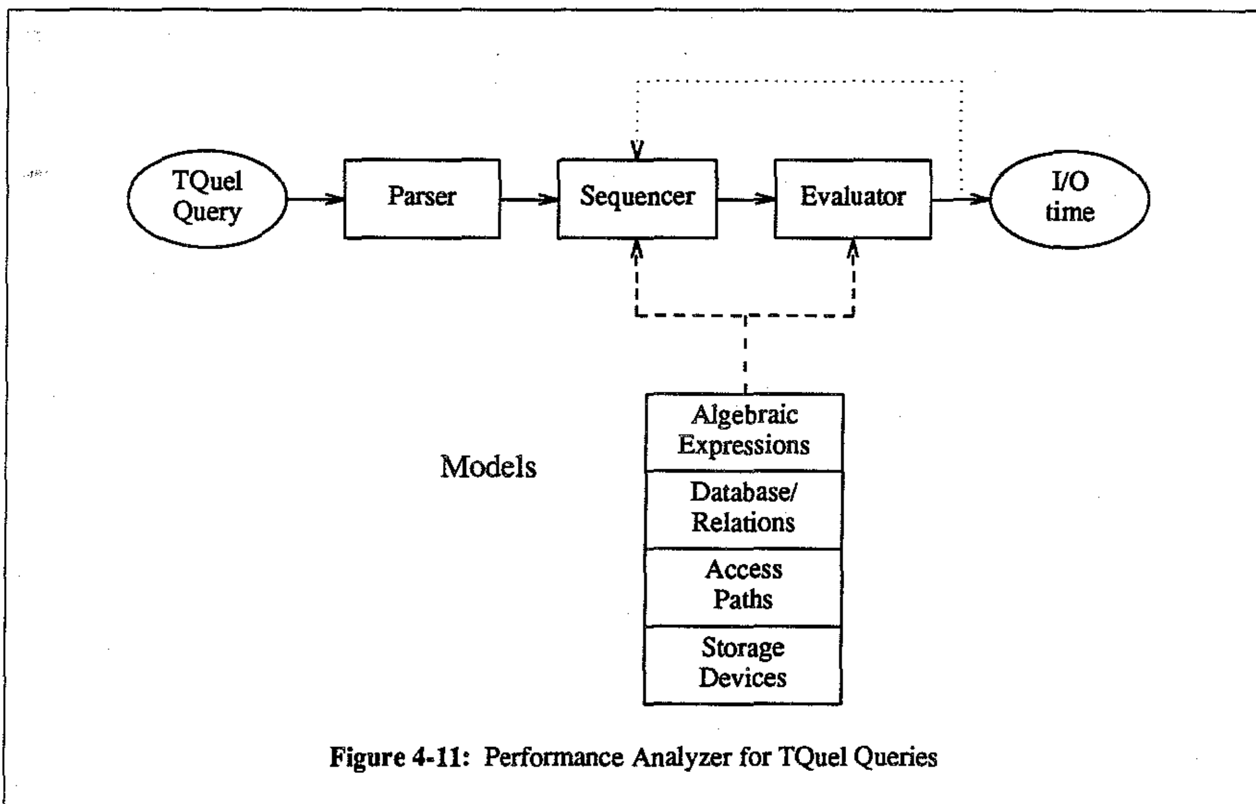


Figure 4-11: Performance Analyzer for TQuel Queries

The *parser* will take TQuel queries and generate a parse tree. The *sequencer* converts the tree into an algebraic expression consisting of algebraic operators and connectives as described in Section 4.1.1.1. Since TQuel is a non-procedural language based on the tuple calculus, there are many ways to process a TQuel query, and many variations of algebraic expressions. The sequencer is the embodiment of the query evaluation and optimization strategy for a particular database management system. Four models described above are all available to it, but the extent of utilizing such information depends upon the system being modeled.

The resulting algebraic expression will be processed by the *evaluator* to compute the input and output cost based on information represented by the set of models. The evaluator converts the algebraic expression to the file primitive expression according to the model of algebraic expressions and the model of database/relations. Next, it converts the file primitive expression to the access path expression, and eventually to the access path cost, using the model of access paths. Finally, it calculates the time required to satisfy the access path cost according to the model of storage devices.

PATQ can be used to test and analyze various alternatives in the design of new access methods, database configurations, or query processing strategies, eliminating the tedious process of case by case implementation or simulation. However, actual implementation of PATQ is beyond the scope of this dissertation, and is left as a future work. In this dissertation, we analyzed the performance of sample queries manually, but in the same manner PATQ would have employed.

PATQ can be extended to be an optimization tool by providing a feedback path, as shown by a dotted line in Figure 4-11, from the evaluator output to the sequencer. The sequencer can generate all possible algebraic expressions for an input parse tree, and can choose the one with the lowest input and output cost as computed by the evaluator. The algebraic expression chosen that way represents the best strategy to minimize the cost of processing the query.

Chapter 5

New Access Methods

As discussed in Section 1.2, databases with temporal support face problems in terms of both space and performance, due to the need for maintaining history data together with current data on line. Conventional access methods such as hashing or ISAM are not expected to be effective for such databases with a large number of temporal versions, which will be demonstrated by the benchmark results in Chapter 6. Other access methods that adapt to dynamic growth better also have various problems as described in Section 1.2.2. Therefore, new access methods and storage structures tailored to the particular characteristics of database management systems with temporal support need to be developed to provide fast response for a wide range of temporal queries without penalizing conventional non-temporal queries.

The first section of this chapter addresses general issues of the *temporally partitioned storage structure*. The second section investigates various formats for the history store which can improve the performance of temporal queries. Then the third section studies issues on how to support *secondary indexing* for databases with temporal support, and the fourth section discusses *attribute versioning* in contrast with *tuple versioning*. Unless specified otherwise, tuple versioning is assumed throughout this dissertation.

5.1. Temporally Partitioned Store

A database with temporal support maintains the history of an enterprise, or the history of activities on the database modeling an enterprise, or the history of both, depending on the type of temporal support. In any case, there can be multiple versions to represent a single entity over a period of time. Thus, the term *version set* is defined to identify a set of versions for one entity. A version set usually has a single key value for all of its versions. But a version set may have multiple keys if there has been key changes, as will be discussed in Section 5.1.4.

As discussed in Section 1.2.1, databases with temporal support contain two distinct types of data, *current* data and *history* data. The characteristics of current data and history data exhibit clear differences in terms of the version count, storage requirements, access frequency, access urgency, and update pattern. These differences make it natural to store and process them separately depending on their individual characteristics. It leads us to the *temporally partitioned storage structure* with two storage areas, the *current store* and the *history store*. The current store contains current versions which can satisfy all non-temporal queries, and possibly some of frequently accessed history versions. The *history store* holds the remaining history versions.

This scheme to separate current data from the bulk of history data can minimize the overhead for non-temporal queries, and at the same time provide a fast access path for temporal queries. It is possible to use different access methods for each of the two. The current store may utilize any conventional access method suitable for a snapshot relation, such as hashing, ISAM, or B-tree. The history store may also use any conventional access method, but several variations are conceivable to exploit the concept of *version* inherent in history data. It is even possible to use different types of storage media for each of the two. For example, history data may be stored on optical disks, while current data are kept on magnetic disks.

This temporally partitioned storage structure can also be regarded as the *reverse differential file*. The scheme of *differential file* represents two versions of data with the *main* file and the *differential* file [Severance 1976]. The main file contains the reference version (R), and is never modified. All changes to the main file are recorded in the differential file, which are either additions (A) or deletions (D). Thus, the current version (C) can be found by $\mathbf{R} \cup \mathbf{A} - \mathbf{D}$. Note that accessing the current version is slower than accessing the old version. On the other hand, the scheme of *reverse differential file* directly represents the current version in the file C. It also records additions (A) and deletions (D) to and from a reference version in a separate file. Then, the current version is readily available from C, and the reference version (R) can be found by $\mathbf{C} \cup \mathbf{D} - \mathbf{A}$. Since $\mathbf{A} \subseteq \mathbf{C}$, A need not be stored separately. They can, instead, be represented as a part of C by marking them with appropriate information, *e.g.* attaching time attributes to each record to show when it was appended. Attaching time attributes to each record also generalizes the number of versions from two to any number.

Storage structures similar to this temporally partitioned scheme have been mentioned in other papers [Ben-Zvi 1982, Katz & Lehman 1984, Lum et al. 1984], but none of them has investigated various characteristics and possible variations, nor has analyzed their performance. There are many issues to be investigated about the temporally partitioned storage structure [Ahn 1986]. This section discusses the *split criteria* specifying 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, and key changes, and the performance with regard to the update count.

5.1.1. Split Criteria

The main objective of the temporally partitioned storage structure in this dissertation is to separate current data from history data so that the overhead for supporting temporal queries can be minimized. Hence the basic criterion is to keep current versions in the *current store*, and to keep history versions in the *history store*. All non-temporal queries can be evaluated by consulting only the current store without any interference from the bulk of history versions. This criterion appears to be quite simple, but there are many complications especially with a historical or a temporal database.

The term *current version* has different implications depending on the temporal type of databases. For a rollback database, the current version of a version set is the version entered into the database most recently for the version set, and has '–' as the value of the *transaction stop* attribute. Such tuples are put into the current store, and the other tuples are put into the history store.

But determining current versions for a historical or a temporal database is complicated by *retroactive* or *proactive* changes, which will be discussed further in Section 5.1.3. For a historical database, the current version has the attributes *valid from* and *valid to* overlapping with the current time. For a temporal database, the current version has the attributes *valid from* and *valid to* overlapping with the current time, and a *transaction stop* value of '–'. If we ignore retroactive or proactive changes for the moment, the current store keeps tuples with a *valid to* value of '∞' for a historical database, and tuples with a *valid to* value of '∞' and a *transaction stop* value of '–' for a temporal database. An extension to the temporally partitioned storage structure with the current and the history stores would be to add the third store, called the *archival* store, which contains tuples with values other than '–' for the *transaction stop* attribute. The

archival store will be consulted only for queries **as of** some moment in the past.

As discussed in Section 1.2.1, current data are in general smaller in volume, but accessed more frequently and urgently, than history data. Thus, the current store can be more efficient than the history store in accessing data. To take advantage of this property, we can relax the basic criterion by keeping some history data, which tend to be accessed rather frequently, in the current store. In this case, care should be taken to limit the amount of history data in the current store so that the performance of non-temporal queries would not suffer from the increased size of the current store. For example, the current store may keep up to two, instead of one, most recent versions for each version set. Furthermore, deletions or proactive changes can be handled following this criterion, as will be discussed later.

It is also possible to adopt the strategy of *vertical partitioning* [Ceri & Pelagatti 1984] which moves some of the current versions, with relatively low access frequencies, to the history store. Though it is not pursued any further in this research, a special case related with this scheme is later described for *proactive* changes. Another factor affecting the criteria is the availability of an access path to history versions, since a version in the history store needs an access path either through some index or through a corresponding version in the current store.

5.1.2. Update Procedures

Unlike snapshot databases relying on *update in place*, databases with temporal support update existing information in a non-destructive way, and maintain *out of date* information as history data. Hence the semantics of **append**, **delete** and **replace** are particularly important in databases with temporal support. Handling **delete** and **replace** is more complicated with the temporally partitioned storage structure, which divides data between the current and the history store according to a split criterion. This section discusses the update procedures for the temporally partitioned storage structure in each type of databases with temporal support. The formal semantics of modification statements for TQuel has been defined elsewhere [Snodgrass 1986].

According to the basic criterion of current data on the current store and history data on the history store, deleted tuples ought be moved to the history store. This reduces the size of the current store, but it becomes necessary to provide an access path to the version set which has no current version, lest the whole

history store be scanned to locate it. The path may be a separate index of deleted tuples, or a combined index involving both the current and the history store, as will be discussed in Section 5.2.3. If the basic criterion is relaxed so that the current store may hold some of history data, deleted tuples may be left in the current store. In this case, there is no need to maintain a separate access path for deleted tuples.

For a rollback database, `append` inserts a tuple with time attributes:

transaction start ← the current time

transaction stop ← '–'

meaning that the tuple is effective from the current time on. `Delete` finds a tuple which satisfies the `where` predicate and has a transaction stop value of '–', then terminates it by changing the transaction stop attribute to the current time. The deleted tuple has been in the current store, and may or may not be moved to the history store depending on the split criteria. Deletion or correction of past tuples, whose transaction stop attribute is not '–', is not allowed in a rollback database.

`Replace` can be described as `delete` followed by `append` in any database. In this *delete and append* scheme, the base tuple is first *deleted* (in the sense of non-snapshot databases) as described above, then a copy of the base tuple with some attributes changed according to the `replace` statement is appended. This scheme works well with conventional storage structures, and is used by the prototype to be described in Chapter 6. But the *delete and append* scheme is not strictly applicable to a rollback database with the temporally partitioned storage structure. The problem is that the base tuple still stays in its place, while the newer version is put into a different location. An alternative is to append into the history store a copy of the base tuple with its transaction stop attribute changed to the current time, then change the base tuple according to the `replace` statement. This *append and change* scheme works well for a rollback database with the temporally partitioned store, and is also better than the *delete and append* scheme for concurrency control and error recovery in that it reduces the critical period while the base tuple is not available.

For a historical database, `append`, `delete`, and `replace` statements have the `valid` clause to specify the period while any of the modification statements will be in effect.

```
range of  h  is   historical_h
delete  h
     valid from t1  to t2
     where (h.id = 500)
```

Figure 5-1: A Delete Statement

The TQuel statement in Figure 5-1 can be regarded as having the *update interval* $[t_1, t_2)$, effective between t_1 and t_2. If no **valid** clause is specified for any modification statement, the default update interval is [now, ∞), where '∞' stands for 'forever'. Let's call a tuple satisfying the **where** predicate the *base tuple*, and assume it has the *base interval* $[t_{vf}, t_{vt})$, effective between t_{vf} and t_{vt}, where t_{vf} and t_{vt} are the values of attributes valid from and valid to. Since $t_1 < t_2$ and $t_{vf} < t_{vt}$, there are six possible relationships between the base interval and the update interval as shown in Figure 5-2.

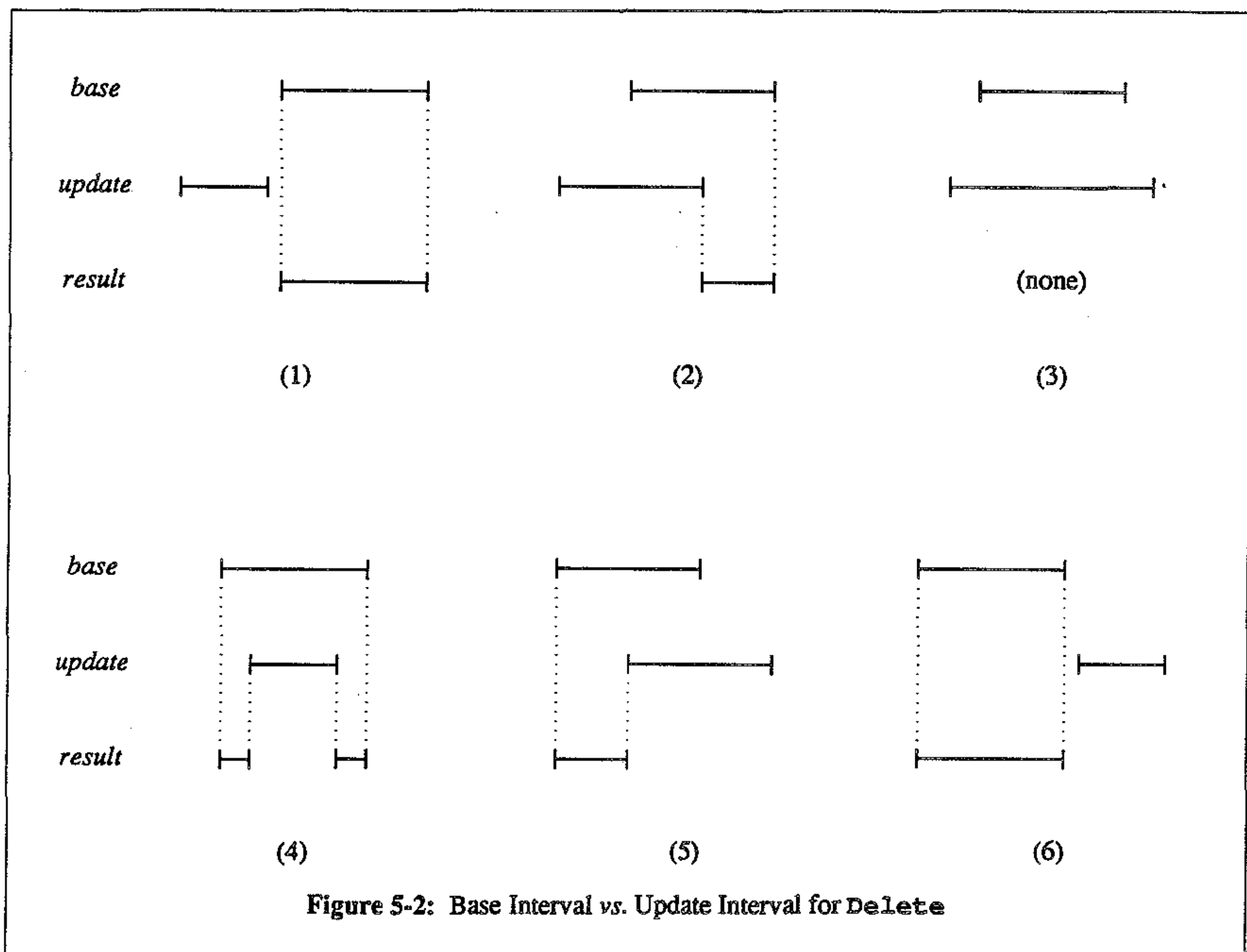


Figure 5-2: Base Interval *vs.* Update Interval for **Delete**

Delete needs to be handled differently for each case, except for cases (1) and (6) which require no

action.

- case (1): $t_2 < t_{vf}$

  The base interval and the update interval do not overlap, so nothing needs to be done.

- case (2): $t_1 < t_{vf} \wedge t_{vf} < t_2 \wedge t_2 < t_{vt}$

  The portion $[t_{vf}, t_2)$ gets deleted. The result is to change the valid from attribute of the base tuple to t_2. The base tuple still stays in its place, whether it is in the current or the history store.

- case (3): $t_1 < t_{vf} \wedge t_{vt} < t_2$

  The base tuple is physically deleted. But the immediate predecessor version of the base tuple, if any, needs to be recognized as the most recent version of the version set in order to maintain an access path to history versions. If the base tuple is in the current store, and `deleted` tuples are kept in the current store, then the immediate predecessor needs to be moved from the history store to the current store.

- case (4): $t_{vf} < t_1 \wedge t_2 < t_{vt}$

  The portion $[t_1, t_2)$, which falls on the middle of the base interval, gets deleted. The result is to change the valid from attribute of the base tuple to t_2, which stays in its place. Then a new tuple, which is the same as the base tuple but with the valid to attribute of t_1, is inserted into the history store.

- case (5): $t_{vf} < t_1 \wedge t_{vt} < t_2$

  The portion $[t_1, t_{vt})$ gets deleted, which changes the valid to attribute of the base tuple to t_1. If the base tuple is in the current store, it may be necessary to move it into the history store depending on the split criteria.

- case (6): $t_{vt} < t_1$

  The base interval and the update interval do not overlap, so nothing needs to be done.

Thus `delete` in a historical database is similar to `replace` in a snapshot database, except for the case (4) which also involves an `append`, and for the cases (1) and (6) which requires no action. Note that `delete` in a rollback database only deals with the case (5), where the time axis represents transaction time.

Handling **`replace`** is more complicated in a historical database than in a rollback database, especially with the temporally partitioned store. To perform **`replace`** in a historical database with the temporally partitioned store, there are also six cases to be examined as shown in Figure 5-3, depending on the relationship between the base interval and the update interval. However, handling **`replace`** is more complicated than **`delete`**, because we need to determine the proper location of the current version and to maintain a history chain, whether explicit or not, for each version set. Basically, we follow the *append and change* scheme, but detailed steps vary significantly for each case.

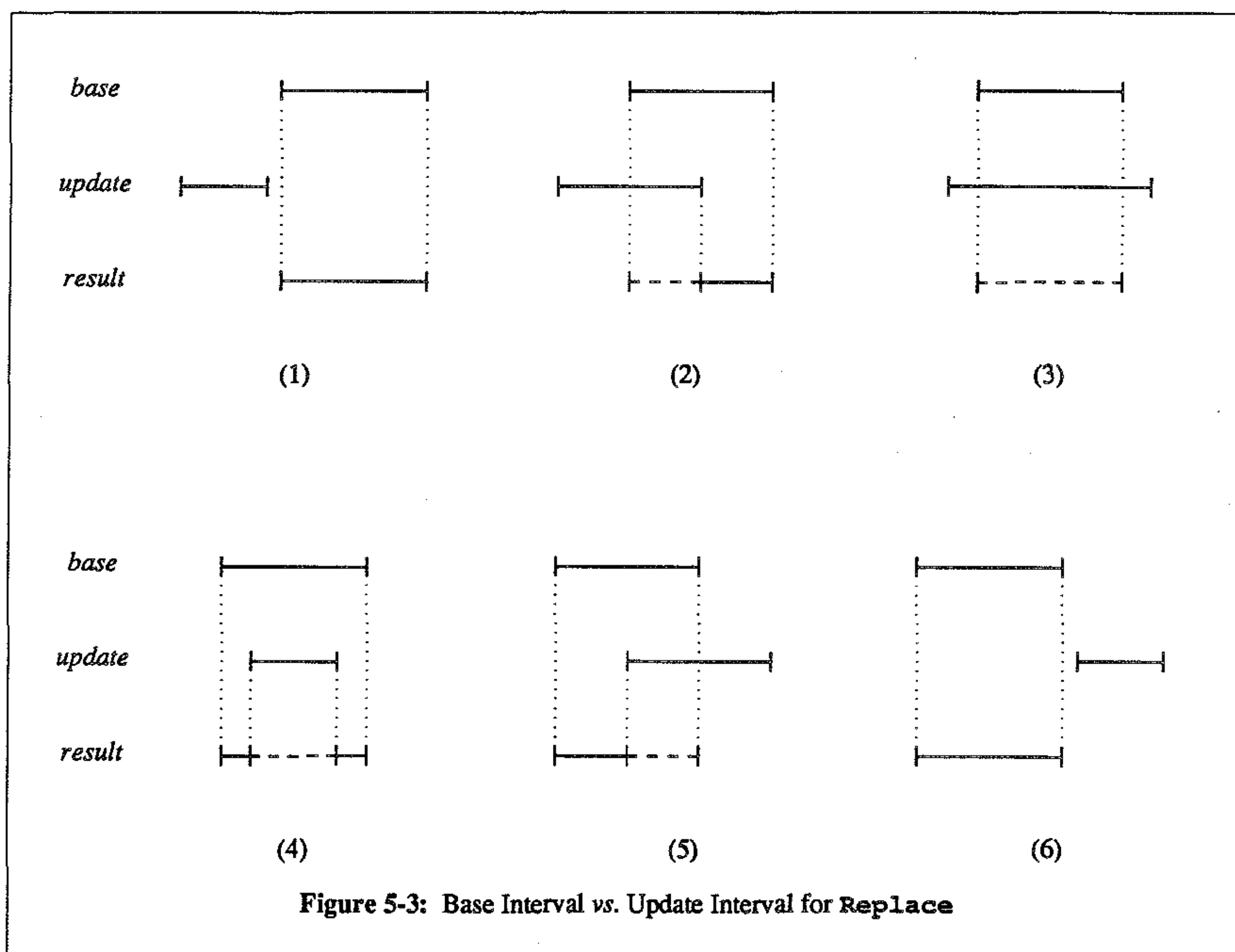


Figure 5-3: Base Interval *vs.* Update Interval for **`Replace`**

- case (1): $t_2 < t_{vf}$

  The base interval and the update interval do not overlap, so nothing needs to be done.

- case (2): $t_1 < t_{vf} \wedge t_{vf} < t_2 \wedge t_2 < t_{vt}$

  The portion $[t_{vf}, t_2)$ gets replaced. First, the new version changed by **`replace`** is put into the history store. Its valid from attribute is set to to t_{vf}, and its valid to attribute is set to t_2. Then, the

base tuple gets its valid from attribute changed to t_2, but still stays in its place, whether it is in the current or the history store.

- case (3): $t_1 < t_{vf} \wedge t_{vt} < t_2$

  The new version changed by `replace` is put into the place of the base tuple. Its valid from attribute is set to to t_{vf}, and its valid to attribute is set to t_{vt}.

- case (4): $t_{vf} < t_1 \wedge t_2 < t_{vt}$

  The portion $[t_1, t_2)$, which falls on the middle of the base interval, gets replaced. First, the new version changed by `replace` is put into the history store. Next, a copy of the base tuple is inserted into the history store with the valid to attribute set to t_1. Then, the base tuple gets its valid from attribute changed to t_2, but still stays in its place, whether it is in the current or the history store.

- case (5): $t_{vf} < t_1 \wedge t_{vt} < t_2$

  The portion $[t_1, t_{vt})$ gets replaced. First, a copy of the base tuple is inserted into the history store with the valid to attribute set to t_1. Then, the new version changed by `replace` is put into the place of the base tuple, whether it is in the current or the history store, with t_{vt} as the value of its valid to attribute. This case is particularly troublesome to the *delete and append* scheme, because the base tuple needs to be moved to the history store. Note that this corresponds to the case of the default `valid` clause for a historical database. This case also corresponds to the only case for a rollback database, except that the time axis for the rollback database represents transaction time.

- case (6): $t_{vt} < t_1$

  The base interval and the update interval do not overlap, so nothing needs to be done.

Though a temporal database supports *transaction time* in addition to *valid time*, modification statements for a temporal database have the same format as those for a historical database. Since the `as of` clause is not allowed in modification statements, transaction time does not participate in `append`, `delete`, or `replace`, except that the transaction stop attribute of the base tuple to be deleted or replaced should have the value of '–'. There are also six possible relationships between the base interval and the update interval in terms of valid time, as shown in Figures 5-2 and 5-3. For each case, `delete` and `replace` for a temporal database are handled in a similar manner to those for a historical database,

but with two exceptions. First, a copy of the base tuple is inserted into the history store with the transaction stop attribute set to the current time, before the base tuple is changed in any manner. This results in adding up to three versions for each `replace`, but provides the capability to capture the history of retroactive and proactive changes completely, as described in Section 3.4. Second, any tuple inserted in the process, except for the copy of the base tuple mentioned above, has the attributes transaction start and transaction stop set to the current time and '–', respectively. In addition, we need to maintain a chain of history versions for each version set, which is further complicated by the fact that each `replace` in a temporal database inserts at least two versions. We order versions affected in each update in reverse order of *valid from* time, then in reverse order of *transaction start* time. This ordering allows us to retrieve recent versions more quickly, especially for queries with the default clause `as of "now"`.

5.1.3. Retroactive or Proactive Changes

For a rollback database, each change is effective from the moment of the transaction, but not so for a historical or a temporal database with the `valid` clause. In the `delete` statement in Figure 5-1 for a historical or a temporal database, if t_1 is earlier than the current time, the change is *retroactive from*, and if t_2 is earlier than the current time, the change is *retroactive to*. If t_1 is later than the current time, the change is *proactive from*, and if t_2 is not '∞' but later than the current time, the change is *proactive to*. Thus a change may be *retroactive from* and *proactive to* at the same time.

Retroactive changes deal with both current and past versions, and can be handled by following the steps outlined for each case of the `delete` and `replace` statements in the previous section. However, proactive changes may involve *future versions* or *versions to be expired* which require special treatment for the temporally partitioned store. For a *proactive from* change, the base tuple is still *current* for the moment, but will expire in time. *Proactive from* `append` or `replace` introduces a future version which will become current some time later. *Proactive to* `replace` introduces both a future version and a version to be expired. A question is how to handle future versions and versions to be expired. It is possible but expensive to maintain a separate index for future versions, and to monitor constantly which versions are becoming current or expired. An alternative is to keep future versions and versions to be expired together with current versions in the current store. When any of those versions is accessed in the course of query processing, it is possible to determine if it has changed its status from *future* to *current* or from

current to *expired*, then move the expired version to the history store.

5.1.4. Key Changes

A *key* of a relation is a smallest set of attributes whose values uniquely identify a tuple, which corresponds to an entity in the entity set modeled by the relation. Formally, a key of a *snapshot* relation r over scheme R is defined as a subset K of R such that for any distinct tuples t_1 and t_2 in r, $t_1(k) \neq t_2(K)$, and no proper subset of K has this property [Maier 1985]. Thus a relation in a conventional snapshot database should not hold two tuples that agree on all the attributes of the key. However, databases with temporal support, which maintain a sequence of versions for each entity, can contain multiple tuples that agree on all the attributes of the key. Hence, the definition of the key needs to be extended for databases with temporal support.

A *key* of a relation r over scheme R in databases with temporal support is a subset K of R such that for any distinct tuples t_1 and t_2 *overlapping in time* in r, $t_1(k) \neq t_2(K)$, and no proper subset of K has this property. Two tuples t_1 and t_2 *overlap in time* if:

- for a rollback relation

$$t_1[\textit{transaction start}] \leq t_2[\textit{transaction stop}] \quad \wedge$$
$$t_2[\textit{transaction start}] \leq t_1[\textit{transaction stop}]$$

- for a historical relation

$$t_1[\textit{valid from}] \leq t_2[\textit{valid to}] \quad \wedge$$
$$t_2[\textit{valid from}] \leq t_1[\textit{valid to}]$$

- for a temporal relation

$$t_1[\textit{valid from}] \leq t_2[\textit{valid to}] \quad \wedge$$
$$t_2[\textit{valid from}] \leq t_1[\textit{valid to}] \quad \wedge$$
$$t_1[\textit{transaction start}] \leq t_2[\textit{transaction stop}] \quad \wedge$$
$$t_2[\textit{transaction start}] \leq t_1[\textit{transaction stop}]$$

The data definition statement `create` in both Quel and TQuel does not enforce the concept of the *key*, in that it does not specify what attributes constitute a key for a relation. Though the formal semantics for `append` defined for TQuel prevents two tuples identical in all the explicit attributes from overlapping

in time [Snodgrass 1986], it is still up to discretion of users to observe the *key* constraint that any new key value entered into a relation either through `append` or `replace` does not overlap with any existing tuple with the same key value. If `append` or `replace` does not insert a new key value overlapping with any existing tuple with the same key value, update procedures for the temporally partitioned store described in Section 5.1.2 ensure that there is at most one active version for each key value at any moment, and thus no two tuples with the same key value overlap in time.

Though the key value identifying an entity is not supposed to change, there are always exceptions, which cause nasty problems in conventional databases when tracking the history of changed identities. However the problem can be handled gracefully in the databases with temporal support, where a sequence of versions for each entity is maintained through physical or virtual links. If the key value of a tuple changes, a new version with the changed key becomes the current version, and the old version is kept as a history version. Thus the history of key values is captured in the same way as the history of other attribute values. But it may be necessary to rearrange the storage structure for the changed key value, if the storage structure depends on the key attributes.

5.1.5. Performance

A query is called *current* or *non-temporal* if it involves only current data and does not concern history data. A non-temporal query for a rollback database has the clause `as of "now"`. For a historical database, a non-temporal query has the clause `when` (t_1 `overlap` ... `overlap` t_i) `overlap "now"` for all the range variable t_i. For a temporal database, a non-temporal query has the clause `when` (t_1 `overlap` ... `overlap` t_i) `overlap "now"` for all the range variable t_i, and the clause `as of "now"`. Hence it is possible to determine at compile time if a query is non-temporal.

According to the split criteria discussed in Section 5.1.1, all non-temporal queries can be evaluated by consulting only the current store without going through the history store. Therefore, maintaining history versions for temporal support does not affect the performance of conventional non-temporal queries concerning only current data. The only overhead is the extra space to hold implicit time attributes and possibly a physical link to history versions, which may increase the relation size and hence the cost to scan the relation when necessary.

For a *temporal* query, it may be necessary to retrieve history versions from the history store. The basic algorithm accesses the current version first through the primary access path. If the temporal predicate of the query does not contain a tuple variable, we can determine the interval which satisfies the predicate. If the interval is found to be a subset of the interval denoted by the time attributes of the current version, there is no need to access the history store, because members of a version set in the history store do not overlap in time with the members of the version set in the current store. Otherwise, it is necessary to follow the chain of history versions through physical or virtual links depending on the format of the history store. However, many variations are conceivable for the structure of the history store, which greatly affects the performance of temporal queries. We can organize the history store in such a way that the cost of accessing the history store can be reduced significantly, as will be discussed next.

5.2. Structures of the History Store

The algorithms and the performance for accessing or updating relations with the temporally partitioned store vary significantly depending on the format of the history store. This section investigates various forms of the history store which can enhance the performance for various types of temporal queries, and analyzes their characteristics. Relative advantages and disadvantages of the various formats are evaluated to determine the cost of supporting temporal queries. In particular, a new method of hashing called *nonlinear hashing* is proposed in Section 5.2.4.2. Note that some formats can be combined together, though each format is discussed here individually.

5.2.1. Reverse Chaining

If history data are stored as a heap without any access mechanism, each request for a history version must scan the whole store, which is often impractical. One solution is *reverse chaining* to link in reverse order all history versions of each version set starting with the current version. Once the current version is located in the current store, its predecessors can be retrieved without scanning the whole history store.

For this purpose, each tuple is augmented with an extra field *nvp* (next version pointer). When a tuple is first inserted into a relation, it is put into the current store with the field *nvp* of *null*. When a tuple is replaced, the version existing in the current store is moved to some other place as described in Section 5.1.2, then a new version is put into its place with the field *nvp* pointing to the predecessor just moved.

This scheme maintains the chain from the most recent to the oldest, and does not change any of existing versions in the history store, except for error correction in historical databases. Since the history store in this scheme works in an append-only mode, it can use write-once media like optical disks. If it is possible to identify attributes which will remain unchanged, *e.g.* keys, those attributes may be excluded from history versions to save space. But unexpected situations such as key changes can cause complications in that case.

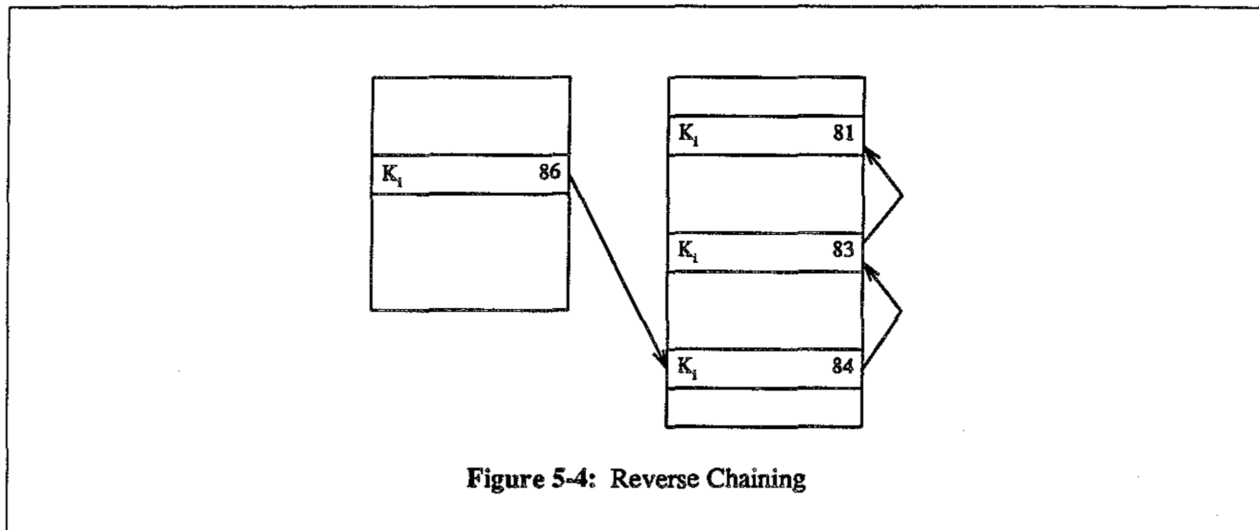


Figure 5-4: Reverse Chaining

For a `retrieve` operation, the current version is located using any access mechanism available for the current store. If the query is temporal, the field *nvp* is examined. If the pointer is null or the query is non-temporal, there is no need to go through the history store. Otherwise, all its predecessors can be found by following the chain of pointers, until a version with the *nvp* of null is reached.

If the interval represented by the temporal predicate can be evaluated as constant, then the performance can be improved by exploiting the fact that all versions are ordered in the reverse order. Instead of following the chain to the end, we can stop traversing history versions when a history version is retrieved whose interval denoted by its time attributes exceeds the constant interval specified by the temporal predicate.

The lower bound for the number of block accesses to perform `retrieve` is $\frac{n}{b}$, when there are n history versions to be retrieved and b is the blocking factor of the history store. This occurs when all history versions are clustered together in the minimum number of blocks. The upper bound for the same

case is n, when no two versions are on the same block.

When a single version set with n history versions is retrieved, the average number of block accesses, assuming uniform distribution, can be evaluated by the formula given by [Yao 1977A].

$$Average\ Block\ Accesses\ (n,f,b) = \frac{f}{b}\left[1-\frac{\binom{f-b}{n}}{\binom{f}{n}}\right] = \frac{f}{b}\left[1-\prod_{i=0}^{n-1}\frac{f-b-i}{f-i}\right] \quad (5.1)$$

where f is the number of records in the history store, and b is the number of records in a block of the history store. Note that reverse chaining maintains an ordering among versions belonging to the same version set, so there is no need to access a block more than once while scanning a chain of versions for a version set.

When several version sets are retrieved to process a query, the procedure to access a chain of versions is repeated for each version set. In this case, a block which contains versions belonging to several version sets may be accessed more than once. Hence the number of block accesses can exceed $\frac{f}{b}$, which is the cost to scan the history store sequentially. Let's assume that each version set has m versions, and that v version sets are retrieved. From the formula (5-1) above, it is possible determine the breakeven point when repeated traversal of history chains is still better than scanning the history store.

$$v' \times \frac{f}{b}\left[1-\prod_{i=0}^{m-1}\frac{f-b-i}{f-i}\right] \leq \frac{f}{b}$$

Thus the number of version sets v' to favor repeated traversal of history chains can be calculated numerically for a given m, the number of versions for each version set.

The access path expression for this format is:

[$FilePath_1$; (P n (S 1) (P 1))]

where $FilePath_1$ is for the current store, and n is the number of history versions. This expression shows that there is a single chain. The head of the chain is located through a pointer, and the chain has n nodes. Each of the node is of one record, and is connected to the predecessor by a pointer.

5.2.2. Accession Lists

If the length of the chain grows long in reverse chaining, it may be too slow to traverse the chain, even when only a small portion of the history versions are of interest. An alternative is to maintain *accession lists* between the current store and the history store.

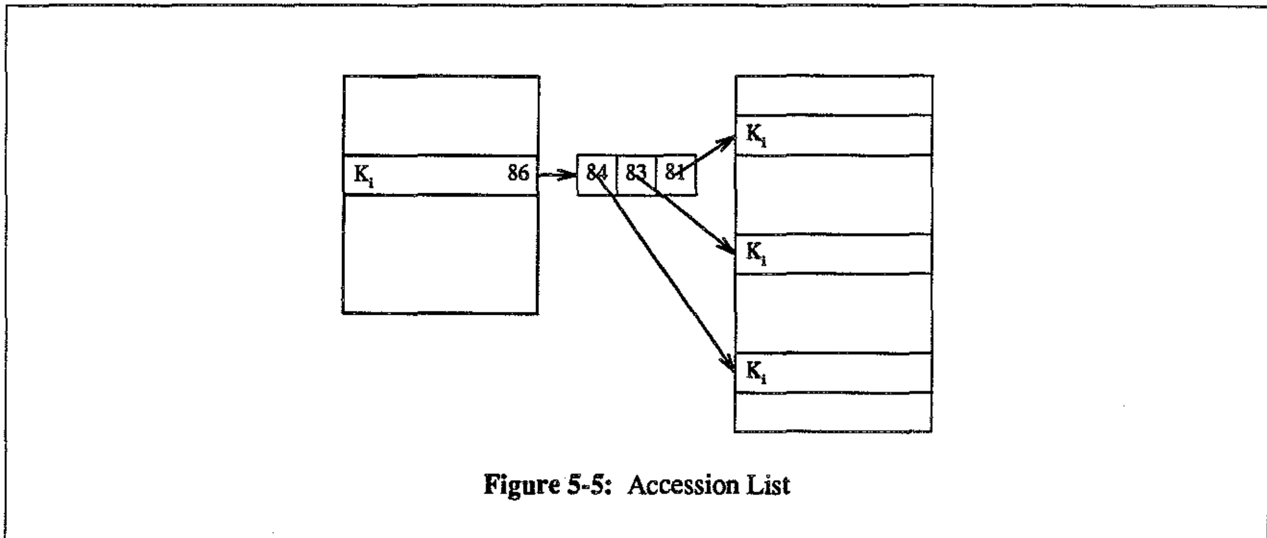


Figure 5-5: Accession List

A tuple is first entered into the current store, with an extra field *alp* (accession list pointer) of null. When a new version replaces the current version, the new version is put into the current store with the field *alp* pointing to an accession list, which is initialized to point to the history version just inserted into the history store. If another version is added into the version set, an entry corresponding to the version is also added into the accession list. Thus the accession list is a full index to history versions for each version set.

It is desirable to include some temporal information for each entry in accession lists, so that temporal predicates can be evaluated without actually accessing history versions. Deciding on the amount of temporal information to be included into accession lists is a question of space time tradeoff.

For a rollback relation, accession lists may contain both of the attributes transaction start and transaction stop (*full accession lists*). Space can be saved by storing only the transaction start attribute (*partial accession lists*) without significant loss of performance, because most version sets are *contiguous*, meaning that the value of the transaction stop attribute is the same as the value of the transaction start attribute of its successor. (Clifford & Warren defined a formal semantics of a historical database based on the *continuity assumption* [Clifford & Warren 1983].) Similar arguments apply to a historical relation, with

the attributes valid from and valid to instead of the attributes transaction start and transaction stop.

For a temporal relation, accession lists may contain up to four time attributes, or some subset of the four attributes, or only one of the four attributes for each version. If two time attributes are included, the attributes valid from and transaction start are recommended for the reason of *contiguity* mentioned above. If only one time attribute is included, the attribute valid from is favored over the attribute transaction start, assuming that the *selectivity* of the `when` clause is smaller than that of the `as of` clause, which is often the case.

For *full* accession lists, only those versions that satisfy the given temporal constraints need to be retrieved from the history store. For *partial* accession lists, it is not possible to evaluate the temporal constraints completely. Hence, all versions which *can* satisfy the constraints based on the partial information are retrieved from the history store to resolve the missing information. Still, the ratio of *false hits* can be significantly reduced compared with the case of no temporal information in accession lists.

Ordering of history versions in accession lists is less critical than reverse chaining, but we still recommend that they be kept in such an order that allows recent versions to be accessed more easily. Hence for a rollback database, versions are maintained in reverse order of transaction start time. For a rollback database, versions are maintained in reverse order of valid from time. For a temporal database, versions are maintained in reverse order of valid from time, then in reverse order of transaction start time.

Including temporal information in accession lists is not an overhead, as it may appear to be. When some time attributes are stored in accession lists as described above, it is not necessary to store the same information in the history store. History versions do not need an extra field *nvp*, as in reverse chaining. Accession lists are also useful to handle *future versions* resulting from proactive changes. The future version may be put either in the current or the history store, pointed to by an entry with appropriate temporal information in accession lists.

Since accession lists are accessed more frequently than history versions, and may be clustered or reorganized for performance reasons, it is better to keep them on magnetic disks. History versions are *append only*, so they may be stored on optical disks.

The access path expression for this format is:

$$[\ FilePath_1\ ;\ (\mathtt{P}\ n\ (\mathtt{P}\ 1))\ ]$$

meaning that there are n chains. Each chain has one node, which in turn is of one record.

The upper bound for the number of block accesses to retrieve all n records is one bigger than that of reverse chaining, owing to an extra disk access for accession lists. Since temporal predicates can be evaluated without accessing the history store, the lower bound for the number of block accesses is just two including a block access for an accession list. On the average, the number of history versions actually retrieved will be much smaller than reverse chaining, though its quantification is difficult due to the variety of temporal predicates.

5.2.3. Indexing

For a snapshot relation, the index is a set of *<value, pointer>* pairs where *value* is a key value and *pointer* is the unique identifier or the address of a tuple containing *value* as the key. For databases with temporal support, the index can be extended to include pointers to history versions. Each entry is of the form $<value, p_c, p_{h_1}, \cdots p_{h_n}>$, where p_c points to the current version, and p_{h_i} with $1 \le i \le n$ points to the i-th history version.

The index entry can even include some temporal information to evaluate temporal predicates without actually accessing data tuples. Then the issue of space time tradeoff on the amount of temporal information discussed above for accession lists similarly apply to this scheme. For example, a temporal relation may have an index with a pointer and four time attributes for each version, or an index with a pointer and just one attribute, *e.g. valid from*, for each version. Figure 5-6 illustrates this scheme, which can be regarded as a combination of conventional indexing and accession lists described above.

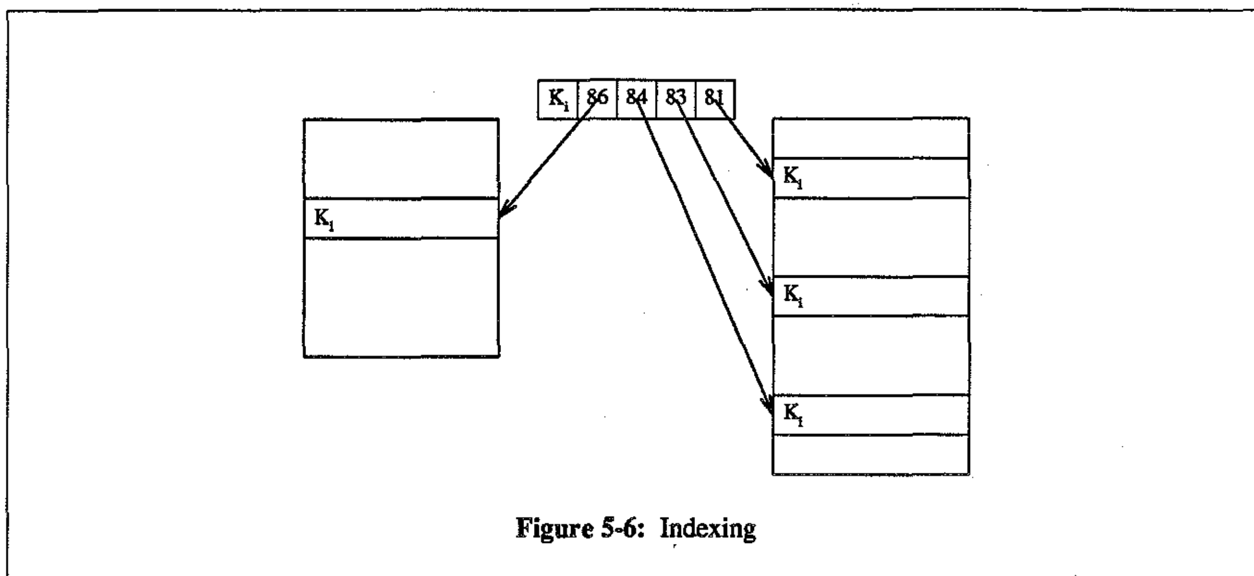


Figure 5-6: Indexing

Indexing is also useful to handle deleted tuples or future versions. Since history versions have an independent access path without going through the current store, all deleted tuples can be put into the history store. The future version may be put either in the current or the history store, pointed to by an index entry.

Its access path expression is:

```
[ FilePath1 ; [(S 1 (P 1)) ?, (S n (P 1)) ]]
```

$FilePath_1$ is for the index, which may take any appropriate storage format itself, and n is the number of history versions. From the index entry, either the current or the history store is accessed. If it is not successful, then the other store is accessed.

Instead of maintaining a pointer for each history version, space can be saved by storing only one entry for the list of history versions. Then each entry is of the form $<value, p_c, p_h>$, where p_c points to the current version. p_h may be the starting address of the chain of history versions, or the address of an accession list for history versions.

A generalization of this scheme is to apply the temporally partitioned structure to the index itself, maintaining two separate indices, one for the current store and the other for the history store. The benefits of the temporally partitioned store considered for storing data similarly apply to this temporally partitioned indexing. By separating current entries from the bulk of history entries, the current index becomes smaller

and more manageable, minimizing the overhead of maintaining history versions on non-temporal queries. The history index can utilize any format developed for the history store to enhance the performance of temporal queries. For example, the current index may be hashed, while the history index has the format of accession lists. Then each entry in the current index is of the form $<value, p_c, p_h>$, as mentioned above. In any case, history versions are *append only* for a rollback or a temporal relation, so they may be stored on optical disks.

Performance characteristics of the indexing scheme is similar to that of accession lists. The upper bound for the number of block accesses to retrieve all n records is n, one less than that of accession lists, without counting the cost to access the index itself. The lower bound for the number of block accesses is just one, without counting the cost to access the index itself. Since temporal predicates can be evaluated by temporal information included in the index, the number of history versions actually retrieved will be much smaller than reverse chaining, though its quantification is difficult due to the variety of temporal predicates.

One problem with indexing is that the format of the current store is tied to indexing, while other schemes allow any format for the current store. Another problem is to handle a query which needs to access records through non-key attributes. It is necessary to maintain the same ordering for the index and the current store, so that the current store can be scanned synchronously with the index.

5.2.4. Clustering

One problem with the schemes discussed thus far is that history versions belonging to a version set are scattered over several blocks. A solution is to *cluster* all versions of each version set into the minimum number of blocks (See Figure 5-7). Clustering significantly reduces the number of disk accesses to retrieve history versions, and thereby improve the performance of temporal queries. However, its update mechanism is more complicated to maintain clustering while achieving a high degree of storage utilization. Clustering can be combined with other schemes described earlier, such as reverse chaining, accession lists, or indexing.

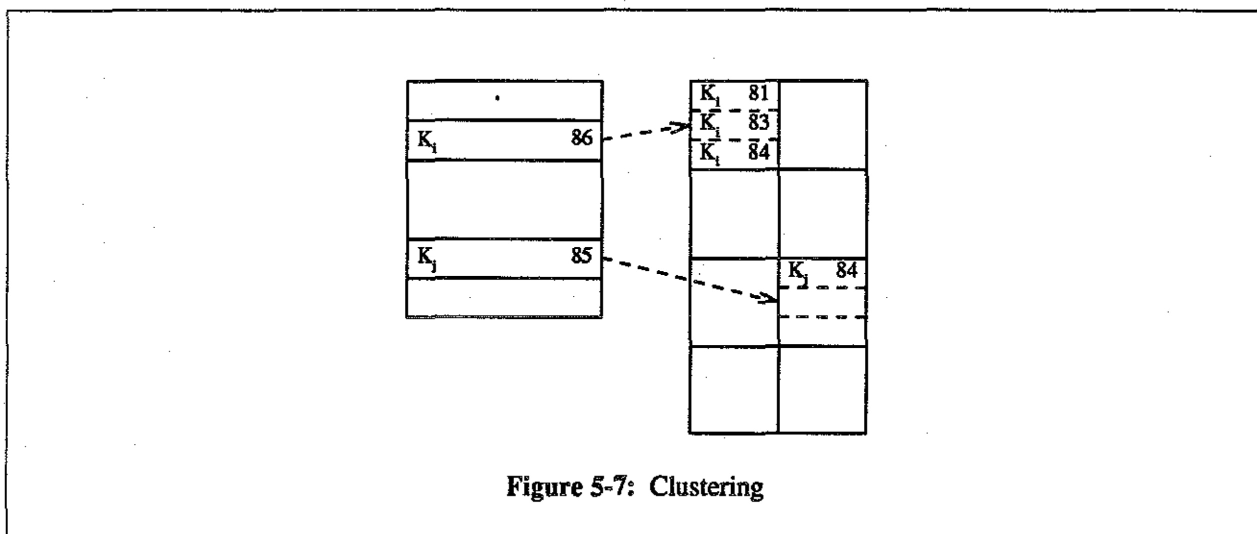


Figure 5-7: Clustering

If we maintain a pointer from each of the current version to its clustered blocks, its access path expression is:

$$[\ FilePath_1\ ;\ (\mathtt{P}\ \left\lceil \frac{n}{b} \right\rceil\ (\mathtt{S}\ b)\ \ (\mathtt{P}\ b))\ ]$$

There are $\frac{n}{b}$ blocks to be accessed to retrieve n history versions, where b is the number of records in a block. Since this scheme requires splitting of blocks when overflow occurs, it is not strictly applicable to optical disks. There are many variations for this scheme, as will be discussed next.

5.2.4.1. Variations

The simplest method is to assign a whole block to each version set with history versions, which results in unacceptably low storage utilization in most cases. This is a special case of *cellular chaining* to be described later, where a cell is a whole block.

A better method is to share the same block for history versions of several version sets. When an overflow occurs, the block is split into two, moving all versions of some selected version sets to a new block. If all versions in the overflowed block belong to one version set, a new block is added as a successor and chained to the original block. In this scheme, $\frac{n}{b}$ blocks need to be accessed to retrieve n history versions, where b is the number of records in a block.

In the temporally partitioned storage structure, there needs to be a link between the current version and its history versions to avoid scanning the whole history store. The link may be either *physical* or *virtual*. A physical link is a pointer physically stored as an implicit attribute of the current version. If some history versions are moved to other location as a result of an overflow, physical pointers in the current store pointing to those versions need to be adjusted accordingly. It is better to move the version set that has caused the overflow in this case, because it is easier to identify the version in the current store which corresponds to the versions being moved in the history store. If it is still necessary to move or compact other versions remaining in the original block, history versions need to maintain *back pointers* to the corresponding versions in the current store to adjust their pointers.

A *virtual link* is a conceptual link implied by some structural information. For example, history versions can be hashed on the primary key so that all versions belonging to a version set are put into one block or its overflow blocks. But the performance of conventional hashing with reasonable storage utilization deteriorates rapidly, as will be discussed further in Chapter 6, if there are excessive key collisions causing long overflow chains.

One way to resolve this problem is to introduce a *scatter table* between the current store and the history store, which can serve as a combination of the physical link and the virtual link [Morris 1968]. A scatter table may have the form of an index or a directory. Each entry in a scatter table corresponds to a value hashed from the primary key of tuples in the current store, and holds a pointer to a block in the history store. When an overflow occurs to a block in the history store, the block is split into two according to a hash function which generates a sequence of different values for each occurrence of overflows. Then a new entry pointing to the new block is added to a scatter table. A scatter table plays a similar role to accession lists, but an entry in a scatter table is shared by several synonymous tuples through a hash function, while an accession list is only for one tuple through a physical link.

Actual implementation of this scheme using a scatter table may adopt one of *variable size* hashing methods based on an index or a directory which can accommodate dynamic growth of a file by splitting a block upon overflow. Examples of such methods are *dynamic hashing, extendible hashing*, and *grid files*, where an index or a directory can be regarded as a scatter table described above.

Dynamic hashing [Larson 1978] maintains an index on hashed keys, where each entry of the index is a pointer to a disk block. Whenever an overflow occurs in a disk block, the block is split into two, and the corresponding index entry is also split into two. The index entries form a forest of binary trees while undergoing a sequence of overflows.

Extendible hashing [Fagin et al. 1979] maintains a directory of 2^d entries on hashed keys, where d is the directory depth. Several directory entries may share the same disk block, but about half of those entries are changed to point to a new block when an overflow occurs to the block and causes a split. The directory is doubled when the number of overflows for a block exceeds the directory depth.

Grid files [Nievergelt et al. 1984] of one dimension can also be used here by maintaining a directory on the hash values of keys. The directory consists of a *linear scale* and a *grid array*. Each element of the grid array holds a pointer to a data block. When an overflows occurs to a block, the block is split by adding a new block. If the overflowed block is shared by more than one grid array elements, one of the elements is changed to point to the new block. Otherwise, one of the intervals denoted by the linear scale is split by adding a new entry, and all the elements of the grid array corresponding to the split interval are also split to accommodate the new block.

All three methods make it possible to retrieve a record at the cost of one block access by locating the index or directory entry for a given key, assuming that the index or the directory is small enough to reside in the main memory. If the index or the directory does not fit into the main memory, one additional disk access is necessary.

There are other variable size hashing methods which can accommodate dynamic growth of the file without maintaining an index or a directory. They are *virtual hashing, linear hashing*, and *modified dynamic hashing*. Virtual hashing [Litwin 1978] doubles the whole file when an overflow occurs, and modifies the hash function for a block which had an overflow. It needs to maintain a bit map to indicate whether each bucket had an overflow or not, and suffers from low storage utilization.

Linear hashing [Litwin 1980] splits a block when an overflow occurs. But the block being split is not the one which had an overflow, but the one marked by the *split pointer* which increases one by one from the initial value of 0. The record which caused an overflow to a block is put into an overflow block chained to the original block, until the split pointer reaches the original block and splits all records in the chain of

the original and the overflow blocks. Though linear hashing extends the file size by one block at a time while maintaining only the split pointer, it still depends on overflow chains which degrade the overall performance.

Linear hashing *with partial expansions* [Larson 1980] is similar to linear hashing, except that two or more blocks are grouped together in adding a new block upon an overflow. It can improve storage utilization while exhibiting comparable performance. Another way to improve storage utilization is to defer splitting until a certain storage utilization is achieved (*controlled split*).

Modified dynamic hashing [Kawagoe 1985] attaches a *logical* address to each block in addition to a physical address. When an overflow occurs to a block, the block is split into two, and the logical address of the block is stored into a list. At the same time, all logical addresses equal to or smaller than that of the split block are changed. This method can locate a block for a given key at the cost of one block access, but needs to maintain a logical address for each block.

5.2.4.2. Nonlinear Hashing

As an improvement over these hashing methods which achieve the effect of clustering for each version set, a new method of hashing termed *nonlinear hashing* is proposed. Its objective is to retrieve records at the cost of exactly *one* block access, even when the file size grows or shrinks dynamically. It maintains a list of overflow addresses, called *overflow list*. Since the overflow list stores an address only when an overflow occurs, it is smaller than a directory or an index which maintains the addresses of all the buckets, and is expected to fit into the main memory. If the size of the overflow list grows too big, it is possible to reduce its size by reorganization.

Nonlinear hashing is similar to linear hashing in that it need not maintain the addresses of all the buckets. But it is better than linear hashing, because it splits an overflowed block, not a block selected in a linear order (hence the name *nonlinear hashing*).

In nonlinear hashing, each record is hashed on the primary key through a hash function h_0 first, whose range is $\{1, 2, ..., n_0\}$ where n_0 is the size of history blocks initially allocated. If a record needs to be inserted into a block which does not have enough free space, an overflow occurs. When an overflow occurs, a new block is appended to the end of the file. Then the overflowed block is split into two by

rehashing records in the block through a *split function* s_i, $i > 0$, where i, termed the *order* of overflow for the block, is the number of overflows that had occurred on the way to locate the block including the latest one. The split function s_i has the range of {0, 1}, and determines whether a record stays in the original block or is moved to the new block. Hence, the hash function h_0 and the split functions s_i should satisfy the constraints:

$$h_0 : K \rightarrow \{1, 2, ..., n_0\}$$

$$s_i : K \rightarrow \{0, 1\} \quad \text{for } i > 0$$

where K is an arbitrary key, and n_0 is the initial file size in blocks.

At this time, the address of the overflowed block is stored into a list, called the *overflow list*, which is initially empty. The *overflow list* is simply a list of addresses where overflows occurred, but it also represents information on the *order* of overflow for a block, and where a new block was added upon an overflow. Such information maintained by the overflow list is in fact sufficient to retrieve a record at the cost of exactly *one* block access, given the key of the record.

To determine the address of the block for a given key K, h_0 is first applied to K. If $h_0(K) = b_0$, where $1 \leq b_0 \leq n_0$, b_0 is called the *initial* address. If b_0 is not an active member of the overflow list, it becomes the *final* address of the block for K. Otherwise, determine the position p_0 of b_0 in the overflow list, and temporarily deactivate b_0 from the list. Then depending on whether $s_1(K)$ is 0 or 1, the next *intermediate* address b_1 becomes b_0 or $n_0 + p_0$, respectively. Now if b_1 is not an active member in the overflow list, it becomes the *final* address of the block for K. If b_1 is again an active member of the overflow list, then repeat the steps similar to those for the case of b_0 being a member of the overflow list, except that each subscript of p_0, b_0, s_1, b_1 is incremented by 1 respectively on each iteration. Note that the subscripts are determined by the *order* of overflows for the block in question, which is the number of overflows which had occurred along the path to the block. The final address of the block for K is determined when b_i for some i is found not to be an active member of the overflow list. At that moment, all inactive members of the overflow list are reactivated.

Once the final address of the block for K is determined, retrieving a record with such a key needs only to search the corresponding block. Whether the search is successful or not, its cost is just one block access, assuming that the overflow list can be kept in main memory. Thus the access path expression is

simply:

[$FilePath_1$; (H 1)]

To insert a record with a key *K*, the block at the final address for the key is checked if it has enough free space to receive the record. If so, the record is simply put into the block. Otherwise, a new block is appended to the end of the file, the original block is split into two, and the address of the original block is added to the overflow list, as mentioned earlier. The cost of an insertion is one or two block accesses depending on whether it involves an overflow.

A series of insertions into a file, whose initial size n_0 is 3, will illustrate how nonlinear hashing handles insertions. A sample hash function h_0 and the split functions s_i to satisfy the constraints of nonlinear hashing are:

$$h_0(K) = K \ mod \ n_0 + 1$$

$$s_i(K) = \frac{K}{n_0 \times 2^{i-1}} \ mod \ 2 \qquad for \ i > 0$$

Some of the split functions for $n_0 = 3$ are:

$$s_1(K) = \frac{K}{3} \ mod \ 2$$

$$s_2(K) = \frac{K}{3 \times 2} \ mod \ 2$$

$$s_3(K) = \frac{K}{3 \times 2^2} \ mod \ 2$$

Let's assume that each block can hold up to three records, and call a record with *n* as the key simply record *n*. If records 12, 29, 10, 30, 16, 25 are inserted in sequence, the file looks like Figure 5-8 (a). Thus far, the overflow list is null.

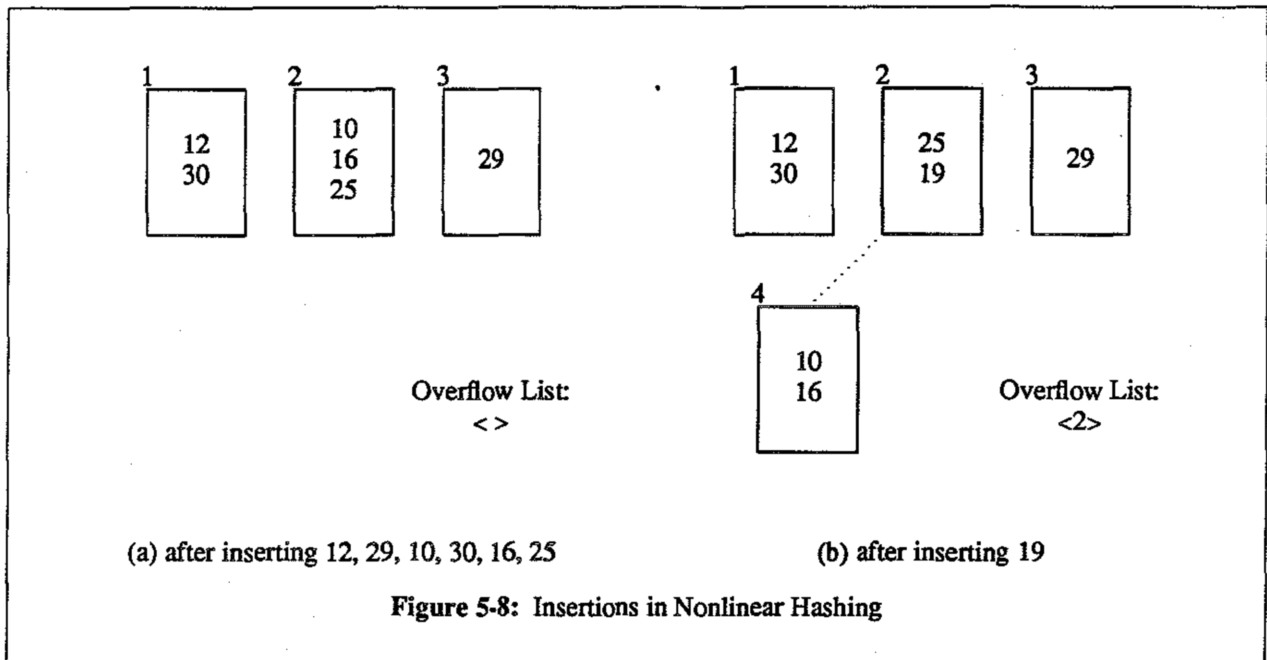
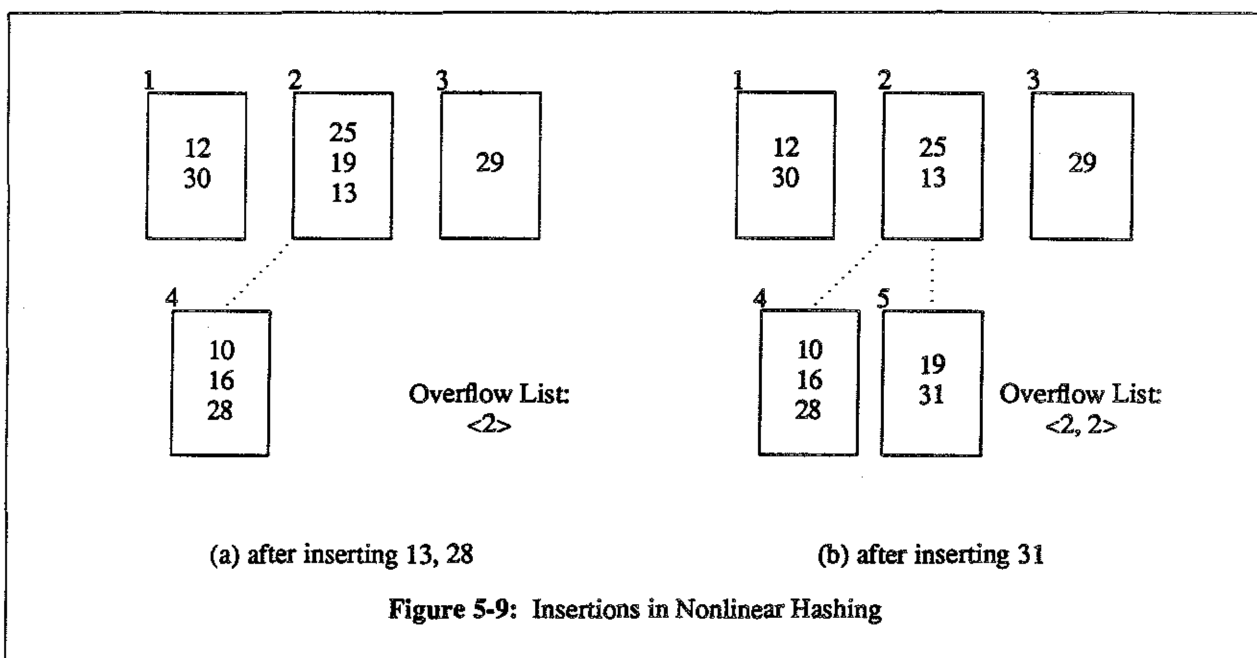


Figure 5-8: Insertions in Nonlinear Hashing

To insert record 19 next, $h_0(19) = 19 \, mod \, 3 + 1 = 2$. Since block 2 is already full, an overflow occurs. So a new block is appended as block 4, and records 10, 16, 25 are rehashed through s_1. Since $s_1(25) = 0$ and $s_1(10) = s_1(16) = 1$, record 25 stays in the original block 2, and records 10 and 16 are moved to the new block 4. The new record 19 is now put into block 2, since $s_1(19) = 0$. The file now looks like Figure 5-8 (b), and the overflow list becomes <2>. Note that the line between block 2 and block 4 is only conceptual, and does not denote any physical link.

For record 13, $h_0(13) = 2$. But there is the address 2 at the position 1 of the overflow list, which is now temporarily deactivated. Since $s_1(13) = 0$, its next intermediate address is still 2. There is no active member with the address 2 in the overflow list, so the final address for record 13 is 1. The record is put into block 2, and the member 2 of the overflow list is reactivated.

To insert record 28, $h_0(28) = 2$. Since block 2 is at the position 1 of the overflow list, and $s_1(28) = 1$, its next intermediate address is $n_0 + p_0 = 3 + 1 = 4$. There is no member with address 4 in the overflow list, so the final address for record 28 is 4. Figure 5-9 (a) shows the current status of the file.



(a) after inserting 13, 28

(b) after inserting 31

Figure 5-9: Insertions in Nonlinear Hashing

To insert record 31, $h_0(31) = 2$. But there is the address 2 at the position 1 of the overflow list, which is now temporarily deactivated. Since $s_1(31) = 0$, its next intermediate address is still 2. There is no more active member with address 2 in the overflow list, so the final address for record 31 is 2. But block 2 is already full. Thus a new block is appended as block 5, and records 25, 19, 13 are rehashed through s_2. Since $s_2(25) = s_2(13) = 0$ and $s_1(19) = 1$, records 25 and 13 stay in the original block 2, and record 19 is moved to the new block 5. The new record 31 is now put into block 5, since $s_2(31) = 1$. The file now looks like Figure 5-9 (b), and the overflow list is <2, 2>.

If we insert record 22 next, $h_0(22) = 2$. Since block 2 is at the position 1 of the overflow list, and $s_1(28) = 1$, its next intermediate address is $n_0 + p_0 = 3 + 1 = 4$. There is no member with address 4 in the overflow list, so the final address for record 28 is 4. But block 4 is already full, so a new block is appended as block 6, and records 10, 16, 28 are rehashed through s_2. Since $s_2(16) = s_2(28) = 0$ and $s_2(10) = 1$, records 16 and 28 stay in the original block, and record 10 is moved to the new block. The new record 22 is put into block 6, since $s_2(22) = 1$, and the overflow list now becomes <2, 2, 4>. Inserting records 22, 34 and 49 next results in Figure 5-10 (a).

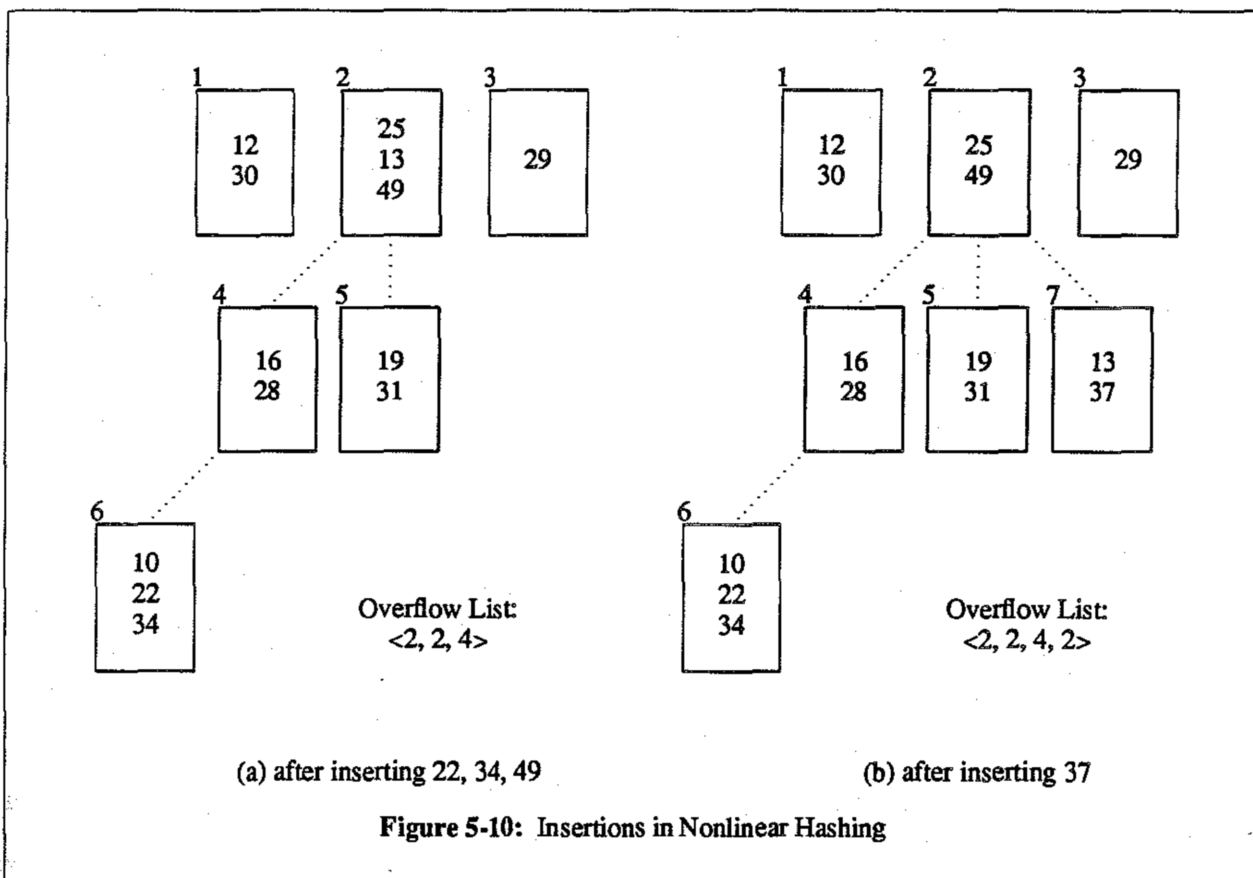


Figure 5-10: Insertions in Nonlinear Hashing

To insert record 37 as a final example, $h_0(37) = 2$. There is the address 2 at the position 1 of the overflow list, which is now temporarily deactivated. Since $s_1(37) = 0$, the next intermediate address is 2. But there is still the address 2 at the position 2 of the overflow list, which is also temporarily deactivated. Now $s_2(37) = 0$, so the next intermediate address is still 2. Block 2 is no longer an active member of the overflow list, so the final address becomes block 2. Since block 2 is already full, a new block is appended as block 7, and records 25, 13, 49 are rehashed through s_3. Since $s_3(25) = s_3(49) = 0$ and $s_3(13) = 1$, records 25 and 49 stay in the original block, and record 13 is moved to the new block. The new record 37 is put into the new block, since $s_3(37) = 1$. The result is Figure 5-10 (b), and the overflow list now becomes <2, 2, 4, 2>.

Now we define some terminology for nonlinear hashing. When an overflow occurs to a block, and a new block is added as a result, we call the original block the *parent* block, and call the new one the *child* block. Children with the same parent are called *siblings*, and a block without a child is called a *leaf*.

The *order* of overflow for a block is the number of overflows that had occurred along the path to locate a record in the block. It is identical to the order of the split function to be used to determine the final address of a record in the block. Now the order of a block can be defined as the number of blocks corresponding to its *ancestors*, its *older* siblings, and its *own* children.

In Figure 5-10 (b), for example, block 2 is the parent of blocks 4, 5 and 7. The order of blocks 1 and 3 is 0. Block 2 has 3 children, so its order is 3. Block 4 has one ancestor and one child, so its order is 2. The order of block 5 and 6 is also 2, since the former has an ancestor and an older sibling, while the latter has two ancestors. Block 7 has an ancestor and two older siblings, so its order is 3.

The children of a block can be found by locating all occurrences of the block number in the overflow list. If the block number is found at a position p_i of the overflow list, the address of the child block corresponding to the position is $n_0 + p_i$, where n_0 is the initial file size. Block 2 in Figure 5-10 (b) occurs at positions 1, 2 and 4 of the overflow list. Since $n_0 = 3$, the children of block 2 are blocks 4, 5 and 7.

The parent f_i of a block b_i can also be determined from the overflow list.

$$f_i = \begin{cases} \textit{none} & \text{if } b_i < n_0 \\ OL\ [b_i - n_0] & \textit{otherwise} \end{cases}$$

where $OL\ [k]$ denotes the k-th member of the overflow list. The parent of block 6 in Figure 5-10 (b) is block 4, since $OL\ [6 - n_0] = OL\ [3] = 4$.

It is also possible to delete a record, and merge two blocks into one if space permits. To delete a record with a key K, the block at the final address is retrieved. If there is no such record, the request fails. Otherwise, such a record is removed from the block. At this moment, we can check the possibility of merging the block with its parent or its child.

If a block is both a leaf and the youngest sibling, and if space permits, then the block can be merged easily into its parent block. This happens when a record is deleted from a parent block whose youngest child is a leaf, or when a record is deleted from a block which is both the leaf and the youngest child. In this case, the address of the parent block is removed from the overflow list, and the addresses of blocks added after the child block are decremented by one each.

For example, deleting records 16 and 28 from Figure 5-10 (b) results in Figure 5-11 (a). Since block 6 is a leaf and the youngest (only) sibling of block 4, we can merge block 6 into block 4, as shown in Figure 5-11 (b). Note that we do not merge block 4 with block 2, though they are also in a parent child relationship.

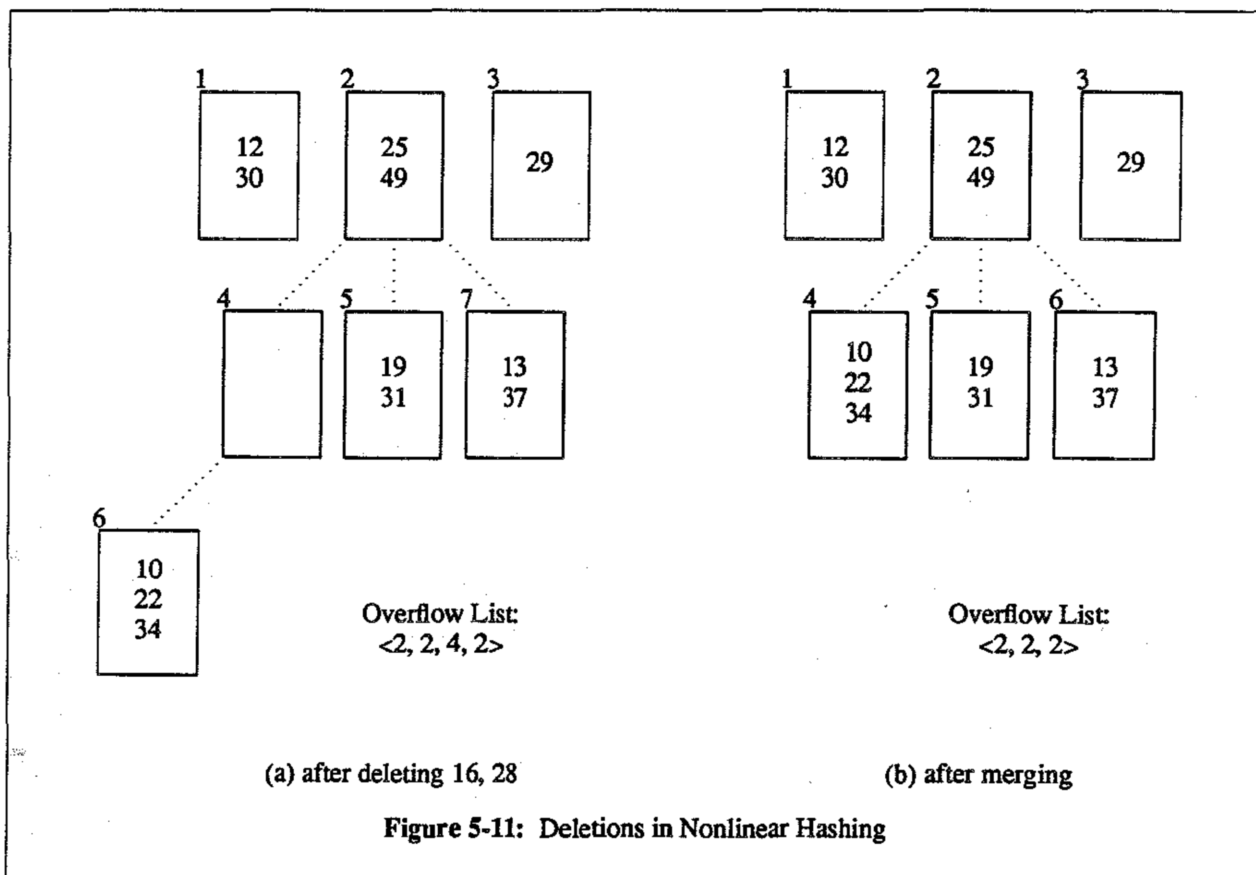


Figure 5-11: Deletions in Nonlinear Hashing

Merging a block which is not the youngest child or not a leaf is more complex. For example, deleting records 10 and 34 from Figure 5-11 (b) results in Figure 5-12 (a). Block 4, which is not the youngest child, can be merged into block 2, moving record 22 to the parent block. But we can't remove the address 2 from the position 1 of the overflow list, because it contains information on the orders of split functions for its younger siblings. Thus we mark such an address by negating it. Note that we have used the scheme in which the address counts from 1, not 0. Now the file and the overflow list look like Figure 5-12 (b).

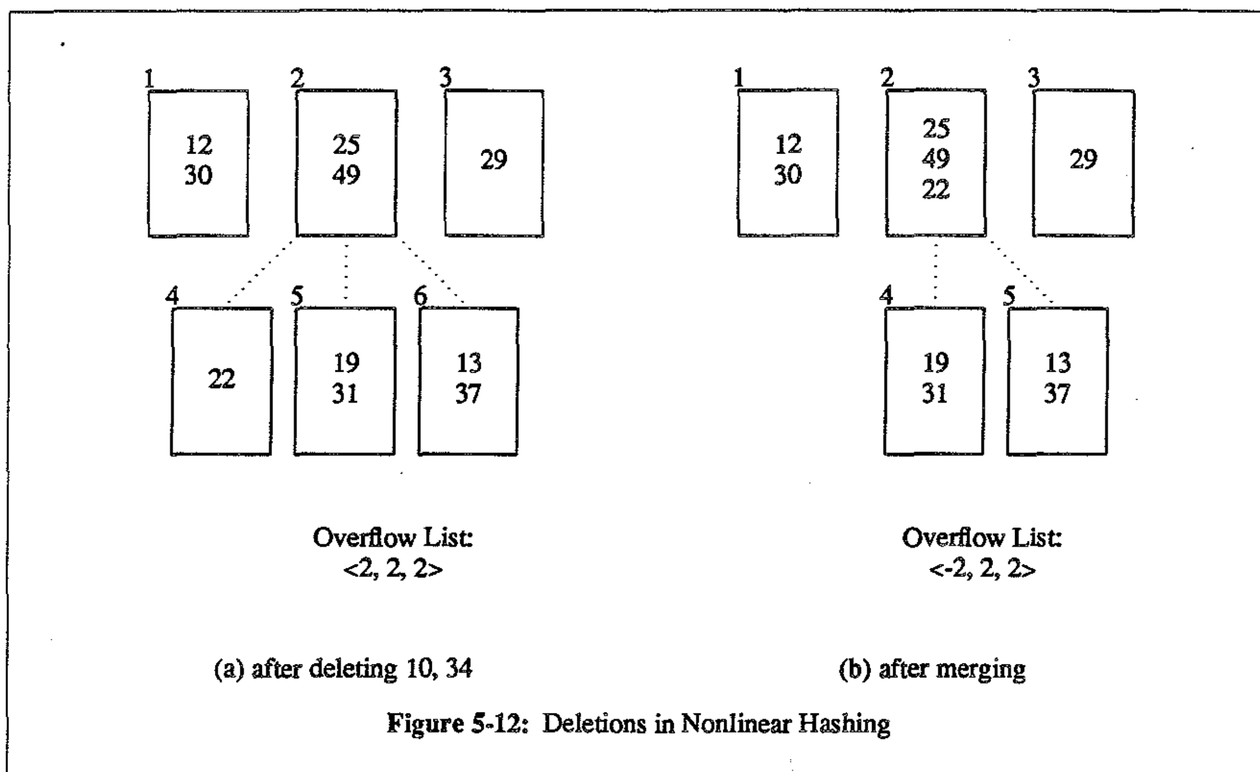


Figure 5-12: Deletions in Nonlinear Hashing

Merging a block which is the youngest child but not a leaf follows the same procedures. A negative member, *e.g.* -*j*, of the overflow list represents that there had been an overflow on block *j*, but the child block, which is not both a leaf and the youngest child, was later merged back into block *j*. Such a member participates in determining the order of overflows, but is not counted in determining the position of an overflow. If block *j* is an initial or an intermediate address for a key, and the next intermediate address is its child block, then *j* is the final address for the key. Note that record 22 in Figure 5-12 (b) should not be rehashed to a new child block, even when another overflow occurs to block 2. Detailed algorithms to retrieve, insert, and delete a record in nonlinear hashing are given in Appendix B.

The size of each entry is dependent on the the total size of a file. For a file with up to about 32,767 blocks, each entry takes 2 bytes. An entry of 4 bytes can handle up to about 2×10^9 blocks, which is about 8×10^{12} = 8 Tera bytes for the block size of 4,096. But the size of the overflow list depends on the number of overflows, not the size of the whole file. Hence the overflow list is small enough to fit into the main memory for most applications. For example, the size of the overflow list is only 32 K bytes for a file of 128 Mega bytes, assuming that a block holds 4 K bytes, the initial size was 16,000 blocks, and there were 16,000 overflows. With the overflow list of 256 K bytes, about 65,000 overflow blocks, or 256 Mega bytes

of overflow blocks, can be supported.

Nonlinear hashing needs to maintain two kinds of information. One is the address of each block which had an overflow, and the other is the occurrence number for each overflow. The overflow list represents such information compactly as a sequential list. Scanning the list to determine the final address of a key is not too expensive either, even when there were many overflows. If we assume that the average *order* of overflows is d, and that there are m entries in the overflow list, the number of entries to be examined appears to be $O(d \times m)$. However, the maximum number of entries to be examined is just m. Since the overflow list is maintained in the order of overflow occurrences, and no child block gets overflowed before its parent, no entry for a child block is ahead of the entry for its parent block in the overflow list. Thus we need to scan the overflow list only once, no matter what the order of overflows is.

The same information can be represented in the form of the *overflow set*, which is a set of [*address*, *position*] pairs. The first element *address* is the address of a block which had an overflow. There are two alternatives for the format of the second element *position*. First, *position* can be a list of numbers to represent the positions of the *address* in the overflow list. Then the overflow set for Figure 5-10 (b) is {[2, (1, 2, 4)], [4, (3)]}. The second alternative is that *position* is a single position number for the *address*, and there are as many entries for each address as there are overflows on the address. Then the overflow set for Figure 5-10 (b) becomes {[2, 1], [2, 2], [2, 4], [4, 3]}. Comparing the two alternatives, the first has to maintain a variable length list for each address, while the second repeats some addresses multiple times.

In either case, the overflow set takes more space than the overflow list. But it is possible to store the overflow set in a randomly accessible format. Hence the number of entries to be examined for determining the final address of a key is reduced to $O(d)$. Various methods of hashing are obvious candidates for this purpose, and nonlinear hashing itself can be applied to maintain the overflow set using the *address* as the key (termed *nested nonlinear hashing*). If the overflow set does not fit into the main memory, the overflow set has to reside on the disk, but the overflow set of the overflow set will be small enough to stay in the main memory.

For example, an overflow list of 20 K bytes can support 10,000 overflow blocks, each of which has 512 entries, assuming that the second alternative above is used with the block size of 4 K bytes. Thus we can support about 5×10^6 overflow blocks, or a file of $2 \times 10^{10} = 20$ Giga bytes. In this case, the average

number of disk accesses to retrieve a record given a key is $d+1$, where d is the average order of overflows.

We can determine the average storage utilization for nonlinear hashing from the analysis result of the extendible hashing [Fagin et al. 1979]. Assuming uniform distribution of keys, the average number of blocks to store n records is $\frac{n}{m \times \ln 2}$, when a block holds up to m records. Since the minimum number of blocks to store n records is $\frac{n}{m}$, the average storage utilization is $\ln 2 = 69.3\%$.

Now the average number of overflow blocks is $\frac{n}{m \times \ln 2} - n_0$, where n_0 is the initial size of the file. Then the average order of overflow is $\log_2 \left\lceil \frac{n}{m \times \ln 2} - n_0 \right\rceil$, assuming uniform distribution.

If the size of overflow list grows big enough to degrade the performance, it is possible to reduce or even eliminate the overflow list through *reorganization*, using a new hash function h_0 with a larger n_0. Nonlinear hashing combined with periodic reorganization can provide excellent performance characteristics with high storage utilization.

As discussed thus far, nonlinear hashing handles dynamic growth and shrinkage of a file through splitting and merging of data blocks. Compared with other variable size hashing methods, it has the advantage of retrieving a record at the cost of exactly one block access, whether successful or not, simply by maintaining the overflow list in main memory.

One problem is the case when a split function fails to divide records of an overflowed block into two groups, *e.g.* when all records have the same key. In that case, we need to maintain a chain of overflow blocks. But accessing the chain of overflow blocks sequentially is not wasteful, because data records were already clustered, and it is usually necessary to retrieve all records belonging to a version set anyhow.

It is conceivable to use time attributes as a part of a key, but there are serious problems with this approach. 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 to be specified as a predicate, especially when the resolution of time values is

fine. Hence, we should be able to support a *range* query on time attributes, which is not possible with some access methods, *e.g.*, hashing. Access methods such as *grid files* [Nievergelt et al. 1984] and *K-D-B trees* [Robinson 1981] can support a multi-attribute key and range queries better, but there is an overhead to maintain the necessary structures.

5.2.5. Stacking

Stacking is a two dimensional implementation of a conceptual cube where all the version sets have an equal number of versions. This is useful when we are interested in the fixed number of most recent versions, where updates are rather periodic and uniformly distributed. For example, *Postgres* stores history data, but discards data older than a specified amount of time [Stonebraker 1986].

When the first history version is put into the history store for a version set, space for d versions is allocated, where d is termed the *depth* of stacking. Subsequent versions are put into the remaining portion of the allocated space. After the predetermined limit d to the number of versions is reached, the next version is put into the place of the oldest version, which becomes lost as if being pushed through the bottom of a stack.

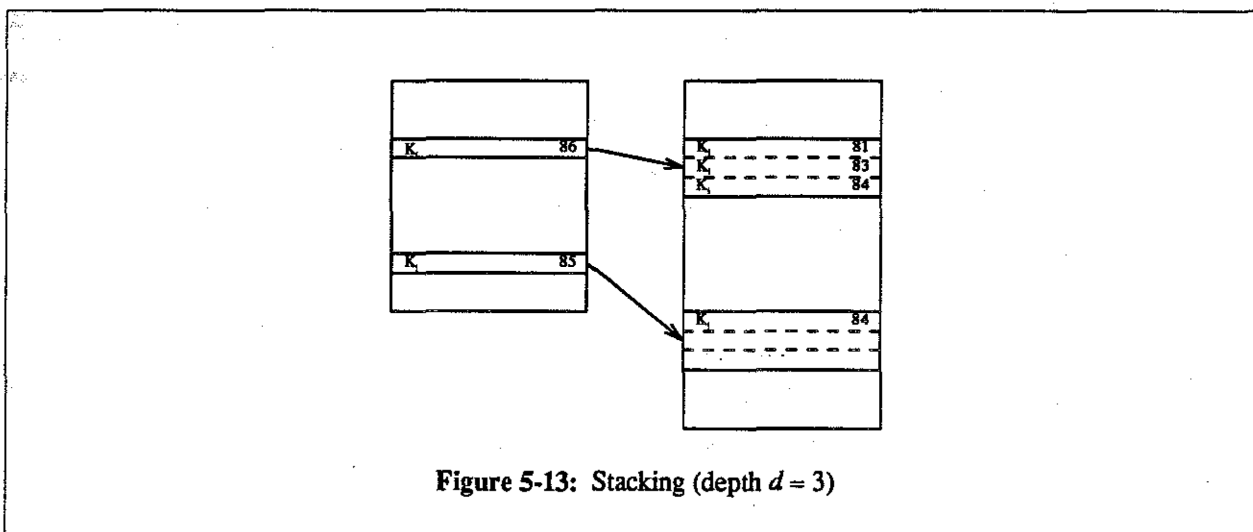


Figure 5-13: Stacking (depth $d = 3$)

This scheme is not strictly applicable to optical disks, since it assumes rewriting of existing data. Its access path expression is:

$$[\ FilePath_1\ ;\ (\mathtt{P}\ d)\ ]$$

where $FilePath_1$ is for the current store, and d is the allocated depth of a stack.

Since the number of history versions to be maintained is predetermined, it is simple to cluster all versions belonging to a version set. Thus, the number of block accesses for retrieving n history versions is just one. Storage utilization is $\frac{u}{d}$ with the maximum of 100%, where u is the update count. Increasing the depth d enables a larger number of versions to be maintained, but storage utilization can be as low as $\frac{1}{d}$. The data being replaced by newer versions may not actually be lost, but can be archived to a lower level storage. Another interesting possibility is to organize the current store as a *shallow* stack, a stack with a small d, then store overflow data into the history store which may use any of the formats discussed in this section.

5.2.6. Cellular Chaining

Cellular chaining is similar to reverse chaining, but attempts to improve the performance by collecting several versions into one cell. The current version initially has an extra field *nvp* (next version pointer) of null. When the first version is inserted into the history store for a version set, a cell is allocated with the size of $c \geq 1$ in the history store. The field *nvp* of the current version now points to the cell, and subsequent versions will be put into the remaining space of the cell. If this space is filled up, another cell is allocated and chained to the predecessor cell.

Its access path expression is:

$$[\ FilePath_1\ ;\ (\mathtt{P}\ \left\lceil \frac{n}{c} \right\rceil\ (\mathtt{S}\ c)\ \ (\mathtt{P}\ c))\ ]$$

where n is the number of history versions, and c is the cell size in records. Since the history store operates in the append only mode, this scheme can use optical disks as well.

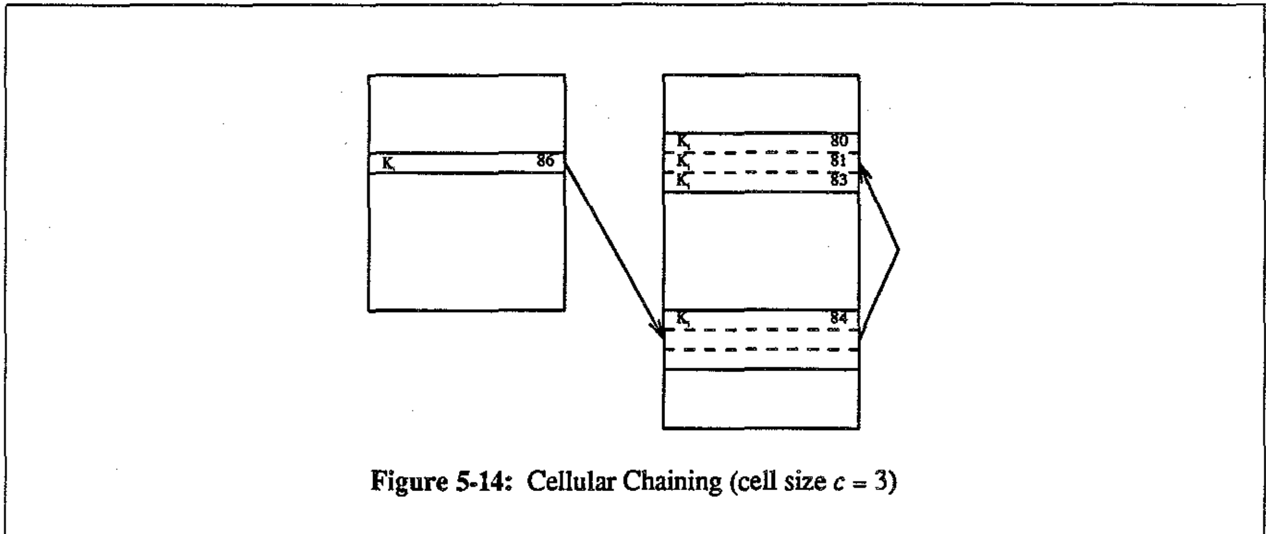


Figure 5-14: Cellular Chaining (cell size $c = 3$)

This can be regarded as a combination of reverse chaining and stacking. It also has the benefit of the clustering scheme, in that the number of blocks to be accessed is reduced as many as c times. The lower bound for the number of block accesses in retrieving n history versions is $\frac{n}{b}$, where b is the blocking factor of the history store. The upper bound for the same case is $\frac{n}{c}$, where c is the cell size of the history store. Thus increasing the cell size c improves the performance.

But a larger cell size tends to lower storage utilization. If the number of version sets are uniformly distributed, expected storage utilization can be calculated as:

$$E\ (Storage\ Utilization) = \left(\frac{1}{c} + \frac{2}{c} + \cdots + \frac{c}{c}\right) \times \frac{1}{c} = \frac{\sum_{i=1}^{c} i}{c^2} = \frac{c+1}{2c}$$

This shows that the average storage utilization is 100% for $c = 1$, which is the same as reverse chaining, ignoring the partially filled block at the end of the history store But the storage utilization falls to about 50% for a reasonably large c. It is possible to improve storage utilization by adjusting the cell size dynamically. The size of the cell can be increased linearly. For example, the first cell of each version set has the size of one, but each time a new cell is allocated for one version set, the cell size increases by one. Or the cell size may be multiplied by some factor, whenever a new cell is allocated for one version set.

5.3. Secondary Indexing

Performance of queries retrieving records through non-key attributes can be improved significantly by *secondary indexing*. This section discusses the types and the structures of secondary indices for databases with temporal support.

5.3.1. Types of Secondary Indices

For a snapshot relation, a secondary index is a set of *<value, pointer>* pairs, where *value* is a secondary key and *pointer* is the unique identifier or the address of the corresponding tuple. Since the *value* is not expected to be unique, there may be several entries for a single *value*. There will be more entries for each *value* in a secondary index for a relation with temporal support, because it maintains history versions in addition to current data. A typical query retrieves only a small subset of all the versions for a given *value*, but temporal predicates to determine which versions satisfy the query can be evaluated only after accessing the data themselves. The number of false hits can be reduced if some or all of temporal information is also maintained in a secondary index. Therefore, extension of the conventional secondary index is desirable for each type of databases with temporal support.

For a rollback database, a secondary index itself can be a rollback relation augmented with attributes transaction start and transaction stop. Then each index entry is a quadruple *<value, pointer, transaction start, transaction stop>*. There is the overhead of 8 bytes for each entry, but the **as of** clause can be evaluated from the information in the secondary index. Only the tuples satisfying the **as of** clause need to be retrieved, significantly enhancing the performance. If the version sets are *contiguous* or nearly *contiguous*, storing only the transaction start attribute can save space without significant loss of performance. The same argument applies to a historical database, when the **valid** clause is substituted for the **as of** clause, and the attributes valid from and valid to are used instead of the attributes transaction start and transaction stop.

For a temporal database, a secondary index may be a rollback relation, a historical relation, or a temporal relation itself. If the index is a rollback relation, the **as of** clause can be evaluated from the information in the index. Then those versions that satisfy the **as of** clause are retrieved from the current or the history store to resolve the **valid** predicate. If the index is a historical relation, those tuples that

fail the `valid` clause need not be accessed to resolve the `as of` predicate. If the index is itself a temporal relation, each index entry is a sextuple `<value, pointer, valid from, valid to, transaction start, transaction stop>`. There is the overhead of 16 bytes for each entry, but temporal predicates of the `valid` and the `as of` clauses can be evaluated completely from the information in the secondary index. It is also possible to store some subsets of the four time attributes, *e.g.* valid from and transaction start, or only one of the two. Storing only a subset saves space, but the number of false hits will increase.

	Snapshot	Rollback	Historical	Temporal
Snapshot Database	√			
Rollback Database	√	√		
Historical Database	√		√	
Temporal Database	√	√	√	√

Figure 5-15: Types of Secondary Indices for Each Type of Databases

The type of secondary indices available for each type of databases is summarized in the Figure 5-15. Deciding which type of secondary index to use for a database with temporal support is a typical question of space time tradeoff.

5.3.2. Structures of Secondary Indices

The size of databases with temporal support is monotonically increasing, and so is the size of secondary indices for such databases. For a large relation especially with temporal support, its secondary index becomes so large that it is important to design a suitable structure which can reduce the access cost for the index. Any conventional storage structures such as heap, hashing, ISAM, *etc.* can be used, but care should be taken for non-temporal queries so that the cost of using the index does not overwhelm the savings achieved from the temporally partitioned store.

Instead of storing all index entries for all the versions into a single file, the index itself can be maintained as a *temporally partitioned* structure having the *current index* for current data and the *history index* for history data. The benefits of the temporally partitioned store considered for storing data similarly apply to secondary indices with the temporally partitioned structure. By separating current entries from the bulk of history entries, the current index becomes smaller and more manageable, minimizing the overhead of maintaining history versions on non-temporal queries. The history index can utilize any format developed for the history store to enhance the performance of temporal queries. For example, the current index may be hashed, while the history index is cellularly chained. Performance comparisons of various structures for secondary indices will be given in Chapter 7.

5.4. Attribute Versioning

The discussion thus far has implicitly assumed *tuple versioning* which maintains multiple versions for updated tuples. The other alternative is *attribute versioning* to maintain versions for each attribute [Clifford & Tansel 1985, Gadia & Vaishnav 1985].

In tuple versioning, each tuple is augmented with time attributes specifying the period while the tuple is in effect. The number and kind of time attributes vary depending on the type of the relation, and whether the relation models an *interval* or an *event*. For simplicity of presentation, we will denote the time attributes as [*time_from*, *time_to*). When a tuple is first inserted, the *time_to* component of the interval is set to '∞', indicating that the tuple is currently valid. A `delete` operation on an existing tuple changes the *time_to* component of the tuple from '∞' to some t_1. The value of t_1 is usually the current time, but it can be specified explicitly by the `delete` statement for a historical database. For a `replace` operation, a new version of the tuple augmented with an appropriate interval is inserted after a *virtual* `delete` operation is executed as above.

For example, an `Employee` relation in a historical database may look like Figure 5-16, showing 4 versions for "John" who received a series of promotions, and a version for "Tom" who quit. An obvious drawback with this approach is the high degree of redundancy owing to duplication of an entire tuple, especially when the changed portion is relatively small compared with the unchanged portion.

Name	Title	Salary	[*time_from*, *time_to*)
John	Programmer	25	[Jun 81, Sep 82)
John	Programmer	30	[Sep 82, Mar 83)
John	Manager	30	[Mar 83, Dec 84)
John	Manager	35	[Dec 84, ∞)
Tom	Programmer	27	[Sep 83, Jun 84)

Figure 5-16: A Relation in Tuple Versioning

In attribute versioning, an attribute is either *static* or *dynamic*, depending on whether its value changes over time. Static attributes, *e.g.* `Name` in Figure 5-17, are constant and simple-valued. On the other hand, each dynamic attribute of a tuple is a set of *<value, interval>* pairs, where handling of the interval [*time_from*, *time_to*) is similar to tuple versioning except that the interval is associated with each version of an attribute value. When a tuple is first appended, the *time_to* component of the interval for each attribute value is set to '∞'. A **`delete`** operation on an existing tuple changes the *time_to* component for each of the current version of dynamic attributes in the tuple from '∞' to some t_1, as described for tuple versioning. **`Append`** and **`delete`** operate on the whole tuple, but the resolution of **`replace`** is the attribute. For a **`replace`** operation, new versions of the changed attributes are inserted with the appropriate time attributes after the **`delete`** operation is executed on the affected attributes. Thus a *tuple* in attribute versioning corresponds to a *version set* in tuple versioning. This results in a non first normal form relation [Jaeschke & Schek 1982]. One restriction on attribute versioning is that the time interval associated with each attribute should be the same for all attributes in a tuple (*homogeneity requirement* [Gadia & Vaishnav 1985]). The remainder of this section describes how to convert one form to the other, compares storage requirements, and discusses how to support attribute versioning with the temporally partitioned storage structure.

5.4.1. Conversion

Attribute versioning and tuple versioning are equivalent in terms of their information contents, which can be proved by induction on the number of updates. Therefore, it is possible to derive one form from the other. One can convert a relation in tuple versioning to one in attribute versioning by following the description above for **`insert,`** **`delete`** and **`replace`** operations. Figure 5-17 is the result of converting the relation in Figure 5-16 to attribute versioning.

Name	Title		Salary	
John	Programmer	[Jun 81, Mar 83)	25	[Jun 81, Sep 82)
	Manager	[Mar 83, ∞)	30	[Sep 82, Dec 84)
			35	[Dec 84, ∞)
Tom	Programmer	[Sep 83, Jun 84)	27	[Sep 83, Jun 84)

Figure 5-17: A Relation in Attribute Versioning

Conversion of a relation from attribute versioning to tuple versioning can be formalized by two operations, UNNEST and SYNCH, disregarding computational efficiency. The UNNEST operation was first introduced by Jaeschke and Schek to transform a non first normal form relation to a first normal form relation [Jaeschke & Schek 1982], and later adopted as the UNPACK operation for a historical relation [Clifford & Tansel 1985, Ozsoyoglu et al. 1985]. Let $r_A(R)$ be a relation over scheme R in attribute versioning. Let $P \in R$ be a particular attribute, and $C_P = R - \{P\}$ be the remaining attributes. Then the UNNEST operation on the attribute P for r_A is:

$$\text{UNNEST}_P(r_A) = \bigcup_{t \in r_A} (\text{UNNEST}_P \, (roman \; \{t\}))$$

where

$$\text{UNNEST}_P(\{t\}) = \begin{cases} \{ t \} & \textit{if } P \textit{ is static (simple-valued)} \\ \{ t' \mid t'[P] \in t[P] \wedge t'[C_P] = t[C_P] \} & \textit{otherwise} \end{cases}$$

Note that the result of an UNNEST operation is a relation preserving the same relation scheme. Applying the UNNEST operation to all attributes of a relation results in a relation in first normal form.

Name	Title		Salary	
John	Programmer	[Jun 81, Mar 83)	25	[Jun 81, Sep 82)
			30	[Sep 82, Dec 84)
			35	[Dec 84, ∞)
John	Manager	[Mar 83, ∞)	25	[Jun 81, Sep 82)
			30	[Sep 82, Dec 84)
			35	[Dec 84, ∞)
Tom	Programmer	[Sep 83, ∞)	27	[Sep 83, ∞)

Figure 5-18: Partial UNNEST'ing of A Relation with Attribute Versions

For example, $\text{UNNEST}_{Name}(Employee)$ on the relation in Figure 5-17 returns the same relation, but

UNNEST$_{Title}$ (UNNEST$_{Name}$ (*Employee*)) results in the relation in Figure 5-18. Note that Figure 5-17 shows 2 tuples, one each for "John" and "Tom", but Figure 5-18 shows 3 tuples, one more for "John".

Repeating the UNNEST operation on each dynamic attribute of a relation results in a relation in first normal form. For example, UNNEST$_{Salary}$ (UNNEST$_{Title}$ (UNNEST$_{Name}$ (*Employee*))) obtains a fully unnested relation as shown in Figure 5-19, which shows 7 tuples, four more for "John".

Name	Title		Salary	
John	Programmer	[Jun 81, Mar 83)	25	[Jun 81, Sep 82)
John	Programmer	[Jun 81, Mar 83)	30	[Sep 82, Dec 84)
John	Programmer	[Jun 81, Mar 83)	35	[Dec 84, ∞)
John	Manager	[Mar 83, ∞)	25	[Jun 81, Sep 82)
John	Manager	[Mar 83, ∞)	30	[Sep 82, Dec 84)
John	Manager	[Mar 83, ∞)	35	[Dec 84, ∞)
Tom	Programmer	[Sep 83, Jun 84)	27	[Sep 83, Jun 84)

Figure 5-19: Full UNNEST'ing of the Relation in Figure 5-17

Another way to obtain a relation in first normal form is to apply, for each tuple, a series of *cartesian products* of unary relations, each of which is an attribute of the tuple. Let $t \in r_A$, $A_i \in R$, $a_{t,i} = \{ p \mid p = t\ [A_i] \}$, $1 \le i \le n$, and n = Degree (R), then

$$\text{UNNEST}_{A_1} (\text{UNNEST}_{A_2} \cdots (\text{UNNEST}_{A_n} (r_A))...)$$

$$= \bigcup_{t \in r_A} (a_{t,1} \times a_{t,2} \cdots \times a_{t,n})$$

In order to obtain a relation in tuple versioning, the result of a series of UNNEST operations needs to be processed by the SYNCH operation. The SYNCH operation on a tuple in the unnested form determines the largest interval during which all the attribute values of the tuple are in effect. A tuple which gives the null interval upon the SYNCH operation may be removed. Let r_U be a relation unnested from r_A in attribute versioning, both over the same relation scheme R. Let further $t \in r_U$, $A_i \in R$, $1 \le i \le n$, n = Degree (R), and $t\ [A_i] = < v_i,\ [time_from_i, time_to_i) >$. Then

$$\text{SYNCH } (t) = \begin{cases} [t_from, t_to) & \text{if } t_from < t_to \\ (\ ) : \textit{the null interval} & \textit{otherwise} \end{cases}$$

where

$$t_from = \max(time_from_i), \quad 1 \le i \le n$$
$$t_to \quad = \min(time_to_i), \quad 1 \le i \le n$$

Results of SYNCH operations on the first three rows in Figure 5-19 are [Jun 81, Sep 82), [Sep 82, Mar 83), and the null interval ( ), respectively. By applying the SYNCH operation to each row of the unnested relation, and removing tuples with null intervals, a relation in attribute versioning can be transformed into one in tuple versioning as in Figure 5-16.

5.4.2. Storage Requirements

Though attribute versioning avoids duplication of static data, there is additional overhead in associating time information with each attribute and maintaining a list of versions for each attribute. Given

t_s : total size of static attributes

n_d : number of dynamic attributes

a_d : average size of dynamic attributes

o_A : size of overhead for each attribute version

o_T : size of overhead for each tuple version

c : average number of updates for each version set

α : average number of attributes modified by an update operation

it is possible to calculate storage requirements for tuple versioning and attribute versioning, s_T and s_A, respectively.

$$s_T = (t_s + n_d * a_d + o_T) * (c + 1)$$

$$s_A = t_s + n_d * (a_d + o_A) + \alpha * c * (a_d + o_A)$$

The number of updates for each version set to favor attribute versioning, c', is one to make s_A smaller than s_T. Therefore,

$$c' > \frac{n_d * o_A - o_T}{t_s + (n_d - \alpha) * a_d + o_T - \alpha * o_A}$$

Since each update modifies at least one attribute ($\alpha \ge 1$),

$$c' > \frac{n_d * o_A - o_T}{t_s + (n_d - 1) * a_d + o_T - o_A}$$

When $o_T = o_A$, which is often the case,

$$c' > \frac{(n_d - 1) * o_A}{t_s + (n_d - 1) * a_d}$$

This result shows that c' is proportional to o_A, inversely proportional to a_d, and relatively unaffected by n_d, which is somewhat surprising. If t_s is 0, or small compared with $(n_d - 1) * a_d$, c' turns out to be simply $\frac{o_A}{a_d}$. Note that o_A is fixed for a particular implementation, so the only variable is the average size of dynamic attributes. In the special case of $n_d = 1$, c' becomes 0, meaning that in this particular situation attribute versioning always wins.

For example, if we have $t_s = 12$, $a_d = 8$, and $n_d = 2$, as in Figure 5-15, and assume 2 time attributes of 4 bytes each with 2 bytes for linking overhead ($o_A = 10$), then $c' > 0.5$ updates per version set would favor attribute versioning.

An advantage for databases in tuple versioning is that the relational theory developed for conventional DBMS's can be utilized to some extent, as they are at least in first normal form. There has been some effort to formalize the concepts and algebra for attribute versioning [Clifford & Tansel 1985, Gadia & Vaishnav 1985, Gadia 1986, McKenzie 1986], but further research is needed in various aspects of query processing in such databases.

5.4.3. Temporally Partitioned Store

Now we look into the question of how to support attribute versioning in the temporally partitioned storage structure to process TQuel queries efficiently. As shown in Figure 5-20, a relation in attribute versioning can be conceptualized as a sparse matrix of nodes, each of which is an attribute version. There are three types of links connecting nodes to one another:

t link : link between tuples

a link : link between attributes

v link : link between attribute versions.

Each of these links can be implemented either physically or virtually. A *physical link* is a physical pointer stored into a node. A *virtual link* is a conceptual link implied by physical contiguity, by physical information such as lengths of tuples and attributes, or even by a hash function. Note that the most current tuple can be found by collecting the most current version from each attribute.

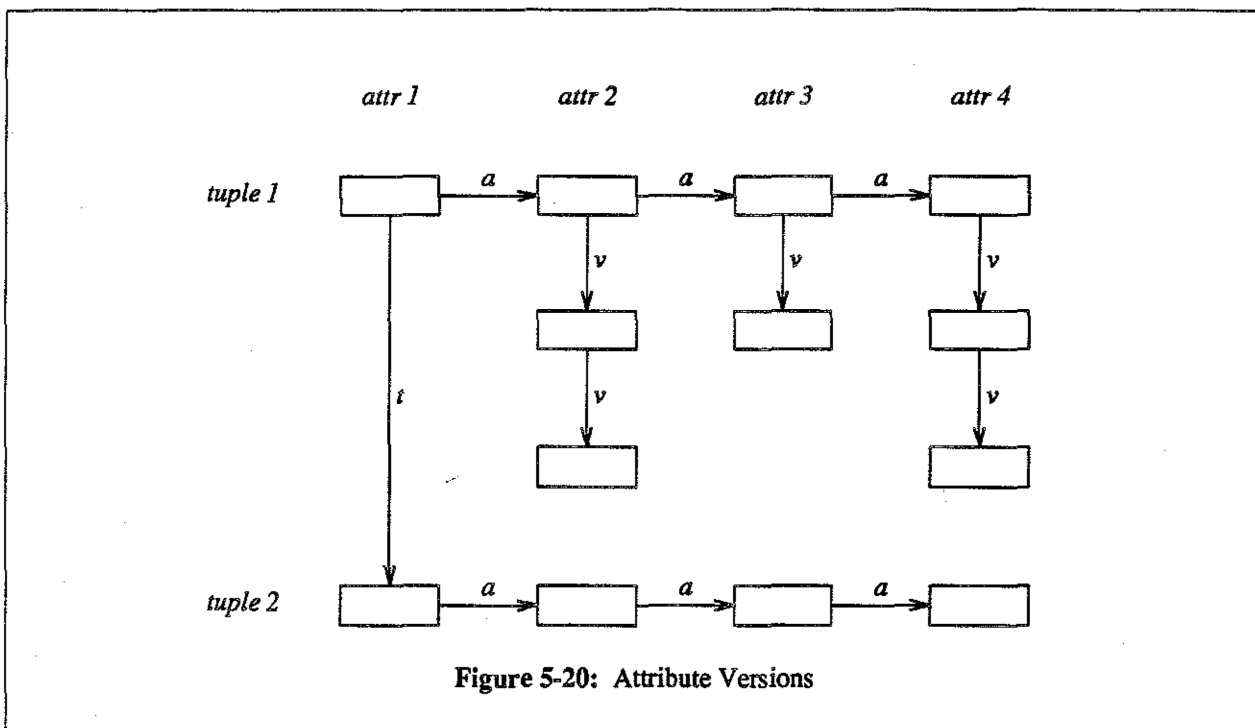


Figure 5-20: Attribute Versions

The formats of the history store discussed in Section 5.2 assumed tuple versioning, but most of them are easily extendible to support attribute versioning. Each tuple in the current store contains exactly one current version for each member attribute. Thus, the *t link* and the *a link* are virtual. Only the *v link* needs to be maintained for each version of dynamic attributes in the history store. For *reverse chaining,* a field *nvp* (next version pointer) is attached to each version of dynamic attributes, and a chain of versions is maintained following similar procedures as described in Section 5.2.1 for tuple versioning. For *clustering,* versions for the same attribute, and then attributes for the same tuple, are clustered together. For *stacking,* space is reserved for a certain number of versions for each dynamic attributes, making the *v link* virtual. For *cellular chaining,* each dynamic attribute maintains a chain of cells whose size is either fixed or variable. For *accession lists,* however, each dynamic attribute in a tuple needs a separate list, which makes management of those lists overly complicated. Indexing or secondary indexing on the history store is not strictly applicable either, because indices need to be maintained for each version of an attribute, not for

each version of a tuple.

5.5. Summary

Section 5.2 investigated six structures for the history store, and Section 5.3 and 5.4 discussed related issues such as secondary indexing and attribute versioning. Various characteristics of those six structures for the history store are compared in Figure 5-21.

Structure	Append-*Only*	Attribute *Versioning*	*Block Accesses Lower Bound*	*Upper Bound*	*Average*
Reverse Chaining	√	√	$\frac{n}{b}$	n	*See (1)*
Accession Lists	√		2	$n+1$	*See (2)*
Indexing	√		1	n	*See (2)*
Clustering		√	$\frac{n}{b}$	$\frac{n}{b}$	$\frac{n}{b}$
Stacking		√	1	1	1
Cellular Chaining	√	√	$\frac{n}{b}$	$\frac{n}{c}$	*See (2)*

Notes:

n : number of history versions for the version set
b : number of tuples in a block
c : number of tuples in a cell
(1) Given in Equation (5.1).
(2) Depending on the given temporal predicate.

Figure 5-21: Structures for the History Store

This table shows for each format whether the format can be implemented as *append only*, and whether it can support *attribute versioning*. The table also compares the lower bound, the upper bound, and the average number of block accesses for each method, when there are n history versions for a versions set. Reverse chaining was implemented to obtain performance data for comparison with the analysis results, as will be discussed in Chapter 7.

PART III

Benchmarks

A prototype temporal database management system has been implemented by extending the snapshot DBMS INGRES. Part three discusses the major features of the prototype and describes the results of the benchmarks run on the prototype. In particular, Chapter 6 is on the prototype with conventional access methods, and Chapter 7 is on the prototype with the new access methods discussed for the temporally partitioned store in Chapter 5.

Chapter 6

Prototype with Conventional Access Methods

A prototype temporal database management system was built by extending the snapshot DBMS INGRES [Stonebraker et al. 1976]. It supports the temporal query language TQuel, described in Section 2.3, and handles all four types of databases: *snapshot*, *rollback*, *historical* and *temporal*. A set of queries were run as a benchmark to study the performance of the prototype on the four types of databases using conventional access methods, and to identify major factors affecting the performance of the prototype. This chapter describes the major features of the prototype, and presents the results of the benchmark as reported in [Ahn & Snodgrass 1986].

6.1. Prototype

There are several approaches to implementing a database management system with temporal support. One initial strategy would be to interpose a layer of code between the user and a conventional snapshot database system. This *layered* approach has a significant advantage of not requiring any change to the complex data structures and algorithms within the snapshot DBMS. However, the performance of such a system will deteriorate rapidly not only for temporal queries but also for non-temporal queries, due to peculiar characteristics of databases with temporal support. There is also an overhead to translate, if possible at all, a temporal query into an equivalent non-temporal query supported by the underlying snapshot DBMS.

An alternative is to integrate temporal support into the DBMS itself, developing new query evaluation algorithms and access methods to achieve reasonable performance for a variety of temporal queries, without penalizing conventional non-temporal queries. There are several issues that must be addressed for this *integrated* approach, such as handling of ever-growing storage size, use of low cost high capacity write-once storage, representation of temporal versions with little redundancy, and efficient access methods for temporal and non-temporal queries [Ahn 1986]. This approach clearly involves substantial

research and implementation effort, yet holds promise for significant performance enhancement.

As an intermediate step towards a fully integrated system, a prototype temporal DBMS was built by extending the snapshot DBMS INGRES [Stonebraker et al. 1976]. Many routines in INGRES to parse, decompose, and interpret queries were modified, and several routines were added for new temporal constructs, but access methods available in INGRES were kept. Thus the performance of the prototype was expected to be less than ideal, rapidly deteriorating for both temporal and non-temporal queries. But it is still useful to identify problems with conventional access methods, and to suggest possible mechanisms for addressing those problems. In addition, the prototype can serve as a comparison point for fully integrated DBMS's to be developed later.

The prototype supports all the augmented TQuel statements: **retrieve, append, delete, replace** and **create**. Temporal clauses in TQuel, such as **valid, when** and **as of**, are fully supported. The prototype also supports all four types of databases: *snapshot, rollback, historical* and *temporal.*

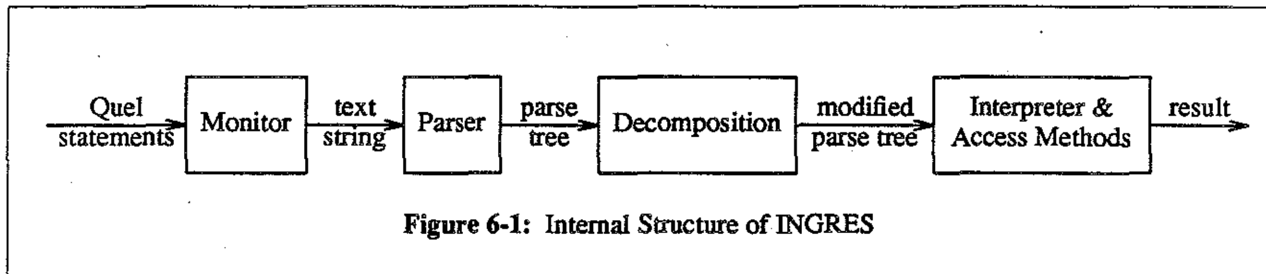


Figure 6-1: Internal Structure of INGRES

Figure 6-1 shows the internal structure of INGRES, and also the structure of the prototype. To handle temporal extensions in TQuel, the parser was modified so that it accepts TQuel statements and generates an *extended* syntax tree with extra subtrees for temporal clauses **valid** and **when.**

```
retrieve (h.id, i.id)
      valid from begin of (h overlap i)   to end of (h extend i)
      where h.id = 500  and   i.amount = 73700
      when  h overlap i
      as of "1981"
```

Figure 6-2: A TQuel Query

For example, a sample query in Figure 6-2 inquires the state of a database **as of** 1981. Retrieved tuples satisfy not only the **where** clause, but also the **when** clause specifying that the two tuples must have coexisted at some moment. The **valid** clause specifies the values of the time attributes *valid from* and *valid to* for result tuples.

The syntax tree for this query looks like Figure 6-3, where the left subtree denotes the target list, and the right subtree represents the predicates **when** and **where**. However, the prototype does not supply default values for the **valid** and **when** clauses if they are omitted in the retrieve statement.

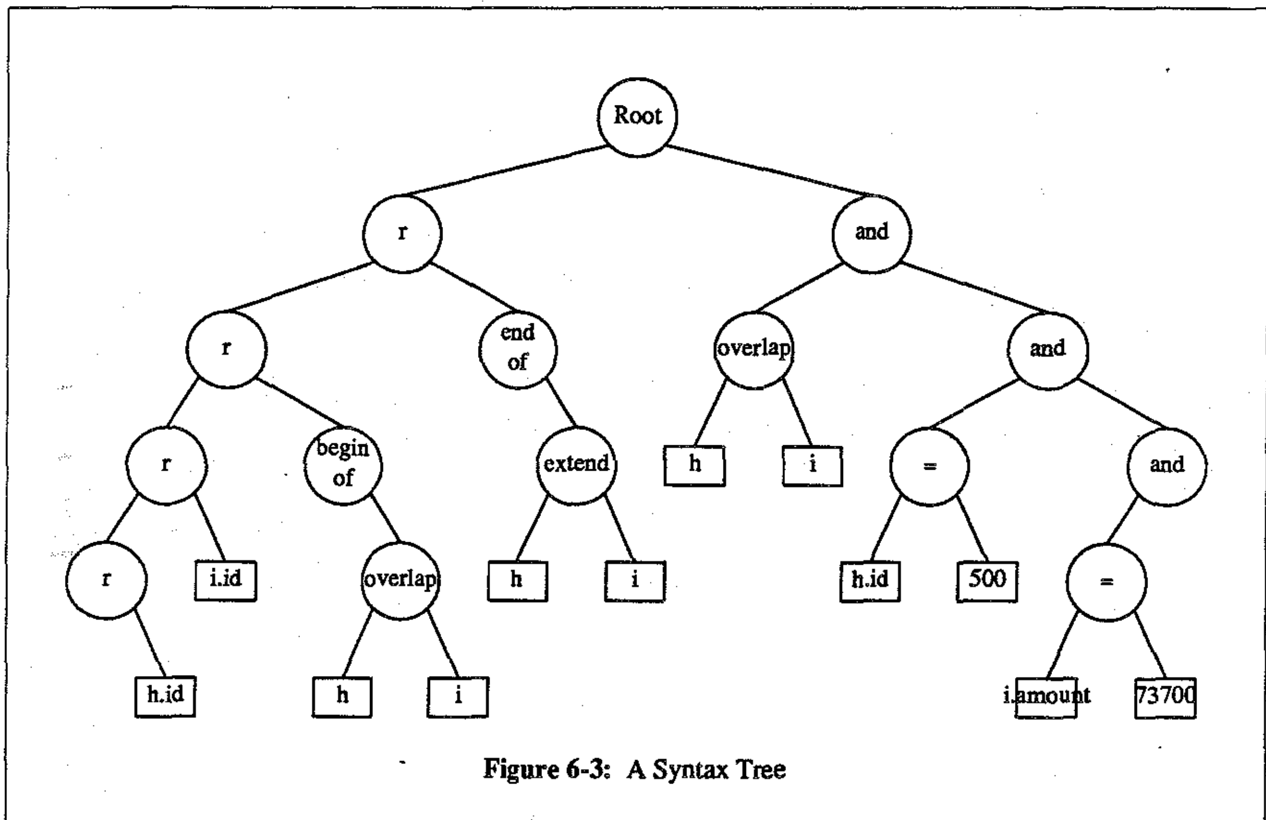


Figure 6-3: A Syntax Tree

The **retrieve** statement uses the clause **as of** t_1 or **as of** t_1 **through** t_2 to specify rollback operations for a rollback or a temporal database. The **as of** clause is not represented in the syntax tree, but sets the external variables `AsOf_start` and `AsOf_stop` to the value of t_1f and t_2 respectively. The default value for `AsOf_start` is the current time, and the default for `AsOf_stop` is the value of `AsOf_start`. These variables specify an interval on the axis of *transaction time* as illustrated in Figure 3-2 and 3-6, and select tuples overlapping with the interval when a rollback or a temporal relation is scanned to interpret the query.

As discussed in Section 5.1.2, data manipulation statements **append, delete** and **replace** use the **valid** clause to specify the update interval. If the **valid** clause is omitted, the prototype supplies the default **valid from** "now" **to** "forever". Though the formal semantics of the **append** statement defined in [Snodgrass 1986] requires to check if there already exists a tuple identical in the explicit attributes during the update interval, the prototype does not perform the integrity checking presently.

For the **delete** or **replace** statement, there are six different cases depending on the relationship between the *base interval* and the *update interval* (Figures 5-2 and 5-3). The prototype properly handles the six different cases for the *delete* statement, using the function of *snapshot* **replace** to update time attributes appropriately for all types of relations. The **replace** statement is performed following the *delete and insert* scheme, as described in Section 5.1.2.

TQuel does not use the **as of** clause in modification statements such as **append, delete**, and **replace**, but the prototype allowed the **as of** clause in those statements to specify values of the transaction start and the transaction stop attributes in creating synthetic relations to be used for a benchmark.

The **create** statement in TQuel specifies the type of a relation, whether snapshot, rollback, historical or temporal, and to distinguish between an interval and an event relation if the relation is historical or temporal. This information on the temporal type of a relation can be represented in three bits. The system relation was modified to store this information for each relation, and to perform appropriate actions depending on the type of a relation in all phases of query processing.

A temporal variable in TQuel can be associated with an *interval* or an *event* relation. An interval relation contains two implicit time attributes *valid from* and *valid to*, while an event relation has only the *valid from* attribute. To support temporal variables, the prototype added a new data type *i_time*, which consists of two time values for the attributes valid from and valid to. For an event variable, the value of the valid to field is set to the value of the valid from field.

Time attributes, whether explicit or implicit, are assigned a distinct data type, TIME_T. A time value is represented internally as a 32 bit integer with the resolution of one second, but externally as a character string. The prototype provides automatic conversion between the internal and the external

representations so that input and output operations can be performed in human readable form. For input, the prototype accepts various formats of character strings commonly used to represent date and time, and recognize values such as 'now', 'forever', and 'oo' (denoting 'forever'). For output, it can express time values with a resolution ranging from a second to a year, as selected by an option. The **copy** statement was also modified to perform input and output operations in batch for relations having time attributes, whether explicit or implicit, represented in various formats.

Some of the decomposition modules were changed to handle the temporal constructs and implicit time attributes. For example, it is necessary to include both time attributes valid from and valid to for a historical or a temporal relation during *one variable detachment* operation [Wong & Youssefi 1976], though only one may be specified in the query itself.

TQuel has temporal operators **begin of, end of, precede, overlap,** and **extend.** Functions to handle these operators were added in the *one variable query processing* portion of the interpreter. Temporal operators compose two types of temporal expressions, *temporal constructor* and the *temporal predicate.* The range of the temporal constructor is an interval of the i_time type, while the range of the temporal predicate is a boolean value. Instead of determining whether a temporal expression is of one type or the other, the prototype evaluates each expression for both cases, and uses the appropriate value depending on the semantics of the expression.

INGRES provides access methods such as heap, hashing, ISAM, and indexing. In this chapter, the prototype uses them without any modification. A new access method, reverse chaining discussed in Chapter 5, was added to the prototype, as will be described in the next chapter.

One of the most important decisions was how to embed a four-dimensional temporal relation into a two-dimensional snapshot relation as supported by INGRES. There are at least five such embeddings [Snodgrass 1986]. The prototype adopts the scheme of augmenting each tuple with two *transaction time* attributes for a rollback and a temporal relation, and one or two *valid time* attributes for a historical and a temporal relation depending on whether the relation models events or intervals.

For a rollback relation, an **append** operation inserts a tuple with the transaction start and the transaction stop attributes set to the current time and "forever" respectively. A **delete** operation on a tuple simply changes the transaction stop attribute to the current time. A **replace** operation first

executes a `delete` operation, then inserts a new version with the transaction start attribute set to the current time. A historical relation follows similar steps for `append`, `delete` and `replace` operations with the valid from and the valid to attributes as the counterparts of the transaction start and the transaction stop attributes. Values of the valid from and the valid to attributes are defaulted to the current time and "`forever`" respectively, but also can be specified by the `valid` clause.

For a temporal relation, an `append` operation inserts a tuple with the transaction start attribute of the current time, and the transaction stop attribute of "`forever`". Attributes valid from and valid to are set as specified by the `valid` clause, or defaulted if is is absent. A `delete` operation on a tuple sets the transaction stop attribute to the current time indicating that the tuple was virtually deleted from the relation. Next a new version with the updated valid to attribute is inserted indicating that the version has been valid until that time. A `replace` operation first executes a `delete` operation as above, then appends a new version marked with appropriate time attributes. Therefore, each `replace` operation in a temporal relation inserts two new versions. This scheme has a high overhead in terms of space, but captures the history of retroactive and proactive changes completely. In addition, all modification operations for rollback and temporal relations in this scheme are *append only*, so write-once optical disks can be utilized. A more detailed discussions of these operations can be found elsewhere [Snodgrass 1986].

The prototype was constructed in about 3 person-months over a period of a year; this figure does not include familiarization with the INGRES internals or with TQuel. About half the changes were modifications, and the rest were additions. The source was increased by 2,900 lines, or about 4.9% of INGRES version 7.10, which is approximately 58,800 lines long.

6.2. Benchmarking the Prototype

We define the *update count* for a tuple as the number of update operations on the tuple, and the *average update count* for a relation as the average of the update counts over all tuples in the relation. We hypothesized that, as the average update count increases, the performance of the prototype with conventional access methods would deteriorate rapidly not only for temporal queries but also for non-temporal ones. We postulated that major factors to affect the performance of a temporal DBMS were the type of a database, the query type, the access methods, the loading factor, and the update count.

A benchmark was run to confirm these hypotheses in various situations, and to determine the rate of performance degradation as the average update count increased. This section describes the details of the benchmark, presents its results, and analyzes the performance data from the benchmark.

6.2.1. A Benchmark

We wanted to compare the performance of the four types of databases described in Chapter 3. For each of the four types, we created two databases, one with a 100% loading factor and the other with a 50% loading factor. As the sample commands for a temporal database in Figure 6-4 show, each database contains two relations, *Type*_h and *Type*_i, where *Type* is one of `Snapshot`, `Rollback`, `Historical`, and `Temporal`.

```
create persistent interval Temporal_h
       (id = i4, amount = i4, seq = i4, string = c96)
modify       Temporal_h to hash on id      where fillfactor = 100

create persistent interval Temporal_i
       (id = i4, amount = i4, seq = i4, string = c96)
modify       Temporal_i to isam on id      where fillfactor = 100
```

Figure 6-4: Creating a Temporal Database

*Type*_h is stored in a hashed file, and *Type*_i is stored in an ISAM file. The loading factor of a file is specified with the **`fillfactor`** parameter in a **`modify`** statement [Woodfill et al. 1981].

Each tuple has 108 bytes of data in four attributes: `id`, `amount`, `seq` and `string`. `Id`, a four byte integer, is the key in both relations. The attributes `Amount` and `string` are randomly generated as integers and strings respectively, and the `seq` attribute is initialized as zero. In addition, rollback and historical relations carry two time attributes, while temporal relations contain four time attributes. The *transaction start* and the *valid from* attributes are randomly initialized to values between Jan. 1 and Feb. 15 in 1980, with the *transaction stop* and the *valid to* attributes set to '`forever`' indicating that they are the current versions. The evolution of these relations will be described shortly.

Each relation is initialized to have 1024 tuples using a **`copy`** statement. The block size in the prototype is 1024 bytes. With 100% loading, there are 9 tuples per block for snapshot relations, and 8 tuples per block for rollback, historical, or temporal relations. Therefore, we need at least 114 blocks for

each snapshot relation, and 128 blocks for each of the others. The actual size depends on the database type, the access method, the loading factor, and the average update count.

```
      range of h  is temporal_h           /* hashed on id */
      range of i  is temporal_i           /* ISAM   on id */

Q01 : retrieve (h.id, h.seq)  where h.id = 500
Q02 : retrieve (i.id, i.seq)  where i.id = 500

Q03 : retrieve (h.id, h.seq)  as of "08:00 1/1/80"
Q04 : retrieve (i.id, i.seq)  as of "08:00 1/1/80"

Q05 : retrieve (h.id, h.seq)  where h.id = 500
            when  h overlap "now"
Q06 : retrieve (i.id, i.seq)  where i.id = 500
            when  i overlap "now"

Q07 : retrieve (h.id, h.seq)  where h.amount = 69400
            when  h overlap "now"
Q08 : retrieve (i.id, i.seq)  where i.amount = 73700
            when  i overlap "now"

Q09 : retrieve (h.id, i.id, i.amount)     where h.id = i.amount
            when  h overlap i and   i overlap "now"
Q10 : retrieve (i.id, h.id, h.amount)     where i.id = h.amount
            when  h overlap i and   h overlap "now"

Q11 : retrieve (h.id, h.seq, i.id, i.seq, i.amount)
            valid from begin of h         to end of i
            when  begin of h precede i
            as of "4:00 1/1/80"
Q12 : retrieve (h.id, h.seq, i.id, i.seq, i.amount)
            valid from begin of (h overlap i)   to end of (h extend i)
            where h.id = 500  and   i.amount = 73700
            when  h overlap i
            as of "now"

Q13 : retrieve (h.id, h.seq)  where h.id = 455
            when  "1/1/82" precede  end of h
Q14 : retrieve (h.id, h.seq)  where h.amount = 10300
            when  "1/1/82" precede  end of h

Q15 : retrieve (h.id, h.seq)  where h.amount = 10300
            as of "1/1/83"
Q16 : retrieve (h.id, h.seq)  where h.amount = 10300
            when  "1/1/82" precede  end of h
            as of "1/1/83"
```

Figure 6-5: Benchmark Queries

Sixteen sample queries with varying characteristics comprise the benchmark as shown in Figure 6-5. These queries were chosen in an attempt to exercise the access methods available in INGRES, to isolate the effects of various TQuel clauses, and to demonstrate the possibility of performance enhancement. The number of output tuples were kept constant regardless of the update count, except for queries Q01, Q02 and Q12.

Q01 retrieves all versions of a tuple (*version scan*) from a hashed file given a key. Q03 is a *rollback* query, applicable only to rollback and temporal databases, retrieving the state of a relation as of some moment in the past. Q05 retrieves the most recent version from a hashed file given a key, while Q07 retrieves the most recent version from a hashed file through a non-key attribute, requiring a sequential scanning of the whole file. Queries Q02, Q04, Q06 and Q08 are counterparts of Q01, Q03, Q05, and Q07 respectively, where the even numbered queries access an ISAM file and the odd numbered access a hashed file. Both Q09 and Q10 join current versions of two relations; Q09 goes through the primary access path of a hashed file and Q10 goes through an ISAM file.

Queries Q05 through Q10 all refer to only the most recent versions. They are termed *non-temporal queries* in the sense that they retrieve the current state of a database as if from a snapshot database. For a snapshot database, the **when** clause in these queries are neither necessary nor applicable. For a rollback database, we use the **as of** clause instead of the **when** clause. For example, **when** x **overlap** "now" will become **as of** "now".

Q11 is a query involving a *temporal join*, a join of two tuples based on temporal information. In this query, the **as of** clause specifies the *rollback* operation shifting the reference point to a past moment. The **when** clause specifies a temporal relationship between two versions, where the value of the valid from attribute in the version from *Type*_h relation is earlier than the corresponding value in the version from *Type*_i relation. The **valid** clause specifies that the transaction start attribute of the result tuple be set to the value of the transaction start attribute in the version from *Type*_h relation, and that the transaction stop attribute of the result tuple be set to the corresponding value in the version from *Type*_i relation. Q12 contains all types of clauses in TQuel, inquiring the state of a database as of 'now' given both temporal and non-temporal constraints. Obviously, Q11 and Q12 are relevant only for a temporal database.

Queries Q13 through Q16 exercise various combinations of the **when** and the **as of** clauses. Query Q13 retrieves tuples whose *valid to* value is later than "1/1/82" through the hashed key. Query Q14 also retrieves tuples whose *valid to* value is later than "1/1/82", but through a non-key attribute. Query Q15 retrieves tuples from the `Temporal_h` relation *as of* "1/1/83" through a non-key attribute. Query Q16 is similar to Q15, but also requires that the *valid to* value is later than "1/1/82".

These sixteen queries were run on each of eight test databases as described earlier; two databases, with the loading factor of 100% and 50% respectively, for each of `Snapshot`, `Rollback`, `Historical`, and `Temporal`. We focused solely on the number of disk accesses per query at a granularity of a block, as this metric is highly correlated with both CPU time and response time. There are a few pitfalls to be avoided with this metric. Disk accesses to system relations are relatively independent of the database type or the characteristics of queries, but more dependent on how a particular DBMS manages system relations. Also, the number of disk accesses varies greatly depending on the number of internal buffers and the algorithm for buffer management. To eliminate such variables, which are outside the scope of this research, we counted only disk accesses to user relations, and allocated only 1 buffer for each user relation so that a block resides in main memory only until another block from the same relation is brought in.

Once performance statistics were collected for all the sample queries, we simulated the uniformly distributed evolution of each database by incrementing the value of `seq` attribute in each of the current versions. The time attributes were appropriately changed for this **replace** operation using the default of **valid from** `"now"` **to** `"forever"` as described in Section 4. Thus a new version (two new versions for temporal relations) of each tuple is inserted, and the average *update count* of the database is incremented by one. Performance on the sample queries were measured after determining the size of each relation appended with new versions. This process was repeated until the average update count reached 15, which we believed high enough to show the relationship between the growth of I/O cost and the average update count. The benchmark was run on a Vax 11/780, consuming approximately 20 hours of CPU time.

6.2.2. Performance Data

Space requirements for various databases were measured as the average update count ranged from 0 to 15. Figure 6-6 shows the data for the average update count of 0 and 14 along with the *growth per update*. The table also shows the *growth rate*, obtained when dividing the growth per update by the size for the update count of 0. These data were useful for analyzing the I/O costs measured in the benchmark.

Type	Snapshot				Rollback				Historical				Temporal			
Loading	100 %		50 %		100 %		50 %		100 %		50 %		100 %		50 %	
Relation	H	I	H	I	H	I	H	I	H	I	H	I	H	I	H	I
Size, UC= 0	166	115	257	259	129	129	257	259	129	129	257	259	129	129	257	259
Size, UC=14	–	–	–	–	1927	1921	2048	2051	1927	1921	2048	2051	3717	3713	3839	3843
Growth per Update	–	–	–	–	128.4	128.0	127.9	128.0	128.4	128.0	127.9	128.0	256.3	256.0	255.9	256.0
Growth Rate	–	–	–	–	1	1	0.5	0.5	1	1	0.5	0.5	1.99	2	1	1

Notes :

Relation H is a hashed file. 'UC' denotes *Update Count.*
Relation I is an ISAM file. '–' denotes *not applicable.*

Figure 6-6: Space Requirements (in Blocks)

From this table, we find that:

- The rollback and the historical databases have the same space requirements.
- The temporal database consumes the same amount of space as the rollback and the historical databases for the update count of 0.
- The temporal database, following the embedding scheme described in Section 6.1, requires almost twice the additional blocks as the update count increases.
- The growth per update for a hashed file varies slightly due to key collisions in hashing.

Input costs for the sample queries on each database were measured as the average update count increased from 0 to 15. Some queries also incurred output costs, which accounted for creating temporary relations to store intermediate results. For example, queries Q09 and Q10 wrote out 56 blocks each, and

Q12 on the historical or the temporal database wrote 4 blocks. Output costs were constant for these queries regardless of the update count, because the size of temporary relations were kept the same for the sample queries. Since the output costs are negligible compared with the input costs, we concentrate on the analysis of the input costs. Appendix C shows the measurement data from the benchmark for the rollback, historical, and temporal databases, each with 100% and 50% loading. Figure 6-7 shows the input costs for the temporal database with 100% loading.

Update Count	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Q01	1	3	5	7	9	11	13	15	17	19	21	23	25	27	29	31
Q02	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32
Q03	129	387	645	903	1153	1411	1669	1927	2177	2435	2693	2951	3201	3459	3717	3975
Q04	128	384	640	896	1152	1408	1664	1920	2176	2432	2688	2944	3200	3456	3712	3968
Q05	1	3	5	7	9	11	13	15	17	19	21	23	25	27	29	31
Q06	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32
Q07	129	387	645	903	1153	1411	1669	1927	2177	2435	2693	2951	3201	3459	3717	3975
Q08	128	384	640	896	1152	1408	1664	1920	2176	2432	2688	2944	3200	3456	3712	3968
Q09	1200	3512	5816	8120	10386	12690	14994	17298	19564	21868	24172	26476	28742	31046	33350	35654
Q10	2233	4539	6845	9151	11449	13755	16061	18367	20665	22971	25277	27583	29881	32187	34493	36799
Q11	385	1155	1925	2695	3457	4227	4997	5767	6529	7299	8069	8839	9601	10371	11141	11911
Q12	131	389	647	905	1163	1421	1679	1937	2195	2453	2711	2969	3227	3485	3743	4001
Q13	1	3	5	7	9	11	13	15	17	19	21	23	25	27	29	31
Q14	129	387	645	903	1153	1411	1669	1927	2177	2435	2693	2951	3201	3459	3717	3975
Q15	129	387	645	903	1153	1411	1669	1927	2177	2435	2693	2951	3201	3459	3717	3975
Q16	129	387	645	903	1153	1411	1669	1927	2177	2435	2693	2951	3201	3459	3717	3975

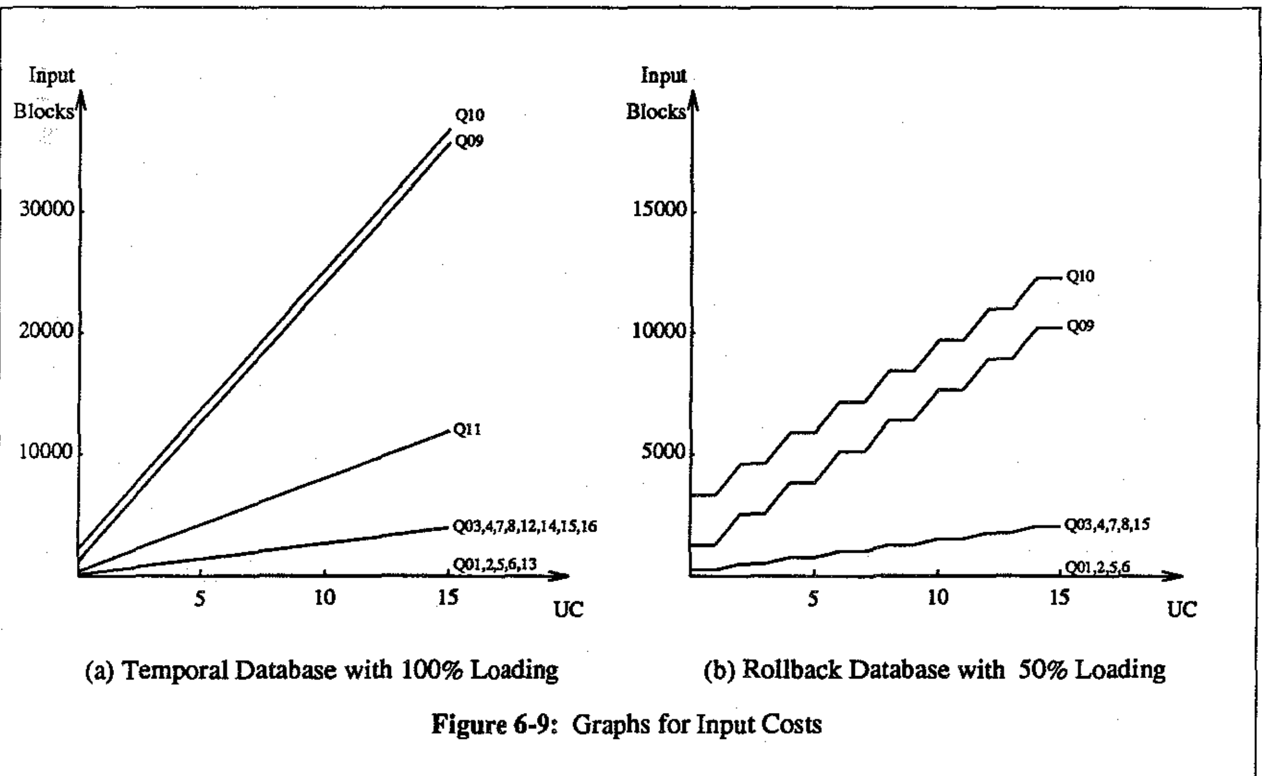
Figure 6-7: Input Costs for the Temporal Database with 100% Loading

Similar tables, a total of 8, were obtained for each database of different types and loading factors. We summarize the input costs for the sample queries on various databases with the average update count of 0 and 14 in Figure 6-8.

Figure 6-8 shows that the rollback and the historical databases exhibit similar performance, while the temporal database is about twice more expensive than rollback and historical databases for the update count of 14. If we draw a graph for the input costs shown in Figure 6-7, we get Figure 6-9 (a). Figure 6-9 (b) is a similar graph for the rollback database with 50% loading, showing jagged lines caused by the odd numbered updates filling the space left over by the previous updates before adding overflow blocks.

Type	Snapshot		Rollback				Historical				Temporal			
Loading	100 %	50 %	100 %		50 %		100 %		50 %		100 %		50 %	
Query	U C	U C	U C		U C		U C		U C		U C		U C	
	0	0	0	14	0	14	0	14	0	14	0	14	0	14
Q01	2	1	1	15	1	8	1	15	1	8	1	29	1	15
Q02	2	3	2	16	3	10	2	16	3	10	2	30	3	17
Q03	–	–	129	1927	257	2048	–	–	–	–	129	3717	257	3839
Q04	–	–	128	1920	256	2048	–	–	–	–	128	3712	256	3840
Q05	2	1	1	15	1	8	1	15	1	8	1	29	1	15
Q06	2	3	2	16	3	10	2	16	3	10	2	30	3	17
Q07	166	257	129	1927	257	2048	129	1927	257	2048	129	3717	257	3839
Q08	114	256	128	1920	256	2048	128	1920	256	2048	128	3712	256	3840
Q09	1585	1276	1141	17242	1271	10240	1197	17298	1327	10296	1200	33350	1333	19256
Q10	2214	3329	2177	18311	3329	12288	2233	18367	3385	12344	2233	34493	3385	21303
Q11	–	–	–	–	–	–	–	–	–	–	385	11141	769	11519
Q12	–	–	–	–	–	–	–	–	–	–	131	3743	259	3857
Q13	–	–	–	–	–	–	1	15	1	8	1	29	1	15
Q14	–	–	–	–	–	–	129	1927	257	2048	129	3717	257	3839
Q15	–	–	129	1927	257	2048	–	–	–	–	129	3717	257	3839
Q16	–	–	–	–	–	–	–	–	–	–	129	3717	257	3839

Figure 6-8: Input Costs for Four Types of Databases



(a) Temporal Database with 100% Loading

(b) Rollback Database with 50% Loading

Figure 6-9: Graphs for Input Costs

6.2.3. Analysis of Performance Data

The graphs in Figure 6-9 show that input costs increase almost linearly with the update count, but with varying slopes for different queries. A question is whether there are any particular relationships independent of query types between the input cost and the average update count, and between the input cost and the database type. To answer this question, we now analyze how each sample query is processed, and identify the dominant operations which can characterize each query.

Though queries Q01 and Q05 are functionally different from each other, one being the *version scanning* and the other a *non-temporal query*, the prototype built with conventional access methods uses the same mechanism to process them. Both queries are evaluated by accessing a hashed file given a key *(hashed access)*. Likewise, Q02 and Q06 requires the access to an ISAM file given a key *(ISAM access)*. Queries Q03, Q04, Q07 and Q08 all need to scan a file, whether hashed or ISAM *(sequential scanning)*.

Processing Q09 first scans an ISAM file sequentially doing selection and projection into a temporary relation *(one variable detachment)*. It then performs one hashed access for each of 1024 tuples in the temporary relation *(tuple substitution)*. Here the dominant operation is the hashed access, repeated 1024 times. Q10 is similar to Q09 except that the roles of the hashed file and the ISAM file are reversed. Hence the dominant operation for Q10 is the ISAM access.

Q11 is evaluated by sequentially scanning one file to find versions satisfying the **as of** clause. For such a version, the other file is sequentially scanned for versions satisfying both the **as of** clause and the **when** clause. Here the dominant operation is the sequential scanning. Processing Q12 requires a sequential scanning and a hashed access to find versions satisfying the **where** clause, then joins them on time attributes according to the **when** clause. Since the number of versions extracted for the join is small enough to fit into one block each, the dominant operation is the sequential scanning.

Query Q13 is similar to queries Q01 and Q05 in that it retrieves a record through the hashed key. Queries Q14 through Q16 are similar to query Q07, which requires sequential scanning, though they have different temporal predicates.

From this analysis, we can divide the input cost into the *fixed* cost and the *variable* cost. The fixed cost is the portion which stays the same regardless of the update count. It accounts for traversing the

directory in the ISAM, or for creating and accessing a temporary relation whose size is independent of the update count. On the other hand, the variable cost is the portion which increases with the update count. It is the result of subtracting the fixed cost from the cost of a query on a database with no update. Operations contributing to the variable cost will grow more expensive as the number of updates on the relation increases.

Now we can define the *growth rate* of the input cost on a database with the update count of *n* as:

$$Growth\ Rate_n = \frac{C_n - C_0}{(variable\ cost) \times n}$$

where

C_n = *input cost for update count of n*

C_0 = *input cost for update count of 0*

The growth rate is the key aspect of an implementation, characterizing the performance degradation as the update count increases. Clearly the ideal would be a growth rate close to 0.

Fixed costs, variable costs and growth rates for the sample queries on various types of databases were calculated. The growth rate was relatively independent of the update count *n*, as suggested by the linearity of cost curves shown in Figure 6-9. Figure 6-10 shows fixed costs, variable costs, and growth rates for the sample queries on the rollback and the temporal databases with the loading factor of 100% and 50% each. The historical database shows the same variable costs and the growth rates as the rollback database, except for Q03, Q04, and Q15 which are not applicable to historical databases. But its fixed costs are the same as the temporal database, except for Q03, Q04, Q11, Q12, Q15, and Q16 which are not applicable.

Type	Rollback						Temporal					
Loading	100 %			50 %			100 %			50 %		
Query	Cost (in Blocks)		Growth	Cost (in Blocks)		Growth	Cost (in Blocks)		Growth	Cost (in Blocks)		Growth
	Fixed	Variable	Rate	Fixed	Variable	Rate	Fixed	Variable	Rate	Fixed	Variable	Rate
Q01	0	1	1	0	1	0.5	0	1	2	0	1	1
Q02	1	1	1	2	1	0.5	1	1	2	2	1	1
Q03	0	129	1	0	257	0.5	0	129	1.99	0	257	1
Q04	0	128	1	0	256	0.5	0	128	2	0	256	1
Q05	0	1	1	0	1	0.5	0	1	2	0	1	1
Q06	1	1	1	2	1	0.5	1	1	2	2	1	1
Q07	0	129	1	0	257	0.5	0	129	1.99	0	257	1
Q08	0	128	1	0	256	0.5	0	128	2	0	256	1
Q09	0	1141	1.01	0	1271	0.5	56	1144	2.01	56	1277	1
Q10	1024	1153	1	2048	1281	0.5	1080	1153	2	2104	1281	1
Q11	–	–	–	–	–	–	0	385	2	0	769	1
Q12	–	–	–	–	–	–	2	129	2	2	257	1
Q13	–	–	–	–	–	–	0	1	2	0	1	1
Q14	–	–	–	–	–	–	0	129	1.99	0	257	1
Q15	0	129	1	0	257	0.5	0	129	1.99	0	257	1
Q16	–	–	–	–	–	–	0	129	1.99	0	257	1

Note :
'–' denotes *not applicable.*

Figure 6-10: Fixed Costs, Variable Costs, and Growth Rates

Rather surprisingly, the growth rate turned out to be independent of the query type and the access method as far as access methods of sequential scanning, hashing or ISAM are concerned. It was, however, highly dependent on the database type and the loading factor. For example, the growth rates for operations such as sequential scanning, hashed access, and access of data blocks in ISAM are all 2.0 in case of the temporal database with 100% loading. On the other hand, the growth rates for similar operations are approximately 0.5 in case of the rollback or the historical database with 50% loading.

From these analyses, we can make several observations as far as access methods of sequential scanning, hashing or ISAM are concerned.

- The fixed and the variable costs are dependent on the query type, the access method and the loading factor, but relatively independent of the database type.
- The growth rate is approximately equal to the loading factor of relations for rollback or historical databases.
- The growth rate of input cost is approximately twice the loading factor of relations for temporal databases.

- The growth rate is independent of the query type and the access method.

The fact that the growth rate can be determined given the database type and the loading factor without regard to the query type or the access method has a useful consequence. From the definition of the growth rate, we can derive the following formula for the cost of a query when the update count is n.

$$\begin{aligned} C_n &= C_0 + (growth\ rate) \times (variable\ cost) \times n \\ &= (fixed\ cost) + (variable\ cost) + (growth\ rate) \times (variable\ cost) \times n \\ &= (fixed\ cost) + (variable\ cost) \times [1 + (growth\ rate) \times n\ ] \end{aligned}$$

Therefore, when the cost of a query on a database with the update count of 0 is known and its fixed portion is identified, it is possible to predict future performance of the query on the database when the update count grows to n. Note that the fixed cost, and hence the variable cost, can even be counted automatically by the system, except when the size of a temporary relation varies greatly depending on the update count.

6.2.4. Non-uniform Distribution

Thus far, we have assumed uniform distribution of updates where each tuple will be updated an equal number of times as the average update count increases. Since the assumption of uniform distribution may appear rather unrealistic, we also ran an experiment with a non-uniform distribution. To simulate a maximum variance case, only 1 tuple was updated repeatedly to reach a certain average update count. We measured performance of queries on the updated tuple and on any of remaining tuples, then averaged the results weighted by the number of such tuples. Since it takes $O(n^2)$ block accesses to update a single tuple for n times, owing to the overflow chain ever lengthening, we repeated the process only up to the update count of 4, which was good enough to confirm our subsequent analysis.

Performance of a query is highly dependent upon whether the tuple participating in the query has an overflow chain. We hypothesized that updating tuples with a high variance would affect the growth rate significantly, owing to the presence of long overflow chains for some tuples and the absence of such chains for others. However, the growth rate averaged over all tuples turned out to remain the same as the uniform distribution case. For example, if we update one tuple in a temporal relation 1024 times, the average update count becomes one. For a query like Q01, a hashed access to any tuple sharing the same block as the changed tuple costs 257 block accesses, while a hashed access to any tuple residing on a block without

an overflow costs just one block access. Therefore, the average cost becomes three block accesses, the same as the uniform distribution case.

We can extend this result to a more general case. If the number of primary blocks is x with 100% loading, there will be approximately $2x$ overflow blocks for the average update count of one in a temporal relation. Let y be the number of primary blocks which have overflow blocks, and z be the number of primary blocks which do not have an overflow, then $y+z=x$. Since the average length of overflow chains is $\frac{2x}{y}$ blocks, the average cost of a hashed access to such a relation will be:

$$\frac{y \times (\frac{2x}{y} + 1) + z \times 1}{y+z} = \frac{y}{x} \times \frac{2x}{y} + \frac{y+z}{x} = 3$$

showing the same result as the more restricted case discussed above.

This reasoning can be generalized for other database types, access methods, loading factors, query types, and update counts in a similar fashion. Now one more observation about the growth rate can be added:

- The growth rate is independent of the distribution of updated tuples.

We conclude that the results from the benchmark we ran under the assumption of uniform distribution are still valid for any other distribution.

6.3. Analysis from Models

The sample queries in Figure 6-5 were also analyzed using the four models discussed in Chapter 4. Full description of the analysis results is given in Appendix D, and Figure 6-11 shows the summary of the input costs for each type of databases with the update count of 0 and 14.

Type	Snapshot		Rollback				Historical				Temporal			
Loading	100 %	50 %	100 %		50 %		100 %		50 %		100 %		50 %	
Query	UC	UC	UC		UC		UC		UC		UC		UC	
	0	0	0	14	0	14	0	14	0	14	0	14	0	14
Q01	1	1	1	15	1	8	1	15	1	8	1	29	1	15
Q02	2	3	2	16	3	10	2	16	3	10	2	30	3	17
Q03	–	–	128	1920	256	2048	–	–	–	–	128	3712	256	3840
Q04	–	–	128	1920	256	2048	–	–	–	–	128	3712	256	3840
Q05	1	1	1	15	1	8	1	15	1	8	1	29	1	15
Q06	2	3	2	16	3	10	2	16	3	10	2	30	3	17
Q07	114	228	128	1920	256	2048	128	1920	256	2048	128	3712	256	3840
Q08	114	228	128	1920	256	2048	128	1920	256	2048	128	3712	256	3840
Q09	1194	1308	1152	17280	1280	10240	1208	17336	1336	10296	1208	33464	1336	19256
Q10	2218	3356	2176	18304	3328	12288	2232	18360	3384	12344	2232	34488	3384	21304
Q11	–	–	–	–	–	–	–	–	–	–	384	11136	768	11520
Q12	–	–	–	–	–	–	–	–	–	–	131	3743	259	3857
Q13	–	–	–	–	–	–	1	15	1	8	1	29	1	15
Q14	–	–	–	–	–	–	128	1920	256	2048	128	3712	256	3840
Q15	–	–	128	1920	256	2048	–	–	–	–	128	3712	256	3840
Q16	–	–	–	–	–	–	–	–	–	–	128	3712	256	3840

Notes :

'UC' denotes *Update Count.* '–' denotes *not applicable.*

Figure 6-11: Analysis Results using Performance Models

To compare the analysis results (Figure 6-11) with the measurement data from the benchmark (Figure 6-8), we calculate the *error rate* as:

$$Error\ Rate = \frac{a - b}{b} \times 100\ \%$$

where

a = cost estimated from the analysis

b = cost measured from the benchmark

Figure 6-12 shows the error rate for each data point. It shows that error rates are generally within about 1% for the rollback, historical, and temporal databases. Interestingly, the biggest errors are found for the snapshot database. The reason is that a snapshot relation with 100% loading can hold 9 tuples, compared with 8 tuples for other types of relations, but the larger number of tuples per block caused extra key collisions due to imperfect nature of the hash function used for hashing. For example, a snapshot relation which can hold 9 tuples per block consumed 166 blocks for 1024 tuples, not 114 tuples as expected

for a perfect hashing [Sprugnoli 1977]. As a result, query Q07 costs 166 blocks accesses to scan a hashed relation, and query Q01 costs two block accesses, not one as expected for hashing, to retrieve a tuple through a hashed key. The unpredictability of key collisions is less visible for other types of relations, which hold a smaller number of tuples per block to incorporate time attributes, but it still contributes to discrepancies between the analysis results and the measurement data.

Type	Snapshot		Rollback				Historical				Temporal			
Loading	100 %	50 %	100 %		50 %		100 %		50 %		100 %		50 %	
Query	UC 0	UC 0	UC 0	14	UC 0	14	UC 0	14	UC 0	14	UC 0	14	UC 0	14
Q01	-50	0	0	0	0	0	0	0	0	0	0	0	0	0
Q02	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Q03	–	–	-1.	-0.	-0.	0	–	–	–	–	-1.	-0.	-0.	+0.
Q04	–	–	0	0	0	0	–	–	–	–	0	0	0	0
Q05	-50	0	0	0	0	0	0	0	0	0	0	0	0	0
Q06	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Q07	-31.	-11.	-1.	-0.	-0.	0	-1.	-0.	-0.	0	-1.	-0.	-0.	+0.
Q08	0	-11.	0	0	0	0	0	0	0	0	0	0	0	0
Q09	-25.	+3.	+1.	+0.	+1.	0	+1.	+0.	+1.	0	+1.	+0.	+0.	0
Q10	-0.	+1.	-0.	-0.	-0.	0	-0.	-0.	-0.	0	-0.	-0.	-0.	+0.
Q11	–	–	–	–	–	–	–	–	–	–	-0.	-0.	-0.	+0.
Q12	–	–	–	–	–	–	–	–	–	–	0	0	0	0
Q13	–	–	–	–	–	–	0	0	0	0	0	0	0	0
Q14	–	–	–	–	–	–	-1.	-0.	-0.	0	-1.	-0.	-0.	+0.
Q15	–	–	-1.	-0.	-0.	0	–	–	–	–	-1.	-0.	-0.	+0.
Q16	–	–	–	–	–	–	–	–	–	–	-1.	-0.	-0.	+0.

Notes :

'UC' denotes *Update Count.*
'0' denotes the *true zero.*
'-0.' denotes a *small negative fraction.*
'–' denotes *not applicable.*
'+0.' denotes a *small positive fraction.*

Figure 6-12: Error Rates in the Analysis Results

We also measured the elapsed time to process the sample queries on the prototype. Figure 6-13 compares the measurement data with the time estimated from the analysis described in Appendix D. This table shows that the differences between the measurement and the estimation is mostly 10 to 20%. There are many factors to affect the elapsed time to process a query, other than input and output costs. Examples are the CPU speed, machine load, scheduling policy, buffer management algorithms, *etc.* Though we analyzed only input and output costs in this research, we could still estimate the elapsed time rather closely.

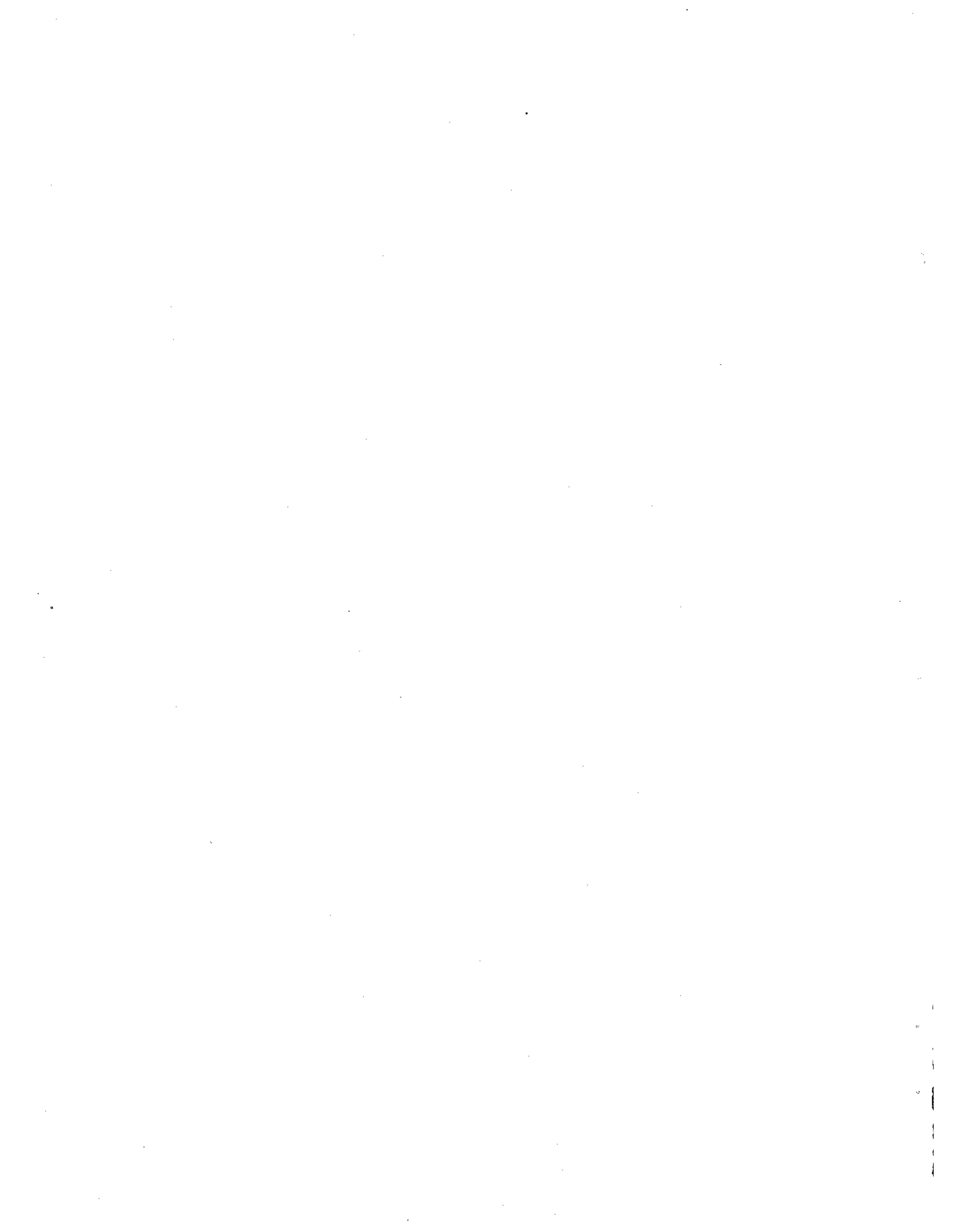
Query	Update Count = 0		Update Count = 14	
	Measured	Estimated	Measured	Estimated
Q09	44.8	36.5	1277	1001
Q10	61.2	68.5	1187	1031
Q11	7.8	7.1	140	205
Q12	4.0	2.5	62	69.2

Figure 6-13: Elapsed Time (in *sec*)

6.4. Summary

A prototype of a temporal database management system was built by extending the snapshot DBMS INGRES. It supports the temporal query language TQuel, a superset of Quel, and handles all four types of databases: snapshot, rollback, historical and temporal. A benchmark with sixteen sample queries was run to study the performance of the prototype on the four types of databases with two loading factors. We analyzed the results of the benchmark, determined the fixed cost and the variable cost for each query, and identified major factors that have the greatest impact on the performance of the system. We also found that the growth rate can be determined by the database type and the loading factor, regardless of the query type, the access method, or even the distribution of updated tuples, as far as the access methods of sequential scanning, hashing or ISAM are concerned. A formula was obtained to estimate the cost of a query on a database with multiple temporal versions, when the cost of a query on the database with a single version is known and its fixed portion is identified.

Input and output costs of the sample queries were also analyzed using the four models discussed in Chapter 4. Estimated costs from the analysis were compared with the measurement data from the benchmark, which showed that the cost of a query in terms of block accesses can be estimated quite accurately (generally within about 1 %) using the four models. The elapsed time to process a query, estimated using the models, was within about 10 to 20% of the measurement data.



Chapter 7

Temporally Partitioned Store

As the results of the benchmark discussed in Chapter 6 indicate, sequential scanning is expensive. Access methods such as hashing and ISAM also suffer from rapid performance degradation due to ever-growing overflow chains. Reorganization does not help to shorten overflow chains, because all versions of a version set share the same key.

A lower loading factor results in a lower growth rate, by reducing the number of overflow blocks in hashing and ISAM. Hence better performance is achieved with a lower loading factor when the update count is high. But there is an overhead for maintaining a lower loading factor both in space and performance when the update count is low. A lower loading factor requires more space for primary blocks. Scanning such a file sequentially (*e.g.* for query Q07 or Q08 in Chapter 6) is more expensive than scanning a file with a higher loading factor. For ISAM, a lower loading factor requires more directory blocks, which may increase the height of the directory. As shown in Figure 6-8 of the previous chapter, for example, query Q10 on the temporal database with the update count of 0 reads in 3385 blocks for 50% loading, significantly higher than 2233 blocks for 100% loading.

We conclude that access methods such as sequential scanning, hashing, or ISAM are not suitable for a database with temporal support. There are other 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], but they also have various problems as indicated in Section 1.2.2. Therefore, new access methods tailored to the particular characteristics of database management systems with temporal support need to be developed to provide fast response for a wide range of temporal queries without penalizing conventional non-temporal queries.

Our solution is the *temporally partitioned* storage structure discussed in Chapter 5, with various formats for the history store, such as *reverse chaining, accession lists, indexing, clustering, stacking,* and

cellular chaining. This chapter describes how the temporally partitioned storage structure was implemented into the prototype, and discusses the performance improvement achieved for the prototype by using the various methods developed in Chapter 5. Issues on secondary indexing will also be discussed.

7.1. Implementation of the Temporally Partitioned Store

The prototype described in Section 6.1 supported TQuel and all four types of databases, yet used the conventional access methods available in INGRES. To improve its performance, *reverse chaining*, among the temporally partitioned storage structures, was subsequently added to the prototype.

The default storage format of a relation in INGRES, and hence in the prototype, is a *heap*. The `modify` statement in Quel converts the storage structure of a relation from one format to another. Major storage options available in INGRES are:

`heap` : for a sequential file

`hash` : for a hashed file

`isam` : for an ISAM file

For example, a statement in Figure 6-4

```
modify      Temporal_h to hash on id      where fillfactor = 100
```

converted the `Temporal_h` relation to a hashed file with the loading factor of 100%.

New options were added to the `modify` statement to specify the format of the *history store* for the temporally partitioned storage structure. They are:

`chain` : for *reverse chaining*

`accessionlist` : for *accession lists*

`index` : for *indexing*

`cluster` : for *clustering*

`stack` : for *stacking*

`cellular` : for *cellular chaining*

For example, the statement

```
modify      Temporal_h to chain on id
```

changes the `Temporal_h` relation to the temporally partitioned store, if it is not already in such a structure. The history store uses *reverse chaining* with the `id` attribute as the key, while the current store maintains the previous format.

Though the **fillfactor** parameter is not relevant for the history store considered here, some formats require additional parameters. *Accession lists* and *indexing* have the parameter **time** to specify the amount of temporal information to be maintained in accession lists or index entries. Allowed values for the **time** parameter are **all** to maintain information on all the time attributes, or a list of time attributes such as **valid from, valid to, transaction start,** and **transaction stop**. For example, we use the following statement to change the history store to the format of accession lists with all the time attributes:

```
modify      Temporal_h to accessionlist on id   where time = (all)
```

Stacking and *cellular chaining* have the parameter **cellsize** to specify the stacking depth or the size of a cell. To change the history store to the format of cellular chaining with up to four tuples in each cell, we use the statement:

```
modify      Temporal_h to cellular on id  where cellsize = 4
```

Issuing another **modify** statement with one of the options **heap, hash**, or **isam** will change the format of the current store accordingly, but the history store will be unaffected. The option **single** was also added to convert a relation from the temporally partitioned structure to the single file structure. Therefore, we can specify that the structure of a relation be changed from a single file to another single file structure, from a single file structure to a temporally partitioned store, from a temporally partitioned store to another temporally partitioned store, or from a temporally partitioned to a single file structure. In this process, we can change the formats of the current and the history store independently of each other. Complete syntax of the extended **modify** statement is given in Appendix A.

The system relation was modified to maintain information on the structure of each relation: whether a relation is of the temporally partitioned structure, and if so, what format is used for the history store. A relation with the temporally partitioned storage structure consists of two physical files, when indices, if any, are not counted: one for the current store and the other for the history store. Opening or closing a relation opens or closes both files together. When a relation is accessed, it is necessary to track the current position for each file.

We can determine at compile time if a query is non-temporal. For a rollback database, a query is non-temporal if it has the clause **as of** "now". For a historical database, a query is non-temporal if it has the clause **when** (t_1 **overlap** ... **overlap** t_i) **overlap** "now" for all the range variables t_i. For a temporal database, a query is non-temporal if it has the clause **when** (t_1 **overlap** ... **overlap** t_i) **overlap** "now" for all the range variables t_i, and the clause **as of** "now". For a non-temporal or current query, the query is evaluated by consulting only the current store without going through the history store, using the conventional access methods provided by INGRES.

For the **delete** or the **replace** statement on a rollback database, there is only one case to be examined for the relationship between the base interval and the update interval. For the **delete** or the **replace** statement on a historical or a temporal database, there are four cases to be examined, ignoring two null cases, for the relationships between the base interval and the update interval as discussed in Section 5.1.2. In the prototype described in Section 6.1, the **replace** operation was performed by following the *delete and append* scheme, because this scheme was simple to implement for all cases.

However, the *delete and append* scheme was found to be inapplicable to the temporally partitioned store, because the base tuple remains in its place, while the newer version is put into a different location. Thus, the system was changed to follow the *append and change* scheme as discussed in Section 5.1.2. We had to examine each case of the relationships between the base interval and the update interval carefully to determine the proper location of the current version, and to maintain a history chain, whether explicit or not, for each version set. Maintaining a chain of history versions for each version set is more complicated for a temporal database, since each **replace** inserts at least two versions. We ordered versions affected in each update in reverse order of *valid from* time, then in reverse order of *transaction start* time. Thus, we can retrieve recent versions more quickly, especially for queries with the default clause **as of** "now".

Accessing a relation with the single file structure involves two steps: one for the main block and the other for overflow blocks. Accessing a relation with the temporally partitioned structure involves another step: following the history chain, whether explicit or implicit. Hence we need to maintain global information on which store provides the tuple being processed now and the tuple to be retrieved next. Algorithms to handle the `delete` and the `replace` statements on different types of relations are given in Appendix E.

For simplicity, the split criterion adopted in implementing the temporally partitioned store was:

- The current store contains current versions, while the history store holds history versions.
- Deleted tuples are kept in the current store.
- Versions to be expired, discussed in Section 5.1.3, are kept in the current store until a new version is inserted.
- Future versions are stored in the current store.

At present, the structure of *reverse chaining* has been actually implemented. The prototype's parser accepts the full BNF syntax, but the remaining components do not support the other options.

7.2. Performance Analysis

This section discusses performance improvement achieved for the prototype by using the various access methods developed in Chapter 5. Performance figures were obtained through performance analysis, as described in Appendix F using the models in Chapter 4. The figures for *reverse chaining* were also compared with the measurement data from the actual implementation to check its validity.

We studied the performance of the access methods on both rollback and temporal databases. We assume that *accession lists* and *indexing* maintain complete temporal information, both *transaction time* and *valid time* as appropriate, separate from history data. The index itself is assumed to be a hashed file, but note that indexing restricts the format of the current store to indexing, as discussed in Section 5.2.3. We also assume that the depth for *stacking* is four, and the cell size for *cellular chaining* is four.

As for *clustering*, we use the method of nonlinear hashing. The average storage utilization for nonlinear hashing is 69.3 %, as discussed in Section 5.2.4.2. However, databases considered in this analysis have high update counts, so each version set consists of more versions than a block can hold.

When a block gets full with versions belonging to a single version set, we need to maintain a chain of overflow blocks. As a result, storage utilization becomes 100% ignoring the last block of each chain.

7.2.1. Performance on a Rollback Database

Space requirements when the update count is 0 or 14 are shown in Figure 7-1 for the `Rollback_h` relation in hashing with 100% loading, and for the same relation with various formats of the temporally partitioned structure. Space requirements for the `Rollback_i` relation are similar to the `Rollback_h` relation except that the ISAM file requires additional space for directories. The table also shows the *growth rate*, which is obtained when the *growth per update* is divided by the size for the update count of 0.

Type	Hashing (100%)	Reverse Chaining	Accession Lists	Indexing	Clustering	Stacking	Cellular Chaining
Size, UC= 0	129	129	129	133	129	129	129
Size, UC=14	1927	1921	1922	1982	1921	(641)	2177
Growth per Update	128.4	128	128.1	128	128	(36.6)	146.3
Growth Rate	1.0	1.0	1.0	1.0	1.0	(0.28)	1.13

Notes :
'UC' denotes *Update Count*.
'(*n*)' denotes that only a partial history is stored.

Figure 7-1: Space Requirements for the `Rollback_h` Relation

From this table, we can make the following observations on the storage requirements of a rollback relation with the temporally partitioned storage structure:

- The temporally partitioned storage structures have the same space requirements as the single file structure when the update count is 0.
- When the update count is not 0, space requirements for reverse chaining, accession lists, indexing, and clustering are about the same.

- When the update count is not 0, space requirements for cellular chaining is larger than the other formats due to unfilled cells.

- When the update count is not 0, storage size for stacking remains the same, but older versions are lost due to stack overflows.

Figure 7-2 shows the input costs for the benchmark queries of Figure 6.5 on the rollback database with 100 % loading. Two columns under the label *Conventional* show the queries costs for the update count of 0 and 14. Then there are six columns to show the costs of queries for the update count of 14 for each format of the history store: *reverse chaining, accession lists, indexing, clustering, stacking,* and *cellular chaining.* When the update count is 0, the cost for any of the temporally partitioned structures is the same as the cost for the conventional case.

Query	Conventional Update Count 0	Conventional Update Count 14	Temporally Partitioned Store for Update Count = 14: Reverse Chaining	Accession Lists	Indexing	Clustering	Stacking	Cellular Chaining
Q01	1	15	15	16	16	3	(2)	5
Q02	2	16	16	17	16	4	(3)	6
Q03	129	1927	129	334	280	129	X	129
Q04	128	1920	128	333	280	128	X	128
Q05	1	15	1	1	2	1	1	1
Q06	2	16	2	2	2	2	2	2
Q07	129	1927	129	129	129	129	129	129
Q08	128	1920	128	128	128	128	128	128
Q09	1141	17242	1141	1141	2162	1141	1141	1141
Q10	2177	18311	2177	2177	2162	2177	2177	2177
Q15	129	3717	129	129	129	129	X	129

Notes :
'X' denotes *not applicable.*
'(*n*)' denotes that only a partial answer is retrieved.

Figure 7-2: The Rollback Database with 100% Loading

The advantage of the temporally partitioned store is evident in processing current queries such as Q05 through Q10. No matter what format is used for the history store, the cost remains constant for any update count. For example, Q10 on the rollback database costs 2177 blocks instead of 18311 blocks when the update count is 14. Query Q05 for indexing costs two block accesses, one more than the other formats, because the current store is also restricted to indexing, while the other formats allow hashing for the current store.

The performance of temporal queries like Q01 and Q02 can be improved by clustering, which collects history versions of each version set into a minimum number of blocks. Since there are 14 history versions for the update count of 14, and each block holds up to 8 tuples according to the assumption in Chapter 6, scanning all history versions for a version set costs two block accesses. Counting the cost to locate the current version in the current store, Q01 costs three block accesses, and Q02 costs four block accesses.

Cellular chaining also provides the benefit of clustering to a certain degree. It takes four cells to hold 14 history versions with the cell size of four according to the assumption. Hence, Q01 costs five block accesses, and Q02 costs six block accesses.

By *stacking*, we can retrieve history versions for each version set at the cost of one block access, but only a limited number of the most recent versions are maintained. Thus, Q01 costs two block accesses, and Q02 costs three block accesses, but those figures are put in parentheses to denote that the answers are only partial. Note that stacking cannot answer queries Q03 and Q04 inquiring the old status of the database, because older versions of history data were discarded due to *stack overflow*.

Accession lists or *indexing* with temporal information in each accession list or an index entry can facilitate temporal queries Q03 and Q04 by evaluating the temporal predicate without accessing history data. If we assume that accession lists maintain complete temporal information for the time attributes *transaction start* and *transaction stop*, each entry consumes 12 bytes for two time attributes and a pointer to a history version, thus 72 entries are contained in each block of 1024 bytes allowing for some overhead. Since there are 14 history versions times 1024 version sets for the update count of 14, the size of the entire accession lists is 200 blocks. Scanning the current store and the accession lists for the `Temporal_h` relation, entries satisfying the **as of** clause are extracted. If we assume that the number of such entries is five, the total cost for Q03 is 334 block accesses (= *129 + 200 + 5*). Likewise, the cost for Q04 is 333 block accesses (= *128 + 200 + 5*).

Similar improvement is also achieved by *indexing*, where each index entry maintains complete temporal information for transaction time. Since each entry with two time attributes plus a key and a pointer takes 16 bytes, and there are 15 versions times 1024 version sets for the update count of 14, the size of the entire index is 275 blocks. We need not scan the current store in indexing, so entries satisfying the

as of clause are extracted while canning the index for the `Temporal_h` relation. Under similar assumptions to the case of accession lists above, the total cost for Q03 or Q04 is 280 block accesses (= *275 + 5*).

The same arguments apply to the historical database with 100% loading, except that queries Q03 and Q04 are not applicable to a historical database. The costs for queries Q09 and Q10 are higher by 56 block accesses each on the historical database than on the rollback database, because *one variable detachment operation* is performed to evaluate the **when** clause for the queries Q09 and Q10, apparently without any benefit.

7.2.2. Performance on a Temporal Database

Space requirements when the update count is 0 or 14 are shown in Figure 7-3 for the `Temporal_h` relation in hashing with 100% loading, and for the same relation with various formats of the temporally partitioned structure. Space requirements for the `Temporal_i` relation are similar to the `Temporal_h` relation except that the ISAM file requires additional space for directories. The table also shows the *growth rate*, which is obtained when the *growth per update* is divided by the size for the update count of 0.

From Figure 7-3, we can make the following observations on the storage requirements of a temporal relation with the temporally partitioned storage structure:

- The temporally partitioned storage structures consume slightly more space than the single file structure when the update count is 0, due to extra space for a physical link to the history chain.
- The temporally partitioned storage structures consume more space than the single file structure when the update count is not 0, due to extra space for maintenance of chaining, indexing or accession lists.
- When the update count is not 0, space requirements for reverse chaining, accession lists, indexing, and clustering are about the same.
- When the update count is not 0, space requirements for cellular chaining can be larger than the other formats if there are unfilled cells.
- When the update count is not 0, storage size for stacking remains the same, but older versions are lost due to stack overflows.

Type	Hashing (100%)	Reverse Chaining	Accession Lists	Indexing	Clustering	Stacking	Cellular Chaining
Size, UC= 0	129	147	147	141	147	147	147
Size, UC=14	3717	4243	3957	4082	4243	(733)	4243
Growth per Update	256.3	292.6	272.1	281.5	292.6	(41.9)	292.6
Growth Rate	1.99	1.99	1.85	2.0	1.99	(0.28)	1.99

Notes :
'UC' denotes *Update Count.*
'(n)' denotes that only a partial history is stored.

Figure 7-3: Space Requirements for the `Temporal_h` Relation

Compared with the table in Figure 7-1, we find that:

- When the update count is 0 with the temporally partitioned storage structure, a temporal relation can consume more space than a corresponding rollback relation due to additional time attributes.
- When the update count is not 0 with the temporally partitioned storage structure, a temporal relation consumes about twice the space of a corresponding rollback relation, because each **`replace`** operation inserts two new versions.

Figure 7-4 shows the input costs for the temporal database with 100% loading. We make the same assumptions as those for the rollback database in Figure 7-2, except that accession lists and indexing also maintains information on *valid time* as well as *transaction time.* The discussion concerning queries Q01 through Q10 on the rollback database with 100% loading similarly applies to those queries on the temporal database with 100% loading.

Performance improvement with the temporally partitioned storage structure is even striking for the temporal database. For queries Q05 through Q10 on any temporally partitioned structure other than indexing, the cost remains constant regardless of the update count. For example, Q10 on the temporal database costs 2251 blocks instead of 34493 blocks when the update count is 14. Note, however, that the costs of queries Q07 through Q10 on a temporally partitioned structure are slightly higher than the corresponding costs on a conventional structure with the update count of 0, because the size of the current

store is bigger than the conventional structure with the update count of 0. As for query Q09 or Q10 on the temporal database with indexing, we need to scan the index and the current store of the `Temporal_i` relation, then repeatedly access the `Temporal_h` relation through the index. The resulting cost is significantly higher than other formats, but is still lower than the conventional case.

Query	Conventional Update Count 0	Conventional Update Count 14	Temporally Partitioned Store for Update Count = 14: Reverse Chaining	Accession Lists	Indexing	Clustering	Stacking	Cellular Chaining
Q01	1	29	29	30	30	5	(2)	8
Q02	2	30	30	31	30	6	(3)	9
Q03	129	3717	4243	776	787	4243	X	4243
Q04	128	3712	4243	776	787	4243	X	4243
Q05	1	29	1	1	2	1	1	1
Q06	2	30	2	2	2	2	2	2
Q07	129	3717	147	147	141	147	147	147
Q08	128	3712	147	147	141	147	147	147
Q09	1200	33350	1227	1227	2218	1227	1227	1227
Q10	2233	34493	2251	2251	2218	2251	2251	2251
Q11	385	11141	12729	2317	2350	12729	X	12729
Q12	131	3743	4274	3989	4114	4250	(737)	4253
Q13	1	29	29	8	8	5	X	8
Q14	129	3717	4243	3957	4082	4243	X	4243
Q15	129	3717	4243	3957	4082	4243	X	4243
Q16	129	3717	4243	3957	4082	4243	X	4243

Notes :
'X' denotes *not applicable*.
'(*n*)' denotes that only a partial answer is retrieved.

Figure 7-4: The Temporal Database with 100% Loading

As for query Q11 which requires a *join* operation on time attributes, the performance can be improved by *accession lists*, where each accession list maintains complete temporal information for the time attributes *transaction start, transaction stop, valid from,* and *valid to*. Since each entry with four time attributes and a pointer to a history version consumes 20 bytes, and there are 28 history versions times 1024 version sets for the update count of 14, the size of the entire accession lists is 624 blocks. Scanning the current store and the entries in the accession lists for the `Temporal_h` relation, the entries satisfying the **as of** clause are extracted. If we assume that the number of such entries is two, and that *tuple substitution* is used to perform a *join*, then the current store and the accession lists for the `Temporal_i` relation are scanned twice to find entries satisfying the **as of** and the **when** clauses. Thus we end up

with scanning the current store and the accession lists three times: once for the `Temporal_h` relation and twice for the `Temporal_i` relation. If we assume that two entries in the accession lists for the `Temporal_i` relation satisfy the **as of** and the **when** clause, then four history versions are actually retrieved from the history store: two from the `Temporal_h` relation and two from the `Temporal_i` relation. So the total cost is 2317 block accesses ($= (147 + 624) \times 3 + 4$), which is a marked improvement from 11141 of the conventional method. The improvement resulted from performing a temporal join on the accession lists, whose size is much smaller than the history data.

Similar improvement is also achieved by *indexing*, where each index entry maintains complete temporal information for the four time attributes. Since each entry with four time attributes plus a key and a pointer takes 24 bytes, and there are 29 versions times 1024 version sets for the update count of 14, the size of the entire index is 782 blocks. Scanning the index for the `Temporal_h` relation, the index entries satisfying the **as of** clause are extracted. Under similar assumptions to the case of accession lists above, the index for the `Temporal_i` relation is scanned twice to find the entries satisfying the **as of** and the **when** clauses. Then the total cost is 2350 block accesses ($= 782 \times 3 + 4$).

Query Q12 is facilitated by *clustering* or *cellular chaining* for the portion of scanning a version set, as discussed for queries Q01 and Q02, but the overall performance is dominated by scanning the `Temporal_i` relation sequentially. Query Q12 will be further discussed in the next section for *secondary indexing*. Note that *stacking* cannot answer query Q11, and provides only a partial answer to query Q12.

Query Q13 is similar to Q01, but Q13 can be improved by *accession lists* or *indexing*. The **when** clause can be evaluated without accessing history data, then only the tuples satisfying the temporal predicate are retrieved. If we assume there are seven such tuples, the cost is eight block accesses, where one extra block accounts for accessing an accession list or an index entry.

Queries Q14 through Q16 retrieve tuples through a non-key attribute, which requires sequential scanning of the entire relation. Maintaining a secondary index can improve the costs of these queries, as will be discussed in the next section.

Access Method	Conventional			Reverse Chaining			Accession Lists			Indexing		
Query	Cost (in Blocks) Fixed	Variable	Growth Rate	Cost (in Blocks) Fixed	Variable	Growth Rate	Cost (in Blocks) Fixed	Variable	Growth Rate	Cost (in Blocks) Fixed	Variable	Growth Rate
Q01	0	1	2	0	1	2	1	1	2	1	1	2
Q02	1	1	2	1	1	2	2	1	2	1	1	2
Q03	0	129	1.99	0	146.3	2	129	22.3	2	5	27	2
Q04	0	128	2	0	146.3	2	129	22.3	2	5	27	2
Q05	0	1	2	0	1	0	0	1	0	1	1	0
Q06	1	1	2	1	1	0	1	1	0	1	1	0
Q07	0	129	1.99	0	146.3	0	0	146.3	0	0	140.7	0
Q08	0	128	2	0	146.3	0	0	146.3	0	0	140.7	0
Q09	56	1144	2.01	56	1170.3	0	56	1170.3	0	56	2162	0
Q10	1080	1153	2	1080	1170.3	0	1080	1024	0	56	2162	0
Q11	0	385	2	0	438.9	2	376	66.9	2	4	81	2
Q12	2	129	2	2	147.3	2	13	137.0	2	3	141.7	2
Q13	0	1	2	0	1	2	1	0.24	2	1	0.24	2
Q14	0	129	1.99	0	146.3	2	0	146.3	2	0	146.3	2
Q15	0	129	1.99	0	146.3	2	0	146.3	2	0	146.3	2
Q16	0	129	1.99	0	146.3	2	0	146.3	2	0	146.3	2

Access Method	Conventional			Clustering			Stacking			Cellular Chaining		
Query	Cost (in Blocks) Fixed	Variable	Growth Rate	Cost (in Blocks) Fixed	Variable	Growth Rate	Cost (in Blocks) Fixed	Variable	Growth Rate	Cost (in Blocks) Fixed	Variable	Growth Rate
Q01	0	1	2	0	1	0.29	(0)	(1)	(0.07)	0	1	0.5
Q02	1	1	2	1	1	0.29	(1)	(1)	(0.07)	1	1	0.5
Q03	0	129	1.99	0	146.3	2	X	X	X	0	146.3	2
Q04	0	128	2	0	146.3	2	X	X	X	0	146.3	2
Q05	0	1	2	0	1	0	0	1	0	0	1	0
Q06	1	1	2	1	1	0	1	1	0	1	1	0
Q07	0	129	1.99	0	146.3	0	0	146.3	0	0	146.3	0
Q08	0	128	2	0	146.3	0	0	146.3	0	0	146.3	0
Q09	56	1144	2.01	56	1170.3	0	56	1170.3	0	56	1170.3	0
Q10	1080	1153	2	1080	1170.3	0	1080	1170.3	0	1080	1170.3	0
Q11	0	385	2	0	438.9	2	X	X	X	0	438.9	2
Q12	2	129	2	2	146.5	2	(2)	(25.3)	(2)	2	146.6	2
Q13	0	1	2	0	1	0.29	X	X	X	0	1	0.5
Q14	0	129	1.99	0	146.3	2	0	146.3	2	0	146.3	2
Q15	0	129	1.99	0	146.3	2	0	146.3	2	0	146.3	2
Q16	0	129	1.99	0	146.3	2	0	146.3	2	0	146.3	2

Note :

'X' denotes *not applicable.*

Figure 7-5: Fixed Costs, Variable Costs, and Growth Rates

In section 6.2.3, we divided the cost of a query into the *fixed* portion and the *variable* portion, then calculated the *growth rate* of the query cost. Following the same procedure, we obtained Figure 7-5 which shows the fixed costs, variable costs, and growth rates for each query on the temporal database with

different formats of the history store. The table shows that:

- For *non-temporal* queries such as Q05 through Q10, the growth rate can be reduced to 0 by using any of the temporally partitioned storage structures.
- For *version scanning, e.g.* queries Q01 and Q02, *clustering* and *cellular chaining* reduce the growth rate, improving the performance.
- For rollback queries such as Q03, Q04, and Q11, *accession lists* and *indexing* reduce the *variable* cost, improving the performance as the update count increases.
- For queries which require sequential scanning, *e.g.* queries Q03, Q04, Q11, and Q12, *reverse chaining, clustering,* or *cellular chaining* exhibits a slightly inferior performance, due to the overhead of storing temporal information.

In conclusion, the temporally partitioned storage structure improve the retrieval performance of databases with temporal support by reducing either the growth rate or the variable cost.

7.3. Secondary Indexing

Queries retrieving records through non-key attributes can be facilitated by secondary indexing. For example, we can create a secondary index, `Temp_h_inx`, on the `amount` attribute of the `Temporal_h` relation using the **`index`** statement in Quel:

```
index on    Temporal_h  is    Temp_h_inx  (amount)
```

Maintaining a secondary index on the attribute `amount` can improve the performance of queries such as Q07, Q08, Q12, and Q14 through Q16.

As discussed in Section 5.3, there are several types of secondary indices, especially for a temporal relation. A secondary index for a temporal relation may be any of *snapshot, rollback, historical,* or *temporal.* To specify the type of a secondary index, we extend the **`index`** statement with the **`as`** *type* clause, where *type* can be any of **`snapshot`**, **`rollback`**, **`historical`**, and **`temporal`**. For example, a statement:

```
index on  Temporal_h   is  Temp_h_inx  (amount)    as    temporal
```

creates a secondary index as a temporal relation.

The default storage structure for a secondary index is a heap, but like any regular relation, its structure can be changed to other format using the `modify` statement. An index may be stored into a single file for all the versions *(single file)*, or may itself be maintained as a *temporally partitioned* structure having a *current index* for current data and a *history index* for history data. In each case, we may choose any access methods such as a heap, hashing, ISAM, *etc*. At present, our prototype supports only the secondary indices as snapshots. The other options were not implemented into the prototype.

Space requirements for various types of secondary indices on the *Temporal_h relation* are shown in Figure 7-6, when the update count is 0 or 14. The table also shows the *growth rate*, which is obtained when the *growth per update* is divided by the size for the update count of 0. Compared with the table in Figure 7-3, a secondary index consumes from 8% to 21% of the space required by the relation itself.

Type	as Snapshot	as Rollback	as Historical	as Temporal
Size, UC= 0	11	19	19	27
Size, UC=14	295	531	531	782
Growth per Update	20.3	36.6	36.6	53.9
Growth Rate	1.85	1.93	1.93	2.0

Note :
'UC' denotes *Update Count.*

Figure 7-6: Space Requirements for a Secondary Index

For the *snapshot* index, each entry needs eight bytes, four for the secondary key and four for a pointer. Since a block of 1024 bytes can store 101 entries, 11 blocks are needed for 1024 tuples when the update count is 0. When the update count is 14, there are 29 versions multiplied by 1024 tuples; hence 295 blocks are needed for the single file index.

For the *rollback* or the *historical* index, each entry needs 16 bytes, four for the secondary key, four for a pointer, and eight for two attributes of transaction time or valid time. So a block of 1024 bytes can

store 56 entries, and there are 29 versions multiplied by 1024 tuples when the update count is 14; hence 531 blocks are needed for the single file index.

For the *temporal* index, each entry needs 24 bytes, four for the secondary key, four for a pointer, eight for two attributes of valid time, and eight for two attributes of transaction time. So a block of 1024 bytes can store 38 entries, and there are 29 versions multiplied by 1024 tuples when the update count is 14; hence 782 blocks are needed for the single file index.

Figure 7-7 compares the *snapshot* index with the *rollback* index in terms of the costs of sample queries on the temporal database with the update count of 14. Performance figures in this table were derived analytically; as an example, the cost of query Q16 is analyzed in Appendix F. Note that the existence or the structure of secondary indices do not affect the performance of other queries which do not involve the secondary access path.

Query	Conventional		Indexed as Snapshot				Indexed as Rollback			
	Update Count		as Single		as Partitioned		as Single		as Partitioned	
	0	14	as Heap	as Hash	as Heap	as Hash	as Heap	as Hash	as Heap	as Hash
Q07	129	3717	324	30	12	2	560	30	20	2
Q08	128	3712	324	30	12	2	560	30	20	2
Q12	131	3743	355	61	355	62	591	61	591	62
Q14	129	3717	324	30	324	31	560	30	560	31
Q15	129	3717	324	30	324	31	543	13	543	14
Q16	129	3717	324	30	324	31	543	13	543	14

Note :

All values are for a temporal database with a 100% loading and the update count of 14.

Figure 7-7: Secondary Indexing as Snapshot or Rollback

If the index is stored as a heap, queries Q07 and Q08 cost 324 block accesses each, 295 index blocks plus 29 data blocks. This is in fact more expensive than the simple temporally partitioned store without any index, though better than the conventional structure. Hence, we must take care that the cost of using an index does not overwhelm the saving obtained from using the temporally partitioned store. If the index is hashed, the cost is reduced to 30 block accesses with 1 index block and 29 data blocks.

If we follow the temporally partitioned scheme maintaining a separate index for current data, there are only 1024 entries in the current index, requiring 11 index blocks. Each of Q07 and Q08 costs 12 blocks

with a heap index, while it costs only 2 blocks with hashing. Note the difference between 3717 blocks and 2 blocks for processing the same query.

Query Q12 can also benefit from secondary indexing, since it is no longer necessary to scan the `Temporal_i` relation sequentially. If the index is stored as a single heap, Q12 costs 355 block accesses, where 295 block accesses are needed to scan the index. If the index is stored as a single hash, the cost is reduced to 61 block accesses.

Queries Q14 through Q16 are similar to queries Q07 and Q08 in that they are one relation queries and their costs can be reduced significantly with secondary indexing. However, queries Q14 through Q16, like Q12, are temporal queries, and need to access history data regardless of the storage structure. Thus the temporally partitioned index is not better than the single file index for queries Q12 and Q14 through Q16. In fact, the index as a temporally partitioned hash costs one more block access than the index as a single hash, because each index needs to be hashed separately.

The *rollback* index is effective for processing queries with the **as of** clause, such as Q15 and Q16. The **as of** predicate can be evaluated with information from index entries, and only the tuples that satisfy the predicate need to be retrieved.

If the index is stored as a single hash, query Q15 costs 13 block accesses, assuming that there are 12 tuples satisfying the **as of** clause among 29 candidates. Storing the index as a temporally partitioned hash, query Q15 costs 14 block accesses, one block access more than as a single hash, since each index needs to be hashed separately. However, storing the rollback index as a heap increases the query costs over the snapshot index, due to the bigger size of the rollback index.

Figure 7-8 compares the *historical* index with the *temporal* index in terms of the costs of sample queries on the temporal database with the update count of 14. The discussion on the rollback index similarly applies to the historical index, except that the historical index maintains two attributes of valid time instead of transaction time, and that the historical index is effective for processing queries with the **when** clause like Q14 and Q16. If the index is stored as a single hash, Q14 or Q16 costs 8 block accesses, assuming that there are 7 tuples satisfying the **when** clause among 29 candidates.

Query	Conventional Update Count		Indexed as Historical				Indexed as Temporal			
			as Single		as Partitioned		as Single		as Partitioned	
	0	14	as Heap	as Hash	as Heap	as Hash	as Heap	as Hash	as Heap	as Hash
Q07	129	3717	532	2	20	2	783	2	28	2
Q08	128	3712	532	2	20	2	783	2	28	2
Q12	131	3743	563	61	563	62	814	61	814	62
Q14	129	3717	538	8	538	9	789	8	789	9
Q15	129	3717	560	30	560	31	794	13	794	14
Q16	129	3717	538	8	538	9	786	5	786	6

Note :

All values are for a temporal database with a 100% loading and the update count of 14.

Figure 7-8: Secondary Indexing as Historical or Temporal

The *temporal* index combines the benefits of the rollback index and the historical index, effective for processing queries with the `as of` or `when` clause. The temporal predicate can be evaluated completely with information from index entries, and only the tuples that satisfy the predicate need to be retrieved.

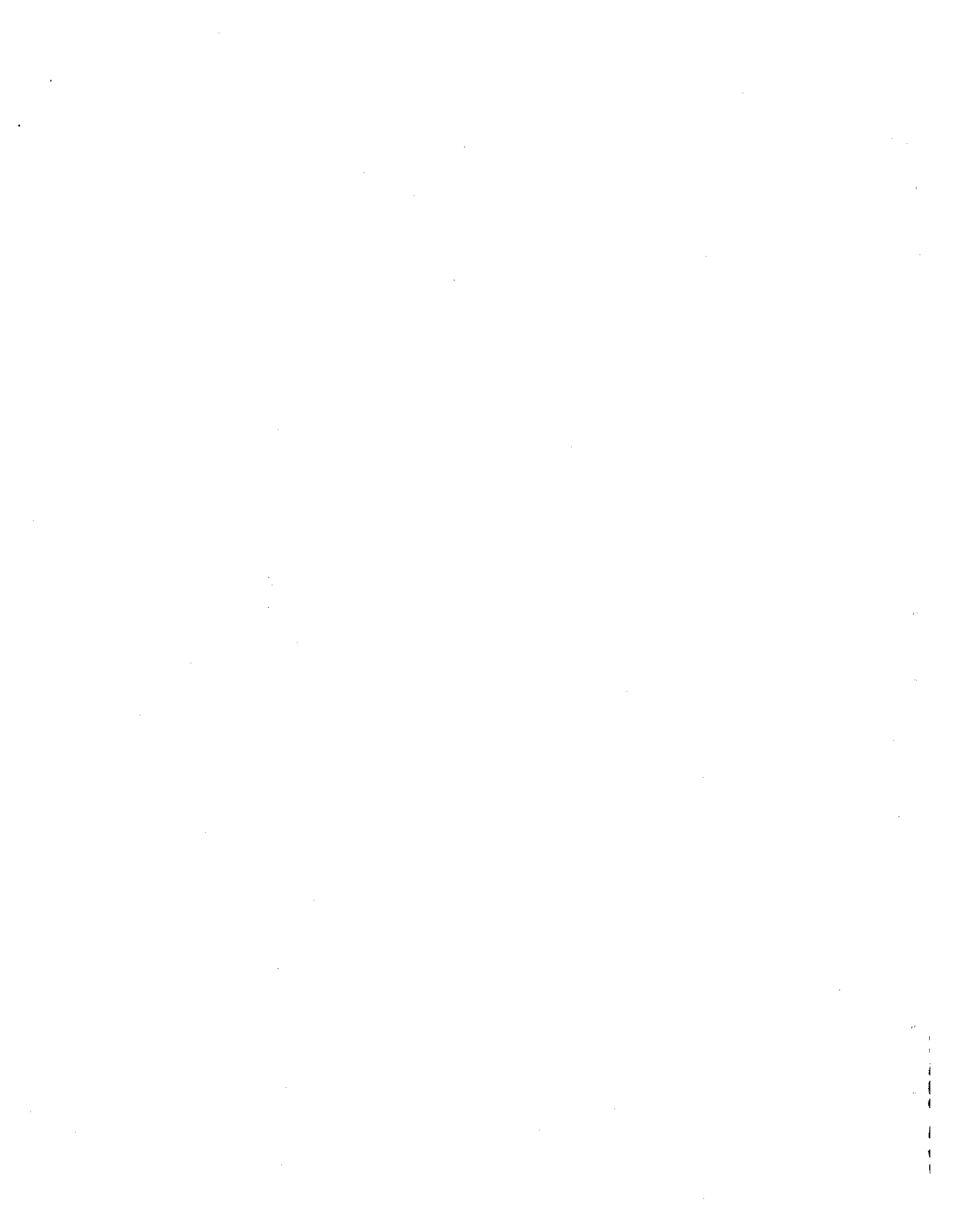
If the index is stored as a single hash, Q16 costs only 5 block accesses, assuming that there are 4 tuples satisfying both the `when` and the `as of` clauses among 29 candidates. However, storing the temporal index as a heap increases the cost of queries over any other types of indices, due to the bigger size of the temporal index.

Now we can make the following observations on the types of secondary indices, based on the analysis of query costs as shown in Figures 7-7 and 7-8.

- The temporally partitioned index is good for non-temporal queries.
- For temporal queries, the cost of a query for the temporally partitioned heap index is equal to the cost of the query for the single heap index.
- For temporal queries, the cost of a query for the temporally partitioned hash index is more expensive by one block access than the cost of the query for the single hash index.
- The rollback secondary index is good for queries with the `as of` clause.
- The historical secondary index is good for queries with the `when` clause.
- The temporal secondary index is good for queries with either or both of the `when` and the `as of` clauses.

- It is desirable to provide a secondary index with the random access mechanism such as hashing.

- If there is no random access mechanism for a secondary index, storing a large amount of temporal information in index entries degrades the performance due to the bigger size.



PART IV

Conclusions

Thus far, various issues on database management systems with temporal support have been examined with emphasis on the implementation aspects. Chapter 8 presents the summary and conclusions of this dissertation and discusses the future work to be pursued in the area.

Chapter 8

Conclusions and Future Work

8.1. Conclusions

The thesis of this research 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.

To demonstrate this thesis, we have developed a set of models to characterize the various phases of query processing in database management systems with temporal support. We have investigated various formats of the *temporally partitioned storage structures*, and analyzed their performance using the performance models. We also implemented a prototype temporal database system incorporating one of the temporally partitioned structures, and ran a benchmark to measure the performance of sample queries on the prototype.

The measurement data and the analysis results indicate that the temporally partitioned store can improve the performance of various temporal queries, while eliminating a performance penalty on conventional non-temporal queries. Query costs estimated from the analysis were compared with the measurement data in Section 6.3, which showed that the performance of database systems with temporal support can be analyzed quite accurately using the four models.

Major contributions of this research achieved in this process are:

- A taxonomy of time to classify database types in terms of temporal support was developed.
    - Three distinct kinds of time with orthogonal semantics in database management systems were identified. They are *transaction time*, *valid time*, and *user-defined time*.
    - Depending on the capability to support either or both of transaction time and valid time, databases were classified into four types: *snapshot*, *rollback*, *historical*, and *temporal*.

- Four models forming a hierarchy were developed to characterize query processing in database systems with temporal support.

  * The *algebraic expression* was defined to describe procedurally the process of evaluating TQuel queries.

  * The *file primitive expression* was defined to characterize the input and output operations.

  * The *model of algebraic expressions* was developed to map the algebraic expression to the file primitive expression.

  * The *model of database/relations* was developed to represent the characteristics of a database and relations.

  * The *access path expression* was defined to characterize a path taken through a storage structure to satisfy an access request.

  * The *model of access paths* was developed to map the file primitive expression to the access path expression.

  * The *model of storage devices* was developed to represent the characteristics of storage devices, and to map the access path expression to the *access path cost.*

- The *temporally partitioned storage structure* was investigated to improve the performance of temporal queries without penalizing conventional non-temporal queries.

  * Various issues on the temporally partitioned structure were examined.

  * Update procedures for `delete` and `replace` were developed and analyzed.

  * Six formats of the *history store* were developed, analyzed, and compared with one another. They are *reverse chaining, accession lists, indexing, clustering, stacking,* and *cellular chaining.*

  * A new form of hashing, termed *nonlinear hashing,* was developed.

  * *Tuple versioning* and *attribute versioning* were compared with each other, and the conversion process from one form to the other was formalized.

  * Issues on *secondary indexing* for databases with temporal support were examined.

- 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 INGRES.

  * A benchmark was run on the prototype to identify problems with conventional access methods.

  * The benchmark also provided performance data to be compared with analysis results from the performance models.

  * Among the temporally partitioned storage structures, *reverse chaining* was incorporated in the prototype to enhance its performance.

- The feasibility of providing temporal support for database management systems without penalizing conventional non-temporal queries was demonstrated.

8.2. Future Work

This research has addressed some of the major issues on providing temporal support for database management systems, yet many issues still remain to be investigated.

The first area of further work is to implement various formats of the history store developed in Section 5.2. We have implemented *reverse chaining* as described in Chapter 7. The other structures can be added incrementally to the system.

We need to analyze the cost to update temporally partitioned storage structures. A preliminary study indicates that the output cost for such an operation is slightly higher due to the overhead for maintaining a specific structure, but its input cost can be lower than conventional structures. The actual cost will be heavily dependent on the number of buffers and buffering algorithms. The CPU cost involved in maintaining the temporally partitioned storage structures also needs to be considered.

This research has dealt with only the core of TQuel. It will be interesting to investigate implementation issues on temporal aggregates [Snodgrass & Gomez 1986], implement them, and analyze their performance.

Throughout this research, we assumed that all temporal information is complete and accurate, which is not true in many cases. Some work has been done to classify information as *determinant* or *indeterminant*, and to define the *before* predicate using three-valued logic [Snodgrass 1982]. We still need to study issues on how to handle different semantics of *null*, *unknown*, or *uncertain*, and on how to support *incomplete* temporal information.

Though this research has studied new access methods to improve the performance of temporal queries, further work is needed to develop new query processing algorithms for optimization of temporal queries. For example, our prototype database system performs the *temporal join* operation using the method of *tuple substitution*, as INGRES computes the conventional join. New methods tailored to the particular characteristics of temporal join may be developed to improve the performance.

We described the *Performance Analyzer for TQuel Queries (PATQ)* in Section 4.2.2, which utilizes the four models developed to characterize the various phases of temporal query processing. We analyzed the performance of sample queries manually in Chapters 6 and 7. However, this analysis could be automated by implementing the PATQ. In addition, the PATQ could be extended to be an optimization tool as discussed in Section 4.2.2.

Nonlinear hashing was developed in this research to cluster tuples belonging to the same version set, but it can be used for other applications to retrieve a record at the cost of one disk access. Further work is needed to analyze characteristics of split functions, study its performance in a highly dynamic environment, and extend it to *nested* nonlinear hashing briefly described in Section 5.2.4.2.

Supporting time in database management systems not only adds to the functionality for various applications, but also can benefit internal DBMS operations. Though this research has not addressed issues on concurrency control, recovery, or synchronization of distributed databases, such issues need to be studied to utilize the temporal information inherent in databases with temporal support.

As discussed in Section 1.1.2, database management systems with temporal support expand the area of database applications, bringing a wide range of benefits. However, many interesting issues remain to be investigated, some of which were listed in this section. It is a challenge to pursue these issues for realizing the full potential of such systems.

Bibliography

[Aghili & Severance 1982] Aghili, J. and D. Severance. *A Pratical Guide to the Design of Differential Files for Recovery of On-Line Databases. ACM Transactions on Database Systems,* 7, No. 4, Dec. 1982, pp. 540-565.

[Ahn 1986] Ahn, I. *Towards an Implementation of Database Management Systems with Temporal Support,* in *Second International Conference on Data Engineering,* IEEE. Feb. 1986, pp. 374-381.

[Ahn & Snodgrass 1986] Ahn, I. and R. Snodgrass. *Performance Evaluation of a Temporal Database Management System,* in *Proceedings of ACM SIGMOD International Conference on Management of Data,* Washington, DC: May 1986, pp. 96-107.

[Ammon et al. 1985] Ammon, G., J. Calabria and D. Thomas. *A High-Speed, Large-Capacity, Jukebox Optical Disk System. IEEE Computer,* 18, No. 7, July 1985, pp. 36-46.

[Anderson 1982] Anderson, T.L. *Modeling Time at the Conceptual Level,* in *Improving Database Usability and Responsiveness,* Ed. P. Scheuermann. Jerusalem, Israel: Academic Press, 1982, pp. 273-297.

[Andler 1979] Andler, S. A. *Predicate Path Expressions: A High-level Synchronization Mechanism.* PhD. Diss. Computer Science Department, Carnegie-Mellon University, Aug. 1979.

[Ariav & Morgan 1981] Ariav, G. and H.L. Morgan. *MDM: Handling the time dimension in generalized DBMS.* Working Paper. The Wharton School, University of Pennsylvania. May 1981.

[Ariav & Morgan 1982] Ariav, G. and H.L. Morgan. *MDM: Embedding the Time Dimension in Information Systems.* TR 82-03-01. Department of Decision Sciences, The Wharton School, University of Pennsylvania. 1982.

[Ariav 1984] Ariav, G. *Preserving the Time Dimension in Information Systems.* PhD. Diss. The Wharton School, University of Pennsylvania, Apr. 1984.

[Batory 1981] Batory, D. *B+ Trees and Indexed Sequential Files: A Performance Comparison,* in *Proceedings of ACM SIGMOD International Conference on Management of Data,* May 1981, pp. 30-39.

[Batory & Gotlieb 1982] Batory, D. and C. Gotlieb. *A Unifying Model of Physical Databases. ACM Transactions on Database Systems,* 7, No. 4, Dec. 1982, pp. 509-539.

[Batory 1985] Batory, D. *Modeling The Storage Architecture Of Commercial Database Systems. ACM Transactions on Database Systems,* 10, No. 4, Dec. 1985, pp. 463-528.

[Bayer & McCreight 1972] Bayer, R. and E. McCreight. *Organization and Maintenance of Large Ordered Indexes. Acta Informatica,* 1, No. 3 (1972), pp. 173-189.

[Bayer & Unterauer 1977] Bayer, R. and K. Unterauer. *Prefix B-Trees. ACM Transactions on Database Systems,* 2, No. 1, Mar. 1977, pp. 11-26.

[Ben-Zvi 1982] Ben-Zvi, J. *The Time Relational Model.* PhD. Diss. UCLA, 1982.

[Bernstein & Goodman 1980] Bernstein, P. and N. Goodman. *Timestamp-Based Algorithms for Concurrency Control in Distributed Database Systems*, in *Proceedings of the Conference on Very Large Databases*, Oct. 1980.

[Bloom 1970] Bloom, B. *Space/Time Trade-offs in Hash Coding with Allowable Errors. Communications of the Association of Computing Machinery*, 13, No. 7, July 1970, pp. 422-426.

[Bolour et al. 1982] Bolour, A., T.L. Anderson, L.J. Debeyser and H.K.T. Wong. *The Role of Time in Information Processing: A Survey. SigArt Newsletter*, 80, Apr. 1982, pp. 28-48.

[Breutmann et al. 1979] Breutmann, B., E. F. Falkenberg and R. Mauer. *CSL: A language of defining conceptual schemas*, in Data Base Architecture. Amsterdam: North Holland, Inc., 1979.

[Bubenko 1977] Bubenko, J.A., Jr. *The Temporal Dimension in Information Modeling*, in Architecture and Models in Data Base Management Systems. North-Holland Pub. Co., 1977.

[Cardenas 1973] Cardenas, A. *Evaluation and Selection of File Organization - A Model and System. Communications of the Association of Computing Machinery*, 16, No. 9, Sep. 1973, pp. 540-548.

[Cardenas 1975] Cardenas, A. *Analysis and Performance of Inverted Data Base Structures. Communications of the Association of Computing Machinery*, 18, No. 5, May 1975, pp. 253-263.

[Ceri & Pelagatti 1984] Ceri, S. and G. Pelagatti. *Distributed Databases Principles & Systems*. NY: McGraw-Hill, 1984.

[Chen & Vitter 1984] Chen, W. and J. Vitter. *Analysis of New Variants of Coalesced Hashing. ACM Transactions on Database Systems*, 9, No. 4, Dec. 1984, pp. 616-645.

[Cheung 1982] Cheung, T. *Estimating Block Accesses and Number of Records in File Management. Communications of the Association of Computing Machinery*, 25, No. 7, July 1982, pp. 484-487.

[Christodoulakis 1983] Christodoulakis, S. *Estimating Block Transfers and Join Sizes*, in *Proceedings of ACM SIGMOD International Conference on Management of Data*, May 1983, pp. 40-54.

[Christodoulakis 1984] Christodoulakis, S. *Implications of Certain Assumptions in Database Performance Evaluation. ACM Transactions on Database Systems*, 9, No. 2, June 1984, pp. 163-186.

[Cichelli 1980] Cichelli, R. *Minimal perfect hash functions made simple. Communications of the Association of Computing Machinery*, 23, No. 1, Jan. 1980, pp. 17-19.

[Clifford & Tansel 1985] Clifford, J. and A. Tansel. *On An Algebra For Historical Relational Databases: Two Views*, in *Proceedings of ACM SIGMOD International Conference on Management of Data*, 1985, pp. 247-265.

[Clifford & Warren 1983] Clifford, J.A. and D.S. Warren. *Formal Semantics for Time in Databases. ACM Transactions on Database Systems*, 8, No. 2, June 1983, pp. 214-254.

[Codd 1979] Codd, E.F. *Extending the Database Relational Model to Capture More Meaning. ACM Transactions on Database Systems*, 4, No. 4, Dec. 1979, pp. 397-434.

[Coffman & Eve 1970] Coffman, E. and J. Eve. *File Structures Using Hashing Functions. Communications of the Association of Computing Machinery*, 13, No. 7, July 1970, pp. 427-432.

[Comer 1979] Comer, D. *The Ubiquitous B-tree. Computing Surveys*, 11, No. 2 (1979), pp. 121-138.

[Copeland 1982] Copeland, G. *What If Mass Storage Were Free?. IEEE Computer*, 15, No. 7, July 1982, pp. 27-35.

[Copeland & Maier 1984] Copeland, G. and D. Maier. *Making Smalltalk a Database System*, in *Proceedings of ACM SIGMOD International Conference on Management of Data*, Ed. B. Yormark. Association for Computing Machinery. Boston, MA: June 1984, pp. 316-325.

[Fagin et al. 1979] Fagin, R., J. Nievergelt, N. Pippenger and H. Strong. *Extendible Hashing - A Fast Access Method for Dynamic Files. ACM Transactions on Database Systems*, 4, No. 3, Sep. 1979, pp. 315-344.

[Findler & Chen 1971] Findler, N. and D. Chen. *On the problems of time retrieval, temporal relations, causality, and coexistence*, in *Proceedings of the International Joint Conference on Artificial Intelligence*, Imperial College: Sep. 1971.

[Fujitani 1984] Fujitani, L. *Laser Optical Disk: The Coming Revolution in On-Line Storage. Communications of the Association of Computing Machinery*, 27, No. 6, June 1984, pp. 546-554.

[Gadia & Vaishnav 1985] Gadia, S. and J. Vaishnav. *A Query Language For A Homogeneous Temporal Database*, in *Proceedings of the ACM Symposium on Principles of Database Systems*, Apr. 1985.

[Gadia 1986] Gadia, S. *Toward a Multihomogeneous Model For a Temporal Database*, in *Second International Conference on Data Engineering*, IEEE. Feb. 1986, pp. 390-397.

[Goldberg & Robson 1983] Goldberg, A. and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[Gremillion 1982] Gremillion, L. *Designing a Bloom Filter for Differential File Access. Communications of the Association of Computing Machinery*, 25, No. 9, Sep. 1982, pp. 600-604.

[Hammer & McLeod 1981] Hammer, M. and D. McLeod. *Database Description with SDM: A Semantic Database Model. ACM Transactions on Database Systems*, 6, No. 3, Sep. 1981, pp. 351-386.

[Hawthorn & Stonebraker 1979] Hawthorn, P. and M. Stonebraker. *Performance Analysis of a Relational Data Base Management System*, in *Proceedings of ACM SIGMOD International Conference on Management of Data*, 1979, pp. 1-12.

[Held 1978] Held, G. and M. Stonebraker. *B-trees Re-examined. Communications of the Association of Computing Machinery*, 21, No. 2, Feb. 1978, pp. 139-143.

[Held et al. 1975] Held, G.D., M. Stonebraker and E. Wong. *INGRES--A relational data base management system. Proceedings of the 1975 National Computer Conference*, 44 (1975), pp. 409-416.

[Hoagland 1985] Hoagland, A. *Information Storage Technology: A Look at the Future. IEEE Computer*, 18, No. 7, July 1985, pp. 60-67.

[Hsiao & Harary 1970] Hsiao, D. and F. Harary. *A Formal System for Information Retrieval from Files. Communications of the Association of Computing Machinery*, 13, No. 2, Feb. 1970, pp. 67-73.

[IBM 1981] IBM *SQL/Data-System, Concepts and Facilities*. Technical Report GH24-5013-0. IBM. Jan. 1981.

[Jaeschke & Schek 1982] Jaeschke, G. and H. Schek. *Remarks on the Algebra of Non First Normal Form Relations*, in *Proceedings of the ACM Symposium on Principles of Database Systems*, 1982, pp. 124-137.

[Jones et al. 1979] Jones, S., P. Mason and R. Stamper. *LEGOL 2.0: a relational specification language for complex rules. Information Systems*, 4, No. 4, Nov. 1979, pp. 293-305.

[Jones & Mason 1980] Jones, S. and P.J. Mason. *Handling the Time Dimension in a Data Base*, in

Proceedings of the International Conference on Data Bases, Ed. S. M. Deen and P. Hammersley. British Computer Society. University of Aberdeen: Heyden, July 1980, pp. 65-83.

[Katz & Lehman 1984] Katz, R.H. and T. Lehman. *Database Support for Versions and Alternatives of Large Design Files. IEEE Transactions on Software Engineering,* SE-10, No. 2, Mar. 1984, pp. 191-200.

[Kawagoe 1985] Kawagoe, K. *Modified Dynamic Hashing,* in *Proceedings of ACM SIGMOD International Conference on Management of Data,* May 1985, pp. 201-213.

[Klopprogge 1981] Klopprogge, M.R. *TERM: An approach to include the time dimension in the entity-relationship model,* in *Proceedings of the Second International Conference on the Entity Relationship Approach,* Oct. 1981.

[Larson 1978] Larson, P. *Dynamic Hashing. BIT,* 18 (1978), pp. 184-201.

[Larson 1980] Larson, P. *Linear Hashing with Partial Expansions,* in *Proceedings of the Conference on Very Large Databases,* 1980, pp. 224-232.

[Larson 1981] Larson, P. *Analysis of Index-Sequential Files with Overflow Chaining. ACM Transactions on Database Systems,* 6, No. 4, Dec. 1981, pp. 671-680.

[Larson 1982] Larson, P. *Performance Analysis of Linear Hashing with Partial Expansions. ACM Transactions on Database Systems,* 7, No. 4, Dec. 1982, pp. 566-587.

[Larson & Ramakrishna 1985] Larson, P. and M. Ramakrishna. *External Perfect Hashing,* in *Proceedings of ACM SIGMOD International Conference on Management of Data,* May 1985, pp. 190-199.

[Litwin 1978] Litwin, W. *Virtual Hashing: A Dynamically Changing Hashing,* in *Proceedings of the Conference on Very Large Databases,* 1978, pp. 517-523.

[Litwin 1980] Litwin, W. *Linear Hashing: A New Tool For File And Table Addressing,* in *Proceedings of the Conference on Very Large Databases,* 1980, pp. 212-223.

[Luk 1983] Luk, W. *On Estimating Block Accesses In Database Organizations. Communications of the Association of Computing Machinery,* 26, No. 11, Nov. 1983, pp. 945-947.

[Lum et al. 1971] Lum, V., P. Yuen and M. Dodd. *Key-to-Address Transform Techniques: A Fundamental Study on Large Existing Formatted Files. Communications of the Association of Computing Machinery,* 14, No. 4, Apr. 1971, pp. 228-239.

[Lum et al. 1984] Lum, V., 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 International Conference on Management of Data,* Ed. B Yormark. Association for Computing Machinery. Boston, MA: June 1984, pp. 115-130.

[Maier 1985] Maier, D. *The Theory of Relational Databases.* Computer Science Press, 1985.

[March & Severance 1977] March, D. and D. Severance. *The Determination of Efficient Record Segmentations and Blocking Factors for Shared Files. ACM Transactions on Database Systems,* 2, No. 3, Sep. 1977, pp. 279-296.

[MCKENZIE 1986] McKenzie, E. *Bibliography: Temporal Databases.* 1986. (Unpublished paper.)

[McKenzie 1986] McKenzie, L.E. and R. Snodgrass. *An Incremental Temporal Relational Algebra.* 1986. (In preparation.)

[Mendelson 1980] Mendelson, H. *A New Approach to the Analysis of Linear Probing Schemes*. *Journal of the Association of Computing Machinery*, 27, No. 2, July 1980, pp. 474-483.

[Mendelson 1982] Mendelson, H. *Analysis of Extensible Hashing*. *IEEE Transactions on Software Engineering*, 8, No. 6, Nov. 1982, pp. 611-619.

[Morris 1968] Morris, R. *Scatter Storage Techniques*. *Communications of the Association of Computing Machinery*, 11, No. 1, Jan. 1968, pp. 38-43.

[Nestor et al. 1982] Nestor, J., W. Wulf and D. Lamb. *IDL - Interface Description Language*. Technical Report. Computer Science Department, Carnegie-Mellon University. June 1982.

[Nievergelt et al. 1984] Nievergelt, J., H. Hinterberger and K. C. Sevcik. *The Grid File: An Adaptable, Symmetric Multikey File Structure*. *ACM Transactions on Database Systems*, 9, No. 1, Mar. 1984, pp. 38-71.

[Ozsoyoglu et al. 1985] Ozsoyoglu, G., Z. Ozsoyoglu and F. Mata. *A Language and a Physical Organization Technique for Summary Tables*, in *Proceedings of ACM SIGMOD International Conference on Management of Data*, 1985, pp. 3-16.

[Ramamohanarao & Sacks-Davis 1984] Ramamohanarao, K. and R. Sacks-Davis. *Recursive Linear Hashing*. *ACM Transactions on Database Systems*, 9, No. 3, Sep. 1984, pp. 369-391.

[Richard 1980] Richard, P. *Evaluation of the Size of a Query Expressed in Relational Algebra*, in *Proceedings of ACM SIGMOD International Conference on Management of Data*, May 1980, pp. 155-163.

[Robinson 1981] Robinson, J. *The K-D-B Tree: A Search Structure for Large Multidimensional Dynamic Indexes*, in *Proceedings of ACM SIGMOD International Conference on Management of Data*, May 1981, pp. 10-18.

[Satyanarayanan 1983] Satyanarayanan, M. *A Methodology for Modelling Storage Systems and its Application to a Network File System*. PhD. Diss. Computer Science Department, Carnegie-Mellon University, Mar. 1983.

[Scheuermann 1977] Scheuermann, P. *Concepts of a data base simulation language*, in *Proceedings of ACM SIGMOD International Conference on Management of Data*, Aug. 1977, pp. 145-155.

[Schueler 1977] Schueler, B. *Update Reconsidered*, in *Architecture and Models in Data Base Management Systems*. Ed. G. M. Nijssen. North Holland Publishing Co., 1977.

[Sernadas 1980] Sernadas, A. *Temporal Aspects of Logical Procedure Definition*. *Information Systems*, 5, No. 3 (1980), pp. 167-187.

[Severance 1975] Severance, D. *A Parametric Model of Alternative File Structures*. *Information Systems*, 1, No. 2 (1975), pp. 51-55.

[Severance 1976] Severance, D. *Differential Files: Their Application to the Maintenance of Large Databases*. *ACM Transactions on Database Systems*, 1, No. 3, Sep. 1976, pp. 256-267.

[Siler 1976] Siler, K. *A Stochastic Evaluation Model for Database Organizations in Data Retrieval Systems*. *Communications of the Association of Computing Machinery*, 19, No. 2, Feb. 1976, pp. 84-95.

[Snodgrass 1982] Snodgrass, R. *Monitoring Distributed Systems: A Relational Approach*. PhD. Diss. Computer Science Department, Carnegie-Mellon University, Dec. 1982.

[Snodgrass & Ahn 1985] Snodgrass, R. and I. Ahn. *A Taxonomy of Time in Databases*, in *Proceedings of ACM SIGMOD International Conference on Management of Data*, Ed. S. Navathe. Association for Computing Machinery. Austin, TX: May 1985, pp. 236-246.

[Snodgrass 1986] Snodgrass, R. *The Temporal Query Language TQuel. ACM Transactions on Database Systems (to appear)*, (1986).

[Snodgrass & Ahn 1986] Snodgrass, R. and I. Ahn. *Temporal Databases. IEEE Computer*, 19, No. 9, Sep. 1986.

[Snodgrass & Gomez 1986] Snodgrass, R. and S. Gomez. *Aggregates in the Temporal Query Language TQuel.* Technical Report TR86-009. Computer Science Department, University of North Carolina at Chapel Hill. Mar. 1986.

[Sprugnoli 1977] Sprugnoli, R. *Perfect hash functions: A single probe retrieving method for static sets. Communications of the Association of Computing Machinery*, 20, No. 11, Nov. 1977, pp. 841-850.

[Stonebraker et al. 1976] Stonebraker, M., E. Wong, P. Kreps and G. Held. *The Design and Implementation of INGRES. ACM Transactions on Database Systems*, 1, No. 3, Sep. 1976, pp. 189-222.

[Stonebraker 1981] Stonebraker, M. *Operating System Support for Database Management. Communications of the Association of Computing Machinery*, 24, No. 7, July 1981, pp. 412-418.

[Stonebraker 1986] Stonebraker, M. *Inclusion of New Types in Relational Data Base Systems*, in *Proceedings of the International Conference on Data Engineering*,, IEEE Computer Society. Los Angeles, CA: IEEE Computer Society Press, Feb. 1986, pp. 262-269.

[Svobodova 1981] Svobodova, L. *A reliable object-oriented data depository for a distributed computer*, in *Proceedings of the ACM Symposium on Operating System Principles*, Dec. 1981, pp. 47-58.

[Teorey & Das 1976] Teorey, T. and K. Das. *Application of an analytical model to evaluate storage structures*, in *Proceedings of ACM SIGMOD International Conference on Management of Data*, June 1976, pp. 9-19.

[Teorey & Fry 1980] Teorey, T. and J. Fry. *The Logical Record Access Approach to Database Design. ACM Computing Surveys*, 12, No. 2, June 1980, pp. 179-211.

[Teorey & Fry 1982] Teorey, T. and J. Fry. *Design of Database Structures.* Prentice-Hall, Inc., 1982.

[Whang et al. 1983] Whang, K. *Estimating Block Accesses in Database Organizations: A Closed Noniterative Formula. Communications of the Association of Computing Machinery*, 26, No. 11, Nov. 1983, pp. 940-944.

[Wiederhold 1981] Wiederhold, G. *Databases for Health Care.* New York, NY: Springer-Verlag, 1981.

[Wiederhold 1984] Wiederhold, G. *Databases. IEEE Computer*, 17, No. 10, Oct. 1984, pp. 211-223.

[Wong & Youssefi 1976] Wong, E. and K. Youssefi. *Decomposition - A Strategy for Query Processing. ACM Transactions on Database Systems*, 1, No. 3, Sep. 1976, pp. 223-240.

[Woodfill et al. 1981] Woodfill, J., P. Siegal, J. Ranstrom, M. Meyer and E. Allman. *Ingres Reference Manual.* Version 7 ed. 1981.

[Yao & Merten 1975] Yao, S. and A. Merten. *Selection Of File Organization Using An Analytic Model*, in *Proceedings of the Conference on Very Large Databases*, Sep. 1975, pp. 255-267.

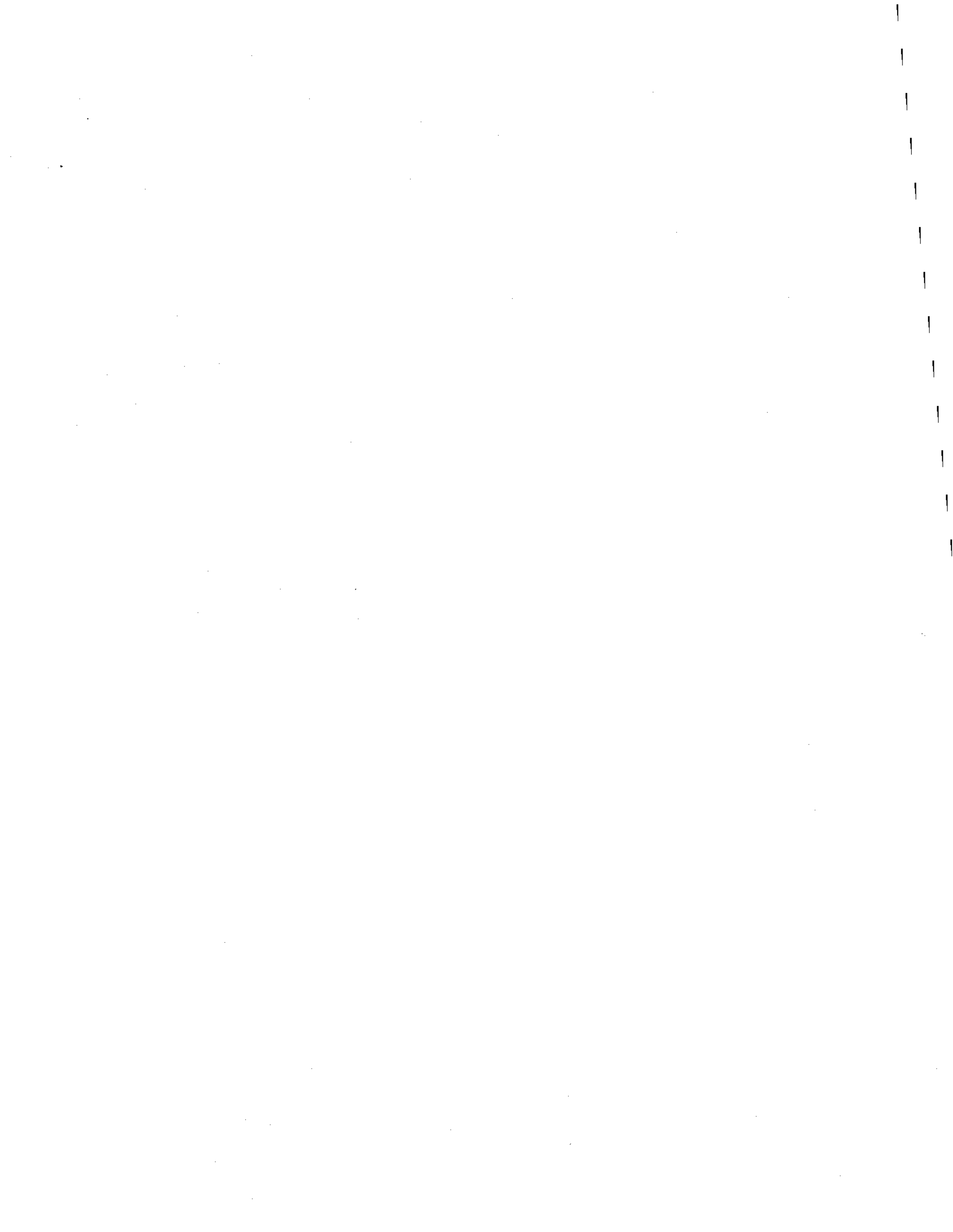
[Yao 1977A] Yao, S. *Approximating Block Accesses in Database Organizations. Communications of the Association of Computing Machinery*, 20, No. 4, Apr. 1977, pp. 260-261.

[Yao 1977B] Yao, S. *An Attribute Based Model for Database Access Cost Analysis. ACM Transactions on Database Systems*, 2, No. 1, Mar. 1977, pp. 45-67.

[Yao & DeJong 1978] Yao, S. and D. Dejong. *Evaluation of Database Access Paths*, in *Proceedings of ACM SIGMOD International Conference on Management of Data*, 1978, pp. 66-77.

[Yao 1979] Yao, S. *Optimization of Query Evaluation Algorithms. ACM Transactions on Database Systems*, 4, No. 2, June 1979, pp. 133-155.

[Yu et al. 1978] Yu, C., W. Luk and M. Siu. *On the Estimation of the Number of Desired Records with Respect to a Given Query. ACM Transactions on Database Systems*, 3, No. 1, Mar. 1978, pp. 41-56.



Appendix A

TQuel Syntax in BNF

TQuel is a superset of Quel, so a legal Quel statement is also a legal TQuel statement. TQuel augments five Quel statements: **retrieve**, **append**, **delete**, **replace**, and **create**. The syntax for these statements are shown below, as defined in the appendix of [Snodgrass 86].

In addition, two Quel statements, **modify** and **index**, have been extended in Chapter 7 to accomodate the temporally partitioned storage structure and secondary indexing for databases with temporal support. We note that the prototype currently supports a very limited subset of the allowed options.

```
<TQuel>            ::=    <retrieve stmt>
                    |     <append stmt>
                    |     <delete stmt>
                    |     <replace stmt>
                    |     <create stmt>

<create stmt>      ::=    create <persistent> <history> <attribute spec>

<persistent>       ::=    ε | persistent

<history>          ::=    ε | interval | event

<retrieve stmt>    ::=    <retrieve head> <retrieve tail>

<retrieve head>    ::=    retrieve <into> <target list> <valid clause>

<retrieve tail>    ::=    <where clause> <when clause> <as of clause>

<into>             ::=    ε | unique | <relation>
                    |     into <relation> | to <relation>

<target list>      ::=    ε | ( <tuple variable>.all ) | ( <t_list> )

<t_list>           ::=    <t_elem> | <t_list> , <t_elem>

<t_elem>           ::=    <attribute> <is> <expression>

<is>               ::=    is | = | by

<append stmt>      ::=    append <to> <target list> <mod tail>
```

```
<to>               ::=    <relation>  |  to  <relation>

<delete stmt>      ::=    delete  <tuple variable>  <mod tail>

<replace stmt>     ::=    replace  <tuple variable>  <target list>  <mod tail>

<mod tail>         ::=    <valid clause>  <where clause>  <when clause>

<valid clause>     ::=    <valid>  <from clause>  <to clause>
                    |     <valid>  <at clause>

<valid>            ::=    ε  |  valid

<from clause>      ::=    ε  |  from  <event expr>

<to clause>        ::=    ε  |  to    <event expr>

<at clause>        ::=    at  <event expr>

<where clause>     ::=    ε  |  where  <boolean expr>

<when clause>      ::=    ε  |  when   <temporal pred>

<as of clause>     ::=    ε  |  as of  <event expr>  <through clause>

<through clause>   ::=    ε  |  through  <event expr>

<event expr>       ::=    <event elem>
                    |     begin of  <either expr>
                    |     end of    <either expr>
                    |     ( <event expr> )

<interval expr>    ::=    <interval elem>
                    |     <either expr>  overlap  <either expr>
                    |     <either expr>  extend   <either expr>
                    |     ( <interval expr> )

<either expr>      ::=    <event expr>  |  <interval expr>

<event elem>       ::=    <tuple variable>  associated with an event relation

<interval elem>    ::=    <tuple variable>  associated with an interval relation
                    |     <temporal const>

<temporal const>   ::=    <string>

<temporal pred>    ::=    <interval elem>
                    |     <event elem>
                    |     <either expr>    precede  <either expr>
                    |     <either expr>    overlap  <either expr>
                    |     <either expr>    equal    <either expr>
                    |     <temporal pred>  and      <temporal pred>
                    |     <temporal pred>  or       <temporal pred>
                    |     ( <temporal pred> )
                    |     not  <temporal pred>

<modify stmt>      ::=    modify  <relation>  to  <store spec>
```

```
                              <on attr>  <parameters>

<store spec>      ::=     isam      |  cisam  |  hash      |  chash
                  |       heap      |  cheap  |  heapsort  |  cheapsort
                  |       truncated
                  |       chain     |  index  |  accessionlist
                  |       cluster   |  stack  |  cellular

<on attr>         ::=     ε  |  on  <key list>

<key list>        ::=     <key order>
                  |       <key list>  ,  <key order>

<key order>       ::=     <attribute>  <order>

<order>           ::=     ε  |  :  <ascend>  |  :  <descend>

<ascend>          ::=     a  |  ascending

<descend>         ::=     d  |  descending

<parameters>      ::=     ε  |  where  <parm list>

<parm list>       ::=     <parm>
                  |       <parm list>  ,  <parm>

<parm>            ::=     fillfactor  =  <integer>
                  |       minpages    =  <integer>
                  |       maxpages    =  <integer>
                  |       cellsize    =  <integer>
                  |       time        =  ( <time list> )

<time list>       ::=     <time attr>
                  |       <time list>  ,  <time attr>

<time attr>       ::=     all
                  |       valid from         |  valid to
                  |       transaction start  |  transaction stop

<index stmt>      ::=     index on  <relation>  is  <index name>
                                ( <attr list> )  <index type>

<attr list>       ::=     <attribute>
                  |       <attr list>  ,  <attribute>

<index type>      ::=     ε  |  as  <temporal type>

<temporal type>  ::=      snapshot  |  rollback  |  historical  |  temporal
```

In this description, the following non-terminals, which are identical to their Quel counterparts, were used:

<relation> the name of a relation

<tuple variable> the name of a tuple variable

`<attribute>`	the name of an attribute
`<attribute spec>`	a list of names and types for the user specified attributes
`<string>`	a string constant
`<boolean expr>`	returns a value of type boolean
`<expression>`	returns a value of type integer, string, floating point, or temporal

Appendix B

Nonlinear Hashing

Algorithms for nonlinear hashing to insert, delete, and retrieve a record given its key K are presented in this appendix. Procedures described here are in bold fonts. The algorithms have been implemented and tested in the C language.

There are two parameters for nonlinear hashing: n_0 and `minLoading`. n_0 is the initial size of the file, and `minLoading` is the minimum loading factor for a block, below which a merge operation is triggered on deleting a record from the block. Procedure **compute** has the parameter `order` which is treated as a *call-by-reference* or a *call-by-result* parameter.

```
(*      Retrieve a record with key K.
 *          parameters
 *              K         : key of the record
 *          return value
 *              record K: when successful
 *              ERROR    : when failed
 *)
function retrieve (K):
begin
        b1 ← compute (K);
        getBlock (b1);              (* read block b1 *)

        if (record K in block b1) then
            return (record K);
        else
            return (ERROR);
end;

(*      Insert a record with key K.
 *          parameters
 *              K         : key of the record
 *          return value
 *              address  : final address of the inserted block
 *)
function insert (K):
begin
        b1 ← compute (K, order);
        getBlock (b1);              (* read block b1 *)

        if (block b1 is full) then
            final ← split (b1, K, order);
```

```
        else                         (* enter record K into block b1 *)
        begin
            final ← b1;
            enterRec (K, b1);
        end;
        return (final);
end;

(*      Delete a record with key K.
 *          parameters
 *              K        : key of the record
 *          return value
 *              OK       : when successful
 *              ERROR    : when failed
 *)
function delete (K):
begin
        b1 ← compute (K, order);
        getBlock (b1);               (* read block b1 *)

        if (K in block b1) then      (* remove record K from block b1 *)
            remove (K, b1);
        else
            return (ERROR);

        if (loading of b1 < minLoading) then
            try_merge (b1);
        return (OK);
end;

(*      Determine the address for key K.
 *          parameters
 *              K        : key of the record
 *              order    : variable parameter for order of overflow
 *          return value
 *              b1       : final address for key K
 *)
function compute (K, VAR order):
begin
        b1 ←  hashFn (K);
        count ← 1;          (* to count the order of overflow *)
        marker ← 0;         (* to mark a position in OverflowList *)

        while (TRUE) do
        begin
            member ← (first b1 or -b1 after the position
                          pointed to by marker in OverflowList);
            if (no such member) then
                break;
            else
            begin
                p1 ← (position of member in OverflowList
                                  without counting negative entries);
                marker ← p1;
            end;
```

```
            (* b1 had an overflow *)
            if (splitFn (K, count) = 0) then
                ;                           (* original block *)

            else
            begin                           (* child block *)
                if (member < 0) then        (* merged back *)
                    break;
                else
                    b1 ← child (b1, p1);
            end;

            count ← count + 1;
        end;

        order ← count;
        return (b1);
end;

(*      Hash function.
 *          parameters
 *              K           : key
 *          return value
 *              hashed address : { 1 .. n0 }
 *)
function hashFn (K):
begin
        return (K mod n0+1);
        (* Addresses start from 1
         * so that they can be negated for merging *)
end;

(*      Split functions.
 *          parameters
 *              K           : key
 *              ord         : order of overflow
 *          return value
 *              split value : { 0, 1 }
 *)
function splitFn (K, ord):
begin
        return (K / (n0 × 2^(ord-1)) mod 2);
end;

(*      Split block b1 into two.
 *          parameters
 *              b1          : address of the block to be split
 *              Ki          : key of the record to be inserted
 *              order       : order of overflow
 *          return value
 *              address     : final address for record Ki
 *)
function split (b1, Ki, order):
begin
        append b1 to OverflowList;
```

```
        append a new block b2 at the end;

        for each record Kr in block b1 do
        begin
                if (splitFn (Kr, order) = 0) then
                        enterRec (Kr, b1);
                else
                        enterRec (Kr, b2);
        end;

        if (splitFn (Ki, order) = 0) then
        begin
                enterRec (Ki, b1);
                return (b1);
        end;

        else
        begin
                enterRec (Ki, b2);
                return (b2);
        end;
end;

(*      Try merging block b1 with its parent or child.
 *          parameters
 *              b1          : address of the block
 *          return value
 *              OK          : when successful
 *              ERROR       : when failed
 *)
function try_merge (b1):
begin
        young ← TRUE;
        if (leaf (b1))
        begin
            if (parent (b1) = ERROR) then
                return (ERROR);             /* one of initial blocks */

            if (b1 is not the youngest child) then
                young ← FALSE;

            return (merge (b1, parent (b1), young));
        end

        (* b1 is not a leaf *)
        children ← (a list of all child of b1 in reverse order);

        (* youngest child is at the head of the list *)
        for each child ch1 in children do
        begin
            if (merge (ch1, b1, young) = OK) then
                return (OK);
            young ← FALSE;
        end;

        return (ERROR);
```

```
end;

(*      Merge block b1 into block b2.
 *          parameters
 *              b1          : address of block b1
 *              b2          : address of block b2
 *              young       : TRUE if b1 is the youngest child
 *          return value
 *              OK          : when successful
 *              ERROR       : when failed
 *)
function merge (b1, b2, young):
begin
        if (block b2 has room for all records in block b1) then
        begin
            move all records of block b1 to b2;
            discard block b1;
            adjust addresses for blocks whose address is higher than b1;

            if (young = TRUE) then
                remove b2 from OverflowList;
            else
                negate b2 in OverflowList;
            return (OK);
        end;
        else  return (ERROR);
end;

(*      Find the the child of block b1.
 *          parameters
 *              b1          : address of the block
 *              p1          : position of b1 in OverflowList
 *          return value
 *              address of the child block
 *)
function child (b1, p1):
begin
        return (p1+n0);
end;

(*      Find the the parent of block b1.
 *          parameters
 *              b1          : address of the block
 *          return value
 *              address of the parent block
 *              ERROR       : when there is none
 *)
function parent (b1):
begin
        if (b1<n0) then  return (ERROR);
        else  return (OverflowList [b1-n0]);
end;

(*      Check if block b1 is a leaf.
 *          parameters
 *              b1          : address of the block
```

```
 *          return value
 *              TRUE     : when b1 is a leaf
 *              FALSE    : when b1 is not a leaf
 *)
function leaf (b1):
begin
        if (there is b1 in OverflowList) then
            return (FALSE);
        else
            return (TRUE);
end;
```

Appendix C

Benchmark Results

This appendix presents the measurement data from the benchmark discussed in Chapter 6.

Query	Update Count															
	0		1		2		3		4		5		6		7	
	In	Out	In	Out	In	Out	In	Out	In	Out	In	Out	In	Out	In	Out
Q01	1	0	2	0	3	0	4	0	5	0	6	0	7	0	8	0
Q02	2	0	3	0	4	0	5	0	6	0	7	0	8	0	9	0
Q03	129	0	258	0	387	0	516	0	645	0	774	0	903	0	1024	0
Q04	128	0	256	0	384	0	512	0	640	0	768	0	896	0	1024	0
Q05	1	0	2	0	3	0	4	0	5	0	6	0	7	0	8	0
Q06	2	0	3	0	4	0	5	0	6	0	7	0	8	0	9	0
Q07	129	0	258	0	387	0	516	0	645	0	774	0	903	0	1024	0
Q08	128	0	256	0	384	0	512	0	640	0	768	0	896	0	1024	0
Q09	1141	0	2304	0	3456	0	4608	0	5760	0	6912	0	8064	0	9178	0
Q10	2177	0	3330	0	4483	0	5636	0	6789	0	7942	0	9095	0	10240	0
Q15	129	0	258	0	387	0	516	0	645	0	774	0	903	0	1024	0
Query	Update Count															
	8		9		10		11		12		13		14		15	
	In	Out	In	Out	In	Out	In	Out	In	Out	In	Out	In	Out	In	Out
Q01	9	0	10	0	11	0	12	0	13	0	14	0	15	0	16	0
Q02	10	0	11	0	12	0	13	0	14	0	15	0	16	0	17	0
Q03	1153	0	1282	0	1411	0	1540	0	1669	0	1798	0	1927	0	2048	0
Q04	1152	0	1280	0	1408	0	1536	0	1664	0	1792	0	1920	0	2048	0
Q05	9	0	10	0	11	0	12	0	13	0	14	0	15	0	16	0
Q06	10	0	11	0	12	0	13	0	14	0	15	0	16	0	17	0
Q07	1153	0	1282	0	1411	0	1540	0	1669	0	1798	0	1927	0	2048	0
Q08	1152	0	1280	0	1408	0	1536	0	1664	0	1792	0	1920	0	2048	0
Q09	10330	0	11482	0	12634	0	13786	0	14938	0	16090	0	17242	0	18356	0
Q10	11393	0	12546	0	13699	0	14852	0	16005	0	17158	0	18311	0	19456	0
Q15	1153	0	1282	0	1411	0	1540	0	1669	0	1798	0	1927	0	2048	0

Figure C-1: I/O Cost for the Rollback DBMS with 100% Loading

	Update Count															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Rollback_h	129	258	387	516	645	774	903	1024	1153	1282	1411	1540	1669	1798	1927	2048
Rollback_i	129	257	385	513	641	769	897	1025	1153	1281	1409	1537	1665	1793	1921	2049
sum	258	515	772	1029	1286	1543	1800	2049	2306	2563	2820	3077	3334	3591	3848	4097

Figure C-2: Space for the Rollback DBMS with 100% Loading

Query	Update Count															
	0		1		2		3		4		5		6		7	
	In	Out	In	Out	In	Out	In	Out	In	Out	In	Out	In	Out	In	Out
Q01	1	0	1	0	2	0	2	0	3	0	3	0	4	0	4	0
Q02	3	0	3	0	4	0	4	0	5	0	5	0	6	0	6	0
Q03	257	0	257	0	514	0	514	0	767	0	771	0	1024	0	1024	0
Q04	256	0	256	0	512	0	512	0	768	0	768	0	1024	0	1024	0
Q05	1	0	1	0	2	0	2	0	3	0	3	0	4	0	4	0
Q06	3	0	3	0	4	0	4	0	5	0	5	0	6	0	6	0
Q07	257	0	257	0	514	0	514	0	767	0	771	0	1024	0	1024	0
Q08	256	0	256	0	512	0	512	0	768	0	768	0	1024	0	1024	0
Q09	1271	0	1271	0	2560	0	2560	0	3840	0	3840	0	5120	0	5120	0
Q10	3329	0	3329	0	4610	0	4610	0	5887	0	5891	0	7168	0	7168	0
Q15	257	0	257	0	514	0	514	0	767	0	771	0	1024	0	1024	0

Query	Update Count															
	8		9		10		11		12		13		14		15	
	In	Out	In	Out	In	Out	In	Out	In	Out	In	Out	In	Out	In	Out
Q01	5	0	5	0	6	0	6	0	7	0	7	0	8	0	8	0
Q02	7	0	7	0	8	0	8	0	9	0	9	0	10	0	10	0
Q03	1281	0	1281	0	1538	0	1538	0	1791	0	1795	0	2048	0	2048	0
Q04	1280	0	1280	0	1536	0	1536	0	1792	0	1792	0	2048	0	2048	0
Q05	5	0	5	0	6	0	6	0	7	0	7	0	8	0	8	0
Q06	7	0	7	0	8	0	8	0	9	0	9	0	10	0	10	0
Q07	1281	0	1281	0	1538	0	1538	0	1791	0	1795	0	2048	0	2048	0
Q08	1280	0	1280	0	1536	0	1536	0	1792	0	1792	0	2048	0	2048	0
Q09	6400	0	6400	0	7680	0	7680	0	8960	0	8960	0	10240	0	10240	0
Q10	8449	0	8449	0	9730	0	9730	0	11007	0	11011	0	12288	0	12288	0
Q15	1281	0	1281	0	1538	0	1538	0	1791	0	1795	0	2048	0	2048	0

Figure C-3: I/O Cost for the Rollback DBMS with 50% Loading

	Update Count															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Rollback_h	257	257	514	514	767	771	1024	1024	1281	1281	1538	1538	1791	1795	2048	2048
Rollback_i	259	259	515	515	771	771	1027	1027	1283	1283	1539	1539	1795	1795	2051	2051
sum	516	516	1029	1029	1538	1542	2051	2051	2564	2564	3077	3077	3586	3590	4099	4099

Figure C-4: Space for the Rollback DBMS with 50% Loading

Query	Update Count															
	0		1		2		3		4		5		6		7	
	In	Out	In	Out	In	Out	In	Out	In	Out	In	Out	In	Out	In	Out
Q01	1	0	2	0	3	0	4	0	5	0	6	0	7	0	8	0
Q02	2	0	3	0	4	0	5	0	6	0	7	0	8	0	9	0
Q05	1	0	2	0	3	0	4	0	5	0	6	0	7	0	8	0
Q06	2	0	3	0	4	0	5	0	6	0	7	0	8	0	9	0
Q07	129	0	258	0	387	0	516	0	645	0	774	0	903	0	1024	0
Q08	128	0	256	0	384	0	512	0	640	0	768	0	896	0	1024	0
Q09	1197	56	2360	56	3512	56	4664	56	5816	56	6968	56	8120	56	9234	56
Q10	2233	56	3386	56	4539	56	5692	56	6845	56	7998	56	9151	56	10296	56
Q13	1	0	2	0	3	0	4	0	5	0	6	0	7	0	8	0
Q14	129	0	258	0	387	0	516	0	645	0	774	0	903	0	1024	0

Query	Update Count															
	8		9		10		11		12		13		14		15	
	In	Out	In	Out	In	Out	In	Out	In	Out	In	Out	In	Out	In	Out
Q01	9	0	10	0	11	0	12	0	13	0	14	0	15	0	16	0
Q02	10	0	11	0	12	0	13	0	14	0	15	0	16	0	17	0
Q05	9	0	10	0	11	0	12	0	13	0	14	0	15	0	16	0
Q06	10	0	11	0	12	0	13	0	14	0	15	0	16	0	17	0
Q07	1153	0	1282	0	1411	0	1540	0	1669	0	1798	0	1927	0	2048	0
Q08	1152	0	1280	0	1408	0	1536	0	1664	0	1792	0	1920	0	2048	0
Q09	10386	56	11538	56	12690	56	13842	56	14994	56	16146	56	17298	56	18412	56
Q10	11449	56	12602	56	13755	56	14908	56	16061	56	17214	56	18367	56	19512	56
Q13	9	0	10	0	11	0	12	0	13	0	14	0	15	0	16	0
Q14	1153	0	1282	0	1411	0	1540	0	1669	0	1798	0	1927	0	2048	0

Figure C-5: I/O Cost for the Historical DBMS with 100% Loading

	Update Count															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Historical_h	129	258	387	516	645	774	903	1024	1153	1282	1411	1540	1669	1798	1927	2048
Historical_i	129	257	385	513	641	769	897	1025	1153	1281	1409	1537	1665	1793	1921	2049
sum	258	515	772	1029	1286	1543	1800	2049	2306	2563	2820	3077	3334	3591	3848	4097

Figure C-6: Space for the Historical DBMS with 100% Loading

Query	Update Count															
	0		1		2		3		4		5		6		7	
	In	Out	In	Out	In	Out	In	Out	In	Out	In	Out	In	Out	In	Out
Q01	1	0	1	0	2	0	2	0	3	0	3	0	4	0	4	0
Q02	3	0	3	0	4	0	4	0	5	0	5	0	6	0	6	0
Q05	1	0	1	0	2	0	2	0	3	0	3	0	4	0	4	0
Q06	3	0	3	0	4	0	4	0	5	0	5	0	6	0	6	0
Q07	257	0	257	0	514	0	514	0	767	0	771	0	1024	0	1024	0
Q08	256	0	256	0	512	0	512	0	768	0	768	0	1024	0	1024	0
Q09	1327	56	1327	56	2616	56	2616	56	3896	56	3896	56	5176	56	5176	56
Q10	3385	56	3385	56	4666	56	4666	56	5943	56	5947	56	7224	56	7224	56
Q13	1	0	1	0	2	0	2	0	3	0	3	0	4	0	4	0
Q14	257	0	257	0	514	0	514	0	767	0	771	0	1024	0	1024	0

Query	Update Count															
	8		9		10		11		12		13		14		15	
	In	Out	In	Out	In	Out	In	Out	In	Out	In	Out	In	Out	In	Out
Q01	5	0	5	0	6	0	6	0	7	0	7	0	8	0	8	0
Q02	7	0	7	0	8	0	8	0	9	0	9	0	10	0	10	0
Q05	5	0	5	0	6	0	6	0	7	0	7	0	8	0	8	0
Q06	7	0	7	0	8	0	8	0	9	0	9	0	10	0	10	0
Q07	1281	0	1281	0	1538	0	1538	0	1791	0	1795	0	2048	0	2048	0
Q08	1280	0	1280	0	1536	0	1536	0	1792	0	1792	0	2048	0	2048	0
Q09	6456	56	6456	56	7736	56	7736	56	9016	56	9016	56	10296	56	10296	56
Q10	8505	56	8505	56	9786	56	9786	56	11063	56	11067	56	12344	56	12344	56
Q13	5	0	5	0	6	0	6	0	7	0	7	0	8	0	8	0
Q14	1281	0	1281	0	1538	0	1538	0	1791	0	1795	0	2048	0	2048	0

Figure C-7: I/O Cost for the Historical DBMS with 50% Loading

	Update Count															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Historical_h	257	257	514	514	767	771	1024	1024	1281	1281	1538	1538	1791	1795	2048	2048
Historical_i	259	259	515	515	771	771	1027	1027	1283	1283	1539	1539	1795	1795	2051	2051
sum	516	516	1029	1029	1538	1542	2051	2051	2564	2564	3077	3077	3586	3590	4099	4099

Figure C-8: Space for the Historical DBMS with 50% Loading

Query	Update Count															
	0		1		2		3		4		5		6		7	
	In	Out	In	Out	In	Out	In	Out	In	Out	In	Out	In	Out	In	Out
Q01	1	0	3	0	5	0	7	0	9	0	11	0	13	0	15	0
Q02	2	0	4	0	6	0	8	0	10	0	12	0	14	0	16	0
Q03	129	0	387	0	645	0	903	0	1153	0	1411	0	1669	0	1927	0
Q04	128	0	384	0	640	0	896	0	1152	0	1408	0	1664	0	1920	0
Q05	1	0	3	0	5	0	7	0	9	0	11	0	13	0	15	0
Q06	2	0	4	0	6	0	8	0	10	0	12	0	14	0	16	0
Q07	129	0	387	0	645	0	903	0	1153	0	1411	0	1669	0	1927	0
Q08	128	0	384	0	640	0	896	0	1152	0	1408	0	1664	0	1920	0
Q09	1200	56	3512	56	5816	56	8120	56	10386	56	12690	56	14994	56	17298	56
Q10	2233	56	4539	56	6845	56	9151	56	11449	56	13755	56	16061	56	18367	56
Q11	385	0	1155	0	1925	0	2695	0	3457	0	4227	0	4997	0	5767	0
Q12	131	4	389	4	647	4	905	4	1163	4	1421	4	1679	4	1937	4
Q13	1	0	3	0	5	0	7	0	9	0	11	0	13	0	15	0
Q14	129	0	387	0	645	0	903	0	1153	0	1411	0	1669	0	1927	0
Q15	129	0	387	0	645	0	903	0	1153	0	1411	0	1669	0	1927	0
Q16	129	0	387	0	645	0	903	0	1153	0	1411	0	1669	0	1927	0
Query	Update Count															
	8		9		10		11		12		13		14		15	
	In	Out	In	Out	In	Out	In	Out	In	Out	In	Out	In	Out	In	Out
Q01	17	0	19	0	21	0	23	0	25	0	27	0	29	0	31	0
Q02	18	0	20	0	22	0	24	0	26	0	28	0	30	0	32	0
Q03	2177	0	2435	0	2693	0	2951	0	3201	0	3459	0	3717	0	3975	0
Q04	2176	0	2432	0	2688	0	2944	0	3200	0	3456	0	3712	0	3968	0
Q05	17	0	19	0	21	0	23	0	25	0	27	0	29	0	31	0
Q06	18	0	20	0	22	0	24	0	26	0	28	0	30	0	32	0
Q07	2177	0	2435	0	2693	0	2951	0	3201	0	3459	0	3717	0	3975	0
Q08	2176	0	2432	0	2688	0	2944	0	3200	0	3456	0	3712	0	3968	0
Q09	19564	56	21868	56	24172	56	26476	56	28742	56	31046	56	33350	56	35654	56
Q10	20665	56	22971	56	25277	56	27583	56	29881	56	32187	56	34493	56	36799	56
Q11	6529	0	7299	0	8069	0	8839	0	9601	0	10371	0	11141	0	11911	0
Q12	2195	4	2453	4	2711	4	2969	4	3227	4	3485	4	3743	4	4001	4
Q13	17	0	19	0	21	0	23	0	25	0	27	0	29	0	31	0
Q14	2177	0	2435	0	2693	0	2951	0	3201	0	3459	0	3717	0	3975	0
Q15	2177	0	2435	0	2693	0	2951	0	3201	0	3459	0	3717	0	3975	0
Q16	2177	0	2435	0	2693	0	2951	0	3201	0	3459	0	3717	0	3975	0

Figure C-9: I/O Cost for the Temporal DBMS with 100% Loading

	Update Count															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Temporal_h	129	387	645	903	1153	1411	1669	1927	2177	2435	2693	2951	3201	3459	3717	3975
Temporal_i	129	385	641	897	1153	1409	1665	1921	2177	2433	2689	2945	3201	3457	3713	3969
sum	258	772	1286	1800	2306	2820	3334	3848	4354	4868	5382	5896	6402	6916	7430	7944

Figure C-10: Space for the Temporal DBMS with 100% Loading

Query	Update Count															
	0		1		2		3		4		5		6		7	
	In	Out	In	Out	In	Out	In	Out	In	Out	In	Out	In	Out	In	Out
Q01	1	0	2	0	3	0	4	0	5	0	6	0	7	0	8	0
Q02	3	0	4	0	5	0	6	0	7	0	8	0	9	0	10	0
Q03	257	0	514	0	767	0	1024	0	1281	0	1538	0	1791	0	2048	0
Q04	256	0	512	0	768	0	1024	0	1280	0	1536	0	1792	0	2048	0
Q05	1	0	2	0	3	0	4	0	5	0	6	0	7	0	8	0
Q06	3	0	4	0	5	0	6	0	7	0	8	0	9	0	10	0
Q07	257	0	514	0	767	0	1024	0	1281	0	1538	0	1791	0	2048	0
Q08	256	0	512	0	768	0	1024	0	1280	0	1536	0	1792	0	2048	0
Q09	1333	56	2616	56	3896	56	5176	56	6456	56	7736	56	9016	56	10296	56
Q10	3385	56	4666	56	5943	56	7224	56	8505	56	9786	56	11063	56	12344	56
Q11	769	0	1538	0	2303	0	3072	0	3841	0	4610	0	5375	0	6144	0
Q12	259	4	516	4	773	4	1030	4	1287	4	1544	4	1801	4	2058	4
Q13	1	0	2	0	3	0	4	0	5	0	6	0	7	0	8	0
Q14	257	0	514	0	767	0	1024	0	1281	0	1538	0	1791	0	2048	0
Q15	257	0	514	0	767	0	1024	0	1281	0	1538	0	1791	0	2048	0
Q16	257	0	514	0	767	0	1024	0	1281	0	1538	0	1791	0	2048	0

Query	Update Count															
	8		9		10		11		12		13		14		15	
	In	Out	In	Out	In	Out	In	Out	In	Out	In	Out	In	Out	In	Out
Q01	9	0	10	0	11	0	12	0	13	0	14	0	15	0	16	0
Q02	11	0	12	0	13	0	14	0	15	0	16	0	17	0	18	0
Q03	2305	0	2562	0	2815	0	3072	0	3329	0	3586	0	3839	0	4096	0
Q04	2304	0	2560	0	2816	0	3072	0	3328	0	3584	0	3840	0	4096	0
Q05	9	0	10	0	11	0	12	0	13	0	14	0	15	0	16	0
Q06	11	0	12	0	13	0	14	0	15	0	16	0	17	0	18	0
Q07	2305	0	2562	0	2815	0	3072	0	3329	0	3586	0	3839	0	4096	0
Q08	2304	0	2560	0	2816	0	3072	0	3328	0	3584	0	3840	0	4096	0
Q09	11576	56	12856	56	14136	56	15416	56	16696	56	17976	56	19256	56	20536	56
Q10	13625	56	14906	56	16183	56	17464	56	18745	56	20026	56	21303	56	22584	56
Q11	6913	0	7682	0	8447	0	9216	0	9985	0	10754	0	11519	0	12288	0
Q12	2315	4	2572	4	2829	4	3086	4	3343	4	3600	4	3857	4	4114	4
Q13	9	0	10	0	11	0	12	0	13	0	14	0	15	0	16	0
Q14	2305	0	2562	0	2815	0	3072	0	3329	0	3586	0	3839	0	4096	0
Q15	2305	0	2562	0	2815	0	3072	0	3329	0	3586	0	3839	0	4096	0
Q16	2305	0	2562	0	2815	0	3072	0	3329	0	3586	0	3839	0	4096	0

Figure C-11: I/O Cost for the Temporal DBMS with 50% Loading

	Update Count															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Temporal_h	257	514	767	1024	1281	1538	1791	2048	2305	2562	2815	3072	3329	3586	3839	4096
Temporal_i	259	515	771	1027	1283	1539	1795	2051	2307	2563	2819	3075	3331	3587	3843	4099
sum	516	1029	1538	2051	2564	3077	3586	4099	4612	5125	5634	6147	6660	7173	7682	8195

Figure C-12: Space for the Temporal DBMS with 50% Loading

Appendix D

Performance Analysis (1)

Cost of each query in Figure 6.5 is analyzed using the four models discussed in Chapter 4. We assume that the queries are executed on the temporal database with 100% loading, as described in Section 6.2.1. The database can be described as follows in IDL's ASCII external representation [Nestor et al. 1982] according to the *model of database/relations*, where *uc* denotes the update count, either 0 or 14.

```
database
[    name "Temporal_100";
     relations
     {
          relation
          [    name                    "Temporal_h";
               temporalType            temporalInterval;
               attributes
               <    A1:       attribute
                    [    name          "Id";
                         type          typeInteger;
                         length        4;
                         selectivity   1/1024;
                         volatility    0;
                    ]
                    A2:       attribute
                    [    name          "Amount";
                         type          typeInteger;
                         length        4;
                         selectivity   1/1024;
                         volatility    0;
                    ]
                    A3:       attribute
                    [    name          "Seq";
                         type          typeInteger;
                         length        4;
                         selectivity   1;
                         volatility    1;
                    ]
                    A4:       attribute
                    [    name          "String";
                         type          typeString;
                         length        96;
                         selectivity   0;
                         volatility    0;
```

```
        ]
    >;
    tupleSize            108;
    tupleCount           1024;
    updateCount          uc;
    storageType          Hash;
    keys
    <    key
         [    name       "hash_key";
              attributes <  A1^;  >
         ]
    >;
    loadingFactor        1;
    blockSize            1024;
]
relation
[   name                 "Temporal_i";
    temporalType         temporalInterval;
    attributes
    <    A1:       attribute
         [    name        "Id";
              type        typeInteger;
              length      4;
              selectivity 1/1024;
              volatility  0;
         ]
         A2:       attribute
         [    name        "Amount";
              type        typeInteger;
              length      4;
              selectivity 1/1024;
              volatility  0;
         ]
         A3:       attribute
         [    name        "Seq";
              type        typeInteger;
              length      4;
              selectivity 1;
              volatility  1;
         ]
         A4:       attribute
         [    name        "String";
              type        typeString;
              length      96;
              selectivity 0;
              volatility  0;
         ]
    >;
    tupleSize            108;
    tupleCount           1024;
    updateCount          uc;
    storageType          Isam;
    keys
    <    key
         [    name       "isam_key";
```

```
                        attributes    <  A1^;  >
                    ]
                >;
                loadingFactor          1;
                blockSize              1024;
            ]
        }
]
#
```

The cost to process a query can be analyzed using the four models developed in Section 4.1. A TQuel query is represented by an *algebraic expression*, which is mapped to the *file primitive expression* according to the *model of algebraic expressions* and the *model of database/relations*. Then the *model of access paths* maps the file primitive expression into the *access path expression*, which is converted to the elapsed time according to the *model of storage devices*. The analysis here was performed manually, and subscripts *i* and *o*, as in APE_i and APE_o, denote input and output, respectively.

§ Q01

```
        retrieve (h.id, h.seq)    where    h.id = 500
```

* Algebraic Expression

```
{[ L1: Select     (h, h.id = 500);
       Project    (L1, h.id, h.seq)          ]}
```

• if $uc = 0$

* File Primitive Expression

```
        Read  (Hash, 0)
```

* Access Path Expression

APE_i :

```
        (H 1)
```

* Access Path Cost

$APC_i = C(APE_i) = C((\text{H } 1))$

$= 1$ *random access* $= 31.3$ *msec*

• if $uc = 14$

* File Primitive Expression

```
        Read  (Hash, 28)
```

* Access Path Expression

APE_i :

```
        (H 1 (P 28 (S 1) (P 1)))
```

* Access Path Cost

$APC_i = C(APE_i) = C((\text{H } 1\ (\text{P } 28\ (\text{S } 1)\ (\text{P } 1))))$

$= 29$ *random accesses* ≈ 908 *msec*

§ Q02:

```
retrieve (i.id, i.seq)   where   i.id = 500
```

* Algebraic Expression

```
{[ L1: Select    (i, i.id = 500);
       Project   (L1, i.id, i.seq)        ]}
```

• if $uc = 0$

* File Primitive Expression

```
Read (Isam, 0)
```

* Access Path Expression

APE_i :

```
(P 1 (P 1))
```

* Access Path Cost

$APC_i = C\,(APE_i) = 2$ *random accesses* $= 62.6$ *msec*

• if $uc = 14$

* File Primitive Expression

```
Read (Isam, 28)
```

* Access Path Expression

APE_i :

```
(P 29 (P 1))
```

* Access Path Cost

$APC_i = C\,(APE_i) = 30$ *random accesses* $= 939$ *msec*

§ Q03:

```
retrieve (h.id, h.seq)   as of   "08:00 1/1/80"
```

* Algebraic Expression

```
{[ L1: AsOf      (h, "08:00 1/1/80", "08:00 1/1/80");
       Project   (L1, h.id, h.seq)        ]}
```

• if $uc = 0$

* File Primitive Expression

```
Read (Heap, 128)
```

* Access Path Expression

APE_i :

```
(U 128)
```

* Access Path Cost

$APC_i = C\,(APE_i)$

$= 1$ *random access* $+ 127$ *sequential accesses* $\approx 2{,}370$ *msec*

• if $uc = 14$

* File Primitive Expression

```
Read (Heap, 3712)
```

* Access Path Expression
  APE_i :
  ```
  (U 3712)
  ```

* Access Path Cost
  APC_i $= C(APE_i)$
  $= 1$ *random access* $+ 3{,}711$ *sequential accesses* $\approx 68{,}300$ *msec*

§ Q04:

```
retrieve (i.id, i.seq)    as of    "08:00 1/1/80"
```

* Algebraic Expression
  ```
  {[ L1: AsOf       (i, "08:00 1/1/80", "08:00 1/1/80");
         Project    (L1, i.id, i.seq)          ]}
  ```

• the rest is the same as Q03.

§ Q05:

```
retrieve (h.id, h.seq)    where    h.id = 500
         when     h overlap "now"
```

* Algebraic Expression
  ```
  {[ L1: Select     (h, h.id = 500);
     L2:When        (L1, h overlap "now");
         Project    (L2, h.id, h.seq)          ]}
  ```

• the rest is the same as Q01.

§ Q06:

```
retrieve (i.id, i.seq)    where    i.id = 500
         when     i overlap "now"
```

* Algebraic Expression
  ```
  {[ L1: Select     (i, id = 500);
     L2: When       (L1, L1 overlap "now");
         Project    (L2, L2.id, L2.seq)        ]}
  ```

• the rest is the same as Q02.

§ Q07:

```
retrieve (h.id, h.seq)    where    h.amount = 69400
         when     h overlap "now"
```

* Algebraic Expression
  ```
  {[ L1: Select     (h, h.amount = 69400);
     L2: When       (L1, h overlap "now");
         Project    (L2, h.id, h.seq)          ]}
  ```

• the rest is the same as Q03.

§ Q08:

```
retrieve (i.id, i.seq)    where    i.amount = 73700
         when     i overlap "now"
```

*   Algebraic Expression

```
{[ L1: Select     (i, i.amount = 69400);
   L2: When       (L1, i overlap "now");
       Project    (L2, i.id, i.seq)        ]}
```

• the rest is the same as Q04.

§ Q09

```
retrieve (h.id, i.id, i.amount)
         where    h.id = i.amount
         when     h  overlap  i    and     i  overlap "now"
```

*   Algebraic Expression

```
{[L1: When        (i, i overlap "now");
  L2: Project     (L1, i.id, i.amount,
                          i.valid_from, i.valid_to)];
  L3: Temporary   (L2);
 [L4: Join        (h, L3, TS, h.id = i.amount & h overlap i);
      Project     (L4, h.id, i.id, i.amount)          ]}
```

• if $uc = 0$

*   File Primitive Expression

```
          Read  (Heap, 128)               +
        ( Read  (Heap, 19) * 2 - 1        +
          Write (Heap, 19) * 3 - 1 )      +
          Read  (Heap, 19)                +
          Read  (Hash, 0) * 1024)
```

*   Access Path Expression

APE_i :

```
        (U 128)              +
        (U 19)  * 2 - 1      +
        (U 19)               +
        (H 1)  * 1024
```

APE_o :

(U 19) * 3 - 1

*   Access Path Cost

APC_i = C ((U 128)) + C ((U 19) * 2 - 1) + C ((U 19)) + C ((H 1)) * 1024
= 1,028 *random accesses* + 180 *sequential accesses* ≈ 35,500 *msec*

APC_o = C ((U 19) * 3 - 1)
= 3 *random accesses* + 53 *sequential accesses* ≈ 1,070 *msec*

• if $uc = 14$

*   File Primitive Expression

```
          Read  (Heap, 3712)              +
        ( Read  (Heap, 19) * 2 - 1        +
          Write (Heap, 19) * 3 - 1 )      +
          Read  (Heap, 19)                +
```

```
Read  (Hash, 28) * 1024
```

* Access Path Expression

APE_i :

```
(U 3712)                        +
(U 19) * 2 - 1                  +
(U 19)                          +
(H 1 (P 28 (S 1) (P 1))) * 1024
```

APE_o :

(U 19) * 3 - 1

* Access Path Cost

APC_i = C ((U 3712)) + C ((U 19) * 2 - 1) + C ((U 19)) + C ((H 1 (P 28 (S 1) (P 1)))) * 1024
= 29,700 *random accesses* + 3,764 *sequential accesses* ≈ 999,000 *msec*

APC_o = C ((U 19) * 3 - 1)
= 3 *random accesses* + 53 *sequential accesses* ≈ 1,070 *msec*

§ Q10:

```
retrieve (i.id, h.id, h.amount)  where   i.id = h.amount
         when    h overlap i     and     h overlap "now"
```

* Algebraic Expression

```
{[L1: When       (h, h overlap "now");
  L2: Project    (L1, h.id, h.amount,
                          h.valid_from, h.valid_to)];
  L3: Temporary  (L2);
 [L4: Join       (i, L3, TS, i.id = h.amount & h overlap i);
      Project    (L4, i.id, h.id, h.amount)          ]}
```

• if $uc = 0$

* File Primitive Expression

```
   Read  (Heap, 128)              +
 ( Read  (Heap, 19) * 2 - 1       +
   Write (Heap, 19) * 3 - 1 )     +
   Read  (Heap, 19)               +
   Read  (Isam, 0) * 1024)
```

* Access Path Expression

APE_i :

```
(U 128)                 +
(U 19) * 2 - 1          +
(U 19)                  +
(P 1 (P 1)) * 1024
```

APE_o :

(U 19) * 3 - 1

* Access Path Cost

APC_i = C (APE_i)
= 2,052 *random accesses* + 180 *sequential accesses* ≈ 67,500 *msec*

APC_o = C ((U 19) * 3 - 1)
= 3 *random accesses* + 53 *sequential accesses* ≈ 1,070 *msec*

• if $uc = 14$

* File Primitive Expression

```
    Read  (Heap, 3712)          +
  ( Read  (Heap, 19) * 2 - 1    +
    Write (Heap, 19) * 3 - 1 )  +
    Read  (Heap, 19)            +
    Read  (Isam, 28) * 1024
```

* Access Path Expression

APE_i :

```
(U 3712)            +
(U 19) * 2 - 1      +
(U 19)              +
(P 29 (P 1)) * 1024
```

APE_o :

```
(U 19) * 3 - 1
```

* Access Path Cost

APC_i = $C(APE_i)$
= 30,724 *random accesses* + 3,764 *sequential accesses* ≈ 1,030,000 *msec*

APC_o = C ((U 19) * 3 - 1)
= 3 *random accesses* + 53 *sequential accesses* ≈ 1,070 *msec*

§ Q11:

```
retrieve (h.id, h.seq, i.id, i.seq, i.amount)
         valid   from begin of h        to end of i
         when    begin of h precede i
         as of   "4:00 1/1/80"
```

* Algebraic Expression

```
{ L1: AsOf    (h, "4:00 1/1/80", "4:00 1/1/80");
 [L2: Join    (L1, i, TS, beignOf (h) precede i);
  (   Project (L2, h.id, h.seq, i.id, i.seq, i.amount),
      Valid   (L2, From, beginOf (h)),
      Valid   (L2, To,   endOf   (i))
  )]    }
```

• if $uc = 0$

* File Primitive Expression

```
Read  (Heap, 128)      +
Read  (Heap, 128) * 2
```

* Access Path Expression

APE_i :

```
(U 128)       +
(U 128) * 2
```

* Access Path Cost

APC_i = $C(APE_i)$
= 3 *random accesses* + 381 *sequential accesses* ≈ 7,100 *msec*

• if $uc = 14$

* File Primitive Expression

```
Read  (Heap, 3712)      +
```

```
                Read  (Heap, 3712) * 2
```

* Access Path Expression

APE_i :

```
(U 3712)        +
(U 3712) * 2
```

* Access Path Cost

APC_i $= C(APE_i)$

$= 3$ *random accesses* $+ 11{,}133$ *sequential accesses* $\approx 205{,}000$ *msec*

§ Q12:

```
retrieve (h.id, h.seq, i.id, i.seq, i.amount)
        valid   from begin of (h overlap i)  to end of (h extend i)
        where   h.id = 500       and      i.amount = 73700
        when    h overlap i
        as of   "now"
```

* Algebraic Expression

```
{(([L1: Select   (h, h.id = 500);
    L2: Project  (L1, h.id, h.seq)        ];
    L3: Temporary        (L2)
  ),
  ([L4: Select   (i, i.amount = 73700);
    L5: Project  (L4, i.id, i.amount, i.seq)       ];
    L6: Temporary        (L5)
  ));
   [L7: Join    (L3, L6, TS, h overlap i);
   (    Project  (L7, h.id, h.seq, i.id, i.seq, i.amount),
        Valid    (L7, From, beginOf (overlap (h, i))),
        Valid    (L7, To,   endOf   (extend  (h, i)))
   )]   }
```

• if $uc = 0$

* File Primitive Expression

```
    Read  (Hash, 0)               +
  ( Read  (Heap, 1) * 2 - 1       +
    Write (Heap, 1) * 3 - 1)      +
    Read  (Heap, 128)             +
  ( Read  (Heap, 1) * 2 - 1       +
    Write (Heap, 1) * 3 - 1)      +
    Read  (Heap, 1) * 0           +
    Read  (Heap, 1) * 0
```

* Access Path Expression

APE_i :

```
(H 1)            +
(U 1) * 2 - 1    +
(U 128)          +
(U 1) * 2 - 1    +
(U 1) * 0        +
(U 1) * 0
```

APE_o :

```
(U 1) * 3 - 1)         +
```

(U 1) * 3 - 1)

* Access Path Cost

APC_i $= C(APE_i)$

$= 4$ *random accesses* + 127 *sequential accesses* ≈ 2,460 *msec*

APC_o $= C(APE_o)$

$= 2$ *random accesses* + 2 *sequential accesses* = 99.4 *msec*

• if $uc = 14$

* File Primitive Expression

```
      Read  (Hash, 28)              +
    ( Read  (Heap, 1) * 2 - 1       +
      Write (Heap, 1) * 3 - 1)      +
      Read  (Heap, 3712)            +
    ( Read  (Heap, 1) * 2 - 1       +
      Write (Heap, 1) * 3 - 1)      +
      Read  (Heap, 1) * 0           +
      Read  (Heap, 1) * 0
```

* Access Path Expression

APE_i :

```
(H 1 (P 28 (S 1) (P 1))) +
(U 1) * 2 - 1            +
(U 3712)                 +
(U 1) * 2 - 1            +
(U 1) * 0                +
(U 1) * 0
```

APE_o :

```
(U 1) * 3 - 1)           +
(U 1) * 3 - 1)
```

* Access Path Cost

APC_i $= C(APE_i)$

$= 32$ *random accesses* + 3,711 *sequential accesses* ≈ 69,300 *msec*

APC_o $= C(APE_o)$

$= 2$ *random accesses* + 2 *sequential accesses* = 99.4 *msec*

§ Q13:

```
retrieve (h.id, h.seq)   where   h.id = 455
         when   "1/1/82" precede  end of h
```

* Algebraic Expression

```
{[ L1: Select  (h, h.id = 455);
   L2: When    (L1, "1/1/82" precede endOf (h));
       Project (L1, h.id, h.seq)          ]}
```

• the rest is the same as Q01.

§ Q14:

```
retrieve (h.id, h.seq)   where   h.amount = 10300
         when   "1/1/82" precede  end of h
```

* Algebraic Expression

```
{[ L1:  Select   (h, h.amount = 10300);
   L2:  When     (L1, "1/1/82" precede endOf (h));
        Project  (L2, h.id, h.seq)          ]}
```

• the rest is the same as Q07.

§ Q15:

```
retrieve (h.id, h.seq)   where   h.amount = 10300
         as of    "1/1/83"
```

* Algebraic Expression

```
{[ L1:  Select   (h, h.amount = 10300);
   L2:  AsOf     (L1, "1/1/83", "1/1/83");
        Project  (L2, h.id, h.seq)          ]}
```

• the rest is the same as Q07.

§ Q16:

```
retrieve (h.id, h.seq)   where   h.amount = 10300
         when     "1/1/82" precede  end of h
         as of    "1/1/83"
```

* Algebraic Expression

```
{[ L1:  Select   (h, h.amount = 10300);
   L2:  When     (L1, "1/1/82" precede endOf (h));
   L3:  AsOf     (L2, "1/1/83", "1/1/83");
        Project  (L3, h.id, h.seq)          ]}
```

• the rest is the same as Q07.

Appendix E

Update Algorithms

This appendix shows the algorithms to handle **delete** and **replace** on rollback, historical, or temporal relations using the *temporally partioned store*, as discussed in Section 5.1.2.

```
(*      Update a relation with a temporally partitioned store.
 *          parameters
 *              mode      : mdDELETE or mdREPLACE
 *              rel       : relation to be updated
 *              baseTup   : base tuple (to be updated)
 *              baseTid   : tuple-id of the base tuple
 *              updateTup : new tuple to replace the base tuple
 *          return value
 *              OK        : when successful
 *              ERROR     : when failed
 *)
function update_t (mode, rel, baseTup, baseTid, updateTup):
begin

    case (mode) of
        mdREPLACE:
            begin
                saveTid ← baseTid;
                cc ← delete_t (rel, baseTid, baseTup,
                                      updateTup, mdREPLACE);
                baseTid ← saveTid;

                case (cc) of
                    NoRep:      (* no overlaping interval for replace *)
                        ;       (* no action needed *)
                    BaseTup:
                        cc ← replace (rel, baseTid, baseTup, TRUE);
                    OK:
                        cc ← replace (rel, baseTid, updateTup, TRUE);
                    ERROR:
                        return (cc);
                end;
            end;

        mdDELETE:
            cc ← delete_t (rel, baseTid, baseTup, updateTup, mdDELETE);
    end;
    return (cc);
end;

(*      Delete a tuple from a temporally partitioned store.
```

```
 *           parameters
 *                rel        : relation to be updated
 *                baseTup    : base tuple (to be updated)
 *                baseTid    : tuple-id of the base tuple
 *                updateTup  : new tuple to replace the base tuple
 *                mode       : mdDELETE or mdREPLACE
 *           return value
 *                OK         : when successful
 *                ERROR      : when failed
 *)
function delete_t (rel, baseTup, baseTid, updateTup, mode):
begin
    (* determine the temporal type *)
    if (rel->temporalType = S_HISTORICAL) then
    begin
        if (rel->temporalType = S_PERSISTENT) then          (* temporal *)
            return (delete_temporal
                      (rel, baseTid, baseTup, updateTup, mode));
        else                                                (* historical *)
            return (delete_historical
                      (rel, baseTid, baseTup, updateTup, mode));
    end;

    else if (rel->temporalType = S_PERSISTENT) then         (* rollback *)
            return (delete_rollback
                      (rel, baseTid, baseTup, updateTup, mode));
    else                          (* snapshot relation *)
        return (ERROR);
end;

(*       Delete a tuple from a historical relation.
 *           parameters
 *                rel        : relation to be updated
 *                baseTup    : base tuple (to be updated)
 *                baseTid    : tuple-id of the base tuple
 *                updateTup  : new tuple to replace the base tuple
 *                mode       : mdDELETE or mdREPLACE
 *           return value
 *                OK         : when successful
 *                NoRep      : when no need to replace
 *                BaseTup    : when baseTup is to be replaced
 *                ERROR      : when failed
 *)
function delete_historical (rel, baseTup, baseTid, updateTup, mode):
begin

    base_validFrom    ← (valid from value of the base   tuple);
    base_validTo      ← (valid to   value of the base   tuple);
    update_validFrom ← (valid from value of the update tuple);
    update_validTo    ← (valid to   value of the update tuple);

    cc ← OK;
    if (update_validFrom ≤ base_validFrom) then
    begin
        update_validFrom ← base_validFrom;
        if (update_validTo ≤ base_validFrom) then (* case (1) *)
            cc ← NoRep;
```

```
    else if (update_validTo < base_validTo) then      (* case (2) *)
    begin    (* base_validFrom < update_validTo < base_validTo *)
        base_validFrom ← update_validTo;

        if (mode = mdDELETE) then
        begin
            ins_rep ← replace;
            tmptup ← baseTup;
        end;
        else          (* mdREPLACE *)
        begin
            ins_rep ← insert_history;
            tmptup ← updateTup;
        end;

        cc ← ins_rep (rel, baseTid, tmptup, TRUE);
        if (mode = mdREPLACE) then
        begin
            cc ← BaseTup;
            if (rel->storeSpec = ReverseChaining) then
                set_nva (rel, baseTup, baseTid);
        end;
    end;

    else              (* base_validTo < update_validTo *)
    begin             (* case (3) *)
        update_validTo ← base_validTo;
        if (mode = mdDELETE) then    cc ← delete (rel, baseTid);
    end;
end;

else if (update_validFrom < base_validTo) then
begin          (* case (4) or (5) *)
               (* base_validFrom < update_validFrom < base_validTo *)
    if (mode = mdDELETE  ∧  base_validTo ≤ update_validTo) then
        ins_rep ← replace;
    else     (* mdREPLACE or (4) of mdDELETE *)
    begin
        ins_rep ← insert_history;
        tmp_tid ← baseTid;
    end;

    tmp_t ← base_validTo;
    base_validTo ← update_validFrom;
    cc ← ins_rep (rel, baseTid, baseTup, TRUE);
    base_validTo ← tmp_t;

    if (update_validTo < base_validTo) then          (* case (4) *)
    begin
        base_validFrom ← update_validTo;

        if (mode = mdDELETE) then
        begin
            ins_rep ← replace;
            tmptup ← baseTup;
            if (rel->storeSpec = ReverseChaining) then
                set_nva (rel, baseTup, baseTid);
```

```
                    baseTid ← tmp_tid;
                end;
                else (* mdREPLACE *)
                begin
                    ins_rep ← insert_history;
                    tmptup ← updateTup;
                    if (rel->storeSpec = ReverseChaining) then
                        set_nva (rel, updateTup, baseTid);
                end;

                cc ← ins_rep (rel, baseTid, tmptup, TRUE);
                if (mode = mdREPLACE) then
                begin
                    cc ← BaseTup;
                    if (rel->storeSpec = ReverseChaining) then
                        set_nva (rel, baseTup, baseTid);
                end;
            end;

            else if (mode = mdREPLACE) then
            begin               (* base_validTo ≤ update_v : case (5) *)
                update_validTo ← base_validTo;
                if (rel->storeSpec = ReverseChaining) then
                    set_nva (rel, updateTup, baseTid);
            end;
        end;
        else            (* base_validTo < update_validFrom : case (6) *)
            cc ← NoRep;
        return (cc);
    end;

    (*      Delete a tuple from a temporal relation.
     *          parameters
     *              rel        : relation to be updated
     *              baseTup    : base tuple (to be updated)
     *              baseTid    : tuple-id of the base tuple
     *              updateTup  : new tuple to replace the base tuple
     *              mode       : mdDELETE or mdREPLACE
     *          return value
     *              OK         : when successful
     *              NoRep      : when no need to replace
     *              BaseTup    : when baseTup is to be replaced
     *              ERROR      : when failed
     *)
    function delete_temporal (rel, baseTup, baseTid, updateTup, mode):
    begin

        base_validFrom     ← (valid from        value of the base   tuple);
        base_validTo       ← (valid to          value of the base   tuple);
        base_transStart    ← (transaction start value of the base   tuple);
        base_transStop     ← (transaction stop  value of the base   tuple);

        update_validFrom   ← (valid from        value of the update tuple);
        update_validTo     ← (valid to          value of the update tuple);
        update_transStart  ← (transaction start value of the update tuple);
        update_transStop   ← (transaction stop  value of the update tuple);
```

```
cc    ← OK;
if (update_validFrom ≤ base_validFrom) then
begin
     update_validFrom ← base_validFrom;
     if (update_validTo ≤ base_validFrom) then           (* case (1) *)
         cc ← NoRep;

     else if (update_validTo < base_validTo) then        (* case (2) *)
     begin    (* base_validFrom < update_validTo < base_validTo *)
         base_transStop ← update_transStart;
         tmp_tid ← baseTid;
         cc ← insert_history (rel, baseTid, baseTup, TRUE);
         base_validFrom ← update_validTo;
         base_transStart ← update_transStart;
         base_transStop ← TIME_MAX;

         if (mode = mdDELETE) then
         begin
              ins_rep ← replace;
              if (rel->storeSpec = ReverseChaining) then
                  set_nva (rel, baseTup, baseTid);
              tmptup ← baseTup;
              baseTid ← tmp_tid;
         end;
         else           (* mdREPLACE *)
         begin
              ins_rep ← insert_history;
              if (rel->storeSpec = ReverseChaining) then
                  set_nva (rel, updateTup, baseTid);
              tmptup ← updateTup;
         end;

         cc ← ins_rep (rel, baseTid, tmptup, TRUE);
         if (mode = mdREPLACE) then
         begin
              cc ← BaseTup;
              if (rel->storeSpec = ReverseChaining) then
                  set_nva (rel, baseTup, baseTid);
         end;
     end;

     else               (* base_validTo < update_validTo *)
     begin              (* case (3) *)
         update_validTo ← base_validTo;
         base_transStop ← update_transStart;

         if (mode = mdDELETE) then     ins_rep ← replace;
         else ins_rep ← insert_history;          (* mdREPLACE *)

         cc ← ins_rep (rel, baseTid, baseTup, TRUE);
         if (mode = mdREPLACE  ∧  rel->storeSpec = ReverseChaining) then
                  set_nva (rel, updateTup, baseTid);
     end;
end;
else if (update_validFrom < base_validTo) then
begin         (* case (4) or (5) *)
              (* base_validFrom < update_validFrom < base_validTo *)
```

```
base_transStop ← update_transStart;
tmp_tid ← baseTid;
cc ← insert_history (rel, baseTid, baseTup, TRUE);

base_transStart ← update_transStart;
base_transStop ← TIME_MAX;
if (mode = mdDELETE  ∧  base_validTo ≤ update_validTo) then
begin
    ins_rep ← replace;
    if (rel->storeSpec = ReverseChaining) then
        set_nva (rel, baseTup, baseTid);
    baseTid ← tmp_tid;
end;
else             (* mdREPLACE *)
begin
    ins_rep ← insert_history;
    if (rel->storeSpec = ReverseChaining) then
        set_nva (rel, baseTup, baseTid);
end;

tmp_t ← base_validTo;
base_validTo ← update_validFrom;
cc ← ins_rep (rel, baseTid, baseTup, TRUE);
base_validTo ← tmp_t;

if (update_validTo < base_validTo) then          (* case (4) *)
begin
    base_validFrom ← update_validTo;
    if (mode = mdDELETE) then
    begin
        ins_rep ← replace;
        if (rel->storeSpec = ReverseChaining) then
            set_nva (rel, baseTup, baseTid);
        baseTid ← tmp_tid;
        tmptup ← baseTup;
    end;
    else         (* mdREPLACE *)
    begin
        ins_rep ← insert_history;
        if (rel->storeSpec = ReverseChaining) then
            set_nva (rel, updateTup, baseTid);
        tmptup ← updateTup;
    end;

    cc ← ins_rep (rel, baseTid, tmptup, TRUE);
    if (mode = mdREPLACE) then
    begin
        cc ← BaseTup;
        if (rel->storeSpec = ReverseChaining) then
            set_nva (rel, baseTup, baseTid);
    end;
end;

else             (* base_validTo ≤ update_validTo : case (5) *)
begin
    update_validTo ← base_validTo;
    if (rel->storeSpec = ReverseChaining) then
```

```
                set_nva (rel, updateTup, baseTid);
            end;
        end;

        else        (* base_validTo < update_validFrom : case (6) *)
            cc ← NoRep;
        return (cc);
end;

(*      Delete a tuple from a rollback relation.
 *          parameters
 *              rel       : relation to be updated
 *              baseTup   : base tuple (to be updated)
 *              baseTid   : tuple-id of the base tuple
 *              updateTup : new tuple to replace the base tuple
 *              mode      : mdDELETE or mdREPLACE
 *          return value
 *              OK        : when successful
 *              ERROR     : when failed
 *)
function delete_rollback (rel, baseTup, baseTid, updateTup, mode):
begin

    base_transStart   ← (transaction start value of the base   tuple);
    base_transStop    ← (transaction stop  value of the base   tuple);
    update_transStart ← (transaction start value of the update tuple);
    update_transStop  ← (transaction stop  value of the update tuple);

    (* in a rollback relation, base_transStart ≤ update_transStart *)
    base_transStop ← update_transStart;

    if (mode = mdDELETE) then    ins_rep ← replace;
    else ins_rep ← insert_history;        (* mdREPLACE *)

    cc ← ins_rep (rel, baseTid, baseTup);
    if (mode = mdREPLACE  ∧  rel->storeSpec = ReverseChaining) then
        set_nva (rel, updateTup, baseTid);
    return (cc);
end;

(*      Insert a tuple into the history store, if partitioned.
 *          parameters
 *              rel   : relation to be updated
 *              tuple : tuple to be inserted
 *              tid   : tuple-id to be set on insertion
 *          return value
 *              OK        : when successful
 *              ERROR     : when failed
 *)
function insert_history (rel, tid, tuple):
begin
    if (rel->storeSpec = ReverseChaining) then
        begin
            insert tuple into the history store;
            set tid to the tuple-id of the inserted tuple;
        end;
    else
```

```
        begin
            insert tuple into the single store;
            set tid to the tuple-id of the inserted tuple;
        end;
    end;
    if (successful) then  return (OK);
    else  return (ERROR);
end;

(*      Set the field nva (next version address).
 *          parameters
 *              rel   : relation to be updated
 *              tuple : tuple whose nva field is to be set
 *              tid   : value of the nva field
 *)
procedure set_nva (rel, tuple, tid):
begin
        set the nva field of tuple to tid;
end;
```

Appendix F

Performance Analysis (2)

Costs of some sample queries on the temporal database with the update count of 14 are analyzed using the four models discussed in Chapter 4. We analyze the query costs for various formats of the history store, as discussed in Chapter 7, assuming that the database uses the temporally partitioned storage structure. Analysis of query costs for the temporal database with the conventional structure was given in Appendix D.

§ Q01

```
retrieve (h.id, h.seq)   where   h.id = 500
```

* Algebraic Expression

```
{[ L1: Select    (h, h.id = 500);
       Project   (L1, h.id, h.seq)          ]}
```

• for Reverse Chaining

* File Primitive Expression

```
Read  (Hash, 0)  +
Read  (Chain, 28)
```

* Access Path Expression

APE_i :

```
[ (H 0) ; (P 28 (S 1) (P 1)) ]
```

* Access Path Cost

APC_i $= C(APE_i) = 29$ *random accesses* ≈ 908 *msec*

• for Accession Lists

* File Primitive Expression

```
Read  (Hash, 0)  +
Read  (Accessionlist, 28)
```

* Access Path Expression

APE_i :

```
[ (H 0) ; (P 28 (P 1))]
```

* Access Path Cost

APC_i $= C(APE_i) = 30$ *random accesses* $= 939$ *msec*

• for Indexing

* File Primitive Expression

```
Read  (Hash, 0)  +
Read  (Index, 29)
```

* Access Path Expression

APE_i :

```
[ (H 0) ; [(S 1 (P 1)) ?, (S 28 (P 1)) ]]
```

* Access Path Cost
$APC_i = C(APE_i) = 30$ *random accesses* $= 939$ *msec*

• for Clustering

* File Primitive Expression
```
Read  (Hash, 0)  +
Read  (Cluster, 28, 8)
```

* Access Path Expression
APE_i :

$$[\ (H\ 0)\ ;\ (P\ \left\lceil \frac{28}{8} \right\rceil\ (S\ 8)\ (P\ 8))]$$

* Access Path Cost
$APC_i = C(APE_i) = 5$ *random accesses* ≈ 157 *msec*

• for Stacking

* File Primitive Expression
```
Read  (Hash, 0)  +
Read  (Stack, 28, 4)
```

* Access Path Expression
APE_i :
```
[ (H 0) ; (P 4)]
```

* Access Path Cost
$APC_i = C(APE_i) = 2$ *random accesses* $= 62.6$ *msec*

• for Cellular Chaining

* File Primitive Expression
```
Read  (Hash, 0)  +
Read  (Cellular, 28, 4)
```

* Access Path Expression
APE_i :

$$[\ (H\ 0)\ ;\ (P\ \left\lceil \frac{28}{4} \right\rceil\ (S\ 4)\ (P\ 4))]$$

* Access Path Cost
$APC_i = C(APE_i) = 8$ *random accesses* ≈ 250 *msec*

§ Q03:
```
retrieve (h.id, h.seq)   as of   "08:00 1/1/80"
```

* Algebraic Expression
```
{[ L1: AsOf      (h, "08:00 1/1/80", "08:00 1/1/80");
       Project   (L1, h.id, h.seq)           ]}
```

• for Reverse Chaining, Clustering, or Cellular Chaining

* File Primitive Expression

```
Read  (Heap, 147)        +
Read  (Heap, 4096)
```

* Access Path Expression
  APE_i :

```
(U 147)        +
(U 4096)
```

* Access Path Cost
  APC_i $= C(APE_i)$
  $= 2$ *random access* $+ 4{,}241$ *sequential access* $\approx 78{,}100$ *msec*

• for Accession Lists

* File Primitive Expression

```
Read  (Heap, 147)        +
Read  (Heap, 624)        +
Read  (Accessionlist, 5)
```

* Access Path Expression
  APE_i :

```
(U 147)        +
(U 624)        +
(S 5 (P 1))
```

* Access Path Cost
  APC_i $= C(APE_i)$
  $= 7$ *random access* $+ 769$ *sequential access* $\approx 14{,}400$ *msec*

• for Indexing

* File Primitive Expression

```
Read  (Heap, 782)        +
Read  (Index, 5)
```

* Access Path Expression
  APE_i :

```
(U 782)        +
(S 5 (P 1))
```

* Access Path Cost
  APC_i $= C(APE_i)$
  $= 6$ *random access* $+ 781$ *sequential access* $\approx 14{,}600$ *msec*

§ Q09

```
retrieve (h.id, i.id, i.amount)
         where  h.id = i.amount
         when   h overlap i   and   i overlap "now"
```

* Algebraic Expression

```
{[L1: When       (i, i overlap "now");
  L2: Project    (L1, i.id, i.amount,
                         i.valid_from, i.valid_to)];
  L3: Temporary  (L2);
 [L4: Join       (h, L3, TS, h.id=i.amount & h overlap i);
```

```
                Project    (L4, h.id, i.id, i.amount)        ]}
```

• for Reverse Chaining, Accession Lists, Clustering, Stacking, or Cellular Chaining

* File Primitive Expression

```
    Read  (Heap, 147)                  +
  ( Read  (Heap, 19) * 2 - 1           +
    Write (Heap, 19) * 3 - 1 )         +
    Read  (Heap, 19)                   +
    Read  (Hash, 0) * 1024)
```

* Access Path Expression

APE_i :

```
(U 147)                +
(U 19) * 2 - 1         +
(U 19)                 +
(H 1) * 1024
```

APE_o :

(U 19) * 3 - 1

* Access Path Cost

APC_i = *C* ((U 147)) + *C* ((U 19) * 2 - 1) + *C* ((U 19)) + *C* ((H 1)) * 1024
= 1,028 *random accesses* + 199 *sequential accesses* ≈ 35,800 *msec*

APC_o = *C* ((U 19) * 3 - 1)
= 3 *random accesses* + 53 *sequential accesses* ≈ 1,070 *msec*

• for Indexing

* File Primitive Expression

```
    Read  (Heap, 114)                  +
  ( Read  (Heap, 19) * 2 - 1           +
    Write (Heap, 19) * 3 - 1 )         +
    Read  (Heap, 19)                   +
    Read  (Index, 1) * 1024)
```

* Access Path Expression

APE_i :

```
(U 114)                +
(U 19) * 2 - 1         +
(U 19)                 +
(H 1 (P 1))
```

APE_o :

(U 19) * 3 - 1

* Access Path Cost

APC_i = *C* (APE_i)
= 2,052 *random accesses* + 166 *sequential accesses* ≈ 43,000 *msec*

APC_o = *C* ((U 19) * 3 - 1)
= 3 *random accesses* + 53 *sequential accesses* ≈ 1,070 *msec*

§ Q11:

```
retrieve (h.id, h.seq, i.id, i.seq, i.amount)
         valid   from begin of h        to end of i
         when    begin of h precede i
```

```
as of   "4:00 1/1/80"
```

* Algebraic Expression

```
{ L1: AsOf    (h, "4:00 1/1/80", "4:00 1/1/80");
 [L2: Join    (L1, i, TS, beignOf (h) precede i);
  (   Project (L2, h.id, h.seq, i.id, i.seq, i.amount),
      Valid   (L2, From, beginOf (h)),
      Valid   (L2, To,   endOf   (i))
  )]    }
```

- for Reverse Chaining, Clustering, or Cellular Chaining

* File Primitive Expression

```
    Read  (Heap, 147)      +
    Read  (Heap, 4096)     +
  ( Read  (Heap, 147)      +
    Read  (Heap, 4096) ) * 2
```

* Access Path Expression

APE_i :

```
    (U 147)            +
    (U 4096)           +
  ( (U 147)            +
    (U 4096) ) * 2
```

* Access Path Cost

$APC_i = C(APE_i)$

= 6 *random accesses* + 12,723 *sequential accesses* ≈ 234,000 *msec*

- for Accession Lists

* File Primitive Expression

```
    Read  (Heap, 147)              +
    Read  (Heap, 624)              +
  ( Read  (Heap, 147)              +
    Read  (Heap, 624) ) * 2        +
    Read  (Accessionlist, 4)
```

* Access Path Expression

APE_i :

```
    (U 147)            +
    (U 624)            +
  ( (U 147)            +
    (U 624) ) * 2      +
    (S 4 (P 1))
```

* Access Path Cost

$APC_i = C(APE_i)$

= 10 *random accesses* + 2,307 *sequential accesses* ≈ 42,800 *msec*

- for Indexing

* File Primitive Expression

```
  Read  (Heap, 782)        +
  Read  (Heap, 782) * 2    +
```

```
Read  (Index, 4)
```

* Access Path Expression

APE_i :

```
(U 782)            +
(U 782) ) * 2      +
(S 4 (P 1))
```

* Access Path Cost

APC_i = $C(APE_i)$

= 7 *random accesses* + 2,343 *sequential accesses* ≈ 43,300 *msec*

§ Q16:

```
retrieve (h.id, h.seq)  where  h.amount = 10300
         when   "1/1/82" precede  end of h
         as of  "1/1/83"
```

* Algebraic Expression

```
{[ L1: Select    (h, h.amount = 10300);
   L2: When      (L1, "1/1/82" precede endOf (h));
   L3: AsOf      (L2, "1/1/83", "1/1/83");
       Project   (L3, h.id, h.seq)        ]}
```

• with a Secondary Index, on the `Amount` Attribute, as a Snapshot Single Heap

* File Primitive Expression

```
Read  (Heap, 295)     +
Read  (Index, 29)
```

* Access Path Expression

APE_i :

```
(U 295)        +
(S 29 (P 1))
```

* Access Path Cost

APC_i = $C(APE_i)$

= 30 *random accesses* + 294 *sequential accesses* ≈ 6,350 *msec*

• with a Secondary Index, on the `Amount` Attribute, as a Snapshot Single Hash

* File Primitive Expression

```
Read  (Hash, 0)  +
Read  (Index, 29)
```

* Access Path Expression

APE_i :

```
(H 0)          +
(S 29 (P 1))
```

* Access Path Cost

APC_i = $C(APE_i)$ = 30 *random accesses* = 939 *msec*

• with a Secondary Index, on the `Amount` Attribute, as a Temporal Partitioned Heap

* File Primitive Expression

```
Read (Heap, 27)     +
Read (Heap, 755)    +
Read (Index, 4)
```

* Access Path Expression

APE_i :

```
(U 27)       +
(U 755)      +
(S 4 (P 1))
```

* Access Path Cost

$APC_i = C(APE_i)$

$= 6$ *random accesses* $+ 780$ *sequential accesses* $\approx 15{,}500$ *msec*

• with a Secondary Index, on the Amount Attribute, as a Temporal Partitioned Hash

* File Primitive Expression

```
Read (Hash, 0)  +
Read (Hash, 0)  +
Read (Index, 4)
```

* Access Path Expression

APE_i :

```
(H 0)        +
(H 0)        +
(S 4 (P 1))
```

* Access Path Cost

$APC_i = C(APE_i) = 6$ *random accesses* ≈ 188 *msec*