NESTED BITEMPORAL RELATIONAL DATA MODEL

by

CANAN EREN

A dissertation submitted to the Graduate Faculty in Computer Science in partial fulfillment of the requirements for the degree of Doctor of Philosophy, The City University of New York

2008

INFORMATION TO USERS

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleed-through, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

This manuscript has been read and accepted for the
Graduate Faculty in Computer Science in satisfaction of the
dissertation requirement for the degree of Doctor of Philosophy.

Prof. Abdullah Tansel Uz

04/29/2008 Chair of Examining Committee

Prof. Ted Brown

04/29/2008 Executive Officer

Assoc. Prof. Dr. Susan Imberman

Assoc. Prof. Dr. Richard Holowczak

Prof. Dr. Reda Alhajj
Supervisory Committee

THE CITY UNIVERSITY OF NEW YORK

Abstract

NESTED BITEMPORAL RELATIONAL DATA MODEL

by

Canan Eren

Advisor: Prof. Abdullah Uz Tansel

In this dissertation, we propose a nested bitemporal relational data model, using nested relations, and we also develop algebra and calculus languages for this model. The fundamental construct for representing temporal data is a bitemporal atom that consists of five parts: a value, its validity period, and the time this data is recorded in the database. Bitemporal data is attached to attributes, and arbitrary levels of nesting are allowed. The algebra includes operations to manipulate bitemporal data, to restructure nested bitemporal relations, and to rollback database to a designated state in the past. We have also defined the concept of 'context' for using bitemporal data: bitemporal context, historical context, and current context. BtSQL, a preprocessor for the bitemporal SOL language, query syntax allows end users to incorporate bitemporal, current and historical context. We defined and implemented BtSQL for the specification of bitemporal queries in different context. It translates a bitemporal query specification to a standard SQL statement. BtSQL includes select, insert, delete, and update statements of SQL extended for bitemporal relational databases. A prototype implementation has been completed in an object-relational database system to demonstrate the feasibility of the model defined in this thesis.

Acknowledgements

Above all, I am deeply grateful to my supervisor, Prof. Dr. Abdullah Uz Tansel**,** for giving me the chance to finish this thesis. I would like to thank him for his guidance, many good ideas, constant support at all times, and constructive criticism during the course of this work. I hope that his standards have now become mine. His contribution to this work is inestimable.

Many thanks are also due to the external examiners of this thesis, Associate Professor Dr. Susan Imberman, Associate Professor Dr. Richard Holowscak and Professor Dr. Reda Alhajj, for their very helpful comments, which were much appreciated.

I would like to express my sincere gratitude to Professor Dr. Irem Ozkarahan for initiating and encouraging me to finish my thesis. I am thankful to the Department of Computer Science at Dokuz Eylul University for providing me the environment to study.

I am also thankful to my family – especially to my sister Nalan Benakay – for their motivation, constant support, and prayers.

Special thanks to my sons Eray and Ege for their understanding for the time I was away from them to finish this thesis. I could not have done this without their love. This thesis is dedicated to them.

Table of Contents

List of Tables

List of Figures

Chapter 1

1. INTRODUCTION

It is difficult if not impossible to identify a substantial computer application that does not evolve over time. Consider employee data, which may include employee salary, department, and title; all of these attributes change over time. As another example, student histories typically include past, present, and future data on enrollments, grades, degree programs, and degrees awarded. The same is true in fields outside of the academic world: In the financial markets, businesses must track cash flow or account balances over time for each customer. Other examples of ever-changing time-related data include stock market data; patient medical records, with diagnoses, X-rays, and lab tests; reservation systems for airlines, car rentals, and hotels; data warehousing records; and spatial databases. Built-in time-management support greatly increases the functionality of a database application. It is desirable for a database system to maintain past, present, and possibly future versions of data. Such databases are called "temporal databases" [Ta90].

Developments in storage systems, computer hardware, and software paved the way to storing massive amounts of data. In the early 1970s, researchers built prototype database management systems (DBMSs), and software vendors developed commercial DBMSs. In the 1980s, commercial relational DBMSs became available. However, database management systems did not include capabilities for time management. Consequently, application designers developed time-dependent applications mostly in an ad hoc manner.

Research on temporal databases focused mainly on the definition of a data model that could support the essential semantics of temporal data presentation, temporal data storage, efficient temporal query evaluation, and temporal implementation strategies. The first book on temporal databases, by Tansel et al., appeared in 1993 [T^{+}93]. It is a

collection of papers by pioneers in the field, covering in detail the theory, design, and implementation issues of temporal databases. It included a glossary of temporal database concepts. An updated version of this glossary is in [JDB+98]. The August 1995 issue of *IEEE Transactions on Data and Knowledge Engineering* was dedicated to temporal and real-time databases. Workshops bringing together researchers interested in the development of tools for the management of temporal data were held in Arlington, Texas, in 1993, Zurich in 1995, and Dagstuhl in 1997 and 1998. Also, a spatial and temporal databases symposium was held in Redondo Beach, California, in 2001 [JSST01]. In 2000, a book was published on time granularities in databases [BJW00]. Another book by Snodgrass [Sn00], published in 2000, deals with the development of temporal database applications in SQL. International symposium, TIME, on temporal representation and reasoning brings together researchers from distinct research areas involving the management of temporal data as well as the reasoning about temporal aspects of information for the last fourteen years.

Although there has been an extensive research effort in temporal database in the past 30 years, no standardized temporal database model or query language has been defined by any major standardization organization such as the ISO. However, we expect that temporal data management will eventually become an integral part of database management systems and their query languages. Software vendors already include some limited temporal support in their DBMS software.

1.1 Database Management Systems

Data is a known fact that is recordable and has an implicit meaning. Data can be composed of anything from simple values such as integers, real numbers, characters and strings to more complex values such as texts, images, sounds and videos. A *database* is nothing more than a collection of related data obtained from a source that possibly interacts with events in the real world, and serves a user community that is actively interested in the contents of the database.

During the 1960s, data was stored in files and organized with respect to the application that created and used the data. The file systems created by these early data processing systems resulted excessive data redundancy and data was not independent from the devices it was stored on from changes in application software. In the early seventies, database systems were introduced to overcome the disadvantages of the file systems. The aim was to minimize the application specific solutions and have a general-purpose software system so that different application can share the same data. General-purpose software that facilitates the process of defining, constructing, manipulating, sharing, protecting data among various users and applications is called a *Data Base Management System* (DBMS).

DBMS have to support a variety of tasks and requirements [EN04]. A DBMS should manage not only large amounts of data but should also manage persistent data efficiently. The definition, modification, and retrieval of data must be supported through languages and operations as well as optimization techniques to retrieve those data. It should allow multiple users to access the same data simultaneously where access might be from the same computer system or via World Wide Web. The DBMS must also provide data consistency, recovery from system crashes, users' authorizations and database security.

1.1.1 Relational Database Management Systems and Query Languages

The relational data model was introduced in 1970 by Codd [Co70]. The model uses the concept of mathematical relation as its basic building block, and it has its theoretical basis in the set theory and first order predicate logic. The relational model is still the dominant data model among other models. Several commercial DBMS implementing the relational data model are also available.

The relational data model represents the database as a collection of *relations* in the form of two-dimensional tables. A row is called a *tuple*; a column header is called an *attribute*; and the table is called a *relation*. The rows of a relation, other than the header row containing the attribute names, are called tuples. Attributes of a relation serve as names for the columns of the relation and they represent the common properties of a real

world object called an entity or a relationship. A *relation schema* R, denoted by R(A_1, A_2, ..., A_n), is made up of a relation name R and a list of attributes A_1, A_2, ..., A_n. The *degree* of a relation is the number of attributes *n* of its relation schema. The relational model requires that each component of each tuple be atomic; that is, it must be of some elementary type such as integer, real, character, string or date. Such relations are defined as being in the First Normal Form (1NF). An example of a relation schema for a relation of degree 4 is shown in table 1.1. EMPLOYEE(EMP#, ENAME, SALARY, DEPT) is the relation schema. Each *tuple* describes the properties of an employee. For example, the tuple (101, Tom, 30K, Marketing) contains data about employee Tom, including his employment number, salary and the department he works in. The set of tuples is called an *instance* of the EMPLOYEE relation.

Example 1.1: The EMPLOYEE relation contains data about employees of a company.

Table 1.1: EMPLOYE table

EMP#	ENAME	SALARY	DEPT
101	Tom	30000	Marketing
102	Ann	35000	Sales
103	John	45000	Toys

This *relation schema* can be defined as*:*

R = (EMP#: INTEGER PRIMARY KEY,
Name: CHAR (20),
Salary: INTEGER,
Dept: CHAR (15))

The relational algebra and the relational calculus are the two formal query languages associated with the relational model. While queries in the relational algebra are composed using a collection of operators and relation instance, relational calculus queries are declarative predicate calculus expressions. A basic set of five generic operations, namely set union (∪), set difference (-) and cross product (x) of two relations, and selection (σ), projection (π), are sufficient to express all the relational algebra operations. The result of each of these operations is another relation. The set union operation includes all tuples

that either are in the first or second or both relations. The set difference returns all tuples in the first relation but not in the second relation. The cross product combines all tuples of the first with all tuples of the second relation where the schema of the resulting relation contains all attributes of both relations. Selection operation returns a subset of tuples of a relation that satisfy a selection condition. Projection operation keeps designated attributes from the relation. Other operations can be defined in terms of these five operations. One such operation is join that combines two relations according to a condition specified at the attributes of operand relations. Join is a very common and convenient operation in the relational algebra.

Relational calculus is based on first order predicate logic and allows relational (set) defining operations. Expressions contain formulas and terms that qualify the tuples for the resulting relation, without being explicit about how they should be computed. The relational calculus is nonprocedural. Relational calculus is the basis of the design of commercial query languages such as SQL. Both of these abstract query languages, relational algebra, and relational calculus are equivalent in expressive power.

Structured Query Language, SQL, is the standard language for commercial relational DBMS and has three parts: the data definition language, the query language and the data modification language. The data definition language creates, deletes, and modifies tables, indexes, and views. A base table is a relation that is physically stored in the database. Indexes are access paths to base tables that, for example, allow a faster lookup of specific data in the table. Integrity constraints can be defined on tables when they are created or after their creation. SQL also allows the retrieving of data. It is based on relational calculus, though it has some features from relational algebra. It additionally supports features such as aggregate functions, grouping and ordering of tuples. The data modification statements is used to update, insert and delete data.

The first standard for SQL was SQL-86 and ISO (the International Standards Organization) and ANSI (the American National Standards Institute) released SQL1 in 1986. SQL2 or SQL-92 was published as a standard in 1992. The third version of the

standard, SQL3, provides significant extensions from SQL-92. In particular, SQL3 O-R features include multiset, nested collection types and user-defined types. Another major feature is SQL/XML [ISO03], which defines how SQL can be used together with XML in a database, and is supported by major database vendors.

1.1.2 Entity Relationship Data Models

The Entity-Relationship model (ER model) was proposed by Chen in 1976 and has become a popular high-level conceptual data model that provides concepts and formalisms for the description of the semantics in database modeling. The ER diagram is used to develop an initial database design, and then it is mapped to a data model of a specific DBMS, for example, to the relational data model. This model and its variations are commonly used and many database design tools.

An *entity* is an object in the real world with an independent existence and is described using a set of *attributes*. Similar entities are designated by an entity types. An entity set is a collection of similar entities. Attributes may be atomic or composite and they can be nested arbitrarily to produce complex attributes. A relationship is an association among two or more entities. The number of participating entity types is called a degree of a relationship and relationships may have attributes as well. The relationships have cardinality constraints on the number of entities that can participate in a relationship.

1.1.3 Object-Oriented Data Models

Object-oriented databases (OODBs) were proposed to meet the requirements of different application domains, such as databases for engineering design and manufacturing (CAD/CAM), software engineering, and scientific and statistical computation. The characteristics and requirements of these applications are different from traditional business applications. Parts of these newer applications are more complex structures such as new data types for images and large textual items. Object-oriented approaches model some of these requirements independent from data types and

query languages, where the designer can specify the structure of complex objects and the operations that can be applied on these objects.

While relational database systems based their uniform design on Codd's work [Co70], OODBs do not have commonly agreed upon concepts and notions. Each OODB model proposed includes its own basic object concepts. The Object Database Management Group, ODMG, a consortium of OO database vendors and users, proposed a standard in 1993 that was revised in 1997 and 2001. Their strategy was to define object models and languages and to have those definitions accepted by the standardization institutes, such as ISO and ANSI.

An object-oriented database management system (OODBMS) stores objects and supports object identity (O/D), types or classes, complex objects, encapsulation, inheritance, programming language compatibility, polymorphism and operator overloading, extensibility, versioning, and computational completeness. Objects are identified by OIDS, while literals are identified by their values.

ODMG, proposed as an object model, has defined three languages, ODL (Object Definition Language), OQL (Object Query Language), and OML (Object Manipulation Language) [Ca94]. ODL is used to define the database schema. ODL is independent of programming languages and specifies the interfaces of the object types. The OQL is a declarative query language for the ODMG object model. The OQL syntax is similar to SQL, with additional features such as object identity, inheritance, relationships, polymorphisms, and collections. SQL takes relations as sets; however, OQL deals with sets of objects. Therefore, operations can be applied to objects in a query but OQL does not support an update mechanism. An OQL query embedded in programming languages like SMALLTALK, C++, and Java can return objects. An OML is defined to specify how database objects are retrieved and manipulated within SMALLTALK, C++, or Java programs and follows the syntax of those programming languages. Therefore, OML has ordinary SMALLTALK, C++, or Java expressions and commands like conditionals, calculations, and updates that manipulate structures defined in ODL and OQL. Physical

programs (or sets of constructors) are included to let programmers control the physical storage issues, such as memory management and clustering of objects.

1.2 Motivation

Databases without temporal support, whether relational or otherwise, store only the most recent data. The old values of data are either replaced by new values or completely deleted, i.e., removed from the database. Accessing past data is not an option. However, it is almost impossible to find an application that does not involve time-varying data [Sn95]. Because commercial databases are not capable of handling temporal data, ad hoc solutions are typically developed for incorporating temporal support in applications. As a consequence, without the implementation of such ad hoc solutions, the evolution of real-world phenomena over time cannot be recorded in the database, and only queries that involve the current state of the database can be answered.

Consider the EMPLOYEE relation introduced in Example 1.1 in the previous section, which contains only current data. For instance, a salary increase from 30K to 35K for Employee 101 requires updating the salary value to 35K, which means losing the previous value. All employees' salary histories can be kept by adding another attribute to the EMPLOYEE table and naming it SALARY_TIME. An example is given in Table 1.2. However, this table keeps only the salary history. For tracking the department history, another time attribute is required. A DEPARTMENT_TIME value is required to keep the employees' histories of departments. The SALARY_TIME attribute is not sufficient to keep DEPARTMENT history because, generally, the SALARY and DEPARTMENT of an employee do not change at the same time.

Table 1.2: EMPLOYEE relation with past salary data

EMP#	ENAME	SALARY	SALARY_TIME	DEPT
101	Tom	25K	01/2003	Sales
101	Tom	30K	02/2005	Sales
102	Ann	35K	06/2001	Sales
103	John	35K	06/2003	Toys
103	John	42K	07/2004	Toys
103	John	45K	01/2006	Marketing

Table 1.3 gives an example of employee data where past salary and department values are kept. Each change in SALARY or DEPARTMENT value causes an additional tuple in the relation, while the rest of the information does not change. This solution seems easy, but creates excessive data redundancy and provides limited time-processing capacity.

Table 1.3: EMPLOYEE relation with two time attributes, SALARY_TIME, DEPT_TIME

EMP#	ENAME	SALARY	SALARY_TIME	DEPT.	DEPT_TIME
101	Tom	25K	01/2003	Sales	01/2003
101	Tom	30K	02/2005	Sales	01/2003
101	Tom	30K	02/2005	Marketing	02/2006
102	Ann	35K	06/2001	Sales	06/2001
103	John	35K	06/2003	Toys	06/2003
103	John	42K	07/2004	Toys	06/2003
103	John	45K	01/2006	Marketing	01/2006
103	John	45K	01/2006	Toys	05/2006

Clearly, if temporal data is to be managed properly, a database management system should have native temporal support. A plethora of temporal data models have been proposed. Some of them are extensions to the conventional relational model, and others are completely new approaches.

A temporal database may capture either the history of objects (valid time), or the history of database activity (transaction time). However, bitemporal database systems support valid *time* and *transaction time*, the two orthogonal aspects of time [SA85]. Having only a valid time captures the history of an object but does not preserve the history of retroactive and post-active changes. Having only transaction time does not carry historical or future data, i.e., validity period of data values. On the other hand, bitemporal databases model our changing knowledge of the changing world, and thus associate data values with facts and also specify when the facts were valid, thereby providing a complete history of data values and their changes.

In order to accurately and completely model the real world, both time dimensions are needed. There are many applications that can benefit from the support of both valid and transaction times (i.e., financial, tax, and insurance applications) where auditing is especially important.

1.3 Contributions of the Thesis

Our main objective of this thesis is to define an extension to the relational data model that supports both valid and transaction times, by using attribute time-stamping for a complete representation of the real world. We use nested temporal relations as a basis [Ta97] for adding both time dimensions to define bitemporal realtions. The contribution of this thesis can be summarized as follows:

We model both valid time and transaction time to provide a complete history of data values and their changes. Unlike the earlier proposals that support both valid time and transaction time, our approach maintains the entire history of an object in one tuple. Snodgras's model and BCDM [S^+94] in general keep each temporal attribute in a separate relation. In contrast, Gadia and Bhargava's work is based on homogenous tuples [GB89].

We define a bitemporal relational algebra and a bitemporal relational calculus language. We extend previously defined temporal relational algebra operations for bitemporal data support. In querying bitemporal data, we introduce the concept of contexts: Bitemporal Context, Historical Context, and Current Context. Bitemporal context is useful for auditing queries and investigating the history of corrected errors. Historical context restricts a bitemporal relation to its state at a given time point or interval. Current context is for querying the snapshot state of a bitemporal database. We have also defined the temporal operators, such as Slice and Rollback, at the attribute level. That allows manipulation of temporal attributes while the rest of temporal tuples are kept intact.

The model defined in this thesis has features that are more general than previously proposed models [GB89, BG93, BZ93, S^{+}94, JSS94]. Nested bitemporal relational algebra facilitates query optimization, and nested bitemporal relational calculus forms a basis for designing query languages that support bitemporal data manipulation.

We have developed a preprocessor for the bitemporal SQL (BtSQL) language, designed for translating SQL statement into standard SQL statements. BtSQL includes SELECT, INSERT, DELETE and UPDATE statements of SQL, extended for bitemporal relational databases. We have successfully managed to hide tedious bitemporal query specifications from the user with BtSQL For this purpose, we use ORACLE as the implementation platform.

We have implemented the proposed bitemporal relational data model on top of a commercial database management system, to demonstrate the feasibility of our approach. We considered alternative implementation approaches; 1- a bitemporal atom represented as String implementation (BTA_String) or as an abstract data type (BTA_ADT), 2- a bitemporal atom is defined with SQL (BTA_SQL) or the language Java (BTA_Java), 3- a bitemporal atom stored in nested (SofBTA_Table) or array table collection type (SofBTA_Array).

We evaulated the performance of each implementation method to gain insight into their relative performance. It is our hope that our work will lay the foundation for the implementation of bitemporal relational databases.

1.4 Organization of the Thesis

The rest of the thesis is organized as follows: Chapter 2 presents basic temporal definitions used in the thesis as well as the different concepts proposed in various temporal data models, which are essential to all temporal data models.

Chapter 3 presents a survey of relevant existing temporal database models and query languages. The chapter is divided into three parts. In the first part, temporal relational models are described, and their most important characteristics are presented. The temporal entity-relational model presented in the second part and the object-oriented temporal models are covered in the third part of this chapter. Lastly, XML temporal databases are explained. Different representative research approaches in the field of temporal databases are described, and the same example relation is presented in each of these models, to illustrate the differences in their representational capabilities.

Chapter 4 gives the definition of nested relations and the structure of the nested bitemporal relations, where the Nested Bitemporal Relational Model also is formalized. The Nested Bitemporal Relational Algebra operations for bitemporal context and historical context are formally defined. The Nested Bitemporal Relational Calculus for both contexts is also defined, and the full expressive power of NBRM is demonstrated by a series of examples.

Chapter 5 describes how the Nested Bitemporal Relational Model is implemented, and how the preprocessor BtSQL works. Implementation is based on building the bitemporal atom as String implementation, and as an abstract data type using Oracle9i, a commercial object-relational DBMS.

Finally, Chapter 6 summarises the achievements of this thesis, followed by future work. Appendix A contains a brief description of the prototype implementation that has been undertaken in Oracle9i. The preprocessor BtSQL is elucidated in Appendix B.

Chapter 2

2. BACKGROUND

2.1 Modeling Time

Databases model some mini-world that has a domain, namely, universe U. This universe is the set of all atomic values such as integers, real numbers, characters, strings, and the value null. Some values in U represent time, and T denotes the set of these values [Ta86]. This set is called a *time domain*, and it is ordered by the relationship "≤". There is no known beginning or ending of time. Naturally, we regard time as continuous; it started at some point in the past and is going into the future.

T, the time domain, is *continuous* and consists of all real numbers between relative origin T_0 and ∞. Real numbers are a better approximation for T because they can accommodate any time granularity. Time can be modeled as linear or branching. Depending upon the application, linear time may be considered discrete, dense, or continuous [JDB+98].

Time domain T, as a subset of a real line, is *discrete* if there exists a real number, $\varepsilon > 0$, such that the distance between any two points in T is at least ε. If T is discrete, then it is a finite set. This is because T is bounded, and the time line is isomorphic to the integers. Clifford made the discreteness assumption in [CT85]. T is said to be *dense* if for any two instants t_1 and t_2, an instant t_3 satisfying $t_1 < t_3 < t_2$ is required to exist. Thus, limitless precision is guaranteed and the time line is isomorphic to the rational numbers.

In the *branching* model, time is linear from past to "now," where it then divides into several time lines, each representing a potential sequence of events. Along with any

future path, additional branches may exist. The structure of branching time is a tree rooted at now [OS95].

A finite and discrete time domain is more suitable for databases because the time model is to be implemented on a discrete computing device. Time values range over integers 0, 1, 2, ..., *now*, where 0 represents the relative origin of time and now is the special symbol representing the current time that separates the past from the future. Figure 2.1 shows the time axis.

The smallest nondecomposable unit of time on the time axis is defined as being a t*ime unit* [Ta86], [NA87]. *Instant* [Ga86], *moment* [SA85], and *chronon* [Ar86], [CR87] can be assumed to be the same [BJW00].

2.1.1 Time Instants and Events

A time instant is a *time point* on the time axis, and an event is an isolated instant in time that results in some actions that may cause changes in the states of some objects in the real world. An *event* is defined as occurring at time t if it occurs at any time during the time unit or the chronon represented by t [T^+93]. An *event* is instantaneous and a *state* is something that extends over time. In Figure 2.1, an event e_1 occurred at time point 0, and generated data values that are valid until time point 3, when an event e_2 happened and replaced these data values.

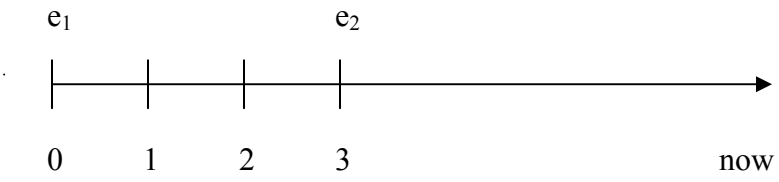


Figure 2.1: Time points and events

2.1.2 Time Intervals

A time *interval* is the time between two events. It may be represented by a set of consecutive time units t_ℓ and t_u where t_ℓ is the starting time instant (lower bound) and t_u is

the ending time instant (upper bound) $[t_\ell , t_u]$. Time intervals are natural subsets of T. There are four possible notations for time interval.

An interval $[t_\ell, t_u]$ is closed at both lower and upper bounds, $\{t \mid t_\ell \le t \le t_u\}$,

$[t_\ell, t_u)$ has a closed lower but an open upper bound, $\{t \mid t_\ell \le t < t_u\}$,

$(t_\ell, t_u]$ has an open lower but a closed upper bound, $\{t \mid t_\ell < t \le t_u\}$,

(t_ℓ, t_u) has open lower and upper bounds, $\{t \mid t_\ell < t < t_u\}$.

A *temporal set* is any set of time points, and a continuous temporal set that contains consecutive time points $\{t_i, t_{i+1}, \ldots, t_{i+n}\}$ is represented either as a closed interval $[t_i, t_{i+n}]$ or as a half-open interval $[t_i, t_{i+n+1})$ [Ta86].

Allen [Al83] defined interval operations. Through the use of the set theoretic operations, union, intersection, and difference are defined for intervals. However, the set of all intervals in T is not closed under these operations.

Example 2.1: Assume three time intervals $I_1 = [10, 20)$, $I_2 = [23, 26)$ and $I_3 = [5, 36)$.
The union of I_1 and I_2 returns a set of intervals:
$I_1 \cup I_2 = \{[10, 20), [23, 26)\}$
The difference of I_3 and I_2 returns a set of intervals:
$I_3 - I_2 = \{[5, 23), [26, 36)\}$
and $I_1 - I_3$ results in an empty set.

There are thirteen different ways of comparing two intervals with each other [Al83], as shown in table 2.1. A and B are time intervals where begin(A) and end(A) represent the starting time instant t_ℓ and the ending time instant t_u, respectively, of time interval A $= [t_\ell, t_u)$.

Table 2.1: Allen's temporal comparison predicates.

Comparison Predicate	**Predicates with Endpoints**
A **before** B A **after** B	end(A) < begin(B) end(B) < begin(A)
A **during** B A **contains** B	(begin(A) > begin(B) ∧ end(A) ≤ end(B)) ∨ (begin(A) ≥ begin(B) ∧ end(A) < end(B)) (begin(B) > begin(A) ∧ end(B) ≤ end(A)) ∨ (begin(B) ≥ begin(A) ∧ end(B) < end(A))
A **overlaps** B A **overlapped** by B	begin(A) < begin(B) ∧ end(A) > begin(B) ∧ end(A) < end(B) begin(B) < begin(A) ∧ end(B) > begin(A) ∧ end(B) < end(A)
A **meets** B A **met_by** B	end(A) = begin(B) end(B) = begin(A)
A **starts** B A **started** by B	begin(A) = begin(B) ∧ end(A) < end(B) begin(A) = begin(B) ∧ end(B) < end(A)
A **finishes** B A **finished_by** B	begin(A) > begin(B) ∧ end(A) = end(B) begin(B) > begin(A) ∧ end(A) = end(B)
A **equals** B	begin(A) = begin(B) ∧ end(A) = end(B)

2.1.3 Temporal Element

A temporal element is the finite union of disjoint time intervals [Ga88]. For instance, {[1, 8) ∪ [10, 20] ∪ [23, 36]} is a temporal element. The set of temporal elements are closed under the set theoretic operations of union, intersection, and complementation, with T (the set of time instants) as its maximum element and ∅ as its minimum element, such that it forms a Boolean algebra [GV85]. This illustrates how time points, intervals, and temporal elements are essential temporal data types for modeling and querying temporal data.

Let A and B be two temporal elements. Then, the union of temporal elements is the temporal element defined as:

$A \cup B = \{t | t \in A \vee t \in B\}$

The intersection of temporal elements is the temporal element defined as:

$A \cap B = \{t | t \in A \wedge t \in B\}$

The difference of temporal elements is the temporal element defined as:

$A - B = \{t | \ t \in A \wedge t \notin B\}$

Example 2.2: Figure 2.2 depicts a time point, a time interval, and a temporal element.

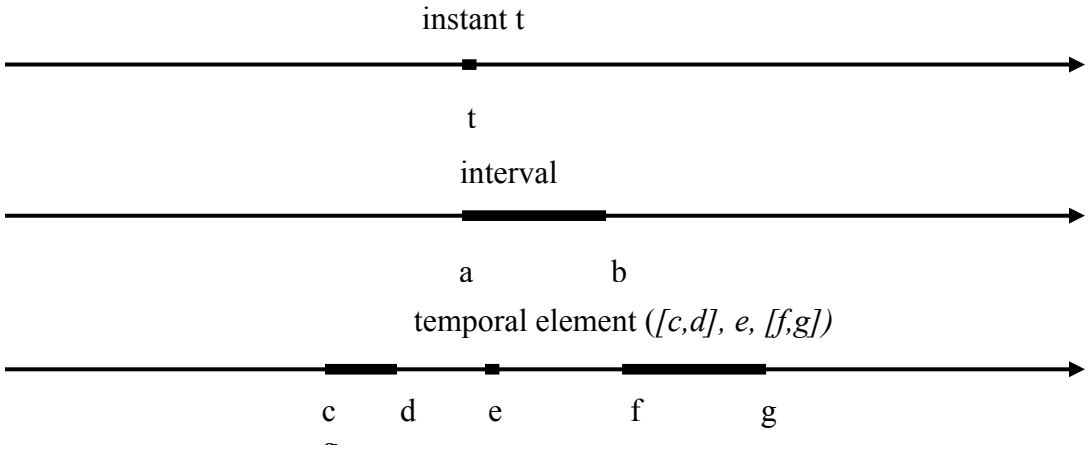


Figure 2.2: Time point, time interval, and temporal element on the time axis

Example 2.3: Figure 2.3 shows the results, given two temporal elements A and B for the set operations union, intersection, complementation (not), and set difference, which are used respectively.

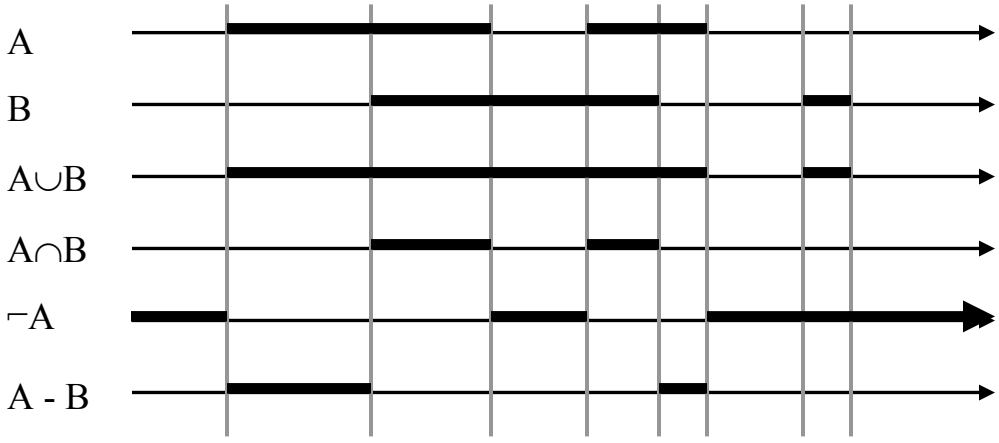


Figure 2.3: Set operations on temporal elements

2.1.4 Time Granularity

For our daily life, the ways to quantify time and to specify certain points in time use different *granularities*. For example, a *calendar* structures time into different time units, such as years, days, minutes, and seconds. It is necessary to consider multiple interrelated temporal domains for multiple time granularities. An instant in a "higher level" domain corresponds to a contiguous set of instants in another, "lower level" domain. Clifford and Rao introduced a temporal universe in [CC88]. A general structure for time domains consists of a totally ordered set of granularities and operations are defined to convert different anchored times to a finer granularity before carrying out the operation. [WJS93] provided semantics for moving up and down a granularity lattice. Various time granularities and their equivalence within calendar systems are defined in [BJW00].

2.2 Representing Temporal Data

Temporal data means that the data are defined to have some time-related information associated with them. A *timestamp* is a time value (time point, time interval, or temporal element) associated with a data value. There are different notions of time that may be orthogonal to each other. To capture different aspects of temporal reality, more than one time dimension was proposed for temporal databases because data values may carry different semantics depending on the time dimension [Sn87].

2.2.1 User-Defined Time

A column of a relation whose domain is time, is not interpreted by the DBMS. Such an attribute is called *user-defined time* [SA86], because the DBMS treats this temporal data like any other ordinary data. For example, a value in an attribute Birthday is an example of user-defined time. The time data are only meaningful to the user, as values are supplied and, if necessary, updated by the user. Data types such as DATE, TIME, and DATE-TIME are used as the domain of user-defined time.

2.2.2 Valid Time

Valid time represents when a fact (data value) was, is, or will be true in the modeled reality. The validity period may be in the past, present, or possibly in the future [SA86]. Timestamps are provided by the user or implied by the user transactions when adding or modifying data.

2.2.3 Transaction Time

Transaction time represents the recording time of the values in the database. The DBMS creates the transaction time, and it cannot be later than the current time. These system-generated values grow monotonically as the database state evolves. Valid time and transaction time are orthogonal time lines; that is, one is independent of the other [Sn87].

2.2.4 Other Time Lines

We can consider other possible time lines. One example is *Decision time*, which may be recorded in the database when a decision is made about the value of an attribute, such as the promotion of an employee to a new position. Because the number and meaning of the decision time(s) of an event varies from application to application, the desirability of building decision-time support into temporal database technologies is unclear. Therefore, decision time has hardly been considered or supported in temporal database proposals. Naturally, including the decision time increases the complexity of the data model.

2.3 Time-Stamping Temporal Data

We will explain time-stamping with time points, time intervals, and temporal elements. It will be explained in the relational data model, but the same idea can be extended to other models such as object-oriented or entity relationship data models. Although the proposed temporal relational data models differ in many aspects, there are two common approaches with respect to where the timestamps are attached: 1) tuple time-stamping, which uses first normal-form (1NF) relations, and 2) attribute-value time-

stamping, which requires non-first normal-form (N1NF) relations. Clifford, Croker, and Tuzhilin formally captured the distinction between tuple time-stamped and attribute time-stamped by the concepts of *temporally ungrouped* and *grouped* data models [CCT94].

2.3.1 Tuple Time-Stamping

Temporal data models with a tuple time-stamping approach stay within 1NF relations, and add timestamps to each tuple in a relation. Update operations require inserting a new tuple into a temporal relation, which results in data redundancy. In case there is more than one temporal attribute, new values on each attribute significantly increase redundancy. Decomposing temporal attributes into separate tables reduces this data redundancy in the tuple time-stamping approach, which results in many small relations. We will show an example only on the SALARY attribute from the EMPLOYEE table.

Tuple time-stamping with time points. Tuple time-stamping with time points extends a relation with a time attribute in any time granularity. Data time-stamped with a time instant (point) is usually assumed to be valid only at the specified instant. Relations containing data time-stamped with time instant (point) are called event tables [Sn95]. Event tables cannot store the duration of when an object was valid in reality. It only states that at time *t* some event occurred, when the fact became valid. The relation stays within the first normal form.

Example 2.4: The example EMPLOYEE table becomes an event table using tuple time-stamping with time points. We named it EMPLOYEE_2.2 as depicted in Table 2.2.

Table 2.2: EMPLOYEE_2.2 tuple time stamped with time points

EMP#	ENAME	SALARY	SALARY_TIME
101	Tom	25K	01/2003
101	Tom	30K	02/2005
102	Ann	35K	06/2001
103	John	35K	06/2003
103	John	42K	07/2004
103	John	45K	01/2006

The event table EMPLOYEE_2.2 contains the dates when people had salary increases. The schema is extended with an additional attribute, SALARY_TIME, to hold the time value of when employers had an increase. Clearly, to determine the validity period of salary requires examination of tuples with the same employee number. Also, note that the fact that John left the company and was rehired cannot be deduced from the values of time stamp attributes. An extreme case in this approach is to copy a temporal relation (or the database) at each time point [CW83], [To97].

Tuple time-stamping with time intervals. Two time attributes representing the time reference of tuples are added to 1NF relations. The first one (FROM) represents the time when an attribute value becomes valid, and the second one (TO) denotes the time when that attribute value is replaced by a new value as a result of an update transaction.

Example 2.5: The EMPLOYEE table in Table 2.3 is extended with two time attributes to store the starting and ending time instants of a valid-time interval.

Table 2.3: EMPLOYEE_2.3 tuple time stamping with time intervals.

EMP#	ENAME	SALARY	FROM	TO
101	Tom	25K	01/2003	01/2005
101	Tom	30K	02/2005	*now*
102	Ann	35K	06/2001	*now*
103	John	35K	06/2003	06/2004
103	John	42K	07/2004	12/2004
103	John	45K	01/2006	*now*

In tuple time-stamping, both valid and transaction times can be included. For each time dimension, time points or time intervals can be used. As an example, we give EMPLOYEE.2.4, where intervals are used to represent valid time and transaction time. Table is extended with four attributes: two attributes are used to capture valid time and two attributes are used to define transaction time.

Table 2.4: EMPLOYEE_2.4 tuple time stamping with both valid and transaction time intervals

EMP#	ENAME	SALARY	START	STOP	FROM	TO
101	Tom	25K	01/2003	01/2005	12/2002	12/2004
101	Tom	30K	02/2005	*now*	01/2005	*now*
102	Ann	35K	06/2001	*now*	05/2001	*now*
103	John	35K	06/2003	06/2004	05/2003	05/2004
103	John	42K	07/2004	12/2004	06/2004	12/2004
103	John	45K	01/2006	*now*	12/2005	*now*

The benefits of the tuple time-stamping approach are that tuple time-stamped relations can be implemented easily on top of commercial relational database systems. Application developers and users understand the structure of a temporal database's relations and can express queries using this approach. Because the tuple time-stamping approach uses 1NF, well-understood storage organization, query optimization, and query evaluation techniques and be adopted. Traditional functional dependencies are easily applied. In contrast, data redundancy is the main disadvantage of this approach.

Other issues in tuple time-stamping. The tuple time-stamping approach splits the object's history into several tuples that create redundancy. Each tuple contains a data value, the beginning time when this data value became valid, and then at the end of its validity. If the validity no longer holds, the tuple's time reference is closed by adding the ending time, and a new tuple with its start time is inserted. Splitting a logical unit of data into more than one tuple is called a *vertical temporal anomaly* [Ga88]. Table 2.5 has two time-related attributes. An update on each time-related attribute causes repetition of all the other data values in the tuple. Because SALARY and DEPARTMENT values do not necessarily change at the same time, these two attributes are split into two relations (Table 2.6-a and Table 2.6-b). This forced splitting of the horizontal format of a relation is called the *horizontal temporal anomaly*. When trying to avoid the horizontal anomaly, the vertical anomaly increases [Ga88].

Table 2.5: An example of a vertical temporal anomaly

EMP #	**NAME**	**SALARY**	**Time_ start**	**Time_ end**	**DEPT.**	**Time_ start**	**Time_ end**
101	Tom	25K	01/2003	01/2005	Sales	01/2003	01/2005
101	Tom	30K	02/2003	*now*	Sales	01/2003	01/2006
101	Tom	30K	02/2005	*now*	Marketing	02/2006	*now*
102	Ann	35K	06/2001	*now*	Sales	06/2001	*now*
103	John	35K	06/2003	06/2004	Toys	06/2003	12/2004
103	John	42K	07/2004	12/2004	Toys	06/2003	12/2004
103	John	45K	01/2006	*now*	Marketing	01/2006	04/2006
103	John	45K	01/2006	*now*	Marketing	05/2006	*now*

Table 2.6-a: An example of a horizontal temporal anomaly

EMP#	**ENAME**	**SALARY**	**Time_ start**	**Time_ end**
101	Tom	25K	01/2003	01/2005
101	Tom	30K	02/2005	*now*
102	Ann	35K	06/2001	*now*
103	John	35K	06/2003	06/2004
103	John	42K	07/2004	12/2004
103	John	45K	01/2006	*Now*

Table 2.6-b: An example of a horizontal temporal anomaly

EMP#	**ENAME**	**DEPARTMENT**	**Time-start**	**Time end**
101	Tom	Sales	01/2003	01/2006
101	Tom	Marketing	01/2006	*Now*
102	Ann	Sales	06/2001	*Now*
103	John	Toys	06/2003	12/2004
103	John	Marketing	01/2006	*Now*

Two tuples are *value equivalent* if their non-timestamp attribute values and the value component of the time-dependent attributes are identical in both tuples. Value-equivalent tuples are similar to duplicates that are not allowed in a conventional relational data model. Value-equivalent tuples are not allowed in a temporal relation, and they must be coalesced. A relation instance is *coalesced* if it does not contain two or more value-equivalent tuples with overlapping or consecutive valid-time periods.

A tuple is temporally *homogeneous* if each of its attributes has values over the same time period. A temporal relation is said to be temporally homogeneous if its tuples are temporally homogeneous [Ga88].

2.3.2 Attribute Time-Stamping

Time-stamps in attribute time-stamping – regardless of being time points, time intervals, or temporal elements – are attached to attributes whose values become temporal atoms. *Atom* is the basic undefined term on which the recursive definitions are built. An atom takes its values from U. A *temporal atom* is a <*t, v*> pair where *t* is a timestamp and *v* is an atomic value. A temporal atom asserts that the value is valid (or recorded) over period *t* [TG89]. The history of values is stored separately for each attribute, and the entire history of an object is stored in one tuple. Thus, values in a tuple that are not affected by an update do not have to be repeated. This approach represents a three-dimensional view of temporal data commonly adopted by researchers in this field [Ta90].

Attribute time-stamping requires that the underlying data model supports N1NF relations, because time-stamped attributes store the attribute value together with its timestamp. All temporal attributes can be included in one relation. This relation may have non-temporal attributes along the temporal attributes such as the EMP# and ENAME attributes in the EMPLOYEE relation in Table 1.1 on page 5.

Attribute time-stamping with time points. A temporal atom's timestamp part can be taken as the time point at which the attribute value becomes valid [CW83].

Using the time point splits the valid (or transaction) time of an attribute value interval between two successive pairs of temporal atoms**.** Therefore, to determine the end-of-validity period of a value, the successor pair needs to be examined. This creates complications in querying data.

Example 2.6: The EMPLOYEE table with time points is depicted in Table 2.7.

Table 2.7: EMPLOYEE_2.7 table with time points

EMP#	ENAME	SALARY	DEPARTMENT
101	Tom	<01/2003, 25K>, <02/2005, 30K>	<01/2003, Sales>, <02/2006, Marketing>
102	Ann	<06/2001, 35K>	<06/2001, Sales>
103	John	<06/2003, 35K>, <07/2004, 42K>, <01/2006, 45K>	<06/2003, Toys>, <01/2006, Marketing>, <05/2006, Toys>

Attribute time-stamping with time intervals. A temporal atom's timestamp part is the interval in which the value is valid, e.g., < [t_l, t_u], v>, where *l* and *u* are the lower and upper bounds of an interval.

Example 2.7: Consider the EMPLOYEE table given in Table 2.8, where time intervals are used. Tom's history of SALARY and DEPARTMENT are included in one tuple. Unlike the tuple time-stamping, the EMP# and ENAME attributes are no longer repeated.

Table 2.8: EMPLOYEE_2.8 table with time intervals

EMP#	ENAME	SALARY	DEPARTMENT
101	Tom	{<[01/2003, 1/2005),25K>, <[02/2005, now], 30K>}	{<[01/2003, 01/2006), Sales>, <[02/2006, now], Marketing>}
102	Ann	{<[06/2001, now], 35K>}	{<[06/2001, now], Sales>}
103	John	{<[06/2003, 6/2004),35K>, <[07/2004, 12/2004], 2K>, <[01/2006, now], 45K >}	{<[06/2003, 12/2004], Toys>, <[01/2006, 04/2006], Market>, <[05/2006, now], Toys>}

Attribute time-stamping with temporal elements. The temporal atom's timestamp part may have a temporal element that consists of several nonoverlapping intervals over which the value is valid. Because temporal elements are closed under the set-theoretic operations of union, intersection, difference, and complementation, using the attribute

time-stamping approach provides additional benefits by simplifying the query specification.

Example 2.8: Consider the EMPLOYEE table given in Table 2.9, where temporal elements are used. Notice that John has non-overlapping time intervals between when he left the company and rejoined.

Table 2.9: EMPLOYEE_2.9 table with temporal elements

EMP #	**ENAME**	**SALARY**	**DEPARTMENT**
101	Tom	{<[01/2003, 01/2005), 25K>, <[02/2005, now], 30K>}	{<[01/2003, 01/2006), Sales>, <[02/2006, now], Marketing>}
102	Ann	{<[06/2001, now], 35K>}	{<[06/2001, now], Sales>}
103	John	{<[06/2003, 06/2004), 35K>, <[07/2004, 12/2004], 42K>, <[01/2006, now], 45K >}	{<[06/2003, 12/2004], Toys>, <[01/2006, 04/2006], Market>, <[05/2006, now], Toys>}

The attribute time-stamping approach offers advantages over tuple time-stamping, though it requires nested relations (N1NF) where tuples have a set of composite values. By using N1NF relations, it avoids redundancy and is definitely more expressive. The attribute time-stamping model supports temporally heterogeneous as well as homogeneous data. This approach models an object's entire history in a single tuple rather than splitting it into several tuples. An attribute time-stamping model provides a more natural view, closer to how a user might perceive reality, and consequently likely to be easier to design or query. Many temporal attributes are allowed within one relation. In addition, temporal attributes can be expressed using different time granularities (i.e., years, days, months, seconds) in the same relation. The attribute time-stamping approach has been criticized as overly complex by many researchers. However, object relational DBMS, SQL3, and advances in database technology allow us to model temporal data with attribute time-stamping by using N1NF relations.

Nevertheless, attribute time-stamped relations present a number of shortcomings. Nested relations are more complex than 1NF relations. Queries can become very

complex, even in those cases where only one relation is involved. Definitions of algebraic operations for an attribute time-stamping temporal database model can become very complicated. Nested relations can be represented in different ways using different structures, and restructuring operations must also be defined. Traditional functional dependencies cannot be applied; new functional dependencies must be defined.

2.4 Temporal Databases

Temporal databases capture different aspects of time to store the history of an object being modeled. According to the taxonomy of temporal databases presented in [SA85], four categories of temporal databases are identified with respect to the valid and transaction times. These four different types are snapshot, historical, rollback, and bitemporal.

Table 1.3, on page 10, with histories of SALARY and DEPARTMENT, describes the different types of temporal databases. The time granularity in this example is interpreted as months. It is assumed that the company knows about the new employment information a month before the effective date, whereas information about raises, transfers, and terminations enters the database after their effective date.

2.4.1 Snapshot Databases

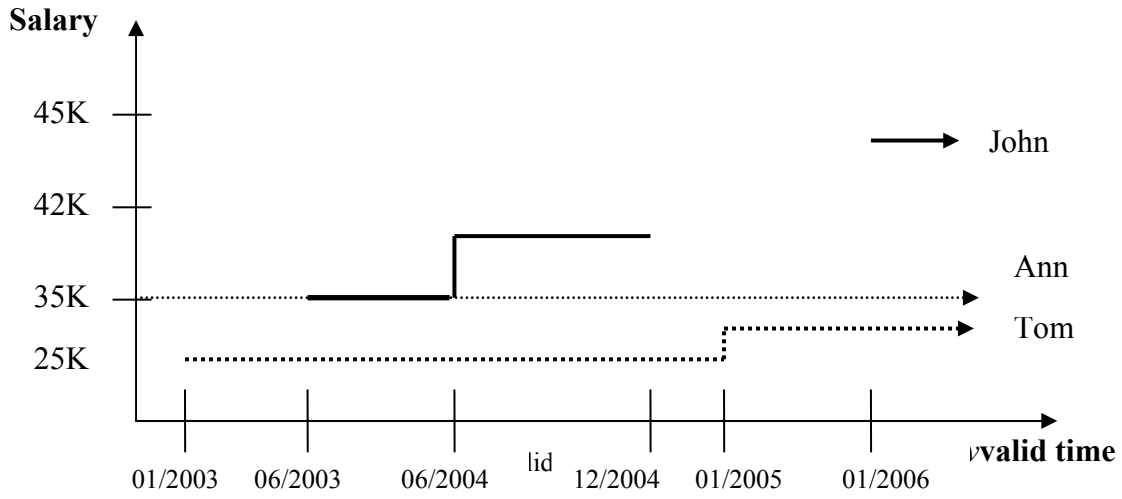
Conventional databases model the dynamic real world. These databases represent the state of an enterprise at one particular time – generally, the current state. When new data values are replaced with old ones, data representing past states are discarded. These types of databases are called *snapshot databases* because they only capture a snapshot of reality; i.e., they contain current data, which are snapshots of the current reality. Snapshot databases only support user-defined time.

2.4.2 Historical Databases

Historical (valid time) databases record database states along the valid-time line. A historical database contains historical (valid time) relations. Its history of values changes

with respect to the real world, storing the history, known as of now. Database states past, present, or future are recorded with valid time. The database user supplies the valid-time values. If an error is discovered, an update operation has to be performed to correct the error, which causes the database state to change. The previous value is discarded and the erroneous data are lost. Therefore, the database may not be viewed as it was in the past. Thus, historical databases are similar to snapshot databases in this respect.

Using the information from Table 1.3, we can create Figure 2.4. Ann has been working since 01/2001, receiving a salary of 35K. John started work in 06/2003, and had a salary increase on 06/2004 from 35K to 42K. On 12/2004, the line representing John's salary history stops; he probably left the company. On 01/2006, the line reappears on the table, indicating that John came back to the company at a salary of 45K.



2.4.3 Rollback (Transaction Time) Databases

A rollback (transaction time) database records data along the transaction time line, or as a sequence of snapshot relations indexed by transaction time – that is, the history of database activities as recorded in the database. Rolling the database back to a state at some time point in the past is possible in a rollback database. Because transaction time is system generated, rollback databases may not record future database states.

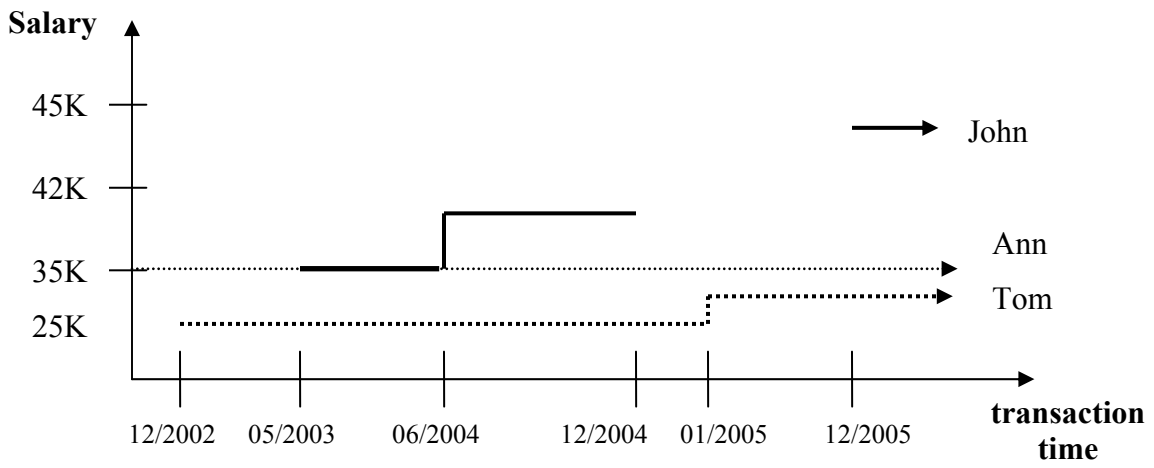


Figure 2.5: Transaction time salary history

2.4.4 Bitemporal Databases

Historical databases only model the history of an object and do not preserve the changes in the system. However, rollback databases do not carry the true changes about an object being modeled. If the reality is to be modeled accurately, both historical and rollback databases should be combined, and the system should support both valid time and transaction time. Bitemporal database systems support both valid time and transaction time. By doing so, the system actually models our changing knowledge of the changing real world and hence associates values with facts, and specifies when the facts were current in the database. That is, the system allows storing retroactive and postactive changes in the database.

Figure 2.6 shows John's salary history on the two dimensional axes, where the horizontal axis is the valid time and the vertical axis is the transaction time. The state of the database can be computed at a desired transaction time by moving through the space of validity rectangles along the horizontal transaction time axis.

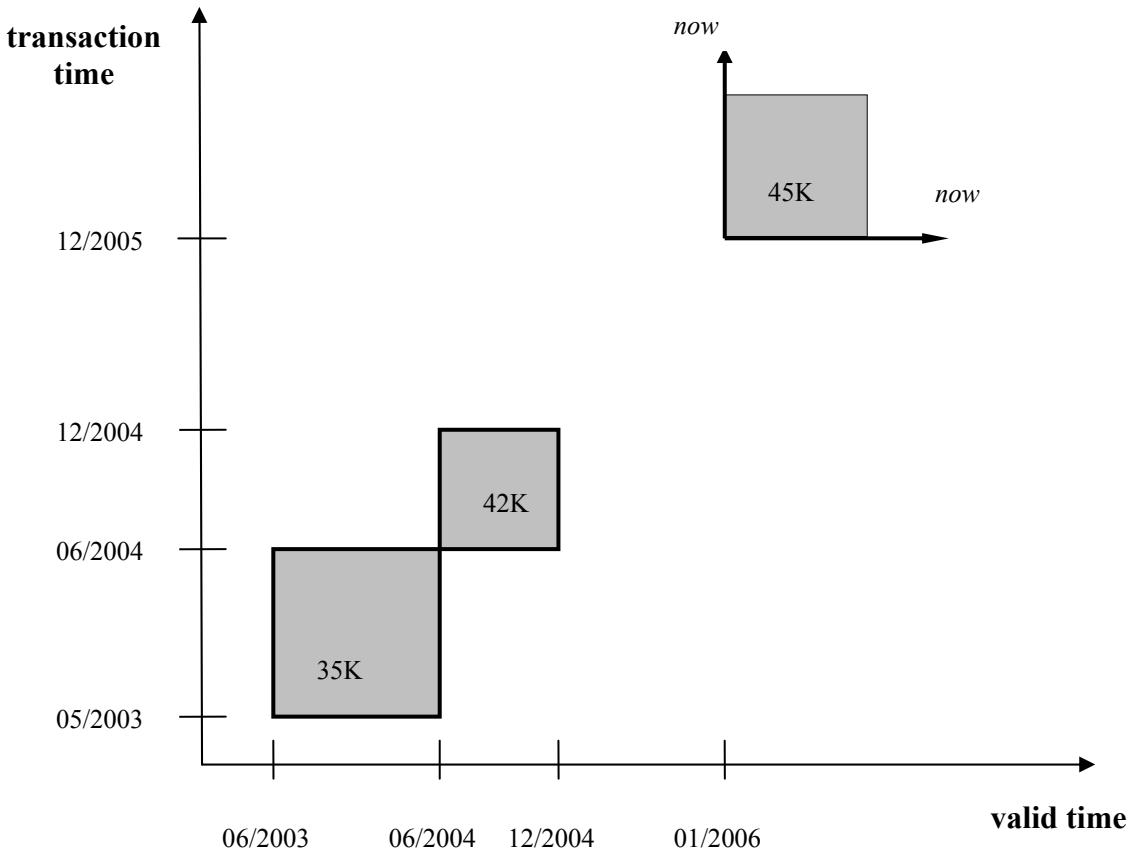


Figure 2.6: Valid and transaction time salary history

2.5 Time Support in SQL92 and SQL3

SQL is firmly established as the predominant database language for database access. While SQL86 and SQL89 did not include any support for temporal data types, some of the temporally-oriented changes were introduced in SQL92 to model time points such as DATE, TIME, and TIMESTAMP.

The DATE data type stores year, month, and day values of a date. The year value can represent the years 0001 through 9999, the month value is 01 through 12, and the day value is limited to values 01 through 31. The month value can apply additional restrictions on the day value to a maximum of 28, 29, 30. The TIME stores the hour, minute, second, and a fraction of second values of a time. The hour value can represent the hours 00 through 23, the minute value is 00 through 59, and the second value is

restricted to 00 through 59. The TIMESTAMP stores the year, month, and day values of a date as well as the hour, minute, and second values of a time. TIME and TIMESTAMP data types can also be specified with precision number *p*, which cannot be negative. The length of DATE, TIME, and TIMESTAMP is 10, 8 and 18 digits, respectively.

Example 2.9: (2001-11-25), (15:15:01) and (1999-02-15 21:10:01) are examples of SQL92 time-related data types, DATE, TIME and TIMESTAMP, respectively.

CURRENT_DATE, CURRENT_TIME, and CURRENT_TIMESTAMP are the three "current" functions in SQL92. While CURRENT_DATE does not take any arguments, the other two have one argument, precision *p*, to specify the fractional seconds' precision to be returned with the time.

Although these data types allow some temporal information to be modeled in a relational database, the support is inadequate, and thus SQL92 is deficient for use in temporal applications. The formulation of accurate queries is dependent on the user's semantic knowledge of the entity being described.

SQL3 times are specific with a relationship to UTC, or Universal Coordinated Time (Greenwich Mean Time). SQL3 requires that every SQL session have associated with it a default offset from UTC that is used for the duration of the session, or until explicitly changed by the user [MS02]. In addition to three specific forms of date and time in SQL92, two new ones are defined in SQL3, TIME WITH TIME ZONE and TIMESTAMP WITH TIME ZONE. They are like TIME and TIMESTAMP, respectively, also include additional information from UTC of the time specified.

Example 2.10: (15:12+04:00) and (1999-02-15 21:10:01-03:00) are examples of SQL3 time-related data types, TIME WITH TIME ZONE and TIMESTAMP WITH TIME ZONE, respectively.

LOCALTIME and LOCALTIMETIMESTAMP are new datetime value functions in SQL3. They both have an optional precision argument. While LOCALTIME returns the current time, LOCALTIMETIMESTAMP returns a timestamp with the number of precision decimal places.

INTERVAL: *year-month* and *day-time* are the two categories of intervals defined in SQL3 and cannot be mixed in any expression. SQL3's intervals are durations of time without an express starting or ending time. PERIOD intervals are a specific time span with a known start and/or finish; but it is not defined or used in SQL3.

Example 2.11: (INTERVAL '5' YEAR) for 5 years,
(INTERVAL '7' MONTH) for 7 months,
(INTERVAL '3-7' YEAR TO MONTH) for three years and seven months,
(INTERVAL DAY(4) TO HOUR) can represent up to 9999 days and up to 24 hours,
(INTERVAL HOUR(3) TO SECOND) can represent up to 40 days,
(INTERVAL MINUTE(4) TO SECOND) can represent almost a week,
PERIOD (2006-09-25 + '5' DAY) results on 2006-09-29.

year-month intervals can contain only a year value, only a month value, or both: INTERVAL YEAR, INTERVAL MONTH, or INTERVAL YEAR TO MONTH, with an optional precision. If precision is not specified, only 2 digits are returned by default. *day-time intervals* can contain only a day value, an hour value, a minute value, and/or a second value: INTERVAL DAY TO HOUR, INTERVAL DAY TO MINUTE, INTERVAL SECOND, INTERVAL DAY TO SECOND, and INTERVAL MINUTE TO SECOND. Note that the use of the term *interval* in SQL3 is different than the definition we have provided for *interval* earlier.

Datetime value expressions allow combining datetime and interval values in SQL3 statements. They operate on date-oriented data types such as DATE, TIME, TIMESTAMP, and INTERVAL. The result of such operations is always another datetime.

Example 2.12: CURRENT_DATE + INTERVAL ‘7’ DAY results as next week.

(TIME ’09:30:00’ AT LOCAL) represents the time of 9:30 in local time zone.

(TIME ’09:30:00’ AT TIME ZONE INTERVAL ‘-7:00’ HOUR TO MINUTE) represents the time in Istanbul.

Two intervals can be added or subtracted to get another interval. A single interval can be multiplied or divided by a numeric constant. Year-month and day-time intervals cannot be mixed in a single interval value expression.

OVERLAPS: This predicate's purpose is to determine whether two periods of time overlap with one another. The format of the OVERLAPS predicate is

event-information OVERLAPS *event-information*

where event-information can be of the form (start-time, duration) or (start-time, end-time), but they don’t have to be the same form. Both start-time and end-time have to be the same datetime, which is DATE, TIME, or TIMESTAMP. For example,

Example 2.13: The following expression results in a value of false because these two do not overlap.

(TIME ‘10:45:00’, INTERVAL ‘1’ HOUR) OVERLAPS

(TIME‘10:00:00’, TIME‘10:30:00’)

But the following expression is true.

(TIME‘10:30:00’, TIME‘11:30:00’) OVERLAPS

(TIME‘11:00:00’, TIME‘12:00:00’)

SQL3’s date capability cannot handle B.C.E. or B.C. (Before the Common Era or Before Christ) dates yet, because of a lack of general agreement over issues such as how to handle the year 0.

Chapter 3

3. A SURVEY OF TEMPORAL DATA MODELS

3.1 Temporal Relational Data Models

In the following, temporal relational proposals are classified with respect to the time dimension they support and where the time-stamps are attached. Time-stamp type is specified for each model. Table 3.1 gives a partial list of the proposed temporal data models. A representative of each category is explained, because it is typical and provides insight into novel research contributions. The tuple time-stamping approach is explained first, valid time models [Ar86], [NA87], [Lo88], [Sa90], transaction time models [JMR91], and both valid and transaction time models (i.e., a bitemporal data model) [BZ93], [Sn93], in that order. Then, attribute time-stamping valid time models [CC87], [Ga86], [Ta86], the transaction time model [BG93], and the both valid time and transaction time model [GB89] are covered. The EMPLOYEE relation from Chapter 2 is used for each model as a running example.

Ariav (TOSQL), Navathe-Ahmed (TSQL), Sarda (HSQL), Snodgrass (TQuel), Tansel (HQuel), and Gadia (HTQuel) have developed query languages for their data models. These temporal query languages are also explained.

Table 3.1: Temporal relational data models

	Valid Time	**Transaction Time**	**Valid + Transaction Time**
Tuple Time-stamping	Ariav [Ar86] NavatheAhmed [NA87] Lorentzos [Lo88] Sarda [Sa90]	Jensen [JMR91]	Ben_Zvi [BZ93] Snodgrass [Sn93]
Attribute Time-stamping	Gadia[Ga86] Tansel [Ta86] Clifford [CC87]	BhargavaGadia [BG93]	GagiaBhargava [GB89]

3.1.1 Tuple Time-stamped Temporal Relational Data Models

3.1.1.1 Tuple Time-stamped Historical Relational Data Models

Ariav's temporally oriented data model (TODM) time-stamps tuples with event-based valid time instants [Ar86]. His model is based on the notion of a *data cube*, which is a three-dimensional extension of the plain relations in which a time dimension is the third dimension. The explicit and inherent order of the tuples preserves the temporal context of the data. To determine how long a state lasted or what the state was at a given time on the chronological order of tuples, an interpolation function is necessary. The state of a relation at any time point is determined by slicing the cube in a depth corresponding to the specified time. Ariav defined object selection, temporal selection, and projection as operations over a data cube to form new cubic views from existing ones. Table 3.2 illustrates Ariav's model.

Because only the event times are stored in the database, it is difficult to determine the accuracy of data at previous time points based upon the data stored in this model. For instance, we cannot deduce that John actually left the company at time point 12/2004 and rejoined at time point 01/2006.

Table 3.2: EMPLOYEE_3.2 with Ariav's model

EMP#	ENAME	SALARY	Event_time
101	Tom	25K	01/2003
101	Tom	30K	02/2005
102	Ann	35K	06/2001
103	John	35K	06/2003
103	John	42K	07/2004
103	John	45K	01/2006

TOSQL, an extension to SQL, is a calculus-based query language proposed by Ariav; its query syntax allows end users to incorporate temporal notions as well as to query only the current view of data. TOSQL includes AT, WHILE, DURING, BEFORE, AFTER, and AS OF constructs.

Navathe-Ahmed's temporal relational model (TRM) models valid time using time points and time intervals; if needed, transaction time can also be incorporated by additional time-stamp attributes [NA93]. In TRM, temporal relations are augmented with two implicit time-stamp attributes, time-start (TS), and time-end (TE), corresponding to the lower and upper bound of a time interval $[T_S, T_E]$, respectively. If $T_S = T_E$, then this model supports events. The primary key is chosen as a time-invariant attribute, and there are two candidate keys (the time-invariant key, plus either TS or TE) for temporal relation. An important contributions of their model are the notion of time normalization, as well as a language construct, which is called the *moving window*. Value-equivalent tuples are required to be coalesced so contiguous intervals are not represented by two value-equivalent tuples in their model.

Navathe and Ahmed classify temporal attributes as synchronous or asynchronous. While synchronous attributes always change at the same time, asynchronous values change independently of each other. Time normalization is defined as decomposing relations into subrelations where all temporal attributes change their values simultaneously. However, in the case that synchronous attributes are not available, time

normalization degenerates to tuple time-stamping over relations, causing the vertical temporal anomaly. Navathe and Ahmed decomposed the EMPLOYEE relation into two relations, since SALARY and DEPARTMENT are asynchronous attributes; see Table 3.3 and Table 3.4.

Table 3.3: EMPLOYEE_3.3 relation for the SALARY attribute.

EMP#	**ENAME**	**SALARY**	**time_start**	**time_end**
101	Tom	25K	01/2003	01/2005
101	Tom	30K	02/2005	*Now*
102	Ann	35K	06/2001	*Now*
103	John	35K	06/2003	06/2004
103	John	42K	07/2004	12/2004
103	John	45K	01/2006	*Now*

Table 3.4: EMPLOYEE_3.4 relation for DEPARTMENT attribute

EMP#	**ENAME**	**DEPARTMENT**	**time-start**	**time_end**
101	Tom	Sales	01/2003	01/2006
101	Tom	Marketing	02/2006	*Now*
102	Ann	Sales	06/2001	*Now*
103	John	Toys	06/2003	12/2004
103	John	Marketing	01/2006	05/2006
103	John	Toys	06/2006	*Now*

Navathe and Ahmed have developed a temporal query language, TSQL, which is a superset of SQL [NA89]. TSQL has new temporal ordering, temporal group-by, moving window, and time-slice components. The WHEN clause specifies predicates on time attributes. The TIME-SLICE clause selects tuples that are fully or partially valid for the specified time interval or for a given time point from a relation. The MOVING WINDOW clause applies to the referred interval rather than the entire lifespan of an

object to provide aggregate information. These components add powerful time-processing capabilities to the language.

Lorentzos proposed the interval extended relational model (IXRM) for valid time databases to support intervals [Lo93]. IXRM integrates N dimensional intervals into the snapshot relational model, and for each attribute an interval may be drawn from any data type, e.g., date, hour. Lorentzos's model supports time-stamps with a nested granularity and the conversion of a time duration to a particular granularity is possible. If the operands are addition compatible, addition between time durations is also possible. Time-stamps are numeric-valued, explicitly defined by the user, and are viewed and updated directly by the user. A time-stamp does not include its upper bound.

Lorentzos defined five operations: compute, fold, unfold, extend, and normalize. In order to use functions, compute is defined in the new model. Fold returns a relation where time is represented as time intervals from an input relation where time is represented as time points. Unfold is the inverse of the fold operation. The difference between fold/unfold and nest/unnest operations is that fold returns intervals from consecutive points, whereas nest returns sets of attribute values. Extend returns a relation from an initial relation that contains a new attribute consisting of all the time points that are extracted from the time intervals. The normalize operation returns a relation where duplicate tuples are eliminated and adjacent or overlapping intervals are merged into one.

In Lorentzos's model, relations are maintained in 1NF, and attributes are time-stamped, because more than one time interval attribute can coexist in the same relation referring to different data. Hence, the history of an object does not consist of a single tuple, but is split up into many. Consequently, his model does not take full advantage of either the 1NF or the attribute time-stamping approach.

Sarda proposed the Historical DBMS (HDBMS), and its query language HSQL, based on the extended relational data model [Sa93]. HDBMS supports only valid time by adding two new attributes to the relation R, namely FROM and TO. HDBMS assumes

that a temporal database is updated in real time, meaning that valid time and transaction time are the same, and no future data are supported. Events (time instants) and time intervals are modeled with a hierarchy of time units, and operations are defined on both sides' closed intervals. A hierarchy of different granularities is provided. The same attribute values with overlapping or consecutive time intervals are coalesced. To allow faster access to the current data, HDBMS separates each historical relation into two union-compatible relations that are transparent to the user. One relation contains the history of the tuples and the other contains current tuples.

In addition to the standard relational algebra operations, Sarda also defined expand, coalesce, and concurrent product as operators. Expand (EX) converts an interval-stamped relation into an instant-stamped relation by repeating for each time instant. Coalesce (CK) is the inverse operation of Expand. Concurrent Product is similar to Cartesian product, except only tuples of two relations are paired to have overlapping time intervals.

HSQL is an extended query language of SQL for HDBMS that treats a nonhistorical relation as a constant relation. HSQL provides the TIME domain for defining the implicit attributes FROM and TO. To obtain a snapshot relation, a time-slice at *now* is applied.

3.1.1.2 Tuple Time-stamped Rollback Relational Data Models

Jensen, Mark, and Roussopoulos presented an implementation model for the standard relational data model extended with transaction time [JMR91]. Their model stores any view reflecting past or present states, and provides efficient support for access to arbitrary time-slices. The stored views are then used for incremental and decremental computations of user requests. Once entered, the data are never deleted, and the database can be rolled back to a previous state.

A *backlog* for a relation, R, is a relation that contains the complete history of the changes on the relation R [JMR91]. A backlog relation consists of two attributes – one with a single time-stamp value provided by the system and the other with a tag indicating whether the request is insert, delete, or modify. The time-slice operator returns the

portion of the values that existed during the specified time argument. The time-slice operator is a fundamental and nonstandard operator such as unit and aggregate formation. The special variable NOW and transaction time-valued expressions are included in the model. Backlogs are accessible via a query language that makes retrieving detailed rollback data relatively easy. The main disadvantage of this system is the constant growth needed for disk space. Table 3.5 shows the running example in Jensen, Mark, and Roussopoulos's model.

Table 3.5: EMPLOYEE_3.5 in backlog model

EMP#	ENAME	SALARY	FROM	Operation
101	Tom	25K	01/2003	Ins
101	Tom	30K	02/2005	Mod
102	Ann	35K	06/2001	Ins
103	John	35K	06/2003	Ins
103	John	42K	07/2004	Delete
103	John	45K	01/2006	Ins

3.1.1.3 Tuple Time-stamped Bitemporal Relational Data Models

Ben-Zvi proposed a model for temporal databases, a temporal query language, storage architecture, indexing, recovery, concurrency and synchronization, and its implementation [BZ93]. Ben-Zvi's model is a bitemporal model with effective time and registration time, known as valid time and transaction time, respectively. Both times start from a fixed point (or original beginning) to define validity and transaction time intervals and expand to infinity. Therefore, data may be valid in the past, present, and future.

In this model, tuples are augmented with five implicit attributes. Effective-time-start and Effective-time-stop are the end-points of the interval for the valid time. The transaction time interval is represented by the Registration-time-start and Registration-time-end attributes. The user provides the Effective-time-start, whereas Registration-time-start is system generated when a data value is recorded in the database. The

Deletion-time attribute records the time when erroneously entered tuples are logically deleted by adding a time-stamp to it.

Ben-Zvi defined a Time-View operator that computes snapshots from a table. He also introduced time-union, time set-difference, time selection, time-projection, and time-join operators for temporal relations. Ben-Zvi prefixed the select statement of SQL with a Time-View clause, and changed the clause supports that query for changes to a database.

Table 3.6 gives our running example in Ben-Zvi's model. Note that none of the tuples are marked "delete" in Table 3.6, because no errors occurred in the database.

Snodgrass proposed a temporal model that supports valid and transaction times, where tuples are time-stamped with either time instants or time intervals [Sn93]. A time instant is mapped to a single attribute if the relation models events. A temporal relation is augmented with four attributes: *from* and *to* attribute pairs are for valid time, and *start* and *stop* attribute pairs are for the transaction time. While (*from–to*) pairs represent the valid time when a tuple is started and stopped, (*start–stop*) attributes represent the transaction time. User-defined time is also supported. Table 3.7 gives the running example in Snodgrass's model.

Table 3.6: EMPLOYEE_3.6 tuple time stamp with valid and transaction times.

EMP#	ENAME	SALARY	Effective time-start	Effective time-stop	Registration time-start	Registration time-stop	Deletion time
101	Tom	25K	01/2003	01/2005			
101	Tom	30K	02/2005	*Now*			
102	Ann	35K	06/2001	*Now*			
103	John	35K	06/2003	06/2004			
103	John	42K	07/2004	12/2004			
103	John	45K	01/2006	*Now*			

Effective time-stop	Registration time-start	Registration time-stop	Deletion time
01/2005	12/2002	01/2005	–
Now	01/2005	*Now*	–
Now	05/2001	*Now*	–
06/2004	05/2003	06/2004	–
12/2004	07/2004	12/2004	–
Now	12/2005	*Now*	–

Snodgrass presented a temporal query language TQuel, which is an extension of Quel, the query language of INGRES. TQuel is based on the predicate calculus. It also supports aggregates, valid time indeterminacy, and schema evolution. Temporal relations are always in a coalesced state, and temporal operations return coalesced results. TQuel adds three new clauses, *valid, when,* and *as of* to the Quel retrieve statement. The *when* clause is for valid time selection, and it is the temporal analogue of the *where* clause. Temporal predicates such as *overlap, during, before, after, precede,* and *equal* are included in TQuel. The *when* clause is associated with the valid time, and the *as-of* clause is associated with the transaction time. Relations can be either snapshot, rollback, historical, or temporal. The "persistent" keyword is used for rollback or temporal relations, the "interval" or "event" keywords for historical or temporal relations; otherwise, the relation is snapshot.

Table 3.7: EMPLOYEE_3.7 in Snodgrass model.

EMP#	**ENAME**	**SALARY**	**FROM**	**TO**	**START**	**STOP**
101	Tom	25K	01/2003	01/2005	12/2002	12/2004
101	Tom	30K	02/2005	*now*	01/2005	*now*
102	Ann	35K	06/2001	*now*	05/2001	*now*
103	John	35K	06/2003	06/2004	05/2003	05/2004
103	John	42K	07/2004	12/2004	06/2004	12/2004
103	John	45K	01/2006	*now*	12/2005	*now*

Jensen, Soo, and Snodgrass proposed the bitemporal conceptual data model, BCDM, that forms the basis for the TSQL (Temporal Structured Query Language) proposal and allows for multiple representation data models [JSS94]. BCDM captures the essential semantics of temporal relations – rather than the representational, as all the other proposed temporal data models do – and is therefore considered conceptual. Relations in BCDM use tuple time-stamping because they consist of a set of tuples. Each tuple includes an implicit attribute value composed of an ordered pair of integers. The first part of the pair denotes when the fact represented by this specific tuple is true in the modeled

reality (valid time), and the second part denotes when it is current in the stored relation (transaction time). Time is represented in BCDM as temporal elements; each single tuple represents the whole history of a fact. Since the time-stamps associated with the tuples are sets of time units, relations in BCDM are in N1NF. Therefore, only homogeneous tuples are supported in the model. Table 3.8 depicts the running example in BCDM model.

Valid time-slice operators and transaction time-slice operators are defined in BCDM, which take as arguments a bitemporal relation, and return a transaction time relation or a valid time relation, respectively, consisting of all tuples valid during the time value.

The BCDM can be mapped to several existing bitemporal representational data models. Jensen, Snodgrass, and Soo describe mappings to and from several representational data models, for example, to the temporal data model in [Sn93], [BZ93] and a N1NF attribute value time-stamped representation schema (not to be confused with a database "schema"). Unfortunately, BCDM is neither appropriate to present stored data to the user nor to physically store data.

TSQL2 is an extension of SQL-92 [S^{+}94]. TSQL2 provides a conceptual model that captures the semantics of temporal relations without being restricted to a specific internal representation. It is based on a tuple time-stamping data model, and only one implicit attribute is allowed in a relation. Value-equivalent tuples are not allowed in a relation. Three time dimensions are supported in TSQL2: user-defined time, valid time, and transaction time; valid time and transaction time are recorded in implicit attributes. In TSQL2, there are six kinds of relations: snapshot relations, valid-time event relations, valid-time state relations, transaction time relations, bitemporal event relations, and bitemporal state relations. Surrogate values are available in TSQL2 to form unique identifiers that can be compared for equality. Surrogates values cannot be seen by the users. The semantics of arithmetic operations that involve time spans and time instants are not explicitly supported in TSQL2. They are left to the calendar as calendar specific operations. Because TSQL2 treats all instants as indeterminate at finer granularities, time

durations that have mixed granularities cannot be represented. SQL-92 aggregates are extended for temporal domains, and new temporal aggregates such as RISING are also included in TSQL2. Transaction time relations may also be vacuumed to remove the old versions. The algebra underlying TSQL2 was defined after the language definition of TSQL2.

Table 3.8: EMPLOYEE_3.8 in BCDM model.

EMP#	ENAME	SALARY	TIME
101	Tom	25K	{(12/2002, 1/2003), (12/2002, 2/2003), (12/2002, 3/2003),…, (12/2002, 1/2005), (1/2003, 1/2003), (1/2003, 2/2003), (1/2003, 3/2003),…, (1/2003, 1/2005), … (01/2005, 1/2003), (01/2005, 2/2003), (01/2005, 3/2003), …, (01/2005, 1/2005)}
101	Tom	30K	{(1/2005, 2/2005), (1/2005, 3/2005), (1/2005, 4/2005),…, (1/2005, 12/2005), (2/2005, 2/2005), (2/2005, 3/2005), (2/2005, 4/2005),…, (2/2005, 12/2005), ,…, (UC, 2/2005), (UC, 3/2005), (UC, 4/2005),…, (UC, now)}
102	Ann	35K	{(5/2001, 6/2001), (5/2001, 7/2001), (5/2001, 8/2001), …, (5/2001, now), (6/2001, 6/2001), (6/2001, 7/2001), (6/2001, 8/2001), …, (6/2001, now), … (UC, 6/2001), (UC, 7/2001), (UC, 8/2001), …, (UC, now)}
103	John	35K	{(5/2003, 2/2003), (5/2003, 3/2003), (5/2003, 4/2003),…, (5/2003, 6/2004), (6/2003, 2/2003), (6/2003, 3/2003), (6/2003, 4/2003),…, (6/2003, 6/2004), … (6/2004, 2/2003), (6/2004, 3/2003), (6/2004, 4/2003),…, (6/2004, 6/2004)}
103	John	42K	{(7/2004, 7/2004), (7/2004, 8/2004), (7/2004, 9/2004),…, (7/2004, 12/2004), (8/2004, 7/2004), (8/2004, 8/2004), (8/2004, 9/2004),…, (8/2004, 12/2004), … (12/2004, 7/2004), (12/2004, 8/2004), (12/2004, 9/2004),…, (12/2004, 12/2004)}
103	John	45K	{(12/2005, 1/2006), (12/2005, 2/2006), (12/2005, 2/2006),…, (12/2005, now), (1/2006, 1/2006), (1/2006, 2/2006), (1/2006, 2/2006),…, (1/2006, now), … (UC, 1/2006), (UC, 2/2006), (UC, 2/2006),…, (UC, now)}

3.1.2 Attribute Time-stamped Temporal Relational Data Models

3.1.2.1 Attribute Time-stamped Historical Relational Data Models

Clifford and Warren [CW83] and Clifford and Tansel [CT85] used time points as time-stamps in their earlier work. Clifford and Croker proposed the historical relational data model (HRDM), which was an extension of their previous work [CCT94]. Clifford and Croker suggested incorporating a time dimension at the attribute level. Lifespan was introduced as an attribute in [CT85] to capture the existence of database objects "birth," "death," and "reincarnation." The lifespan of an object denotes those periods of time during which the database models the properties of that object. Clifford and Croker discussed "what is an appropriate object with which to associate the lifespans". They stated that this could be done on three levels: on the database level, on the relation level, and on the tuple or the attribute level.

A database is a collection of relations that are homogeneous in the temporal dimension if a lifespan is associated at the database level. The database itself is time-stamped, and homogeneity is assumed throughout all the levels. All relations in the database have a valid time period that is a subset of the database validity time. If a lifespan is associated with each relation, all tuples in the relation will have a time period that is a subset of that relation's lifespan, but each relation might have different periods of time. The relation has a lifespan itself because it is also created at some point in time, and possibly dropped at some future time. The tuples are restricted to being contained in the relation, although they do not necessarily need to have the same lifespan. Users explicitly indicate the period of time over which the attribute is defined in that relation by assigning a lifespan to each attribute. Clifford and Tansel explained in detail that attribute time-stamping provides more user control over different temporal properties of the individual attributes [CT85]. The lifespan of an object being modeled is determined by the lifespan of the tuple and the lifespan of the attribute in the schema. Clifford and Croker argued that associating the lifespan on the tuple level provides for more flexibility [CC85].

Clifford and Croker proposed an algebra based on the lifespan and interpolation function for determining attribute values time points [CC85]. Two versions of the selection operation are defined: “select-if” selects on the values, and “select-when” selects on the temporal dimensions. The time-slice operator cuts a snapshot (slice) at the temporal dimension. The “when” operator extracts the time reference of a relation.

Gadia considered modeling with intervals to be inadequate, and in turn defined the temporal element as a finite union of disjoint time intervals that are closed under unions, intersections, and difference. He avoided horizontal and vertical anomalies [Ga88] by capturing the entire history of an object in a tuple.

Gadia’s model is a homogeneous relational model where validity periods of all attributes in a tuple have the same temporal domain [Ga88]. A relation is called homogeneous if all of its tuples are homogeneous. This requirement guarantees that a snapshot of a temporal relation will be a relation without null values. For each attribute, including the key attribute, every value is stored along with its union of time intervals – a temporal element. The key attribute of a relation must be time invariant. Therefore, the key attribute’s time-stamp represents the lifespan of the entity stored in the tuple. Gadia then suggested ways to relax homogeneity where [GY88] defines a heterogeneous relational model. Table 3.10 depicts Gadia’s homogeneous model where all time-stamps of attributes cover the same time, from 06/2003 to now.

Table 3.9: EMPLOEE_3.9 in Clifford and Croker's model.

EMP#	**ENAME**	**SALARY**	**DEPARTMENT**	**Lifespan**
101	[06/2003, now]→Tom	[01/2003, 01/2005)→25K [02/2005, now]→30K	[01/2003, 01/2006)→Sales [02/2006, now]→Marketing	{2003, 2004, 2005, 2006}
102	[06/2003, now]→Ann	[06/2001, now]→35K	[06/2001, now]→Sales	{2003,2004, 2005, 2006}
103	[06/2003, 12/2004)→John [01/2006, now]→John	[06/2003, 06/2004)→35K [07/2004, 12/2004)→42K [01/2006, now]→45K	[06/2003, 12/2004)→Toys [01/2006, 01/2006]→Marketing [06/2006, now)→Toys	{2003, 2004, 2006}

Table 3.10: EMPLOEE_3.10 in Gadia's homogenous model.

EMP#	ENAME	SALARY	DEPARTMENT
101	[06/2003, now] Tom	[01/2003, 01/2005)25K [02/2005, now]30K	[01/2003,01/2006) Sales [02/2006,now] Mark.
102	[06/2003, now] Ann	[06/2001, now]35K	[06/2001, now] Sales
103	[06/2003, 12/2004) John [01/2006, now] John	[06/2003, 06/2004)35K [07/2004, 12/2004)42K [01/2006, now] 45K	[06/2003, 12/2004) Toys [01/2006,05/2006)Mark. [06/2006, now] Toys

Gadia introduced a temporal version of relational algebra and relational calculus where snapshot semantics was used. Relational expressions and temporal expressions form the algebraic expressions where the first one evaluates to TRUE or FALSE, and the second one evaluates to a temporal element, respectively.

Gadia and Vaishnav developed Homogeneous Temporal Query Language, HTQuel, a query where the syntax is similar to QUEL, the query language of INGRES [SWKH76]. HTQuel extracts snapshots from the time cube and operates on them [GV85]. The time specification is implicit in HTQuel; that is, it is handled by set-theoretic operations in expressions manipulating time domains.

Gadia and Nair developed the query language TempSQL, a direct extension of SQL, and based on the N1NF attribute time-stamped relational data model on homogenous relations [GN93]. TempSQL is a tuple calculus language; therefore, optimization of corresponding algebraic expressions is possible. It consists of relational, temporal, and Boolean expressions. A relational expression's outcome is also a relation, which can be queried further. A "valid time selection" operation specifies a temporal expression – which returns a temporal element – and is specified in the WHILE clause. The SELECT – FROM – WHERE statement of SQL is extended with a WHILE clause. Time-stamp referencing is defined by a temporal expression formed with the set theoretic operations $\cup$, $\cap$, and –. Temporal expressions can be used in nested form. Boolean expressions

evaluate to TRUE or FALSE. TempSQL does not provide a special time-slice operator. “First instant” and “last instant” functions are provided to specify event extraction, but event comparison operators are not clearly defined. TempSQL also supports system users and classical users. While system users see the full temporal databases, classical users can only access the currently valid values in the database.

Tansel incorporates a temporal dimension to nested relations by extending the (N1NF) nested relations for modeling and manipulating temporal data [Ta97]. He attached temporal sets to the attributes in his historical relation, and allowed four types of attributes: atomic, set-valued, triplet-valued, and set-triplet valued [Ta86]. Atomic attributes contain atomic values such as integers, reals, and character strings. Set-valued attributes are sets of atomic values. Triplet-valued attributes consist of a valid-time interval $[t_l, t_u)$, together with an atomic value, $< [t_l, t_u)$, value >. If the upper bound of the valid-time interval is *now*, a closed interval $[t_l, now]$ is used. Set-valued attributes are sets of atomic values. An object being modeled may have different time periods. Moreover, time intervals of attribute values may overlap. Furthermore, the histories of attributes in a tuple may contain different time values since the model is heterogeneous. Key attributes can be atomic or triplet-valued, where values may not change over time. Tansel defined algebraic operations and calculus formulas at the attribute level by keeping the rest of the tuple untouched.

Tansel developed algebra and calculus languages for historical relational databases. The historical relational algebra includes pack, unpack, triplet formation, triplet decomposition, drop-time, and slice operations, in addition to conventional relational algebra operations. The pack operation converts atomic and triplet-valued attributes to set-valued and set-triplet-valued attributes, respectively. The unpack operation does the reverse and converts set-valued and set-triplet-valued attributes to atomic and triplet-valued attributes. The triplet decomposition operation decomposes time-interval bounds and data values of a triplet-valued attribute, and triplet-formation is the reverse of triplet decomposition. The drop-time operation discards the time components of a triplet-valued or set-triplet-valued attribute, and converts it into an atomic or a set-valued attribute,

respectively. The slice operation cuts the time of triplets in an attribute according to the time of another attribute. These new operations can transform a non-1NF relation to 1NF and vice-versa.

Table 3.11: EMPLOYEE_3.11 in Tansel's model.

EMP#	**ENAME**	**SALARY**	**DEPARTMENT**
101	Tom	{<[01/2003, 01/2005), 25K>, <[02/2005, now], 30K>}	{<[01/2003, 01/2006), Sales>, < [02/2006, now], Mark.>}
102	Ann	{<[06/2001, now], 35K>}	{<[06/2001, now], Sales>}
103	John	{<[06/2003, 06/2004), 35K>, <[07/2004, 12/2004], 42K>, <[01/2006, now], 45K >}	{<[06/2003, 12/2004], Toys>, <[01/2006, 05/2006], Mark.>, <[06/2006, now], Toys>}

Tansel defined a query language, HQuel for historical relational databases as an extension of Quel, the query language of INGRES [TA86]. New range declarations, set theoretic expressions, and conditions are included in HQuel, as well as aggregate functions. HQuel is based on an extended relational model employing attribute time-stamping and non-1NF relations. In addition to time, HQuel manipulates set-valued attributes. Set comparison, set membership, and temporal comparison are the new comparison operators. The arithmetic comparison operators are used to express an event-time predicate for event comparison. In HQuel, the valid-time projection is defined explicitly in the *retrieve* clause. Triplet-valued attributes can be formed in the *retrieve* clause for obtaining a temporal result. Tansel showed that HQuel is at least as powerful as the other extensions to Quel (see Table 3.14) and has the same expressive power as the relational algebra with structuring operations and a tuple calculus with set operators.

Tansel generalized his extension to nested relations by allowing arbitrary levels of nesting where attribute values can be relations. He represented the history of an attribute by a set of temporal atoms [Ta97]. His model supports both intervals and temporal sets as time-stamps. In his model, the schema of a historical relation can be viewed as a hierarchical tree, where each node is either a leaf or another subtree. Time information is attached only at the leaf level. The time reference of nonleaf nodes is defined to be the

union of the time stamps of its descendents until reaching the leaf nodes. Thus, nested historical relations allow modeling of the history of relationships as well. Tansel and Tin identified requirements crucial in representing temporal data, and provided a metric for comparing the expressive power of temporal query languages [TT97].

Tansel's main contribution is the coexistence of different attribute types in the same relation. Like other researchers, Tansel and Gadia visualize historical relations as three-dimensional structures or cubes – the third dimension being time; but the way they refer to historical data differs. Tansel first flattens the cube with algebraic operators, and then applies relational algebra operations, whereas Gadia extracts snapshots from the cube.

3.1.2.2 Attribute Time-stamped Rollback Relational Data Models

Bhargava and Gadia presented a model called Zero Information Loss in which no information is ever lost [BG93]. Transaction time temporal elements attached to the attributes, where transaction time is determined by the system clock. The Zero Information Loss model is a consistent extension of the snapshot database model, and there are different users: the system user and the classical user.

Bhargava and Gadia proposed a model consisting of three components [BG93]: a data history, an update environment, and a query store. The data store consists of data history relations where attribute values are time-stamped with transaction temporal elements, while the data store is accessible to the system user who can only see the snapshot database equivalence or *now* values. The update environment store monitors additional data surrounding the updates. For each data history relation, there is a shadow relation in the update store with the required attributes, i.e., transaction time (tt) – the primary key of store relation – user id, and an attribute for a possible explanation of why the update was performed. The query store, Q-rel, is a relation storing the queries and the user data indicating when and by whom the query was run. Q-rel is a log of all queries where the past query details are available and accessible by only the system user. Table 3.12-a

shows our running example in Bhargava and Gadia's Model and Table 3.12-b is the update store relation.

Bhargava and Gadia gave a relational algebra for their model, and defined new operators to capture and update the data and query store relations by considering their specific semantics [BG93]. Their model allows the environment of updates and queries to be restructured, and can be used as an auditing database system. However, it requires manual work to actually roll back the data store to a given transaction time.

Table 3.12-a: EMPLOYEE_3.12 in Bhargava and Gadia's model

EMP#	**ENAME**	**SALARY**	**DEPARTMENT**
101	Tom	{[12/2002, 01/2005], 25K>, < [02/2005, now], 30K>}	{<[12/2002, 01/2006), Sales>, <[02/2006, now], Marketing>}
102	Ann	{ [05/2001, now], 35K>}	{<[05/2001, now], Sales>}
103	John	{ [05/2003, 05/2004], 35K>, < [06/2004, 12/2004], 42K>, < [12/2005,now], 45K>}	{<[05/2003, 12/2004], Toys>, <[12/2005, 04/2006], Mark.>, <[05/2006, now], Toys>}

Table 3.12-b: Update store keeps all transaction details.

EMP#	**Transaction time**	**Authorizer**	**User**	**Reason**
101	12/2002	AUT_1	SU_1	New Employee
101	02/2005	AUT_1	SU_2	Raise
101	01/2006	AUT_1	SU_2	Transfer
102	05/2001	AUT_1	SU_1	New Employee
103	05/2003	AUT_1	SU_1	New Employee
103	07/2004	AUT_1	SU_2	Raise
103	12/2004	AUT_1	SU_3	Left
103	12/2005	AUT_1	SU_1	Hire an old employee
103	05/2006	AUT_1	SU_2	Transfer

3.1.2.3 Attribute Time-stamped Bitemporal Relational Data Models

Gadia and Bhargava attached two-dimensional temporal elements as time-stamps to attribute values [BG93]. Their model was designed as a comprehensive formalism for updates and errors. Time-stamps are in the form of [0, now] x [0, ∞] where the first part is transaction time, and the second part is real-world (valid) time.

Table 3.13: EMPLOYEE_3.13 in Gadia and Bhargava's Bitemporal model.

EMP#	ENAME	SALARY	DEPARTMENT
101	[12/2002, now]X[01/2003, θ] Tom	[12/2002, 01/2005]X[01/2003, θ] 25K [02/2005, now] X[01/2003, 01/2005] 25K [02/2005, θ] 30K	[12/2002, 01/2006]X[01/2003, θ] Sales [02/2006, now]X[01/2003, 01/2006] Sales [02/2006, θ] Marketing
102	[05/2001, now]X[06/2001, θ] Ann	[05/2001, now]X[06/2001, θ] 35K	[05/2001, now] X [06/2001, θ] Sales
103	[05/2003, 12/2004]X[06/2003, θ] John [12/2005, now]X[06/2003,12/2004] John [01/2006, θ] John	[05/2003, 06/2004]X [06/2003, 06/2004] 35K [07/2004, 12/2004]X [07/2004, 12/2004] 42K [12/2005,now] X [06/2003, 06/2004] 35K [07/2004, 12/2004] 42K [01/2006, now] 45K	[05/2003, 12/2004]X[06/2003, θ] Toys [05/2003, 12/2005]X[06/2003, 12/2004] Toys [12/2005,now] X[06/2003, θ] Toys [06/2003, 12/2004] Toys [01/2006, θ]Marketing

Gadia and Bhargava introduced three types of users: system, historical, and snapshot [GB89]. The whole database is accessible by the system user, where the span of time reference is [0, now] x [0, ∞]. The system user can exploit the full potential of the model and can query the history of updates and errors. A historical user sees the database as a historical database, whereas the snapshot user sees the database as a snapshot in time, without any time-stamps. A snapshot user querying the database is the same as the classical relational algebra. Gadia and Bhargava used the idea of an anchor to hold the correct knowledge of the history, and gave algebra and calculus languages for bitemporal relations. They also showed that this algebra is a consistent extension of the relational algebra.

Table 3.14 lists the temporal query languages we have reviewed in this section. We have also included the underlying query language that is extended for temporal support and whether the language has formal semantics. Our survey of the temporal data models and temporal query languages is not complete in the sense of reviewing all the proposals in the literature. We have tried to give a fair picture of the existing work by considering representative proposals for the main approaches in temporal databases. We have attempted to provide a base for the bitemporal data model that we will develop in the following chapters.

Table 3.14: Temporal relational query languages.

Data Model	Name	Based on	Formal Semantics
Gadia	HTQuel	Quel	Yes
Tansel	HQuel	Quel	Yes
Snodgrass	TQuel	Quel	Yes
Ben-Zvi	—	SQL	Yes
Navathe-Ahmed	TSQL	SQL	No
Ariav	TOSQL	SQL	No
Sarda	HSQL	SQL	No
Snodgrass	TSQL2	SQL	Yes

3.2 Temporal Entity Relationship Data Models

Elmasri and Wu proposed a temporal extension for an EER model, TEER, and extended the GORDAS Language [EWH85] to handle temporal queries and updates [EW90]. The TEER model includes temporal information on entities, relationships, superclasses/subclasses and attributes. While previous works on temporal extensions to the ER model have dealt mainly with the representation of temporal information, Elmasri and Wu expanded the previous work to formalize a temporal query language. The TEER models only valid time.

The concept of a defined lifespan for entities and relationships has been incorporated into the EER model. A lifespan of an entity ($T(e) \subseteq [O, now]$) could be a continuous time interval, or the union of a number of disjoint time intervals. A temporal value of each attribute of an entity is a partial function from temporal element T to the domain of the attribute. For every entity, system generates a unique id called a surrogate attribute whose

temporal element defines the entity's lifespan. Thus each real world object is member of an entity set and has a surrogate attribute specifying the object's lifespan.

Each relationship instance r is associated with a temporal element T(r), which gives the lifespan of the relationship instance. T(r) must be a subset of the intersection of the temporal elements of the entities e_1, e_2, …, e_n that participate in r, $T(r) \subseteq (T(e_1) \cap T(e_2) \cap \ldots \cap T(e_n)))$ [EW90].

While an entity type is a class, subclasses can be used for additional groupings of entities. There are two ways to specify a subclass; through a predicate or explicitly by the user. In the first case, only the predicate satisfying superclass of entities will be a member of the subclass, therefore all time intervals of superclass will belong to a predicate defined by subclass. In the second case, the user assigns an entity from the superclass to become a member of the subclass and specifies at specific time points whether the entity is to be made a member of the subclass or removed from the subclass.

Elmasri and Wu discussed the concept of temporal Boolean expression, temporal selection, and temporal projection similar to GORDAS query language in TEER model. A temporal Boolean expression is a conditional expression on the attributes and relationships of an entity. The condition c is TRUE if the temporal element is the time for e.

A temporal selection condition compares two temporal elements using the set comparison operators =, ≠, and ⊆, and evaluates to those entities that satisfy the condition. A temporal projection is applied to a temporal entity, and restricts all temporal assignments for that entity to a specific time period specified by a temporal element T. This is similar to the aforementioned "when" operator. Elmasri and Wuu also discussed temporal aggregate functions, temporal restrictions, and temporal update operations.

Example 3.1: Assume EMP# is a surrogate. Then, a relation TEER_EMPLOYEE, similar to the one given in Example 2.6, can be modeled in TEER in the following way.

Table 3.15: TEER_EMPLOYEE relation

SURROGATE	EMP#	ENAME	SALARY	DEPARTMENT
{[2003 – now] → ID1}	101	{[01/2003, now] → Tom}	{<[01/2003, 01/2005] → 25K>, <[02/2005, now] → 35K>}	{<[01/2003, 01/2006] → Sales>, <[02/2006, now] → Marketing>}
{[2001 – now] → ID2}	102	{[06/2001, now] → Ann}	{<[06/2001, now] → 35K>}	{<[06/2001, now] → Sales>}
{[2003 – 2004] → ID3, {[2006 – now] → ID3}	103	{[06/2003, 12/2004] → John, [01/2006, now] → John}	{<[06/2003, 06/2004] → 35K>, <[07/2004, 12/2004] → 42K>, <[01/2006, now] → 45K>}	{<[06/2003, 12/2004] → Toys>, <[01/2006, 05/2006] → Marketing>, <[06/2006, now] → Toys>}

3.3 Temporal Object-Oriented Data Models

The general limitation of the relational model in supporting complex applications has led to research into object data models. An object-oriented approach captures the complex semantics of time by representing it as a basic entity. In addition, the typing and inheritance mechanisms of object-oriented systems enable the various notions of time to be reflected in a single structure.

There have been many temporal object model proposals [SC91, KS92, WD93, GÖ93]. These models differ in the functionality that they offer, however as in relational systems, they assume a set of fixed notions of time.

In the following, we review Wuu and Dayal's extension to OOPAPLEX, as well as Goralwalla and Özsu's extension to TIGUKAT, to show specifically how time is incorporated into object-oriented (OO) databases.

3.3.1 OODAPLEX

Wuu and Dayal defined the object-oriented data model OODAPLEX [WD93], which is based on the functional data model DAPLEX [Sh81]. In their model, time is defined as an abstract data type. Time elements are reference points where objects evolve.

The abstract data type (ADT) {point} is a supertype of all time types. A set type is defined for each *point* time type using the parameterized type mechanism of OODAPLEX. This abstract generic time type carries general semantics of time. The different notions of time—total ordering or linear time versus partial ordering or branching time, discrete versus dense, time instants versus time intervals, and one-dimensional versus multi-dimensional—are modeled as subtypes of time type; implementers are responsible for an appropriate representation of application.

The type point defines (=, <) operators between elements. The subtype of a point may override and redefine these operators. The {point} also defines $\in$, =, $\cup$, $\cap$, - operators on sets, which are inherited. In addition, they support temporal predicates such as *overlap* and *during*. One of the defined subtypes of point is *region*, that in one-dimensional point space is called *interval*. An instance of region contains all points in $[l_b, u_b]$, where l_b is the lower bound and u_b is the upper bound.

The OODAPLEX model is based on functions, and all properties of objects, operations on objects, and relationships among objects are modeled by functions. Similar behaviors and properties of objects are grouped into types. Therefore, a set of functions specified by type is applied to instances of the type. Functions can be of a temporal sort or a property sort. The temporal behavior of an object's functions maps time elements to nontemporal snapshot values of the properties, and they have the time dimension as its domain. A property function takes an OID as its input and returns a temporal function that maps a time value into a snapshot value of that property.

Example 3.2: An example of an abstract object type and its temporal extension, as well as the properties of an employee object modeled as a function shown in Figure 3.1. Temporal employee objects having a name, a salary, and a department property are modeled in OODAPLEX. Attribute time-stamping using the time ADT can be achieved in the following way:

```
type person is object
    function emp# (P:person → S:string)
    function name (P:person → S:string)
type employee is person
    function name (e:employee → n:string)
    function salary (e: employee → f: (t:time → s: money))
    function dept (e: employee → f: (t:time → d: department))
```

Figure 3.1: A temporal abstract object type in OODAPLEX

The function **name** models an attribute that is constant over time. The functions **salary** and **dept** model temporal attribute values of the salary history of an employee and the history of memberships in different departments, respectively.

Functions can model one-dimensional or multi-dimensional time, e.g.,
function salary(e: employee → f:([valid_time: time, transaction_time: time] → s: money))

The function *salary*(e) (t1, t2) returns the employee e's salary at time t1, as recorded by the database at time t2. The OODAPLEX model supports time attributes and object time-stamping.

Object versioning keeps evolving versions for the same object. OODAPLEX has the flexibility to support both attribute versioning and object versioning. In this case, temporal employee objects are modeled with a state function that maps from time to snapshot states of employees. A snapshot state is modeled as a conventional, nontemporal employee type as depicted in Figure 3.2.

```
type employee is object
function state(e: Employee -> f: (t: time -> s : snapshot_employee))
        type snapshot_emp is object
function name (s: snapshot_emp → n:string)
function salary (s: snapshot_emp → m: money)
function dept (s: snapshot_emp → d: department)
```

Figure 3.2: A snapshot state of an object

If an object has a function to be created, destroyed, and reincarnated for a specific type, this function is called the *lifespan* of an object. OODAPLEX supports a **lifespan** function that accepts both a type time domain T and a database DB as input parameters. The union of lifespans or temporal elements of all objects of type T are returned by

function **lifespan** (T). The union of all lifespans of all types in DB is returned by function lifespan (DB).

The OODAPLEX model can enforce several constraints using objects' lifespans among objects, types, and a database, such as:

lifespan (o/T) $\subseteq$ **lifespan** (T) $\subseteq$ **lifespan** (DB)

where lifespan (o/T) denotes the lifespan of object o as an instance of type T.

OODAPLEX models the most general semantics of time which can then be subtyped to represent the various notions of time required by specific applications. However, this requires considerable support from the user.

3.3.2 TIGUKAT

TIGUKAT [GÖ93] is an object-based database management system with a clear distinction between the notions of class and type. Several classes may implement a type; that is, a type defines object characteristics as an interface, whereas a class is used for the implementation of a type, as well as management and maintenance of instances of that type.

The TIGUKAT object model is behavioral that all access and manipulation of objects is based on the application of behaviors to objects. It is uniform that every component of information is modeled as a first-class object with well-defined behavior. Other typical object modeling features supported by TIGUKAT include strong object identity, abstract types, strong typing, complex objects, full encapsulation, multiple inheritance, and parametric types [Go98].

The behavioral and uniform features of the TIGUKAT model are exploited in order to incorporate time; that is, the TIGUKAT temporal object model, consists of an extensible

set of primitive time types with a rich set of behaviors to accommodate different applications that require temporal support.

The TIGUKAT temporal object model provides a means to represent unanchored temporal data, procedures to convert the temporal data to a given granularity, canonical forms for the data, and operations between the data [Go98]. Granularities are accommodated within the context of calendars.

TIGUKAT has a user-defined time structure, and three different kinds of time stamps are supported: time instants, time intervals, and durations of time. The type system of ADT is defined to handle discrete, continuous, and dense time. Temporal behaviors of objects are specified through explicit behavioral types. These types then can be subtyped to model, for example, time-stamps with different granularities. Part of the type lattice for time in TIGUKAT is depicted at Figure 3.3 TIGUKAT supports both valid and transaction time, and these two times can be supported simultaneously for a bitemporal database. Since TIGUKAT model is behavioral, these dimensions of time are represented using separate behaviors.

Attribute and object versioning are handled in a uniform manner. Temporal attribute values are denoted by a set of pairs (t, o), where t is user-defined time, and o is the object instance at time t. Object histories are restricted to an object that is a member of a class.

TQL, TIGUKAT's query language, is an extension of the TIGUKAT object model. It is defined by type and behavioral extensions to the primitive model. The user language is similar to SQL3. It is based on an object calculus with an equivalent object algebra where queries are considered objects and therefore are handled by the system as objects. The underlying OO model and query language are essentially similar in OODAPLEX and TIGUKAT, and their extensions are independent of application requirements.

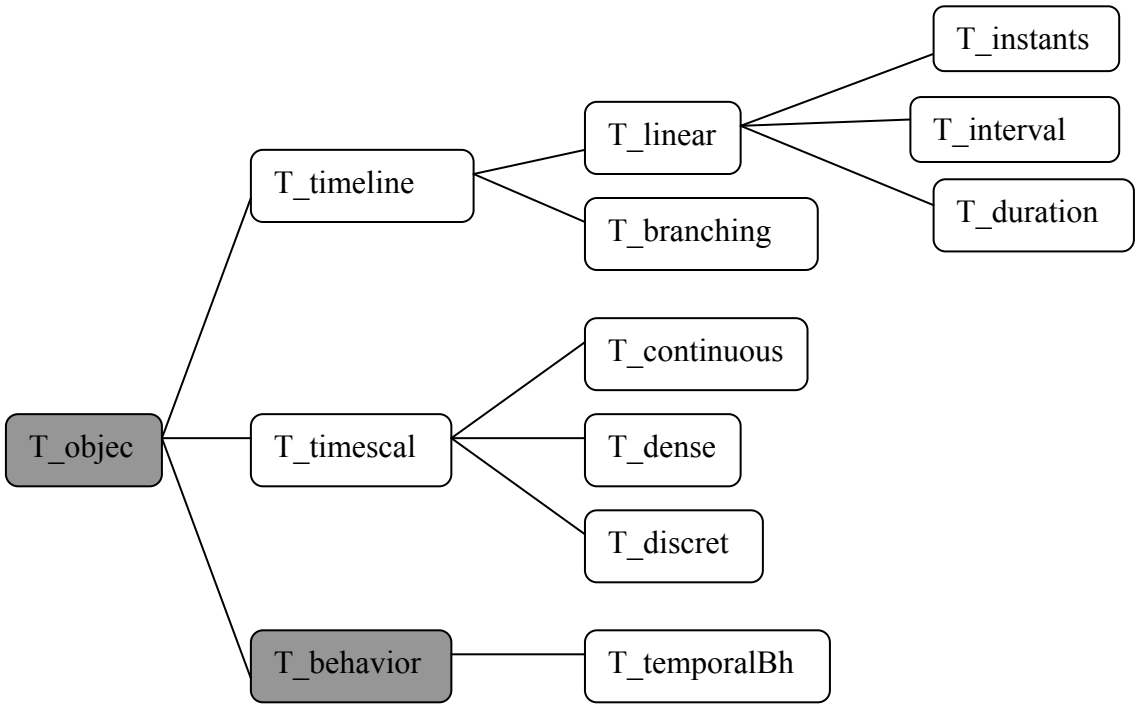


Figure 3.3: Part of the type lattice for time in TIGUKAT

3.4 XML Temporal Data Models

In addition to conventional approaches to implementing temporal databases, database researchers, vendors, and SQL standardization groups started work toward extensions of SQL with XML capabilities [XML], and to support languages such as XQuery [XMLQ] on XML data. The XML language has emerged as the standard for structuring and exchanging data over the Web. XML can be used to provide information about the structure and meaning of certain components of the data displayed on a Web page. The formatting for display aspects can be specified separately, for example through XSL (Extensible Stylesheet Language) [EN04].

The XML document is the basic object in XML. Elements and attributes are the two main structuring concepts that used to construct an XML document. Attributes describe elements in XML. Elements, which can be complex or simple, are identified in a document by their start tag and end tag. The tag names are enclosed between angled brackets <…>, and end tags are further identified b a backslash, </…>. Complex elements are constructed from other elements hierarchically, whereas simple elements contain data values. XML tag names are defined to describe the meaning of the data

elements in the document. This makes it possible to process the data elements in the XML document automatically by computer programs [EN04]. Since XML provides a flexible mechanism to represent complex data, it is adapted in many research areas, i.e., implementations of temporal databases.

Grandi and Mandreoli presented a new <valid> markup tag for XML/HTML documents to support valid time on the Web in [GM00]. In this approach, timestamps are explicitly encoded and temporal documents can then be selectively browsed in accordance with a user-supplied temporal period of interest. In [GS03], a dimension-based method was proposed to manage changes in XML documents; however, how to support queries was not discussed. Some extensions of the XML, such as τXQuery language, proposed in [GSn03] to extend XQuery [XMLQ] with new constructs for temporal support. Wang and Zaniolo proposed the use of XML in publishing and querying database history in [WZ03-a]. They expressed English queries into XQuery and provided performance evaluations between a native XML database and DB2. In addition, they showed that XQuery can be used in temporal databases and discussed the support for transaction time in [WZ03-b]. An archiving technique for scientific data using XML was presented in [BKTT04], but the issue of temporal queries was not discussed. [WZ04] showed that transaction-time, valid-time and bitemporal database histories can be represented in XML and queried using XQuery without requiring extensions of current standards. An XML-based bitemporal data model supports an attribute time stamping approach. [WZZ05] presented the ArchIS system that uses XML to support attribute time stamping approach, XQuery to express powerful temporal queries, temporal clustering, and indexing techniques for managing the actual historical data in a RDBMS, and SQL/XML for executing the queries on the XML views as equivalent queries on the relational DB. The schema proposed in [WZZ05] presents several similarities to that proposed in [BKTT04], but also provides full support for XML query languages. Noh and Gadia showed how XML could be used in temporal databases in [NG06]. By comparing a native XML database with XQuery engine, and XML storage with a temporal query language, they determined that the latter approach is more appropriate to utilizing XML in temporal databases. XQuery queries in [WZ03-a] are different from

[NG06] because data models are different; interval-based vs. temporal element-based, and [NG06] support XPath/XQuery without any extension to XML data models or query languages.

Example 3.3: The running example of EMPLOYEE table can be modeled with XML Representation of the Bitemporal History in [WZ04] as depicted in figure 3.4.

In Figure 3.4, each employee entity is represented as an employee element in the Bitemporal History BH-document, and table attributes are represented as employee element's child elements. Each element in the BH-document is assigned two pairs of attributes *tstart* and *tend* to represent the inclusive transaction time interval, and *vstart* and *vend* to represent the inclusive valid time interval. Elements corresponding to a table attribute value history are ordered by the starting transaction time tstart [WZ04]. The value of tend can be set to now and vend can be set to now.

There have been several proposals for XML query languages, but XPath and XQuery are the two emerged standards. XPath selects nodes from a tree-structured XML document. XQuery permits the specification of more general queries on one or more XML documents. The four main clauses of XQuery is known as a FLWR expression and has the following form:

FOR <variable bindings to individual nodes (elements)>
LET < variable bindings to collection of nodes (elements)>
WHERE <qualifier conditions>
RETURN <query result specification>

Figure 3.4: XML representation of the EMPLOYEE table

```
<EMPLOYEE vstart=”01/2003” vend=”now” tstart=”12/2002” tend=”now”>
<EMP# vstart=”01/2003” vend=”now” tstart=”12/2002” tend=” now”></101>
<ENAME vstart=”01/2003” vend=”now” tstart=”12/2002” tend=” now”>Tom</ENAME>
<SALARY vstart=”01/2003” vend=”01/2005” tstart=”12/2002” tend=”12/2004”></SALARY>
<SALARY vstart=”02/2005” vend=”now” tstart=”01/2005” tend=”now”></SALARY>
<DEPARTMENT vstart=”01/2003” vend=”01/2006” tstart=”12/2002” tend=”12/2005”></DEPARTMENT>
<DEPARTMENT vstart=”02/2006” vend=”now” tstart=”01/2006” tend=”now”></DEPARTMENT>
<SALARY vstart=”” vend=”now” tstart=”” tend=”now”></SALARY>
<DEPARTMENT vstart=”12/2002” vend=”now” tstart=”12/2002” tend=”now”></DEPARTMENT>
<EMP# vstart=”06/2001” vend=”now” tstart=”05/2001” tend=” now”></102>
<ENAME vstart=”06/2001” vend=”now” tstart=”05/2001” tend=” now”>Ann</ENAME>
<SALARY vstart=”06/2001” vend=”now” tstart=”05/2001” tend=”now”>35K</SALARY>
<DEPARTMENT vstart=”06/2001” vend=”now” tstart=”05/2001” tend=”now”>Sales</DEPARTMENT>
<EMP# vstart=”06/2003” vend=”now” tstart=”05/2003” tend=” now”></103>
<ENAME vstart=”01/2003” vend=”now” tstart=”12/2002” tend=” now”>John</ENAME>
<SALARY vstart=”06/2003” vend=”06/2004” tstart=”05/2003” tend=”06/2004”>35K</SALARY>
<SALARY vstart=”07/2004” vend=”12/2004” tstart=”07/2004” tend=”12/2004”>42K</SALARY>
<SALARY vstart=”01/2006” vend=”now” tstart=”12/2005” tend=”now”>45K</SALARY>
<DEPARTMENT vstart=”06/2003” vend=”12/2004” tstart=”05/2003” tend=”11/2004”>Toys</DEPARTMENT>
<DEPARTMENT vstart=”01/2005” vend=”01/2006” tstart=”12/2004” tend=”12/2005”>Marketing</DEPARTMENT>
<DEPARTMENT vstart=”05/2006” vend=”now” tstart=”04/2006” tend=”now”>Toys</DEPARTMENT>
</EMPLOYEE >
```

Chapter 4

4. NESTED BITEMPORAL RELATIONAL MODEL

The traditional relational model introduced by Codd [Co70] requires that relations be in first normal form (lNF), i.e., all values in a relation must be atomic or non-decomposable. The relational model is sufficient for representing objects that have simple domains, but the requirement of the first normal form makes it difficult to model complex objects. Database applications in the areas of Computer-Aided Design (CAD), office automation, text processing, and engineering design systems involve complex objects, and the relational model is insufficient for such applications. The traditional relational model distributes information about objects and their relationships over several different flat tables. This causes queries to be slow and complicated, since excessive joins have to be performed among the various relations in the database.

The nested relational model, N1NF (Non First Normal Form), is a generalization of the traditional relational model without the first normal form requirement. Nested relations overcome a number of limitations imposed by the 1NF restriction. The nested relational model organizes data hierarchically and allows relations to have attributes that have non-atomic values, i.e., values that are themselves relations – subrelations of the relation to which they belong. All data about an object being modeled can be stored within one relation, rather than being distributing over several relations, as in the 1NF model. With the nested relational model, users view the database in a way that is closer to their concept of the real world, because complex objects can be represented as a whole in a single nested relation, instead of being distributed over multiple flat relations.

Relaxing the First Normal Form restriction using Non-First Normal Form relations was first proposed by Makinouchi, [Ma77], followed by a number of other researchers. The nested models that have previously been proposed can be divided into two categories: non-recursive ([JS82], [FT83], [OOM87]) and recursive ([SS86], [RKS88]). Jaeschke and Schek proposed one-attribute nest and unnest operators [JS82]. Fisher and Thomas generalized these nest and unnest operators to multi-attribute operators [FT83]. Schek and Schooll formulated a recursive algebra for nested relations [SS86]. Ozsoyoglu and Ozsoyoglu defined pack (nest), unpack (unnest), and aggregation-by-template operators [OO83]. While recursive operators can be applied repeatedly to the subrelations of a relation, non-recursive operators cannot. Therefore, the main difference in the nature of the operators defined by various researchers is their recursive or non-recursive nature. Because recursive nested relations are beyond the scope of this thesis, only non-recursive models are discussed.

4.1 Basic Concepts and Terminology

In this section we define the basic concepts and terminology of nested relations to provide a basis for the bitemporal relational data model. These will be developed in the following sections. The definition of a nested relation schema is given inductively on the nesting depth of a schema. The basis of this inductive definition is the tuple schema, which consists of a sequence of attribute names that are atoms or nested relation schemas. A nested relation schema is defined on a tuple schema. Each schema has an order that is an ordinal value. An atom has order zero. A tuple schema's order is the same as the maximal order of its components. The EMPLOYEE table in Table 4.1 (below) is an example of a nested table where the DEPARTMENT attribute contains two attributes.

Example 4.1: Following is the definition of the nested relation schema EMPLOYEE depicted in table 4.1.

EMPLOYEE : = relation <e>
e := tuple : <EMP#, ENAME-H, ADDRESS-H, BIRTH-DATE, DEPARTMENT,

SALARY-H>

ENAME-H := relation : < NAME >

ADDRESS-H := relation : < ADDRESS >

DEPARTMENT := relation : < DNAME-H, MANAGER-H>

SALARY-H := relation : < SALARY >

DNAME-H := relation : < DNAME >

MANAGER-H := relation : < MANAGER >

EMP#, NAME, ADDRESS, BIRTH-DATE, DNAME, MANAGER, SALARY: =

tuple <atom>

Table 4.1: A nested relation, EMPLOYEE.

EMP#	**ENAME-H**	**ADDRESS-H**	**BIRTH DATE**	**DEPARTMENT**		**SALARY-H**
	NAME	ADDRESS		DNAME-H	MANAGER-H	SALARY
				DNAME	MANAGER	

Every nested relation R can be uniquely represented as an ordered tree with root R. The atomic attributes form the leaf nodes of the tree, and the nested attributes of the relation are the non-leaf nodes of the tree.

The tree structure offers a clear graphical representation of the nested structure since the schema of a nested relation can be very complex. The number of nesting levels of a relation is equal to the maximum number of nodes that must be traversed from the root to reach an atomic attribute in the tree representation. The root of the relation is by definition at nesting level zero. Two nested relations have the same schema if and only if they contain only common attributes.

The number of nesting levels of a relation is the number of nodes on the longest simple downward path from the root to any atomic attribute in the tree representation. The root's nesting level is zero by definition.

Example 4.2: Figure 4.1 is the schema tree for the relation EMPLOYEE depicted in Table 4.1.

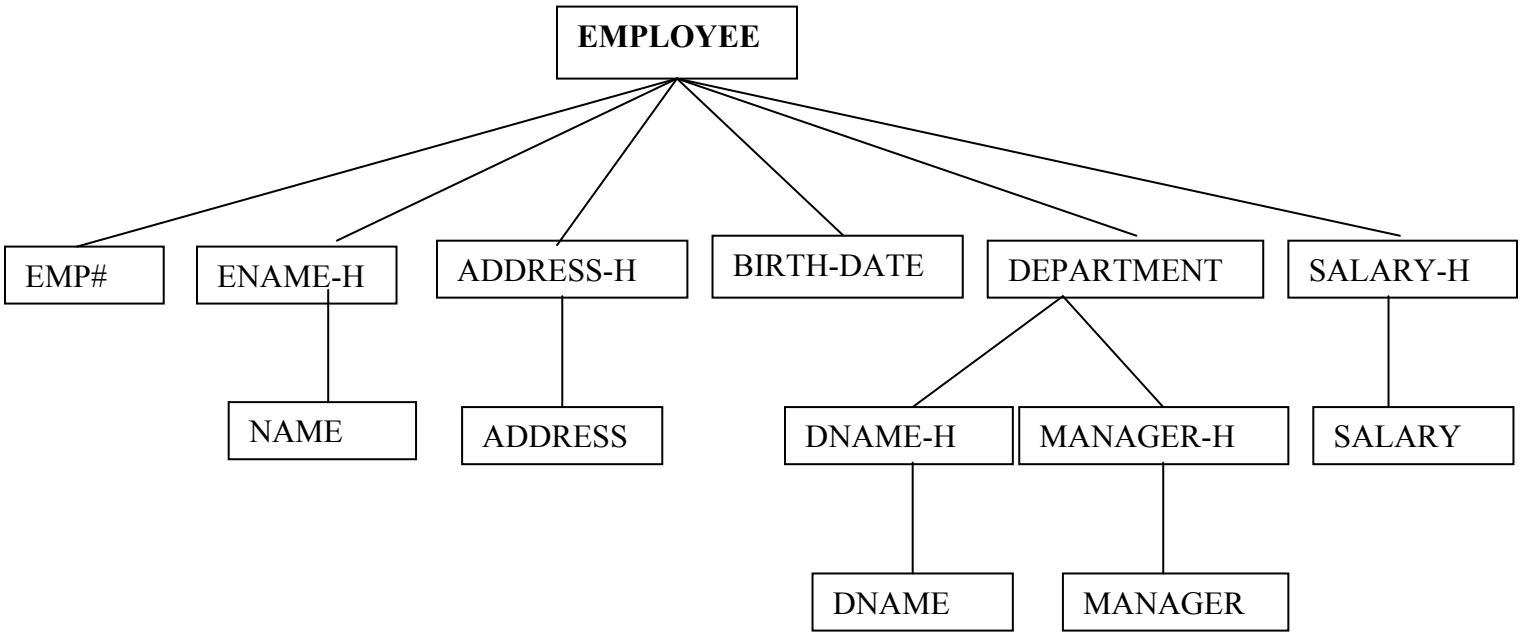


Figure 4.1: Schema tree for the nested relation EMPLOYEE

.

Example 4.3: The nesting level of relation EMPLOYEE is three. Counting the number of nodes from the root node to a specific attribute yields the nesting level of an attribute. For example, the atomic attribute ADDRESS of relation EMPLOYEE (Fig. 4.1) is at nesting level two.

4.2 Nested Bitemporal Relations

A *temporal atom* is a <*t, v*> pair where *t* is a timestamp and *v* is an atomic value (section 2.3.2). A *bitemporal atom* has two-time components and a value component. A bitemporal atom in the form of <TT, VT, V> means V is valid for the time period VT, and this fact is recorded in the database system for the time period TT. A valid time atom has the form <VT, V>, whereas a transaction time atom is in the form of <TT, V>. TT and VT may not be empty: (TT, VT $\subseteq T$) and (V $\in U$). We use the term temporal atom to mean any one of the three types of atom.

The focus of this thesis is on attribute time stamped nested bitemporal relations. Mostly, a few level of nesting would be sufficient to model temporal data. For the sake of generality, arbitrary levels of nesting are allowed. Each attribute's value is recorded as a *bitemporal atom*: <transaction time, valid time, value> triplet, where transaction time and valid time components can be represented as a time point, a time interval or a temporal element. In a bitemporal atom, either the transaction time or the valid time may be missing, but not both. In using time points, a bitemporal atom becomes <TT_i, VT_j, V> where TT_i, VT_j, and V represents transaction time at time point *i,* valid time at time point *j* and the data value, respectively. Using time points require interpolation between time points that complicates the model.

In case the time intervals are used, a bitemporal atom is < [TT_l ,TT_u), [VT_l, VT_u), V>, where TT_l is transaction time lower bound, TT_u is transaction time upper bound, VT_l is valid time lower bound, VT_u is valid time upper bound and *V* is data value. The interval that includes *now* as an upper bound such as [TT_l, *now*] or [VT_l, *now*] is closed and an expanding interval since the value of *now* changes as time progresses. The author of the

thesis opted to use intervals to represent time for the ease of representation. Temporal elements can also be equivalently used.

The definition of a nested bitemporal relation schema is given inductively on the nesting depth of a schema. The basis of this inductive definition is the bitemporal tuple schema, which consists of a sequence of attribute names that are atoms, temporal atoms, or nested relation schemas. A nested temporal relation schema is defined on a tuple schema. Each schema has an order which is an ordinal value. An atom has order zero. A tuple schema's order is the same as the maximal order of its components. The order of a bitemporal relation schema is one more than the order of the tuple schema over which it is defined. Following is the inductive definition of bitemporal tuple and bitemporal nested temporal relation schemas.

Order Zero Schema:

t:= tuple: <t(1), …, t(n)> for n>0. Each t(i) is an atom or a bitemporal atom.

Order k + 1 Schema: Either a bitemporal tuple schema or a bitemporal nested relation schema.

i. **t:= tuple:** <t(1), …, t(n)> for n>0. Each t(i) is an atom, a bitemporal atom, or bitemporal nested relation schema of order k + 1 or less. At least one component of t must be a relation schema of order k + 1.

ii. **r:= relation:** <t> where t is the bitemporal tuple schema <t(1), …, t(n)>. Each t(i) has a schema of order k or less and at least one t(i) has a schema of order k. For notational convenience, we will write **r:= relation:** <t(1), …, t(n)> [Ta97].

4.2.1 Instance of a Schema

$Dom_s(U)$ denotes the domain of interpretation for a schema s, where s is a tuple schema or a relation schema. U^{ba} denotes the domain of interpretation for bitemporal atoms and U^{ba} is $P(T) \times P(T) \times U$ or $P(T) \times U$ where $P(T)$ is the power set of T, and '×' is the Cartesian product operator. The inductive definition of the domain of interpretation for a schema follows [Ta97]:

The order zero schema:

s:= tuple: <s(1), …, s(n)> for n>0. Then, Dom_s (***U***) = $A_1 \times A_2 \times \ldots\ldots \times A_n$

where each A_i for $1 \leq i \leq n$:

$A_i = U$ if s(i) is an atom

$A_i = U^{ba}$ if s(i) is a bitemporal atom, $1 \leq i \leq n$

The order k + 1 schema:

s is a either a bitemporal tuple schema or a bitemporal relation schema.

s:= tuple: <s(1), …, s(n)> where each s(i), $1 \leq i \leq n$, an atom, a bitemporal atom or a relation of order k or less. Then,

Dom_s (U) = $A_1 \times A_2 \times \ldots\ldots \times A_n$

where each A_i for $1 \leq i \leq n$:

$A_i = U$ if s(i) is an atom

$A_i = U^{ba}$ if s(i) is a bitemporal atom, $1 \leq i \leq n$

$A_i = Dom_{s(i)}(U)$ if s(i) is a relation schema of order $k + 1$ or less

s:= relation: <t> where t is **tuple:** <t(1), …, t(n)>, n > 0, t has order k. Thus,

Dom_s (U) = $P(Dom_t(U))$

A nested bitemporal relation may have an attribute type of atomic or bitemporal. A bitemporal attribute may have a valid time temporal atom, a transaction time temporal atom, or a bitemporal atom. If A is a bitemporal attribute, then $A.VT$, $A.TT$, and A.V represent its valid time, transaction time, and value components, respectively.

4.3 Nested Bitemporal Relational Algebra (NBRA)

In querying a bitemporal database, we use a context. There are three commonly used contexts: bitemporal context, historical context, and current context. In bitemporal context, we refer to the entire bitemporal history [TA06]. This context is useful for auditing queries and for investigating the history of corrected errors. In current context,

we refer to only currently valid tuples of a bitemporal relation. In the historical context, we rollback (restrict) a bitemporal relation to its state at a given time (time point or time interval). The notation $R_{t,X}$ stands for the historical context. It refers to the state of the bitemporal relation R at time t. In other words, the attributes in X are rolled back to time t, whereas the rest of the relation R stays intact. Historical context is useful for specifying usual queries involving a state of a bitemporal database. We will first define the bitemporal relational algebra operations for the bitemporal context. Then we will proceed to the definitions of the operations for the historical context.

Let E be all bitemporal relational algebra expressions with the schema

Relation E: < e(1), …, e(n)>. EV(E) represents the evaluation of E.

4.3.1 Bitemporal Context

Set Operations ($\cup, \cap, -$)

In order to apply set operations to two bitemporal nested relations, R and S, they must have the same bitemporal relational schemas. The schema of the resultant structure is the same as the schemas R and S.

$EV(R \cup S) = \{t \mid t \in EV(R) \vee t \in EV(S)\}$

$EV(R \cap S) = \{t \mid t \in EV(R) \wedge t \in EV(S)\}$

$EV(R - S) = \{t \mid t \in EV(R) - t \in EV(S)\}$

Projection (π)

If $X \subseteq atr(E)$ then $EV(\pi_x(E)) = \{s[x] \mid s \in EV(E)\}$

This operation eliminates the identical (duplicate) tuples.

Selection (σ)

$EV(\sigma_F(E)) = \{s \mid s \in EV(E) \wedge F \text{ is true}\}$

The formula F is made up of the conditions in the form of *i* **op** *j*, where **op** is one of the {=, ≠, <, ≤, >, ≥} and *i* and *j* are attribute names or integers showing the position of attributes in E. In a condition, components of a bitemporal atom can be used. As already noted, A.TT, A.VT, and A.V refer to the transaction time, valid time, and value components of attribute A whose values are bitemporal atoms. Moreover, $A.TT_l$, $A.TT_u$, $A.VT_l$, and $A.VT_u$ refer to the lower and upper bounds of the intervals for the transaction and valid times, respectively. Equality (=) and non-equality (≠) tests can be applied to atoms, bitemporal atoms, or sets (relations). On the other hand, comparison operators {<, ≤, >, ≥} can only be applied to atoms or atomic components of bitemporal atoms. Connectives ∧, ∨, and ¬ are used to build complex formulas. For the convenience of the user, we also provide a set membership operator, ∈. However, it can be expressed by other operations of a bitemporal relational algebra [Ta97, GT92]. Using this basic set of formulas, a much larger class of selection formulas can be derived.

Cartesian product (×)

Let R have the schema < r (1), …, r (n)>, and S the schema <s (1), …, s (m)>.

E = R × S. Then, E = <e(1), …, e(n+m)>, where

E (i) = r (i) for 1≤ i ≤ n

e (n+j) = s (j) for 1≤j≤m

E(R × S) = EV(R) × EV(S)

Unnesting (μ) ([SS86], [JS82], [OO83])

The unnesting operation is defined to flatten a nested relation. Unnest is applied on the attribute *i* of a bitemporal relation R where it is a set of atoms or bitemporal atoms. A new tuple is created for each atom or bitemporal atom in the set, and the remaining attribute values are repeated. If unnest is applied on every bitemporal set-valued attribute in relation R, then the result becomes a flat (1NF) relation, made up of atoms or bitemporal atoms.

Let R be a bitemporal relation with the schema <r(1), …, r(n)> where <u(1), …, u(m)> is the schema of r(k), 1≤k≤n.

$E = \mu_k(R)$, where

$e(i) = r(i)$ for $1 \le i \le k-1$

$e(i) = r(i+1)$ for $k \le i \le n-1$

$e(i) = u(i-n+1)$ for $n \le i \le n+m-1$

$EV(E) = \{s \mid \exists r \exists y (r \in EV(R) \wedge y \in r[k] \wedge s[i] = r[i]$ for $1 \le i \le k-1$

$\wedge\ s[i] = r[i+1]$ for $k \le i \le n-1 \wedge s[i] = r[i-n+1]$ for $n \le i \le n+m-1)\}$

The columns created by unnesting are appended to the end of E's schema for notational convenience.

Nesting (v) ([SS86], [JS82], [OO83])

Let R be a bitemporal relation with the schema $<r(1), \ldots, r(n)>$ and $Y = \{i_1, i_2, \ldots, i_k\}$ is the set of k attributes in R, where $0 < k < deg(R)$ for some k, and X is the set of remaining attributes in R, i.e., $\{1, \ldots, n-Y\}$. Let the arity of X be m.

$E = \nu_{m+1 \leftarrow Y}(R)$ where

$e(j) = r(p)$ for $1 \le j \le n-k$, $p \in X$

$e(n-k+1) =$ relation : $< r(i_1), \ldots r(i_k)>$

The evaluation of E is

$EV(E) = \{ s \mid \exists r (r \in EV(R) \wedge s[j] = r[p]$ for $1 \le j \le n-k$, $p \in X$

$\wedge\ s[n-k+1] = \{ z \mid \exists u (u \in E(R) \wedge u[p] = r[p]$ for $p \in X$

$\wedge\ z[j] = u[i_j]$ for $1 \le j \le k)\})\}$

Relation R is partitioned where the attributes in X have the same values. For each partition, one new tuple is formed in EV(E). Newly generated tuple X attribute values are the same, and the values of the attributes in Y are formed into a set that is added as the last column of the result. This newly formed set may not be empty. Repeatedly **a**pplying nesting operations on a flattened version of a nested bitemporal relation reproduces the original structure, provided that certain conditions are met. Note that nest is not always the inverse operation for unnest [JS82].

Slice (§)

Let R be a bitemporal relation, k and p are two of its attributes whose values are bitemporal atoms, $1 \le k, p \le n$, $k \ne p$ and θ is one of $\{\cup, \cap, -\}$,

$E = \S_{\theta, k, p}(R)$

E has the same schema as R and E's evaluation is as follows.

$$EV(E) = \{ s \mid \exists r (r \in EV(R) \wedge s[i] = r[j] \text{ for } 1 \le i \le n, i \ne k, i \ne p$$
$$\wedge\ s[k].V = r[k].V \wedge s[k].TT = r[k].TT$$
$$\wedge\ s[k].VT = r[k].VT\ \theta\ r[p].VT$$
$$\wedge\ s[k].TT \ne \theta \wedge s[k].VT \ne \theta)\}$$

This operation changes the k^{th} attribute's bitemporal time component according to the time of the p^{th} attribute by the specified operator, i.e., union, intersection, or set difference. The result may not be empty, i.e., the resulting transaction time or the valid time may not be empty. The intersection operator implements the "when" predicate of natural languages.

Bitemporal atom decomposition (δ)

Let R have the bitemporal relation schema <r(1), …, r(n)>, k is a bitemporal attribute, $1 \le k \le n$ and $t \le now$.

$E = \delta_k(R)$

This operation splits the k^{th} attribute of R into its components: transaction time, valid time, and the value of the attribute. They are placed at the end of relation R as five new attributes. Note that the bounds of valid time interval and the transaction time interval are represented as four separate attributes.

E's schema is:

e(i) = r(i) for $1 \le i \le k-1$

e(i) = r(i+1) for $k \le i \le n-1$

e(n) := **tuple** <atom>

e (n+1) := **tuple** <atom>

e (n+2) := **tuple** <atom>

e (n+3) := **tuple** <atom>

e (n+4) := **tuple** <atom>

$EV(E) = \{ s \mid \exists r (r \in EV(R_X) \wedge s[i] = r[i] \text{ for } 1 \le i \le k-1$
$\wedge s[i] = r[i+1] \text{ for } k \le i \le n-1$
$\wedge s[n] = r[k].TT_l \wedge s[n+1] = r[k].TT_u$
$\wedge s[n+2] = r[k].VT_l \wedge s[n+3] = r[k].VT_u \wedge s[n+4] = r[k].V)\}$

Bitemporal atom formation (r)

Let R have the bitemporal relation schema <r(1), …, r(n)>, the k^{th}, l^{th}, m^{th}, q^{th} attributes be time attributes, and the p^{th} attribute be an atom.

$E = r_{k, l, m, q, p}(R)$

This operation combines the k^{th}, l^{th}, m^{th}, q^{th}, and p^{th} attributes of R into a new column in E. The number of attributes in relation R reduced by four.

r(k) := **tuple** <atom>

r(l) := **tuple** <atom>

r(m) := **tuple** <atom>

r(q) := **tuple** <atom>

r(p) := **tuple** <atom>

E's schema is:

e(i) = r(j) for $1 \le i \le n-2$, $1 \le j \le n$

e(n-1) = **tuple** := <bitemporal atom> $j \neq k$, $j \neq l$, $j \neq m$, $j \neq q$ and $j \neq p$

$EV(E) = \{ s \mid \exists r (r \in EV(R_X) \wedge s[i] = r[j] \text{ for } 1 \le i \le n-2, 1 \le j \le n,$
$j \neq k, j \neq l, j \neq m, j \neq q, j \neq p$
$\wedge s[n-1].TT_l = r[k] \wedge s[n-1].TT_u = r[l]$
$\wedge s[n-1].VT_l = r[m] \wedge s[n-1].VT_u = r[q] \wedge s[n-1].V = r[p]$
$\wedge s[n-1].TT_l \neq \theta \wedge s[n-1].TT_u \neq \theta$
$\wedge s[n-1].VT_l \neq \theta \; s[n-1].VT_u \neq \theta)\}$

4.3.2 Historical Context

Historical context is defined as a bitemporal relation rolled back to a given time. Bitemporal relational algebra operations work on the rolled back bitemporal relations.

Set Operations ($\cup, \cap, -$)

In order to apply set operations on two rolled back bitemporal relations, R and S, they must have the same relational schemas and refer to the same database state *t*, where $t <$ *now*. The schema of the resultant structure is the same as the schemas of R and S. The notation R_t indicates that the relation R is rolled back to time t. In other words, all the bitemporal attributes of R are rolled back to time t. We can also use a notation such as $R_{t,X}$, where X is a subset of R's bitemporal attributes and t is a time. In this case, rollback is applied only to the attributes in X, and not the entire set of attributes in R.

$EV(R_t \cup S_t) = EV(R_t) \cup EV(S_t)$

$EV(R_t \cap S_t) = EV(R_t) \cap EV(S_t)$

$EV(R_t - S_t) = EV(R_t) - EV(S_t)$

Projection (π)

If $X \subseteq atr(E)$ then $EV(\pi_x(E_t)) = \{s[x] \mid s \in EV((E_t)\}$

This operation eliminates the identical tuples in the result.

Selection (σ)

$EV(\sigma_F(E_t)) = \{s \mid s \in EV(E_t)) \wedge F \text{ is true}\}$

The selection operation applies to the state of the database at a designated time, *t*. That is, first the relation is rolled back to time *t*, and then the selection operation is applied. If time *t* is omitted, it defaults to *now*. The formula F is the same as in bitemporal context.

Cartesian Product ($\times$)

Let R_t have the schema <r(1), …, r(n)> and S_t have the schema <s(1), …, s(m)>.

$E_t = R_t \times S_t$. Then, E = <e(1), …, e(n+m)>, where

$e(i) = r(i)$ for $1 \leq i \leq n$

$e(n+j) = s(j)$ for $1 \leq j \leq m$

$E(R_t \times S_{t'}) = EV(R_t) \times EV(S_{t'})$

Rollback (ρ)

$E_t = \rho_{t, k}(R)$

Let R be a bitemporal relation schema, <r(1), …, r(n)>, where r(k) is a bitemporal attribute. Let t be a time point or an interval such that $t <$ *now*.

The rollback operator is defined as:

$$\rho_{t, k}(R) = \{s \mid (\exists u)\, R(u) \wedge (\, s[i] = u[i] \text{ for } i = 1,\ldots,n;\ i \neq k)$$
$$s[k] = \{\, z \mid (\exists x)(x \in u[k] \wedge$$
$$((\, x.TT_l \leq t < x.TT_u \wedge z = k)$$
$$\vee$$
$$(\, z = x \wedge x.TT_l < t \wedge$$
$$\neg((\exists y)\, y \in u[k] \wedge y.VT \cap x.VT \neq \theta \wedge$$
$$y.TT_l \leq t < x.TT_u\,))))\}\}$$

The result contains bitemporal atoms such that its TT includes the time t. If we want to cut the transaction time of these bitemporal atoms so that y only includes time t, then we can replace in the above formula z = x by the expression $z = <[t, t+1), x.TT, x.V>$

To rollback an entire bitemporal relation, we apply the rollback operation on each bitemporal attribute. Let R be a bitemporal relation with the schema <r(1), …, r(n)> and i, j, k, ... be its bitemporal attributes.

$$\rho(R) = \rho_i\, (\, \rho_j \ldots (\, \rho_k\, (R)))$$

The other operations – nest, unnest, bitemporal atom decomposition, and bitemporal formation – are similarly defined.

4.4 Nested Bitemporal Relational Tuple Calculus (NBRC)

This thesis presents a Nested Bitemporal Relational Calculus (NBRC) for the Nested Bitemporal Relational Model. The bitemporal context, followed by the historical context is defined for the bitemporal relational tuple calculus.

4.4.1 Symbols

. Predicate Names: The database schema has a finite number of relation schemas and one predicate name corresponding to each relation schema, namely, P,Q,R,S,…

. Variables: The finite number of tuple variables are denoted s, t, u, v,… If s is a variable, then s[i] is an indexed variable for i ($1 \leq i \leq deg(s)$), denoting its i^{th} attribute. Variables may be indexed only one level. A variable is associated with a relation schema; therefore it has the same schema and degree as the relation it is associated with. If s[i] is a bitemporal atom, then $s[i].TT_l$, $s[i].TT_u$, $s[i].VT_l$, $s[i].VT_u$, and s[i].V represent the transaction time lower bound, transaction time upper bound, valid time lower bound, valid time upper bound, and value parts of the bitemporal atom, respectively. A variable can only have a tuple schema, whereas an indexed variable can either have a relation schema or be an atom.

c. Constants: a, b, c,… represent the countable number of constant symbols. Each constant is either a tuple or relation.

4.4.2 Well-Formed Formulas for Bitemporal Context

Bitemporal context is useful for auditing queries and investigating the history of corrected errors.

a) Atomic Formulas:

1- P(s); P is a predicate name and *s* is a variable.

2- s[i] **op** r[j], s[i] **op** c, c op s[i] where **op** = {=, ≠, <, ≤, >, ≥} and s[i], r[j], c are atomic.

3- s[i].V **op** p[j].V, s[i].V **op** r[k] , s[i].V **op** c where **op** = {=, ≠, <, ≤, >, ≥} and

s[i], p[j] are bitemporal atoms, r[k] is an atom, c is a constant. In these terms, s[i].TT_l, s[i].TT_u, s[i].VT_l, s[i].VT_u are also allowed.

4- s[i] **op** r[j]; s[i] **op** c where **op** = {=, ≠} and s[i], r[j] and c have the same schema, s[i].VT op r[j].VT allowed if s[i] and r[j] are bitemporal atoms.

5- Formulas involving membership test:

i. s ∈ r[j] where s is a variable or a constant, and r[j] has the schema **relation:** <s>.

ii. s ∈ c where s is a variable or a constant, and c is a constant, and they have the same schema.

iii. s[i] ∈ r [j] where s is an indexed variable whose schema is an atom, and r[j] is an indexed variable with the schema **relation**: <u> and u is an atom. If s[i] is a bitemporal atom, then u, the tuple schema of indexed variable r[j], is also a relation of bitemporal atoms. In this formula, either of the indexed variables can be replaced by a constant of the same type.

iv. s[j] ∈ r[j].VT where s[i] is an indexed variable whose schema is an atom and r[j].VT is also an indexed variable representing a relation schema, which is a bitemporal atom. Either of the indexed variables may be replaced by a constant of the same type. Furthermore s[i].V can also be specified if s[i] is a bitemporal atom.

b) Formulas Containing Logical Operators:

6- If ψ and λ are formulas, then so are λ∧ψ, λ∨ψ, and ¬ψ.

7- If ψ is a formula with the free variable s, then ∃s ψ(s) and ∀s ψ(s) are formulas and s no longer occurs free in ψ.

8- r[i] = { s | ψ (s, u, v,…)} is a formula with free variables s, u, and v, and r does not occur freely in ψ. Indexed variable r[i]'s schema is a set of atoms or bitemporal atoms. Variables u, v, ... are free, and s is bound in the resulting formula, which is called a set constructor.

4.4.3 Well-Formed Formulas for Historical Context

A historical context is defined as a bitemporal relation rolled back to a given time t. Bitemporal relational calculus works on the rolled back bitemporal relations.

a) Atomic Formulas:

1- $P_{X,t}(s)$; P is a predicate name where X is among the attributes of R, and t is a time point, interval, or temporal element. s is a variable.

2- $s[i]_t$ **op** $r[j]_t$, $s[i]_t$ **op** c, c **op** $s[i]_t$ where **op** = {=, ≠, <, ≤, >, ≥} and $s[i]_t$, $r[j]_t$, c are atomic. In these terms, $s[i].TT_l$, $s[i].TT_u$, $s[i].VT_l$,and $s[i].VT_u$ are also allowed.

3- s[i].V **op** p[j].V, s[i].V **op** r[k], s[i].V **op** c where **op** = {=, ≠, <, ≤, >, ≥} and s[i], p[j] are bitemporal atoms with s[i].TT = t, p[j].TT=t and r[k] is an atom, c is a constant.

4- $s[i]_t$ **op** $r[j]_t$; $s[i]_t$ **op** c where **op** = {=, ≠} and $s[i]_t$, $r[j]_t$ and c have the same schema, s[i].VT **op** r[j].VT and s[i].TT **op** r[j].TT allowed if $s[i]_t$ and $r[j]_t$ are bitemporal atoms.

5- Formulas involving membership test:

i. $s \in r[j]_t$ where s is a variable or a constant, and $r[j]_t$ has the schema **relation:** <s>.

ii. $s \in c$ where s is a variable or a constant, and c is a constant, and they have the same schema.

iii. $s[i] \in r[j]$ where s is an indexed variable whose schema is atom, and r[j] is an indexed variable with the schema **relation**: <u>, and u is an atom. If s[i] is a bitemporal atom, then u, the tuple schema of indexed variable $r[j]_t$, is also a relation of bitemporal atoms. In this formula, either of the indexed variables can be replaced by a constant of the same type.

iv. $s[j]_t \in r[j].VT$ where $s[i]_t$ is an indexed variable whose schema is an atom and r[j].VT is also an indexed variable representing a relation schema, which is a bitemporal atom. Either of the indexed variables may be replaced by a constant of the same type. Furthermore s[i].V can also be specified if $s[i]_t$ is a bitemporal atom.

b) Formulas Containing Logical Operators:

6- If $\psi_{t,X}$ and $\lambda_{t,X}$ are formulas, then so are $\lambda_{t,X} \wedge \psi_{t,X}$, $\lambda_{t,X} \vee \psi_{t,X}$, and $\neg\psi_{t,X}$.

7- If $\psi_{t,X}$ is a formula with the free variable s, then $\exists s\ \psi_{t,X}(s)$ and $\forall s\ \psi_{t,X}(s)$ are formulas, and s no longer occurs free in $\psi_{t,X}$.

8- $r[i] = \{ s \mid \psi_{t,X}(s, u, v,\ldots)\}$ is a formula with free variables s, u, and v, and r does not occur freely in $\psi_{t,X}$. Indexed variable r[i]'s schema is a set of atoms or bitemporal atoms. Variables u, v, ... are free, and s is bound in the resulting formula, which is called set constructor.

9- $r[i] = \{s \mid \psi_{t,X}(s, u, v,...) \wedge t \in T\}$ is a formula with free variable s in $\psi_{t,X}$. Indexed variable r[i]'s schema is a set of atoms or bitemporal atoms where its transaction time t' is the element of T.

10- $r[i] = \{s \mid \psi_{t,X}(s, u, v,...) \wedge (\exists u)\ P(u) \wedge (\ s[i] = u[i]$ for $i = 1,\ldots,n;\ i \neq k)$

$s[k] = \{ z \mid (\exists x)(x \in u[k] \wedge ((\ x.TT_l \leq t < x.TT_u \wedge z = k) \vee$

$(\ z = x \wedge x.TT_l < t \wedge \neg((\exists y)\ y \in u[k] \wedge y.VT \cap x.VT \neq \theta \wedge$

$y.TT_l \leq t < x.TT_u\ )))\}\}$

4.4.4 Interpretation of Calculus Objects

U is the universe of atoms. The interpretation of a nested bitemporal relational calculus object is relative to ***U***. The domain of interpretation for a bitemporal atom is $\boldsymbol{U^{ba}}$, the derived set of ***U***. An interpretation of nested bitemporal relational calculus formula is an interpretation of each of its constants (I_c), predicate symbols (I_p), and assignment to each of its free variables (I_s). If constant c is an atom, then $Dom_c(\boldsymbol{U}) = \boldsymbol{U}$ or, if constant c is a bitemporal atom, then $Dom_c(\boldsymbol{U}) = \boldsymbol{U^{ba}}$. P is a predicate name, I_p is a relation instance, and $I_p \in Dom_p(\boldsymbol{U})$. s is a free variable, I_s is a tuple instance and $I_s \in Dom_s(\boldsymbol{U})$. $I_s(i)$ denotes the i^{th} component of I_s.

Formulas are interpreted as *true* or *false.* Rules for the interpretation of formulas are as follows.

a) Formulas with no Logical Operators:

1- P(s) is true if $I_s \in I_p$

2- s[i] op r [j], …, is true if $I_s(i)$ op $I_r(j)$.

3- s[i].V op r[j].V, …, is true if $I_s(i)$.V op $I_r(j)$.V.

4- s[i] = r[j], …, is true if $I_s(i) = I_p(j)$.

5- $s \in r[j]$ is true if $I_s \in I_r(j)$

$s \in r[j]$ is true if $I_s \in I_c$.

$s[i] \in r[j]$ is true if $I_s(I) \in I_r(j)$.

$s[i] \in r[j]$.VT is true if $I_s(i) \in I_r(j)$.VT.

b) Formulas with Logical Operators:

6- $\lambda \vee \psi$ is true if either ψ is true or λ is true.

$\lambda \wedge \psi$ is true if both ψ and λ is true.

$\neg\psi$ is true if ψ is false.

7- $\exists s\ \psi(s)$ is true if there is at least one assignment to s which makes $\psi(s)$ true, i.e., $\psi(s)$ is true for at least one Is. $\forall\ s\ \psi(s)$ is true if $\psi(s)$ is true for any assignment to s.

8- r[i] = $\{s \mid \psi\ (s, u, v,\ldots\}$ is satisfied (true) by the interpretations I_r , I_s , I_u , I_v of the variables r, s, u, v, ..., if the following condition is met. If $I_r(i)$ equals the set of assignments I_s satisfying $\psi(s, u, v,\ldots)$ for the interpretations I_u, I_v,… If there are no such tuples and I_s is empty, then this formula evaluates to false. In other words, the set constructor formula can not create an empty set.

9- r[i] = $\{ s \mid \exists\ t'_1 \leq t' \leq t \wedge \psi(s) \wedge t \in T\}$ is true by the interpretations I_r , I_s of the variables r and s, if $I_r(i)$ equals the set of assignments I_s satisfying ψ (s) where for at least one $t \in T$. If there are no such tuples and I_s is empty, then this formula evaluates to false.

A nested bitemporal relational calculus expression is $\{s^{(k)} \mid \psi(s)\}$, where s is a free variable with arity k, and $\psi(s)$ is a well-formed formula with one free variable, s. An interpretation of this expression is the set of instances of s satisfying the formula $\psi(s)$, i.e., an element of $Dom_s(\boldsymbol{U})$.

4.4.5 Safety of the Nested Bitemporal Relational Calculus

To avoid the creation of infinite relations, we define a safe subset of a nested bitemporal relational calculus as given in [Ul95]. Specifically, we add safety rules for the cases of bitemporal atoms, set membership formulas, and set constructor formulas. Let ψ be a formula with one free variable, s. The formula ψ is safe if it satisfies the following conditions:

a. The '$\forall$' quantifier is not allowed. This is not a restriction since it can always be replaced by the '$\exists$' quantifier.

b. Whenever a '$\vee$' operator is used, the two subformulas connected, say $\psi_1 \vee \psi_2$, should have the same free tuple variables.

c. Consider any maximal conjunct $\psi_1 \wedge \psi_2 \ldots, \wedge \psi_n$ of ψ. Then, all components of tuple variables appearing free in any ψ_i, must be limited in the following sense:

1- If ψ_i is a non-negated atomic formula in the form of R(u), then all components of tuple variable u are limited.

2- If ψ_i is u[i] = c or c = u[i], where c is a constant, then u[i] is limited.

3- If ψ_i is u[i] = v[j] or v[j] = u[i], u[i].TT_l, u[i].TT_u, u[i].VT_l, u[i].VT_u and v[j] is limited then u[i] is limited. Note that rules 2 and 3 apply on the parts of the temporal atoms as well. A temporal atom is limited if both of its temporal set and value components are limited.

4- If ψ_i is u[i]$\in$v[j] and v[j] is limited, then all components of u[i] are limited.

5- If ψ_i is u[i] = {s | ψ '(s, u, v, ...)} and ψ'(s, u, v, ...) is safe, then u[i] is limited.

d. A'$\neg$' may only apply to a formula discussed in five cases of the rule (c). Furthermore, A negated subformula, $\neg\lambda$, can only be part of a larger subformula, $\psi_1 \wedge \psi_2 \ldots, \wedge \psi_n \wedge \neg \lambda \wedge \psi_{n+1} \wedge \psi_{n+2} \ldots, \wedge \psi_{n+m}$ where at least one subformula, ψ_i is not negated.

NBRA and NBRC have the same expressive power. The proof is direct extension of the proof provided in [TT97].

4.5 Nested Bitemporal Relational Queries

We now give examples of the above concepts, using the nested bitemporal relation EMPLOYEE given in Table 4.2 to illustrate the use of nested bitemporal relational algebra operations (Note that MANAGER-H and SALARY-H are column of EMPLOYEE and displaced on the next line.). We use position indexes instead of attribute names to avoid the common attributes introduced by the join operation. Note that notation 3.V, 3.TT, and 3.VT represent the value, transaction time, and valid time bitemporal set components of the attribute number 3, respectively. Consider that *now* is 60 at the time of the queries. Answers to these queries are provided in Figure 4.2.

Query 4.1: What salary values are stored in the database between the times 15 and 30?

$\Pi_{1,7.V}(\sigma_{(7.TT^{-} \geq 15 \wedge 7.TT^{+} \leq 30)}(\mu_6(\text{EMPLOYEE})))$

This is a bitemporal context query, and uses transaction time interval. Applying the unnest operation on the SALARY-H attribute of the EMPLOYEE table creates a new attribute that is the flattened version of the SALARY attribute. It is appended to the EMPLOYEE table, and we refer it as the seventh attribute. The selection operation picks tuples where the transaction time components are between 15 and 30. The projection operation retains EMP# as the first attribute, as well as the value components from the flattened SALARY attribute. The answer to this query is provided in Figure 4.2-a.

Query 4.2: What are the employee numbers of the employees who shared the same addresses at the same time? When was it?

$\Pi_{1,3,8,7}(\S \cap_{,3.VT,7.VT}(\sigma_{3.V=7.V}(\mu_3(\text{EMPLOYEE}) \times (\mu_3(\text{EMPLOYEE})))$

This is a bitemporal context query. Join and time slice operations are used. Applying the unnest operation on the ADDRESS-H attribute of the EMPLOYEE table creates a new attribute, which is appended to the end of the EMPLOYEE table. We refer to it as the seventh attribute, which is the flattened version of the ADDRESS attribute.

Table 4.2: A nested bitemporal relation, EMPLOYEE.

EMP#	**ENAME**	**ADDRESS-H**	**BIRTH DATE**	**DEPARTMENT**
	NAME	ADDRESS		DNAME-H
				DNAME
E_1	James	<[1, *now*], [1, *now*], a_1>	1980	{<[1, 8), [1, 10), Marketing>, <[9, *now*], [11, *now*], Planning>}
E_2	Bob	<[1, *now*], [1, *now*], a_2>	1975	{<[1, 8), [1, 10), Sales>, <[9, *now*], [11, *now*], Planning>}
E_3	Carol	{<[12, 27), [15, 27), a_3>, <[28, 40), [28, 40), a_1>, <[41, 45), [41, 45), a_3>, <[53,*now*], [55, *now*], a_3>}	1990	{<[14, 45), [15, 45), TechSup>, <[53, *now*], [55, *now*], TechSup >}
E_4	Liz	{<[15,56), [18,56), a_4>, <[57, *now*], [57, *now*], a_6>}	1982	<[15, *now*], [18, *now*], Sales>
E_5	Amy	<[10, *now*], [10, *now*], a_5>	1985	<[10, *now*], [10, *now*], Sales>

DEPARTMENT		**SALARY-H**
DNAME-H	MANAGER-H	SALARY
DNAME	MANAGER	
{<[1, 8), [1, 10), Marketing>, <[9, *now*], [11, *now*], Planning>}	<[1, *now*], [1, *now*], Bob>	{<[1, 8), [1, 10), 25K>, <[9, 20), [11, 22), 30K>, <[21,32), [23, 34), 32K>, <[33, *now*], [35, *now*], 40K>}
{<[1, 8), [1, 10), Sales>, <[9, *now*], [11, *now*], Planning>}	<[1, *now*], [1, *now*], James>	{<[1, 13), [1, 15), 25K>, <[14, 32), [16, 34), 32K> <[33, *now*], [35, *now*), 40K>}
{<[14, 45), [15, 45), TechSup>, <[53, *now*], [55, *now*], TechSup >}	{<[14, 45), [15, 45), Amy>, <[53, *now*], [55, *now*], Bob>}	{<[14, 21), [15, 25), 20K>, <[22, 45), [26, 45), 22K>, <[53, *now*], [55, *now*], 25K>}
<[15, *now*], [18, *now*], Sales>	{<[15, 25), [15, *now*], Bob>, <[26, *now*], [15, *now*], Amy>}	{<[15, 27), [18, 30), 22K>, <[28, 39), [31, 42), 24K>, <[40, *now*], [43, *now*], 26K>}
<[10, *now*], [10, *now*], Sales>	<[10, *now*], [10, *now*], James>	{<[10, 30), [10, 34), 25K>, <[31, 44), [35, 48), 28K>, <[45, *now*], [49, *now*], 30K>}

The Cartesian product of this table by itself results in 12 attributes. The selection operation picks tuples where the ADDRESS attributes' value components are equal. The slice operation synchronizes the valid time component of the ADDRESS with respect to other ADDRESS valid time components and hence implements ‘when’. Finally, the projection operation retains EMP#’s and ADDRESS attributes. Figure 4.2-b gives the result.

Query 4.3: Get all records of the departments that ‘Bob’ has worked in that were stored in the database during the time range [5, 25).

$$\Pi_{9.V}(\sigma_{2=\text{‘Bob’}}(\rho_{[5,\,25),\,9}(\mu_7(\mu_5(\text{EMPLOYEE})))))$$

This is a time interval historical context query. Applying the unnest operation on the DEPARTMENT attribute of the EMPLOYEE table creates two new attributes, namely DNAME-H and MANAGER-H. They are appended to the end of the EMPLOYEE table, and we refer them as the seventh and eighth attributes. Applying the unnest operation on the DNAME-H, or the seventh attribute, creates the ninth attribute, which is a flattened version of the DNAME attribute. The rollback operation rolls back the ninth attribute to time value between 5 and 25. The selection operation picks tuples from the second attribute where the value is ‘Bob’, and then the projection operation displays the ninth attribute value components. The result is shown in Figure 4.2-c.

Query 4.4: As of time 50, who was working in the Technical Support Department?

$$\Pi_1(\sigma_{9.V=\text{‘Technical Support’}}(\rho_{[50],\,9}(\mu_8(\mu_5(\text{EMPLOYEE}))))))$$

This is a time point historical context query. Applying the unnest operation on the DEPARTMENT attribute of the EMPLOYEE table creates two new attributes, namely DNAME-H and MANAGER-H. They are appended to the EMPLOYEE table, and we refer to them as the seventh and eighth attributes. Applying the unnest operation on the DNAME-H, or the seventh attribute, creates the ninth attribute, which is a flattened

version of the DNAME attribute. The rollback operation rolls back the ninth attribute to time value 50. The selection operation picks tuples where the value component is equal to ‘Technical Support’, and then the projection operation retains the EMP# as the first attribute. The result is empty.

Query 4.5: Who was Liz’s manager between time 15 and 30 as of [15, 20]?

$$\Pi_9(\sigma_{10=\text{`Liz'} \wedge (9.VTl \geq 15 \wedge 9.VTu \leq 30)}(\mu_2(\rho_{[15,20],9}(\mu_8(\mu_5(\text{EMPLOYEE}))))))$$

This is a time interval historical context query and uses a valid time interval. It is used for error correction. Applying the unnest operation on the DEPARTMENT attribute of EMPLOYEE table creates two new attributes, namely DNAME-H and MANAGER-H They are appended to the end of EMPLOYEE table, and we refer to them as the seventh and eight attributes. Applying the unnest operation on the MANAGER-H, or the eight attribute, creates the ninth attribute, which is a flattened version of the MANAGER attribute. Applying the unnest operation on the NAME-H, or the second attribute, creates the tenth attribute, which is a flattened version of the NAME attribute. The rollback operation rolls back the ninth attribute to the time value between 15 and 20. The selection operation picks tuples from the tenth attribute, where the value is ‘Liz’ and the valid time component is between 15 and 30; then the projection operation displays the ninth attribute. The resultant relation is in Figure 4.2-d gives the result.

Here, Bob was assigned as Liz’s manager at transaction time 15. But at time 17, it was realized that actually Amy was supposed to be her manager. Bitemporal databases are capable of storing such corrections.

Query 4.6: What are the EMP# and managers of employees who were employed by the company between 46 and 52 as known by the database on 58?

$$\Pi 1_{1,9}(\sigma_{(9.VTl \geq 46 \wedge 9.VTu \leq 52)}(\rho_{58,9}(\mu_8(\mu_5(\text{EMPLOYEE})))))$$

This is a time point historical context query and uses a valid time interval. It is used for error correction. Applying the unnest operation on the DEPARTMENT attribute of EMPLOYEE table creates two new attributes, namely DNAME-H and MANAGER-H. They are appended to the end of EMPLOYEE table, and we refer to them as the seventh and eight attributes. Applying the unnest operation on the MANAGER-H, or the eight attribute, creates the ninth attribute, which is a flattened version of the MANAGER attribute. The rollback operation rolls back the ninth attribute to the time point to A. The selection operation picks tuples that the valid time component is between B and C; then the projection operation displays the first and the ninth attribute. The resultant relation is in Figure 4.2-e gives the result.

EMP#	SALARY
E_1	30K
E_1	32K
E_2	32K
E_3	20K
E_3	22K
E_4	22K
E_4	24K
E_5	25K

Figure 4.2-a

TITLE
Sales
Planning

Figure 4.2-c

EMP #	ADDRESS	EMP #	ADDRESS
E_1	<[1, *now*], [1, *now*], a_1>	E_3	<[28, 40), [28, 40), a_1>

Figure 4.2-b

MANAGER
Bob, [15,17), [18, 30)
Amy, [18, 30), [18, 30)

Figure 4.2-d

EMP#	MANAGER
E_1	Bob
E_2	James
E_4	Amy
E_5	James

Figure 4.2-e

Figure 4.2: Query results from Q4.1 to Q4.6

Chapter 5

5. IMPLEMENTATION OF NESTED BITEMPORAL RELATIONAL MODEL

This chapter outlines how the Nested Bitemporal Relational Model presented in the previous chapter can be implemented. Two possibilities are presented to represent bitemporal atom types. The first approach implements it as one string, the second defines it as an abstract data type. For both data types, a bitemporal atom is defined with SQL and the Java language. Figure 5.1 illustrates the implementation plan for bitemporal atom type that results in four types.

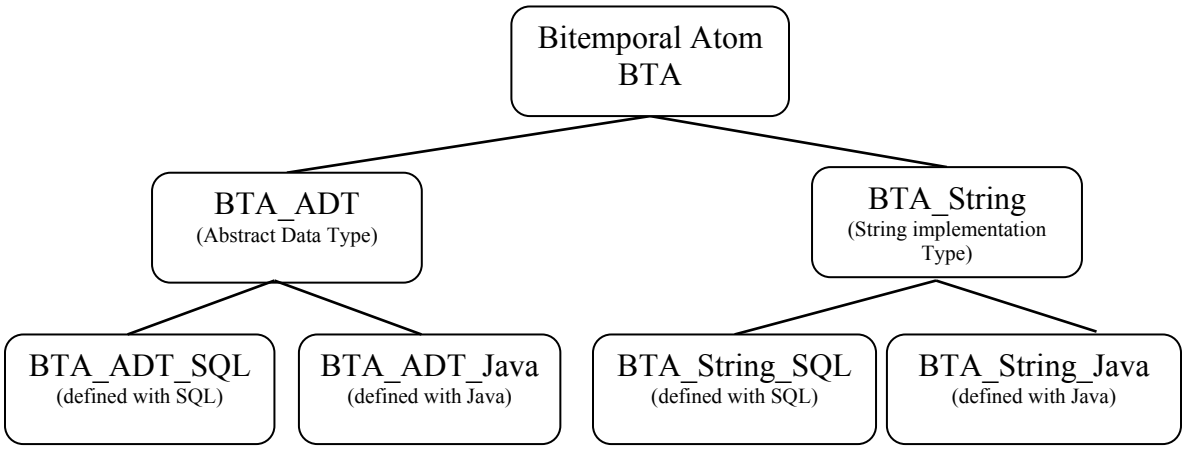


Figure 5.1: Possible implementations of the bitemporal atom type BTA

Each approach has its own merits and limitations. By representing a bitemporal atom as one string, a logically coherent unit is not decomposed over several attributes. Abstract data types hide the complexity of the abstract structures from end users. SQL is tightly integrated with the DBMS, and is therefore easier to use for database applications. The characteristics of Java (i.e., object-oriented, interpreted, architecture neutral, portable,

high-performance, and dynamic) confer it the potential for implementing an object relational database model. In addition, Java is portable, so that the system will run on most platforms without the need to recompile the code. This provides a greater scope of deployment to the of the NBRM model.

A set of bitemporal atoms can be stored in a one column nested or an array table collection type, as depicted in Figure 5.2. While a nested table can have any number of bitemporal atoms, an array table contains a specified maximum number of bitemporal atoms.

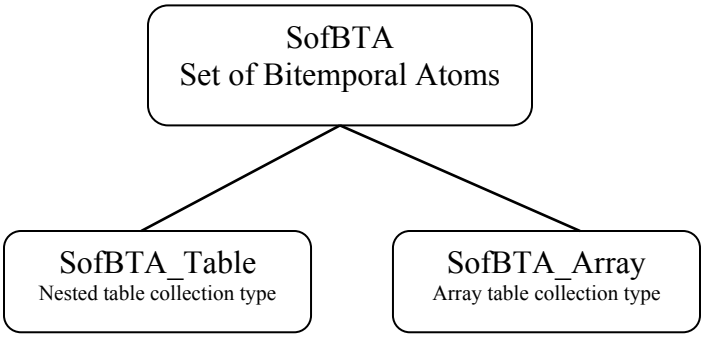


Figure 5.2: Possible implementations of NBR

To test a Nested Bitemporal Relational Model, we conducted experiments to measure the performance of the proposed different implementations against each other. The first implementation illustrates where relationship and entity history is in the same table; that is - an entity is embedded into another relation. The second implementation demonstrates how a typical database where entities and relationships have their relations can be modeled to support bitemporal data. For the experiments, we used a hypothetical company database with over 10 years of past and possible future data. The first implementation database is made up of the employee relation whose schema is given in Table 5.1.

Table 5.1: A nested bitemporal relation for EMPLOYEE.

EMP #	NAME-H	ADDRESS-H	BIRTH - DATE	DEPARTMENT		SALARY-H
	NAME	ADDRESS		DNAME-H	MANAGER-H	SALARY
				DNAME	MANAGER	

We have designed a graphical user interface bitemporal preprocessor, called BtSQL, which is easy to use for application programmers and end-users. BtSQL converts bitemporal queries received from a user into statements in SQL query language and passes them to the DBMS. In order to study the performance of this prototype system, updates and two set of queries were run for eight different databases. While the first group of queries used a bitemporal context, the second group used a historical context to test the overhead of the 'AS-OF' operator. The results of the updates and the queries were analyzed, and major factors that had a measurable impact on the performance of the system were also identified.

The implementation issues and data types are briefly discussed in Section 5.1. The fully supported time slice and 'AS-OF' clauses are explained in Section 5.2. Then, Section 5.3 sketches how these parts were implemented in the prototype object-relational DBMS. How the BtSQL preprocessor works, as well as selected update operations and SQL queries, are fully elucidated in Section 5.4. Section 5.5 implements when entities and relationships have their relations to support bitemporal data. Lastly, Section 5.5 draws lessons from the implementation. The complete BTAs and SofBTA definitions are given in Appendix A.

5.1 Implementation Methodology

5.1.1 Bitemporal Atom Type, BTA

A bitemporal atom is defined in Chapter 4 in the form of <[TT_l, TT_u), [VT_l, VT_u), V>, where TT_l is the transaction time lower bound, TT_u is the transaction time upper bound, VT_l is the valid time lower bound, VT_u is the valid time upper bound, and *V* is the data value. The bitemporal atom uses built-in data type DATE for the lower and upper bounds of transaction and valid times. Time intervals might be in any granularity, i.e., DATE and TIMESTAMP, depending on the application. The value part may be CHARACTER, CHARACTER VARYING, CHARACTER LARGE OBJECT, NUMERIC, DECIMAL, INTEGER, SMALLINT, BIGINT, or BOOLEAN. Figure 5.3 shows the five components stored as one string, BTA_String. Figure 5.4 depicts the bitemporal atom as an abstract data type, BTA_ADT.

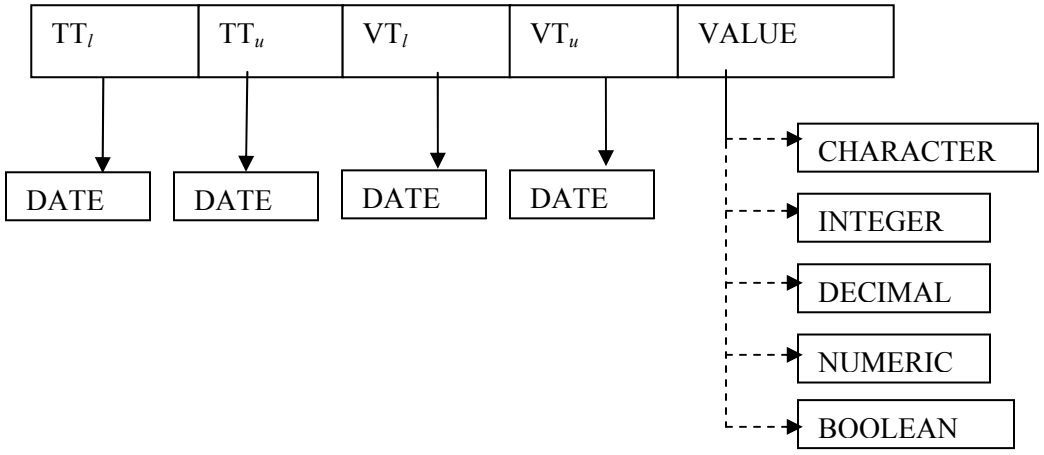


Figure 5.3: Representation of a BTA as string implementation, BTA_String.

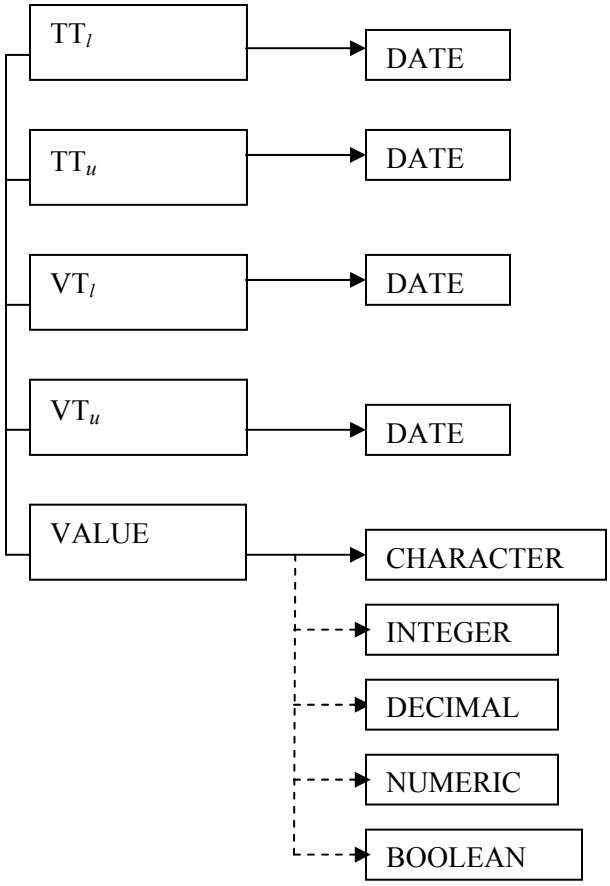


Figure 5.4: Representation of a BTA as an abstract data type, BTA_ADT.

The type system facilities of object-relational databases allow us to define a BTA as one string (BTA_String_SQL) and as a structured abstract data type (BTA_ADT_SQL), as seen in Figures 5.5 and 5.6, respectively. Removing or retrieving a component such as the transaction time lower and/or upper bound as a substring is allowed and used in the query expressions. Once the BTA is defined, it can be used in SQL statements where other built-in types are used.

```
CREATE TYPE BTA_String_SQL AS (
                BTA_String_SQL varchar2(200));
```

Figure 5.5: Definition of BTA as string implementation with SQL, BTA_String_SQL.

```
CREATE TYPE BTA_ADT_SQL AS (
        TRAN_TIME_LOWER_BOUND DATE,
        TRAN_TIME_UPPER_BOUND DATE,
        VALID_TIME_LOWER_BOUND DATE,
        VALID_TIME_UPPER_BOUND DATE,
        VALUE CHARACTER VARYING(50)
)
```

Figure 5.6: Definition of BTA as an ADT, BTA_ADT_SQL.

5.1.2 Custom Bitemporal Atom Java Classes

A SQL user-defined structured data type that encapsulates a set of attributes with a set of methods, can be represented as a Java class within the database. A Java class whose instances are accessible by the database must be created first. Figure 5.7 depicts Java class BTA_String_Java as string implementation. The Java class BTA_ADT_Java in Figure 5.8 contains transaction and valid time bounds with the type of DATE, along with a string type VALUE for the bitemporal atom's value part.

```
public class BTA_String_Java { public String BTA_String_Java;}
```

Figure 5.7: The Java class for a BTA as string implementation, BTA_String_Java.

```
public class BTA_ADT_Java {
        public Date TRAN_TIME_LOWER_BOUND;
        public Date TRAN_TIME_UPPER_BOUND;
        public Date VALID_TIME_LOWER_BOUND;
        public Date VALID_TIME_UPPER_BOUND;
        public String VALUE;
}
```

Figure 5.8: The Java class for a BTA as an ADT, BTA_ADT_Java.

The BTA Java class is compiled with a conventional Java compiler javac. After compilation, the *.class* byte code is loaded into the database. Then, to create the SQLJ object type, the extended SQL CREATE TYPE command must be used by specifying the corresponding Java class in the EXTERNAL NAME clause. The extended SQL CREATE TYPE command populates the database catalog with the external names for the Java class.

Once a BTA_String_Java or BTA_ADT_Java has been created, it can be used as the attribute type of a database table. The database SQL engine can then access and use each field in SQL statements, i.e., the transaction time lower and upper bounds or value part of the BTA.

5.1.3 Nested Bitemporal Relations

Abstract data types can be declared to be the "data type" of an entire table such that the table's attributes are defined by the abstract data type. By in-lining the repeated fields in the table, the reliance on creating another table with its own structure and indexes is removed in collection type tables. Data manipulation operations such as select, insert, and delete can be applied similar to ordinary tables. In the following discussion, for storing sets of bitemporal atoms two kinds of nested bitemporal tables are considered: nested tables and array tables.

A nested table is an unordered collection of elements of the same data type. It can have any number of elements; no maximum number is specified in the definition of the table. Elements of a nested table are actually stored in a separate storage table that contains an attribute that identifies the parent table tuple.

An array is an ordered collection of elements of the same data type. The position of each element is indicated by an index number, and this number is used to access a specific element. When an array is defined, the maximum number of elements it can contain is specified, although this can be changed later. The size of a stored array depends only on the current number of elements in the array and not on the maximum number of elements that it can hold.

A tuple in a nested bitemporal relation is an instance of the structured type on which the table is defined. It gives the instance a unique identity. Having a set of identical abstract data types in a single tuple actually simulates the attribute time-stamping approach with a single-attribute table for each object's time related attributes [TG89]. These are temporally grouped relations [CCT94].

The SQL statement in Figure 5.9 shows that ADDRESS is a nested bitemporal relation whose tuples are of type BTA. The nested bitemporal table ADDRESS is a single-attribute table with a type of BTA. The table can have as many tuples as needed to represent bitemporal atoms.

```
CREATE TYPE ADDRESS AS SofBTA_Table OF BTA;
```

Figure 5.9: The tuples of ADDRESS are as nested table, SofBTA_Table.

The SQL statement in Figure 5.10 specifies that an array type of ADDRESS has no more than ten elements, and each of its data types is a BTA. The ADDRESS table is a single-attribute type table with a type of BTA and with maximum of ten bitemporal atoms (tuples).

```
CREATE TYPE ADDRESS AS SofBTA_Array (10) OF BTA;
```

Figure 5.10: The tuples of ADDRESS are as array table, SofBTA_Array.

Appendix A has the definition of BTA_ADT_SQL, BTA_ADT_Java, BTA_-String_SQL, and BTA_String_Java bitemporal data types. The Nested Bitemporal Relational Database, introduced in Chapter 4, definition is also given in Appendix A with SofBTA_Table and SofBTA_Array collection types.

5.2 Implementing the Nested Bitemporal Relational Algebra

Select, project, cartesian product, and set theoretic operations work as usual in the NBR Algebra. Nest, unnest, bitemporal_atom_decomposition and bitemporal_ atom_formation operations are managed through object-relational structures. Slice and rollback operations are implemented separately.

5.2.1 Slice Operation

The Slice function is given the two bitemporal attributes, and it returns the first and second attributes' value parts along with the common time intervals that they have. It first finds if the given two intervals intersect or not, by comparing lower bounds and upper bounds. If that is the case, then it finds the starting and ending point of this new interval. Lastly, it returns the corresponding value parts along with the common new intervals. Pseudo-code for SLICE operation is presented in Figure 5.11.

```
SLICE(k,p)
For each SofBTA_Table(BTA(i)) in k, i>=1
For each SofBTA_Table(BTA(j)) in p, j>=1

if (p(BTA.TT(j)) != 0 and p(BTA.VT(j))!= 0 and
      k(BTA.VTl(i)) <= p(BTA.VTl(j)) and
      p(BTA.VTu(j)) <= k(BTA.VTu(j)) )
then
      if (k(BTA.TTl(i)) > p(BTA.TTl(j))) then
            TT_LOWER_BOUND= k(BTA.TTl(i))
      else
            TT_LOWER_BOUND = p(BTA.TTl(j))

      if (k(BTA.TTu(i)) < p(BTA.TTu(j))) then
            TT_UPPER_BOUND = k(BTA.TTu(i))
      else
            TT_UPPER_BOUND = p(BTA.TTu(j))

      if (k(BTA.VTl(i)) > p(BTA.VTl(j))) then
            VT_LOWER_BOUND = k(BTA.VTl(i))
      else
            VT_LOWER_BOUND = p(BTA.VTl(j))

      if (k(BTA.VTu(i)) < p(BTA.VTu(j))) then
            VT_UPPER_BOUND = k(BTA.VTu(i))
      else
            VT_UPPER_BOUND = p(BTA.VTu(j))

      Return k(BTA(i).value),
            [TT_LOWER_BOUND, TT_UPPER_BOUND],
            [VT_LOWER_BOUND, VT_UPPER_BOUND],
            p(BTA(j).value);
      else
      Return;
end loop
end loop
```

Figure 5.11: The pseudo-code for SLICE operation

5.2.2 Rollback Operation

NBRM answers queries about past states by rolling the database back to a state some time in the past, through the AS_OF function. To facilitate this process, the SQL syntax is extended by adding the keyword clause AS OF, which requires information about ‘as

of' some earlier time. The AS_OF function receives the attribute name along with transaction time (point) interval. It first finds, for every set of tables in attribute k, if the given interval is in transaction time lower and upper bounds of every BTA by comparing them. If that is the case, then it returns the i^{th} BTA to be included in the query. Pseudo-code for AS_OF operation is presented in Figure 5.12.

```
AS_OF(k, LOWER_BOUND, UPPER_BOUND)
For each SofBTA_Table(BTA(i)) in k, i>=1

if (LOWER_BOUND <= k(BTA.TTl(i)) AND
  k(BTA.TTl(i)) <= UPPER_BOUND)  OR
      (k(BTA.TTl(i)) <= LOWER_BOUND <= k(BTA.TTu(i))) OR
      (k(BTA.TTl(i)) <= UPPER_BOUND <= k(BTA.VTu(i)))
    return k(BTA(i))
  else
    return;
end loop
```

Figure 5.12: The pseudo-code for AS_OF operation

5.3 Database Modifications

We developed the graphical user interface BtSQL, a bitemporal SQL preprocessor, for translating bitemporal SQL statements into standard SQL statements. BtSQL implemented in the Java language and uses JDBC to communicate with the underlying DBMS. BtSQL preserves the validity of end-user programs in the face of differences among vendors and evolving standards. It also shields the end-user from the complexity of the additional operations required by bitemporal modifications and querying, as well as the propagation of updates to enforce the covering constraints. Insert, update, and delete commands can be used to modify the database in BtSQL. We discuss each of these, in turn, followed by how queries are specified and processed in BtSQL.

5.3.1 Insert in BtSQL

When a new entity is inserted into the database, the new tuple is appended to the table. While a valid time must be provided for all bitemporal attributes, the nested table names and system-provided transaction times are available to the INSERT function. For

instance, Figure 5.13-a illustrates the process of inserting ‘MIKE BROWN’ with EMP# 20001, birth date ’10.03.1980’, address ‘West 34th Street NY NY 10292’, into the DEP_ID23 department. ‘TOM WHITE’ is assigned as his manager, and his salary is 25000, starting on January 1, 2007. Figures 5.13-b and 5.13-c display how SQL inserts the same tuple for BTA_String_SQL and BTA_ADT_SQL bitemporal atom type, respectively.

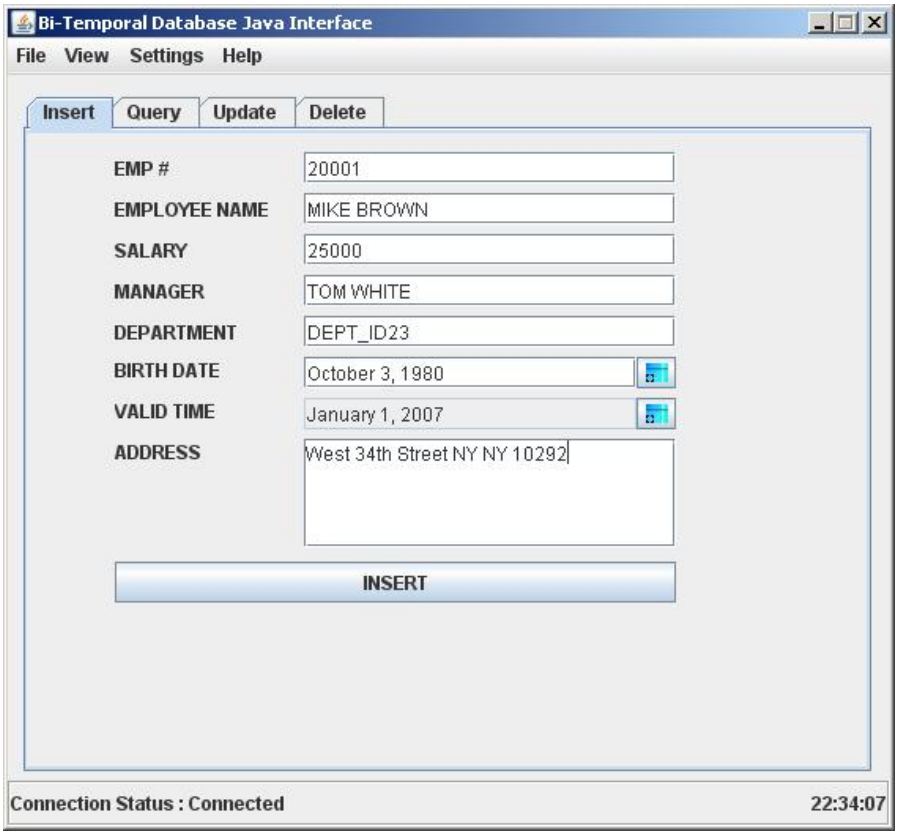


Figure 5.13-a: Inserting a tuple with BtSQL.

```
INSERT INTO EMPLOYEE VALUES
(20001,
NAME(BTA_TYPE(sysdate,'09.09.9999','01.01.2007','09.09.9999',
'MIKE BROWN')),
ADDRESS(BTA_TYPE(sysdate, '09.09.9999','01.01.2007','09.09.9999',
'West 34th Street NY NY 10292')), '10.03.1980',
DEPT_MNG(TYPE_DEPT_MNG(
DEPARTMENT(BTA_TYPE(sysdate, '09.09.9999','01.01.2007','09.09.9999',
'DEPID_23')),
MANAGER(BTA_TYPE(sysdate, '09.09.9999','01.01.2007','09.09.9999',
'TOM WHITE'))),
SALARY(BTA_TYPE(sysdate, '09.09.9999', '01.01.2007', '09.09.9999',
250000)));
```

Figure 5.13-b: Insert a tuple with BTA_String_SQL type.

```
INSERT INTO EMPLOYEE VALUES
(20001,
NAME(BTA_TYPE(to_char(sysdate,'mm.dd.yyyy')||' '||'09.09.9999'||'
'||'01.01.2007' ||' '||'09.09.9999' ||' '||'MIKE BROWN')),
ADDRESS(BTA_TYPE(to_char(sysdate,'mm.dd.yyyy')||' '||'09.09.9999'||'
'||'01.01.2007' ||' '||'09.09.9999' ||' '||'West 34th Street NY NY
10292')),TO_DATE('10.03.1980','mm.dd.yyyy'),
DEPT_MNG(TYPE_DEPT_MNG(
DEPARTMENT(BTA_TYPE(to_char(sysdate,'mm.dd.yyyy')||' '||'09.09.9999'
||' '||'01.01.2007' ||' '||'09.09.9999' ||' '|| 'DEPID_23')),
MANAGER(BTA_TYPE(to_char(sysdate,'mm.dd.yyyy')||' '||'09.09.9999'||'
'||'01.01.2007' ||' '||'09.09.9999' ||' '|| 'TOM WHITE'))),
SALARY(BTA_TYPE(to_char(sysdate,'mm.dd.yyyy')||' '||'09.09.9999'||'
'||'01.01.2007' ||' '||'09.09.9999' ||' '|| to_char(25000))));
```

Figure 5.13-c: Insert a tuple with BTA_ADT_SQL type.

5.3.2 Update in BtSQL

An update operation ‘inserts’ a new bitemporal atom while preserving the old version, which can be on a single tuple, a group of tuples, or all tuples’ bitemporal attributes' value part on the database. The UPDATE page has these choices displayed in different pages.

If an update is on a single tuple, BtSQL displays the allowed bitemporal attribute names, shown in Figure 5.14. After choosing the bitemporal attribute that the update is to be performed, BtSQL asks for the key attribute value and the new value. The system then finds the last bitemporal variable where the valid time upper bound is ‘now’ and replaces the valid time upper bound with the effective date. It next inserts the new bitemporal atom type into the database. Its valid time lower bound gets the “given date” when the change was/is/will be effective. Its transaction time lower bound gets the system-

provided date (sysdate), and both intervals' upper bounds get the value of "now". The bitemporal atom's value part is replaced with the provided new value.

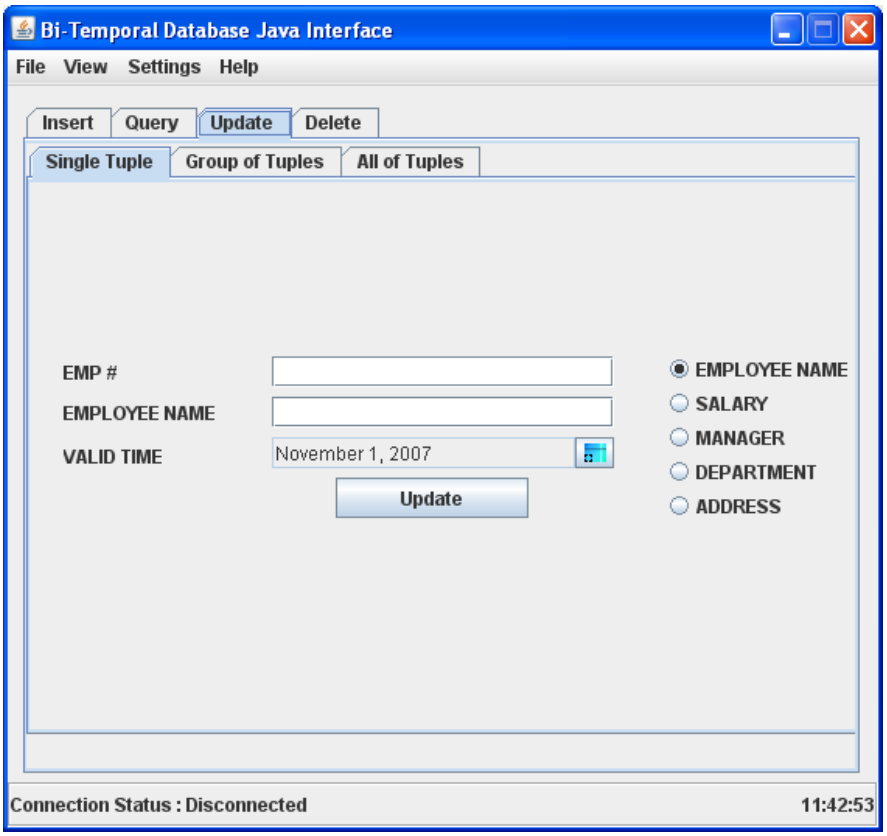


Figure 5.14: Update a tuple with BtSQL.

For example, in the page shown in Figure 5.15-a, the system updates the employee's salary data with EMP# = 12345 to 50,000 effective February 1, 2007. Figures 5.15-b and 5.15-c show the actual SQL code needed to make this update possible, for BTA_String_SQL and BTA_ADT_SQL type, respectively. If an error is discovered, an update operation has to be performed to correct the error. The erroneous data are kept; the correct value part and valid time are updated by using the correct date.

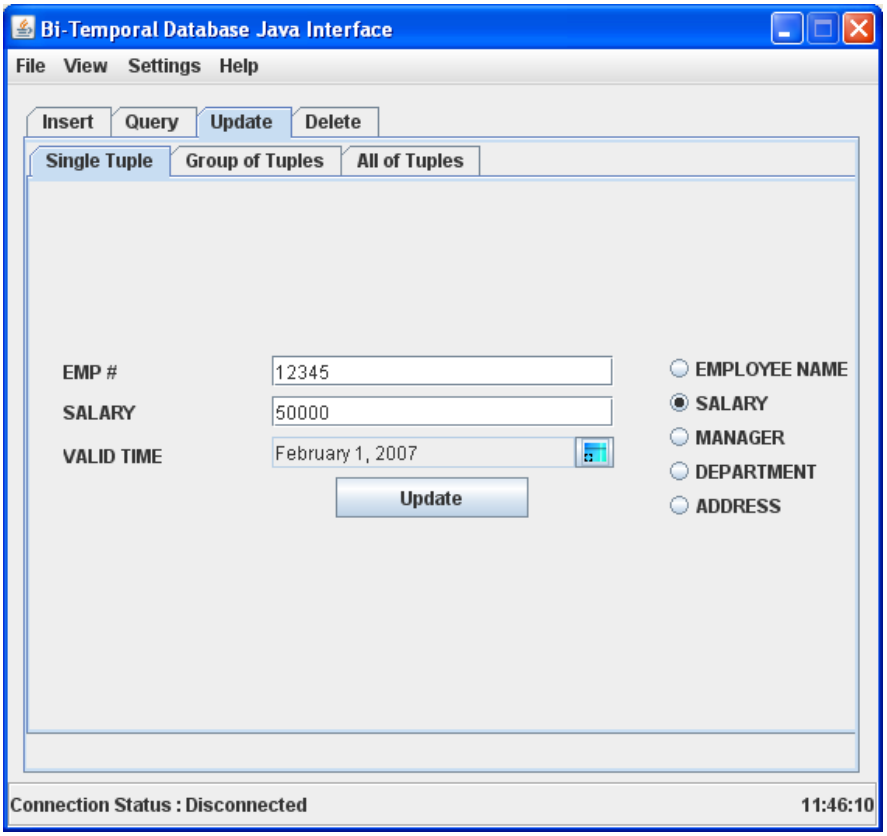


Figure 5.15-a: Single update for salary attribute with BtSQL.

```
UPDATE TABLE(      SELECT E.SALARY
                   FROM EMPLOYEE E
                   WHERE E.EMP#= 12345) SAL
SET SAL.BTA_TYPE = SUBSTR(SAL.BTA_TYPE, 1, 11) || sysdate
||' '|| SUBSTR(SAL.BTA_TYPE, 23, 10) ||' '|| '01.02.2007'
||' '|| SUBSTR(SAL.BTA_TYPE, 45)
WHERE SUBSTR(SAL.BTA_TYPE, 34, 10)='09.09.9999';

INSERT INTO THE(   SELECT E.SALARY
                          FROM EMPLOYEE E
                          WHERE E.EMP# = 12345)
VALUES(bta_type(sysdate||' '||'09.09.9999' ||''||
 '01.02.2007'||' '||'09.09.9999' ||' '|| to char(50000)));
```

Figure 5.15-b: Single update for salary attribute with BTA_String_SQL type.

```
UPDATE TABLE (     SELECT E.SALARY
                   FROM EMPLOYEE E
                   WHERE E.EMP#= 12345) SAL
SET SAL.VALID_TIME_UPPER_BOUND = '01.02.2007',
      SAL.TRAN_TIME_UPPER_BOUND = sysdate
WHERE SAL.VALID_TIME_UPPER_BOUND = '09.09.9999';
INSERT INTO THE ( SELECT E.SALARY
                         FROM EMPLOYEE E
                         WHERE E.SSN = 12345)
VALUES(BTA_TYPE(sysdate,'09.09.9999',
                   '01.02.2007', '09.09.9999', 50000));
```

Figure 5.15-c: Single update for salary attribute with BTA_ADT_SQL type.

If a group of tuples is to be updated, the required bitemporal attributes are selected. Then, BtSQL asks for a condition and the new values, as well as the valid time. The function then adds new bitemporal atom type for tuples, which satisfies the condition. Lastly, in updating all the tuples in the database, the bitemporal attribute(s) are chosen from BtSQL. After receiving the new value(s) and valid time from the user, the function updates all the tuples with the new value and valid time. More update examples are provided in Section 5.4.5.

5.3.3 Delete in BtSQL

If an employee leaves a company, his/her information is typically never deleted from the database – for several reasons. The bitemporal attributes' valid time upper bound is replaced with the provided valid time, and the system provided date is recorded as the transaction time's upper bound. The function receives the EMP# of the employee and the valid time when the employee leaves the company. For every bitemporal attribute, the function finds the last bitemporal atom where the valid time upper bound is 'now', and then replaces it with the given date, when the employee was/is/will no longer be employed. Figure 5.16-a shows the BtSQL's DELETE page, which indicates that EMP# = 13456 will not be working beginning on January 15, 2007. Again, Figures 5.16-b and 5.16-c show the actual SQL code as to how the delete is done for BTA_String_SQL and BTA_ADT_SQL type, respectively.

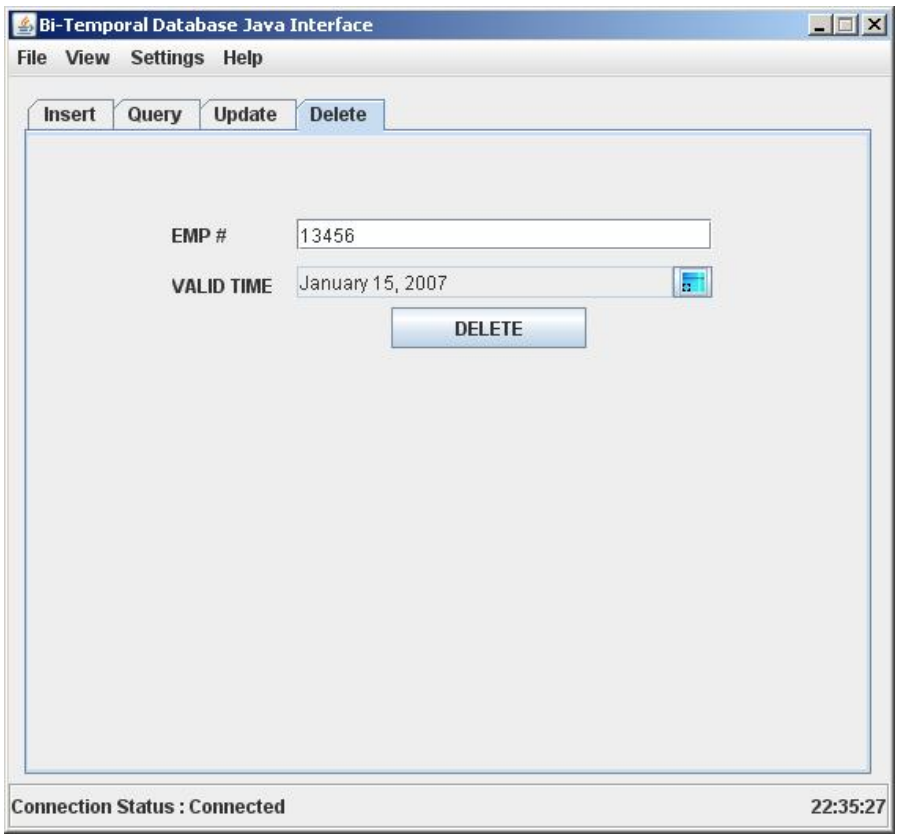


Figure 5.16-a: Delete a tuple with BtSQL

```
UPDATE TABLE (     SELECT E.NAME
                   FROM EMPLOYEE
                   WHERE E.EMP# = 13456) NAM
SET NAM.BTA_TYPE = SUBSTR(NAM.BTA_TYPE, 1, 11) ||
sysdate|| ' '|| SUBSTR(NAM.BTA_TYPE, 23, 10) ||' '||
'01.15.2007'|| ' '|| SUBSTR(NAM.BTA_TYPE, 45)
WHERE SUBSTR(NAM.BTA_TYPE, 34, 10)='09.09.9999';

UPDATE TABLE (     SELECT E.ADDRESS
                   FROM EMPLOYEE
                   WHERE E.EMP# = 13456) ADR
SET ADR.BTA_TYPE = SUBSTR(ADR.BTA_TYPE, 1, 11) ||
sysdate|| ' '|| SUBSTR(ADR.BTA_TYPE, 23, 10) ||' '||
'01.15.2007'|| ' '|| SUBSTR(ADR.BTA_TYPE, 45)
WHERE SUBSTR(ADR.BTA_TYPE, 34, 10)='09.09.9999';

UPDATE TABLE (SELECT DEPARTMENT_HISTORY
               FROM TABLE (SELECT E.DEPT_MNG
                           FROM EMPLOYEE
                           WHERE E.EMP# = 13456)) DEP
SET DEP.BTA_TYPE = SUBSTR(DEP.BTA_TYPE, 1, 11) ||
sysdate|| ' '|| SUBSTR(DEP.BTA_TYPE, 23, 10) ||' '||
'01.15.2007'|| ' '|| SUBSTR(DEP.BTA_TYPE, 45)
WHERE SUBSTR(DEP.BTA_TYPE, 34, 10)='09.09.9999';

UPDATE TABLE (SELECT MANAGER_HISTORY
               FROM TABLE (SELECT E.DEPT_MNG
                           FROM EMPLOYEE
                           WHERE E.EMP# = 13456)) MAN
SET MAN.BTA_TYPE = SUBSTR(MAN.BTA_TYPE, 1, 11) ||
sysdate|| ' '|| SUBSTR(MAN.BTA_TYPE, 23, 10) ||' '||
'01.15.2007'|| ' '|| SUBSTR(MAN.BTA_TYPE, 45)
WHERE SUBSTR(MAN.BTA_TYPE, 34, 10)='09.09.9999';

UPDATE TABLE (SELECT E.SALARY
               FROM EMPLOYEE
               WHERE E.EMP# = 13456) SAL
SET SAL.BTA_TYPE = SUBSTR(SAL.BTA_TYPE, 1, 11) ||
sysdate|| ' '|| SUBSTR(SAL.BTA_TYPE, 23, 10) ||' '||
'01.15.2007'|| ' '|| SUBSTR(SAL.BTA_TYPE, 45)
WHERE SUBSTR(SAL.BTA_TYPE, 34, 10)='09.09.9999';
```

Figure 5.17-b: Delete a tuple with BTA_String_SQL type.

```
UPDATE TABLE (     SELECT E.NAME
                   FROM EMPLOYEE
                   WHERE E.EMP#= 13456) NAM
SET NAM.VALID_TIME_UPPER_BOUND = '01.15.2007',
    NAM.TRAN_TIME_UPPER_BOUND = sysdate
WHERE NAM.VALID_TIME_UPPER_BOUND = '09.09.9999';

UPDATE TABLE (     SELECT E.ADDRESS
                   FROM EMPLOYEE
                   WHERE E.EMP#= 13456) ADR
SET ADR.VALID_TIME_UPPER_BOUND = '01.15.2007',
    ADR.TRAN_TIME_UPPER_BOUND = sysdate
WHERE ADR.VALID_TIME_UPPER_BOUND = '09.09.9999';

UPDATE TABLE (SELECT DEPARTMENT_HISTORY
               FROM TABLE (SELECT E.DEPT_MNG
                           FROM EMPLOYEE
                           WHERE E.EMP#= 13456)) DEP
SET DEP.VALID_TIME_UPPER_BOUND = '01.15.2007',
    DEP.TRAN_TIME_UPPER_BOUND = sysdate
WHERE DEP.VALID_TIME_UPPER_BOUND = '09.09.9999';

UPDATE TABLE (SELECT MANAGER_HISTORY
               FROM TABLE (SELECT E.DEPT_MNG
                           FROM EMPLOYEE
                           WHERE E.EMP#= 13456)) MAN
SET MAN.VALID_TIME_UPPER_BOUND = '01.15.2007',
    MAN.TRAN_TIME_UPPER_BOUND = sysdate
WHERE MAN.VALID_TIME_UPPER_BOUND = '09.09.9999';

UPDATE TABLE (     SELECT E.SALARY
                   FROM EMPLOYEE
                   WHERE E.EMP#= 13456) SAL
SET SAL.VALID_TIME_UPPER_BOUND = '01.15.2007',
    SAL.TRAN_TIME_UPPER_BOUND = sysdate
WHERE SAL.VALID_TIME_UPPER_BOUND = '09.09.9999';
```

Figure 5.17-c: Delete a tuple with BTA_ADT_SQL type.

5.3.4 Queries in BtSQL

BtSQL accepts queries in a bitemporal context, in a current context, or in a historical context. Queries of course involve the relation name listed in the FROM clause. The query selects tuples that satisfy the condition(s) of the WHERE clause, and then projects the result to the attributes listed in the SELECT clause. All the options and flavors of the SELECT statement in SQL can also be used in BtSQL. The preprocessor BtSQL also works with any missing WHERE clause – the same as in regular SQL. It also allows the user to employ the ORDER BY clause, to order the tuples in the result of a query by the values of one or more attributes. GROUP BY, HAVING clauses, and aggregate functions are left for future work.

If a bitemporal attribute's valid time and/or transaction time need to be displayed, they should be specified as VT and/or TT after the bitemporal attribute's name. BTA's valid time and transaction time data can be displayed for all bitemporal attributes. Bitemporal attribute name followed by _VTLB, _VTUB, _TTUB, _TTLB displays valid time upper bound, valid time lower bound, transaction time upper bound, transaction time lower bound, respectively. If two bitemporal attributes' common time intervals – or "when" – need to be queried, then the Slice operation (is its operation or clause) should be chosen. Slice is used in queries as any other clauses independent of the bitemporal context, current context or historical context.

If the query is in a historical context, then the transaction time point or interval is specified in the AS_OF clause, in which all other restrictions as well as capabilities for bitemporal context apply as well. If AS_OF clause is chosen but transaction time value is not specified, it defaults to current context, 'now'.

As an example, Figure 5.18-a displays a query that lists employee numbers and names that currently work in Department 22 and earn more than 100K. The query selects the employee numbers and names that satisfy the conditions DEPARTMENT = 'DEP_ID22' and SALARY>100000. 'now' is chosen for the AS_OF clause which checks if the BTA's transaction time upper bounds are equal to '09.09.9999'. It then passes the result to the EMP# and NAME attributes listed in the SELECT clause. Figures 5.18-b and 5.18-c depict how this query is written with BTA_String_SQL and using BTA_ADT_SQL type, respectively. More query examples can be found in section 5.4.5.

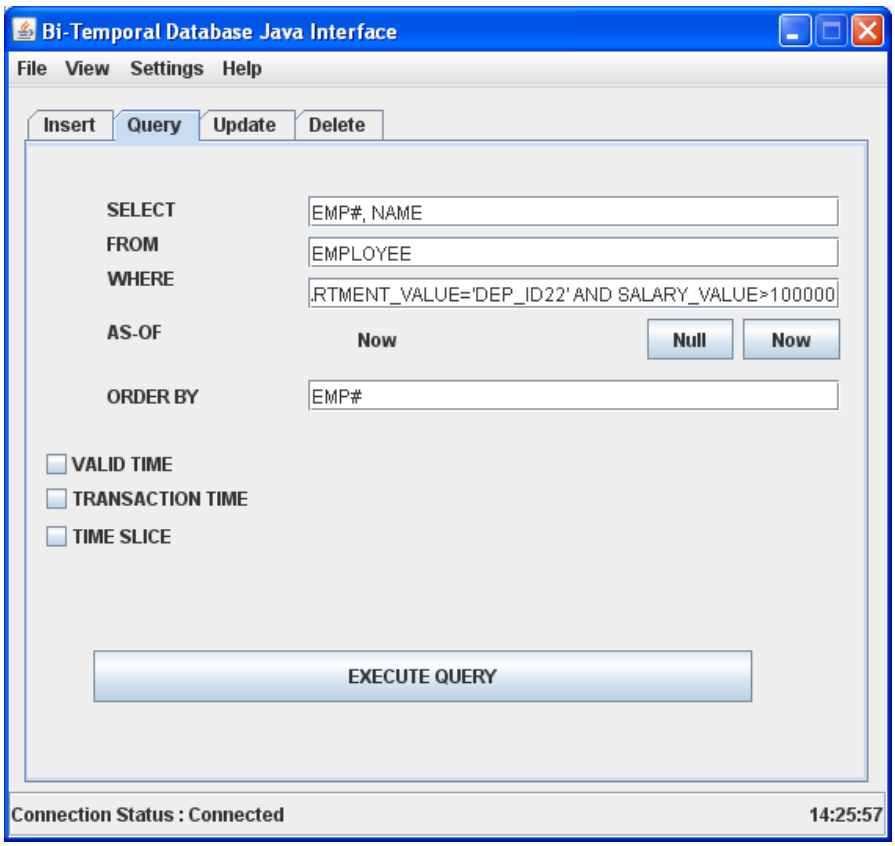


Figure 5.18-a: Snapshot query with BtSQL.

```
SELECT E.EMP#,SUBSTR(NAM.BTA_TYPE, 45) AS NAME
FROM EMPLOYEE E,
      TABLE(E.NAME) NAM,
      TABLE( E.DEPT_MNG) DEP_MAN,
            TABLE( DEP_MAN.DEPARTMENT_HISTORY) DEP,
      TABLE( E.SALARY) SAL
WHERE SUBSTR(DEP.BTA_TYPE, 45) = 'DEP_ID22'
AND SUBSTR(SAL.BTA_TYPE, 45) > 100000
AND SUBSTR(NAM.BTA_TYPE, 34,10) = '09.09.9999'
AND SUBSTR(DEP.BTA_TYPE, 34,10)= '09.09.9999'
AND SUBSTR(SAL.BTA_TYPE, 34,10)= '09.09.9999'
```

Figure 5.18-b: Snapshot query with BTA_String_SQL type.

```
SELECT E.EMP#,NAM.VALUE AS NAME,
FROM EMPLOYEE E,
      TABLE(E.NAME) NAM,
      TABLE(E.DEPT_MNG) DEP_MAN,
            TABLE(DEP_MAN.DEPARTMENT_HISTORY) DEP,
      TABLE(E.SALARY) SAL
WHERE DEP.VALUE = 'DEP_ID22' AND SAL.VALUE >100000
AND NAM.VALID_TIME_UPPER_BOUND ='09.09.9999'
AND SAL.VALID_TIME_UPPER_BOUND ='09.09.9999'
AND DEP.VALID_TIME_UPPER_BOUND ='09.09.9999'
```

Figure 5.18-c: Snapshot query with BTA_ADT_SQL type.

5.4 Performance Evaluation

In order to demonstrate the nested bitemporal relational model proposed in this thesis, we conducted experiments to measure the performance of the model. The experiment was intended to compare the performance of bitemporal tables with different implementations:

- Bitemporal atoms defined as one string written in the native SQL language (BTA_String_SQL) versus the Java language (BTA_String_Java).
- Bitemporal atoms defined as an abstract data type written in the native SQL language (BTA_ADT_SQL) versus the Java language (BTA_ADT_Java).
- Bitemporal atoms stored on nested tables (SofBTA_Table) versus array tables (SofBTA_Array) collection types.

In this experiment, eight databases were created, namely ADT_SQL_Table, ADT_Java_Table, ADT_SQL_Array, ADT_Java_Array, String_SQL_Table, String_-Java_Table, String_SQL_Array, and String_Java_Array. The performance of a bitemporal relational model is represented by the required time for a set of queries and updates over already populated databases.

Each database contains a bitemporal table that has six explicit attributes: EMP#, the primary key of the table, of type INTEGER, and Birthday of type DATE, are both non-

temporal attributes. The other four are bitemporal attributes. NAME, ADDRESS and SALARY bitemporal attributes use either SofBTA_Table or SofBTA_Array type of collection type. DEPARTMENT is a nested table with two columns, DNAME and MANAGER, each one use SofBTA_Table or SofBTA_Array type of collection type. DNAME is recording the department with which the employee is affiliated, and MANAGER is recording the employee's manager.

In comparing the relative performances of the various approaches, answers to the following three questions were explored. First, which implementation method—BTA_String or BTA_ADT – performs faster in terms of database modifications and queries? Second, does the bitemporal atom type that is written in SQL, BTA_ADT_SQL and BTA_String_SQL, perform faster than the one written in Java, BTA_ADT_Java and BTA_String_Java? Third, which approach is the most cost effective – SofBTA_Table or SofBTA_Array collection types for storing the bitemporal history of each object?

The answers to these questions are important, because they should significantly affect bitemporal DBMS design and implementation decisions. For instance, if the SofBTA_Table approach is measurably superior for certain table sizes, and SofBTA_Array approach is superior for other table sizes, then for a given table size, the designer can determine which approach will likely prove to be the most cost effective in a particular situation. The aforesaid questions were answered by running a set of experiments using the same data in all eight tables. Since an existing commercial object relational DBMS was used, a realistic picture of performance is obtained. The performance results were documented, and they clearly indicate the feasibility of the proposed implementation methodologies.

5.4.1 System Configuration

For the experiment, an object relational DBMS was used – Oracle 9i. It was run on a Pentium IV 3.0Ghz PC with 1GB of memory and 1500-3000 Megabytes of system controlled swap space. During the study, the system was used exclusively in our

experiments. The server and client processes ran on the same machine. Oracle database's Java virtual machine was used for handling Java types.

5.4.2 Data Generation

In this step, a set of bitemporal data objects were generated. Since bitemporal data in real-world applications could not be obtained, objects containing bitemporal data were generated synthetically – objects whose bitemporal attributes are random variables drawn from particular distributions between 01.01.1995 and 01.01.2007. The granularity of the DATE values was 'MM.DD.YYYY'. However, all the methods we present are equally valid for any granularity used by the application. 10,000 distinct names and addresses were generated for the testing.

5.4.3 Populating Databases

Unique employee numbers between 10001 and 20000 were used, and increased by one sequentially. Each employee was assigned a birthdate in the MM.DD.YYYY format. MM and DD were integers chosen randomly using uniform distribution between 1 and 12, and 1 and 30, respectively. YYYY is also an integer, and generated using normal distribution, where mean = 1970 and standard deviation = 5. Six thousand tuples had one, two thousand had two, five hundred had three, one thousand had four, and five hundred had five name tags. Names are chosen randomly using normal distribution where mean = 15000 and standard deviation = 1000. The name change also caused an address change. For those tuples that had only one name tag, an additional address tag was provided. Overall, every tuple had two names and addresses tags on average. Table 5.2 depicts the total number of bitemporal atoms on NAME and ADDRESS bitemporal attributes.

Table 5.2: Number of BTAs on NAME and ADDRESS bitemporal attributes.

Number of BTAs	Number of NAME tuples	Number of ADDRESS tuples
1	6000	-
2	2000	8000
4	500	500
3	1000	1000
5	500	500

There were 30 departments and 30 managers from DEP_ID1 through DEP_ID30, and from MANAGER_ID1 through MANAGER_ID30, respectively. Each employee was assigned to one department and one manager at a time. Department and manager numbers were chosen randomly by using normal distribution where mean = 15 and standard deviation = 4.5. Employees changed their departments and managers 5 times on average, and received an additional 5% salary increase when their department changed. Their initial salary value was calculate randomly using normal distribution where mean = 60,000 and standard deviation = 10,000 for each tuple. Every employee had a 3% salary increase on each New Year. Table 5.3 illustrates the selected mean and standard deviation for each attributes. For these updates, it was assumed that the transaction time bounds were within 1 to 10 days less or more from the valid time bounds. For convenience, we assume no null values in the tuples of this relation.

Table 5.3: Selected mean and standard deviation values for attributes.

Attribute Name	Mean	Standard deviation
BIRTHDAY	1970	5
NAME	15000	1000
ADDRESS	15000	1000
MANAGER	15	4.5
DEPT	15	4.5
SALARY	60000	10000

5.4.4 Update Operations and Queries

We conducted three experiments. In the first experiment, we inserted 10,000 tuples to all tables; ADT_SQL_Table, ADT_Java_Table, ADT_SQL_Array, ADT_Java_Array, String_SQL_Table, String_Java_Table, String_SQL_Array, String_Java_Array. We ran other two experiments starting with 10 years of data. Each table thus contains approximately 10,000 tuples, 50,000 set of bitemporal atom tables (SofBTA_Table / SofBTA_Array), and 300,000 bitemporal atoms (BTA_ADT_SQL / BTA_String_SQL / BTA_ADT_Java / BTA_String_Java) that occupy approximately 831 MB.

In the second experiment, the tests were performed by executing modifications as a series of updates for each bitemporal attribute. The update operations are shown below in the preprocessor BtSQL. The first four update operations modify only a single tuple, the next two update modify a group of tuples depending on a condition, and the last one modifies all the tuples on NBRM table. Figures 5.19(a-c) through 5.21(a-c) show how the update1, update 4 and update 7 are written with BtSQL.

In the third experiment, the main goal was to show that the nested bitemporal relational model allows the formulation of useful queries that cannot be easily formulated by the other proposed bitemporal relational models [GB89, BG93, BZ93, S^{+}94, JSS94, CCT94]. To demonstrate the NBRM functionality, we illustrated this point with the two set of queries. The queries were labeled here with the same query numbers that appear in Chapter 4 for easy cross-reference. Queries were designed and run, and the required time was measured over eight different databases. The first group of queries, (Q1 and Q2) used bitemporal context, the second group of queries, (Q3 through Q6) used historical context to test the ‘AS_OF’ operator. Q1 and Q2 queries are on one-level nesting and others are on two-level nesting. Figures 5.22(a-c) through 5.27(a-c) show how these queries are written with BtSQL and SQL.

Update 1: Change the name of an employee to ‘KAMERON JUANA_ONCE’, valid from 07.07.2007, whose EMP# is 19955.

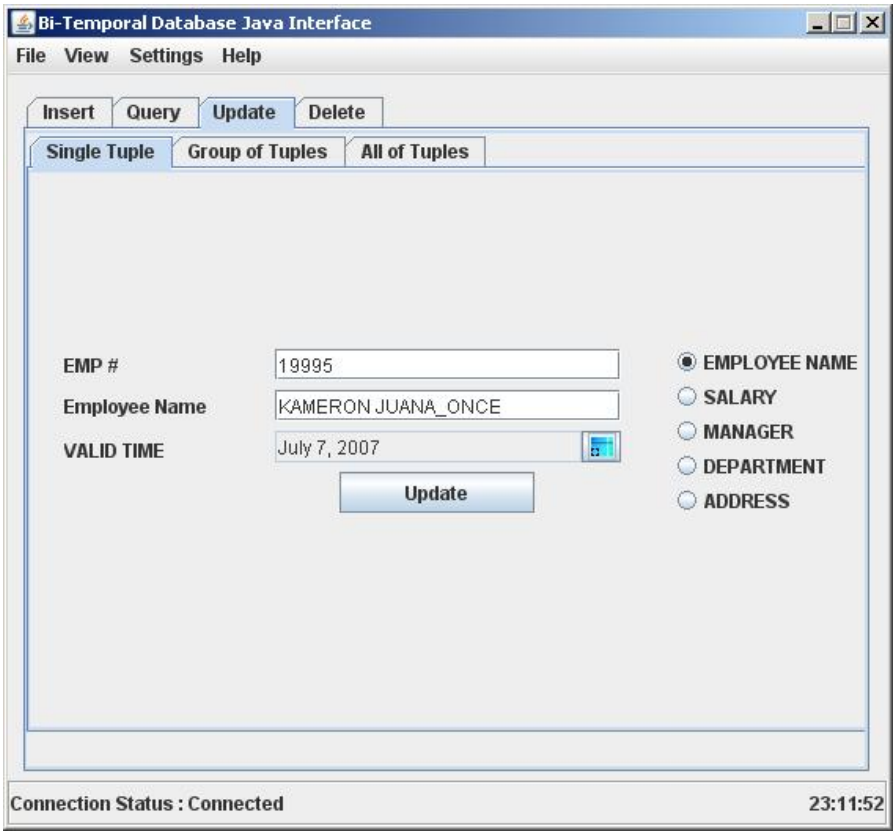


Figure 5.19-a: Update1 with BtSQL.

```
UPDATE TABLE(      SELECT E.NAME
                   FROM EMPLOYEE E
                   WHERE E.EMP#= 19955) NAM
SET NAM.BTA_TYPE = SUBSTR(NAM.BTA_TYPE, 1,
11)||sysdate||' '|| SUBSTR(NAM.BTA_TYPE, 23, 10)||'
'||'07.07.2007'||' '|| SUBSTR(NAM.BTA_TYPE, 45)
WHERE SUBSTR(NAM.BTA_TYPE, 34, 10)='09.09.9999';

INSERT INTO THE ( SELECT E.NAME
                         FROM EMPLOYEE E
                         WHERE E.EMP# = 19955)
VALUES(BTA_TYPE(sysdate ||' '||'09.09.9999'||' '||
            '07.07.2007'||' '||'09.09.9999'||' '||
            'KAMERON JUANA ONCE'));
```

Figure 5.19-b: Update with BTA_String_SQL type.

```
UPDATE TABLE (     SELECT E.NAME
                   FROM EMPLOYEE E
                   WHERE E.EMP#= 19955) NAM
SET NAM.VALID_TIME_UPPER_BOUND = '07.07.2007',
      NAM.TRAN_TIME_UPPER_BOUND = sysdate
WHERE NAM.VALID_TIME_UPPER_BOUND = '09.09.9999';

INSERT INTO THE ( SELECT E.NAME
                         FROM EMPLOYEE E
                         WHERE E.EMP# = 19955)
VALUES (BITEMPORAL_NUMBER(sysdate,'09.09.9999',
      '07.07.2007', '09.09.9999',' KAMERON JUANA_ONCE'));
```

Figure 5.19-c: Update1 with BTA_ADT_SQL type.

Update 2: Change the address of an employee whose EMP# is 19955 to 'Washington Street North Grosvenordale CT 06255', valid from 07.07.2007.

Update 3: Change the employee with EMP# =19955 so his/her department becomes 'DEP_ID12', valid from 07.07.2007.

Update 4: Change the salary of an employee with EMP# = 19955 to 65000 valid from 07.07.2007.

Update 5: Assign MANAGER_ID20 to employees who work for DEP_ID12, valid from 07.07.2007.

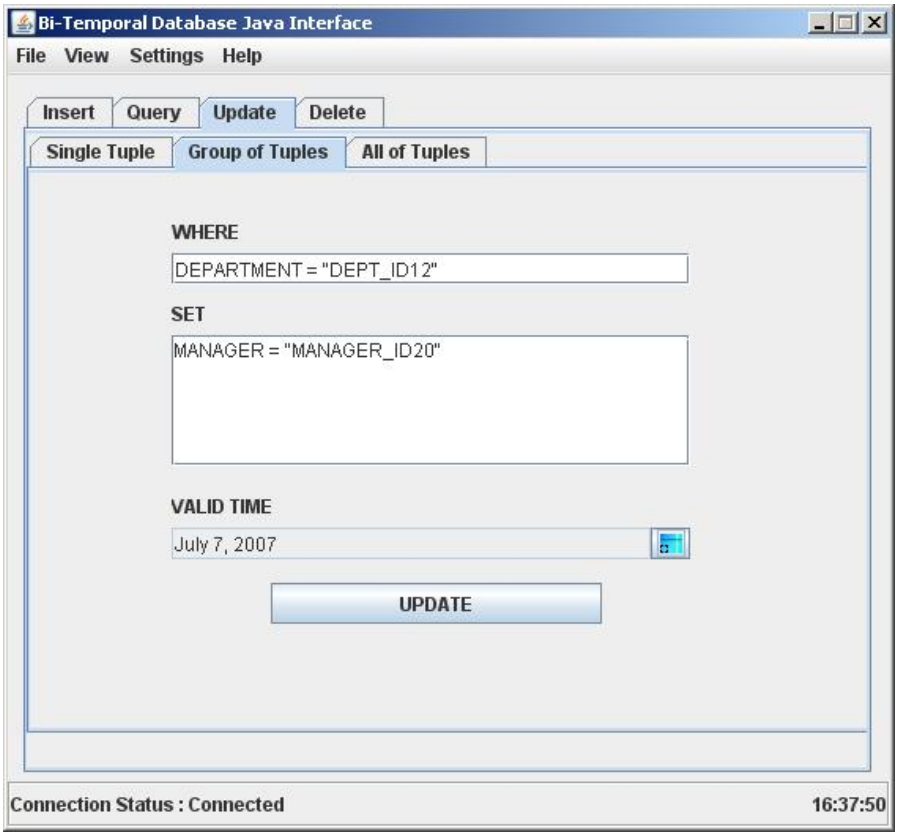


Figure 5.20-a: Update5 with BtSQL.

```
BEGIN
FOR X_C1 IN (      SELECT E.EMP#
                   FROM EMPLOYEE E,
                   TABLE(E.DEPT_MNG) DM,
                         TABLE(DM.DEPARTMENT_HISTORY) D,
                         TABLE(DM.MANAGER_HISTORY) M
                   WHERE SUBSTR(D.BTA_TYPE, 45 ) = 'DEP_ID12'
             AND SUBSTR(M.BTA_TYPE, 45 ) != 'MANAGER_ID20'
             AND SUBSTR(D.BTA_TYPE, 34, 10 ) = '09.09.9999')
LOOP
MY_PKG.SP_UPDATE_MANAGER(X_C1.SSN,'MANAGER_ID20','07.07.2007');
END LOOP;
END;
```

Figure 5.20-b: Update5 with BTA_String_SQL type.

```
BEGIN
FOR X_C1 IN (      SELECT E.EMP#
                   FROM EMPLOYEE E,
                   TABLE(E.DEPT_MNG) DM,
                         TABLE(DM.DEPARTMENT_HISTORY) D,
                         TABLE(DM.MANAGER_HISTORY) M
                   WHERE D.VALUE = 'DEP_ID12'
                   AND M.VALUE != 'MANAGER_ID20'
                   AND D.VALID_TIME_UPPER_BOUND = '09.09.9999')
LOOP
MY_PKG.SP_UPDATE_MANAGER(X_C1.SSN,'MANAGER_ID20','07.07.2007');
END LOOP;
END;
```

Figure 5 20-c: Update5 with BTA_ADT_SQL type.

Update 6: Change MANAGER_ID13 to MANAGER_ID05 for all employees, valid from 07.07.2007.

Update 7: Give a 5% salary increase to all employees, valid from 07.07.2007.

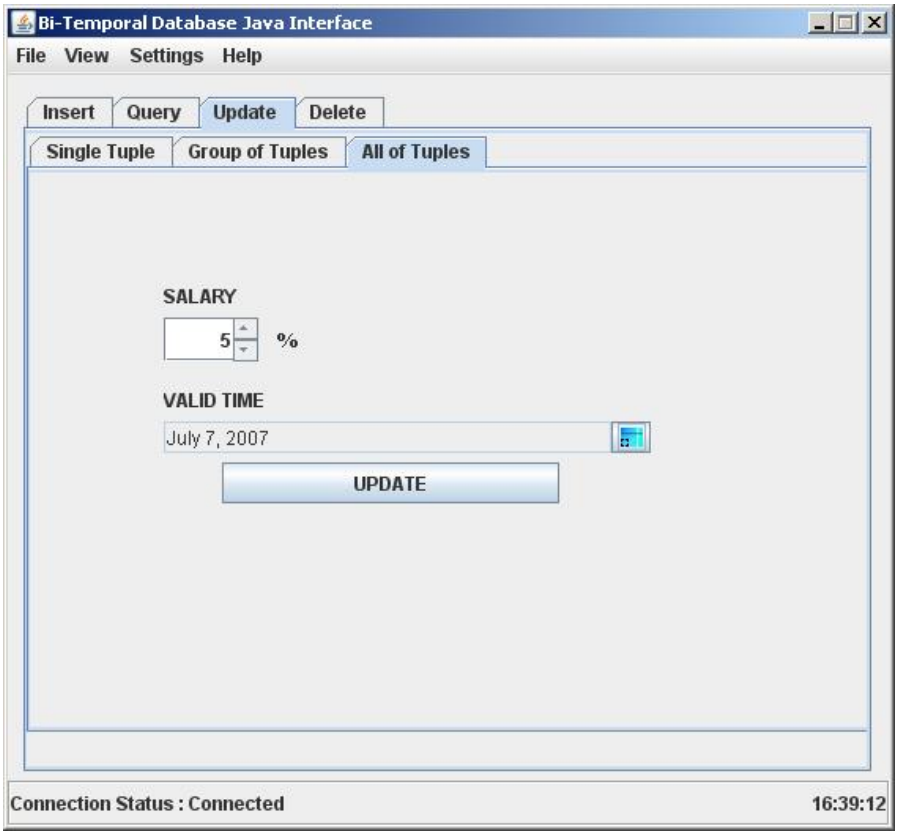


Figure 5.21-a: Update7 with BtSQL.

```
BEGIN
FOR X_C1 IN (SELECT E.EMP#
             FROM EMPLOYEE E, TABLE(E.SALARY) SAL
             WHERE SUBSTR(SAL.BTA_TYPE, 34, 10)= '09.09.9999')
LOOP
MY_PKG.SP_SALARY_INCREASE(5,'07.07.2007');
END LOOP;
END;
```

Figure 5.21-b: Update7 with BTA_String_SQL type.

```
BEGIN
FOR X_C1 IN (SELECT E.EMP#
             FROM EMPLOYEE E, TABLE(E.SALARY) SAL
             WHERE SAL.VALID_TIME_UPPER_BOUND = '09.09.9999')
LOOP
MY_PKG.SP_SALARY_INCREASE(5,'07.07.2007');
END LOOP;
END;
```

Figure 5.21-c: Update7 with BTA_ADT_SQL type.

Query 1: What salary values are stored in the database between the times 01.01.2001 and 01.01.2006?

This is a bitemporal context query, and uses transaction time interval. The selection operation picks tuples where the transaction time components are between 01.01.2005 and 01.01.2006. The projection operation retains the EMP# as the first attribute, as well as the value and other four components from the SALARY bitemporal attribute. Because AS_OF clause is not chosen, it defaults to ‘now’.

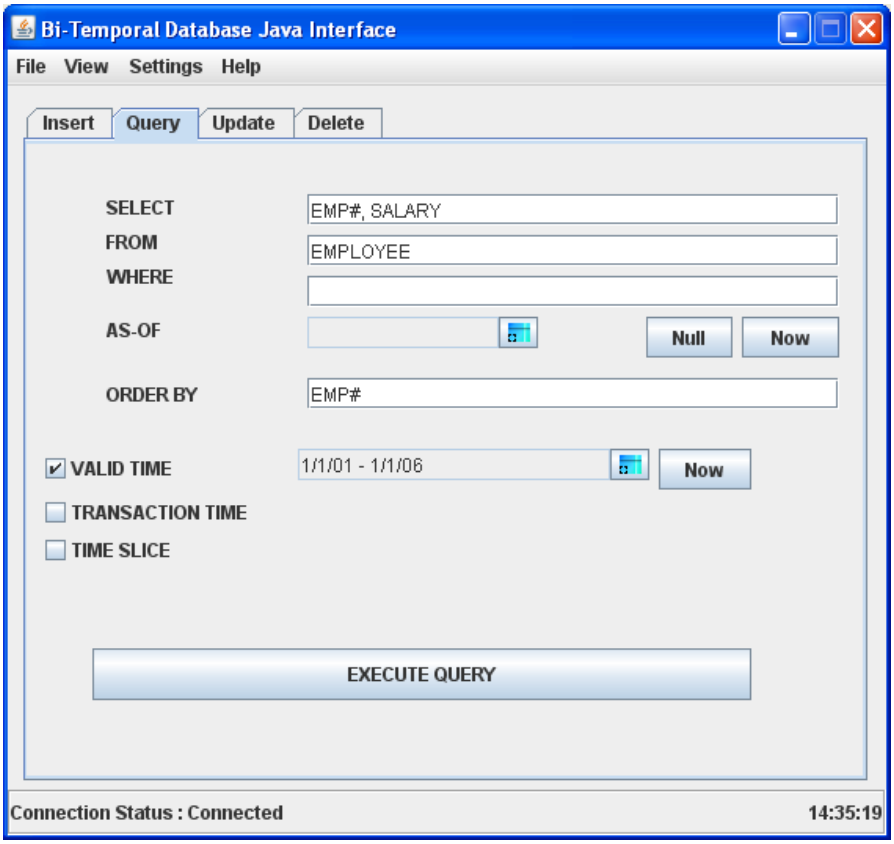


Figure 5.22-a: Query1 with BtSQL.

```
SELECT E.EMP#,
SUBSTR(SAL.BTA_TYPE, 45) AS SALARY,
SUBSTR(SAL.BTA_TYPE, 1,10) AS TT_LOWERBOUND,
SUBSTR(SAL.BTA_TYPE, 12,10) AS TT_UPPERBOUND,
SUBSTR(SAL.BTA_TYPE, 23,10) AS VT_LOWERBOUND,
SUBSTR(SAL.BTA_TYPE, 34,10) AS VT_UPPERBOUND
FROM EMPLOYEE E, TABLE(E.SALARY) SAL
WHERE SUBSTR(SAL.BTA_TYPE, 23,10) BETWEEN '01.01.2001' AND
'01.01.2006'
```

Figure 5.22-b: Query1 with BTA_String_SQL type.

```
SELECT E.EMP#,
SAL.VALUE AS SALARY,
SAL.TRAN_TIME_LOWER_BOUND AS TT_LOWERBOUND,
SAL.TRAN_TIME_UPPER_BOUND AS TT_UPPERBOUND,
SAL.VALID_TIME_LOWER_BOUND AS VT_LOWERBOUND,
SAL.VALID_TIME_UPPER_BOUND AS VT_UPPERBOUND
FROM EMPLOYEE E, TABLE(E.SALARY) SAL
WHERE SAL.VALID_TIME_LOWER_BOUND BETWEEN '01.01.2001' AND
'01.01.2006'
```

Figure 5.22-c: Query1 with BTA_ADT_SQL type.

Query 2: What is the employee number of the employees who shared the same addresses at the same time? When was it?

This is also a bitemporal context query. Join and slice operations are used. The selection operation picks tuples where the ADDRESS attributes' value components are equal. The time slice operation synchronizes the valid time component of the ADDRESS with respect to the ADDRESS_A valid time component and hence implements ‘when’. Finally, the projection operation retains EMP#’s, ADDRESS bitemporal attributes value components along with common valid time lower and upper bounds.

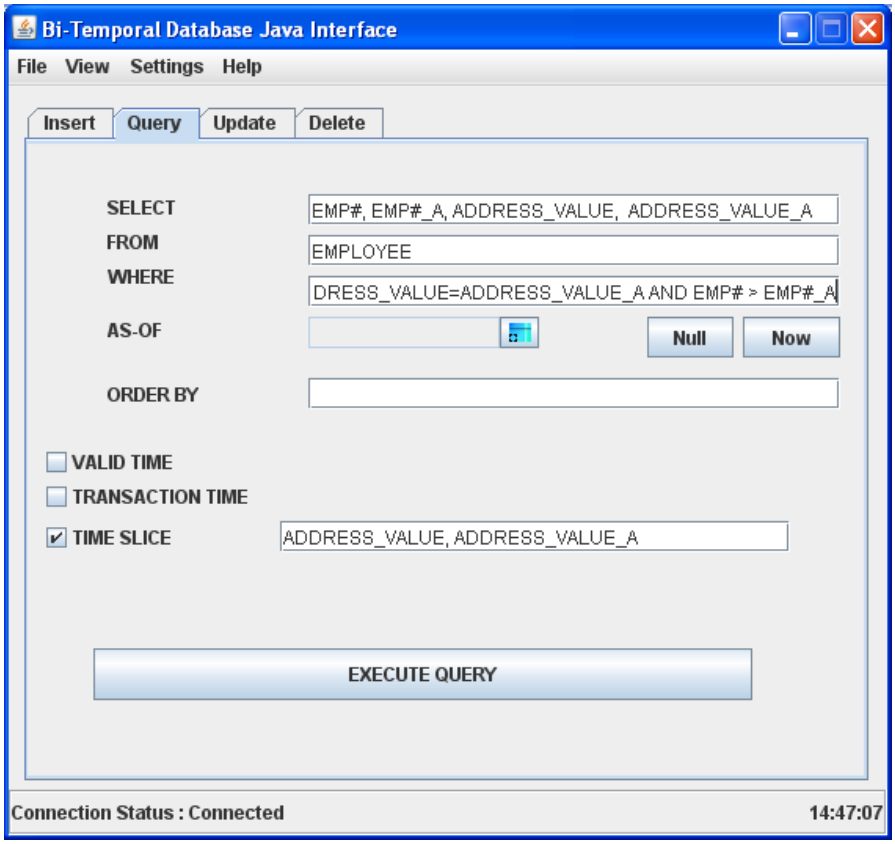


Figure 5.23-a: Query2 with BtSQL.

```
SELECT E.EMP#, SUBSTR(A.BTA TYPE, 45),
       E1.EMP#, SUBSTR(B.BTA_TYPE, 45)
FROM EMPLOYEE E, TABLE(E.ADDRESS) A,
       EMPLOYEE E1, TABLE(E1.ADDRESS) B
WHERE SUBSTR(A.BTA_TYPE, 45)=SUBSTR(B.BTA_TYPE, 45)
AND E.EMP# > E1.EMP#
AND MY_PKG.TIME_SLICE(E.EMP#, E1.EMP#,
     (A.BTA TYPE, 45)=SUBSTR(B.BTA TYPE, 45))
```

Figure 5.23-b: Query2 with BTA_String_SQL type.

```
SELECT E.EMP#, A.ADDRESS.VALUE, E1.EMP#, B.ADDRESS.VALUE
FROM EMPLOYEE E, TABLE(E.ADDRESS) A,
       EMPLOYEE E1, TABLE(E1.ADDRESS) B
WHERE A.VALUE = B.VALUE
AND E.EMP# > E1.EMP#
AND MY PKG.TIME SLICE(E.EMP#, E1.EMP#, A, B)
```

Figure 5.23-c: Query2 with BTA_ADT_SQL type.

Query 3: Get all records of the departments in which the employee CANAN EREN has worked in the database during the date range [‘01.01.2004’, ’12.12.2005’].

This is a historical context query with time interval. The AS_OF operation rolls back the department attribute to time value interval ‘01.01.2004’, ’12.12.2005’. The selection operation picks tuples from the name attribute where the value is ‘CANAN EREN’, and then the projection operation displays the department attribute value and valid time components.

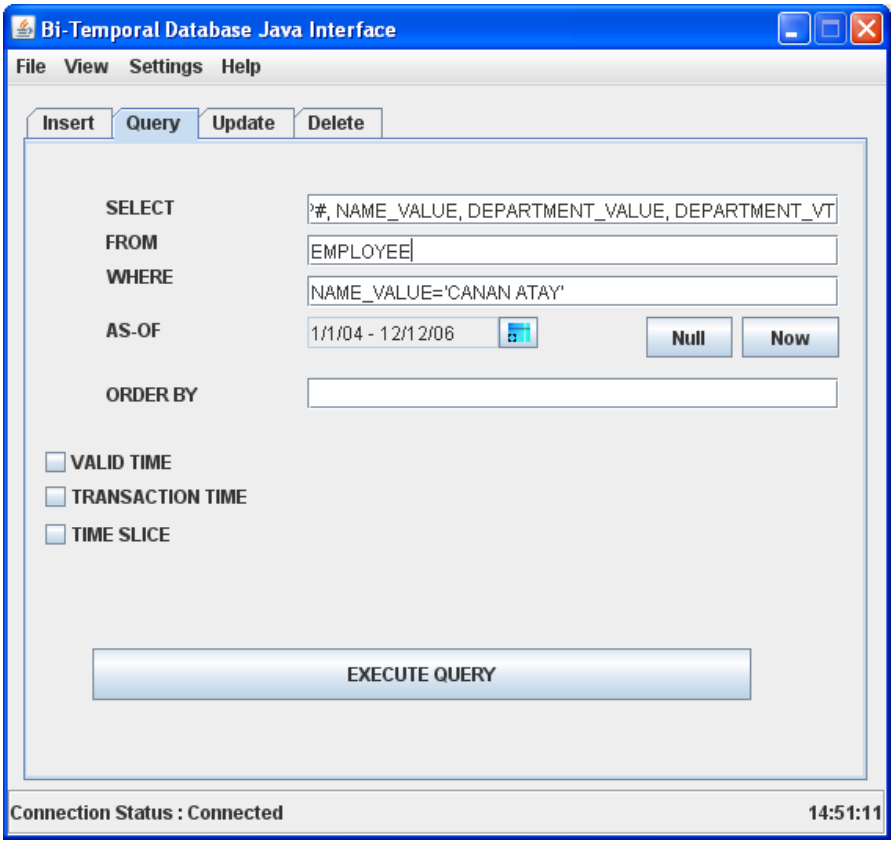


Figure 5.24-a: Query3 with BtSQL.

```
SELECT E.EMP#,
SUBSTR(NAM.BTA_TYPE, 45) AS NAME,
SUBSTR(DEP.BTA_TYPE, 45) AS DEPARTMENT,
SUBSTR(DEP.BTA_TYPE, 23,10) AS VT_LOWERBOUND,
SUBSTR(DEP.BTA_TYPE, 34,10) AS VT_UPPERBOUND
FROM EMPLOYEE E, TABLE(E.NAME) NAM,
      TABLE(E.DEPT_MNG) DEP_MAN,
      TABLE(DEP_MAN.DEPARTMENT_HISTORY) DEP
WHERE MY_PKG.AS_OF(DEP.BTA_TYPE, '01.01.2004', '12.12.2005')=1
AND SUBSTR(MAN.BTA_TYPE, 45) LIKE '%CANAN EREN%'
```

Figure 5.24-b: Query3 with BTA_String_SQL type.

```
SELECT E.EMP#,NAM.VALUE AS NAME,
DEP.VALUE AS DEPARTMENT,
DEP.VALID_TIME_LOWER_BOUND AS VT_LOWERBOUND,
DEP.VALID_TIME_UPPER_BOUND AS VT_UPPERBOUND
FROM EMPLOYEE E, TABLE(E.NAME) NAM,
      TABLE(E.DEPT_MNG) DEP_MAN,
      TABLE(DEP_MAN.DEPARTMENT_HISTORY) DEP
WHERE MY_PKG.AS_OF(VALUE(DEP), '01.01.2004','12.12.2005')=1
AND NAM.VALUE LIKE '%CANAN EREN%'
```

Figure 5.24-c: Query3 with BTA_ADT_SQL type.

Query 4: As of time 01.01.2006, who was working in the DEP_ID22 department?

This is a historical query retrieving the state of a table as of '01.01.2006' in the past. The selection operation picks tuples where value component is equal to 'DEP_ID22'. The projection operation retains the EMP#, name attribute value part, department attribute's name and valid time components.

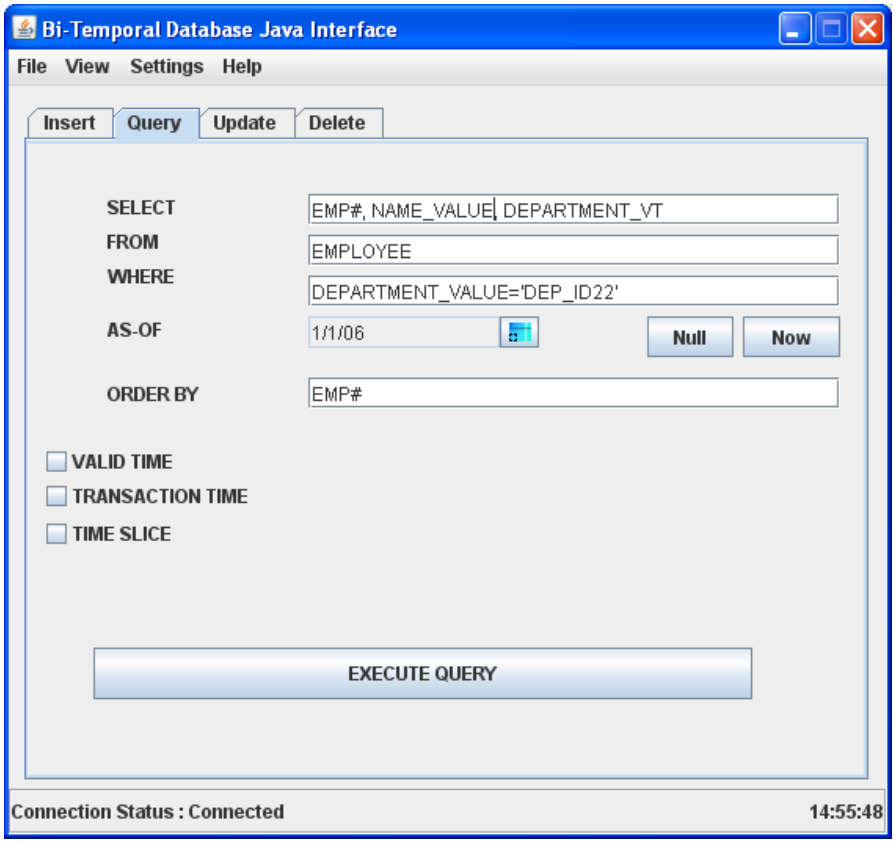


Figure 5.25-a: Query4 with BtSQL.

```
SELECT E.EMP#, SUBSTR(NAM.BTA TYPE, 45) AS NAME,
SUBSTR(DEP.BTA_TYPE, 22,11) AS VT_LOWERBOUND,
SUBSTR(DEP.BTA_TYPE, 33,11) AS VT_UPPERBOUND
FROM EMPLOYEE E, TABLE(E.NAME) NAM,
      TABLE(E.DEPT_MNG) DEP_MAN,
      TABLE(DEP_MAN.DEPARTMENT_HISTORY) DEP
WHERE MY_PKG.AS_OF(DEP.BTA_TYPE, '01.01.2006')=1
AND SUBSTR(DEP.BTA TYPE, 45) = 'DEP ID22'
```

Figure 5.25-b: Query4 with BTA_String_SQL type.

```
SELECT E.EMP#, NAM.VALUE AS NAME,
DEP.VALID_TIME_LOWER_BOUND AS VT_LOWERBOUND,
DEP.VALID_TIME_UPPER_BOUND AS VT_UPPERBOUND
FROM EMPLOYEE E, TABLE(E.NAME) NAM,
      TABLE(E.DEPT_MNG) DEP_MAN,
      TABLE(DEP_MAN.DEPARTMENT_HISTORY) DEP
WHERE MY_PKG.AS_OF(VALUE(DEP), '01.01.2006')=1
AND DEP.VALUE = 'DEP ID22'
```

Figure 5.25-c: Query4 with BTA_ADT_SQL type.

Query 5: Who was CANAN EREN's manager between the times '06.06.2003' and '01.01.2007' as known to the database system within the time range ['06.06.2003', '08.08.2006']?

This historical query retrieves the state of a table between '06.06.2003' and '08.08.2006'. The manager attribute's bitemporal atom's valid time is compared with the lower bound of '06.06.2003' and the upper bound of '01.01.2007', respectively. This query lists if any error correction was made. The selection operation picks tuples from the name attribute, where the value is 'CANAN EREN' and the valid time component is between '06.06.2003' and '01.01.2007'; then the projection operation displays the manager attribute's all components.

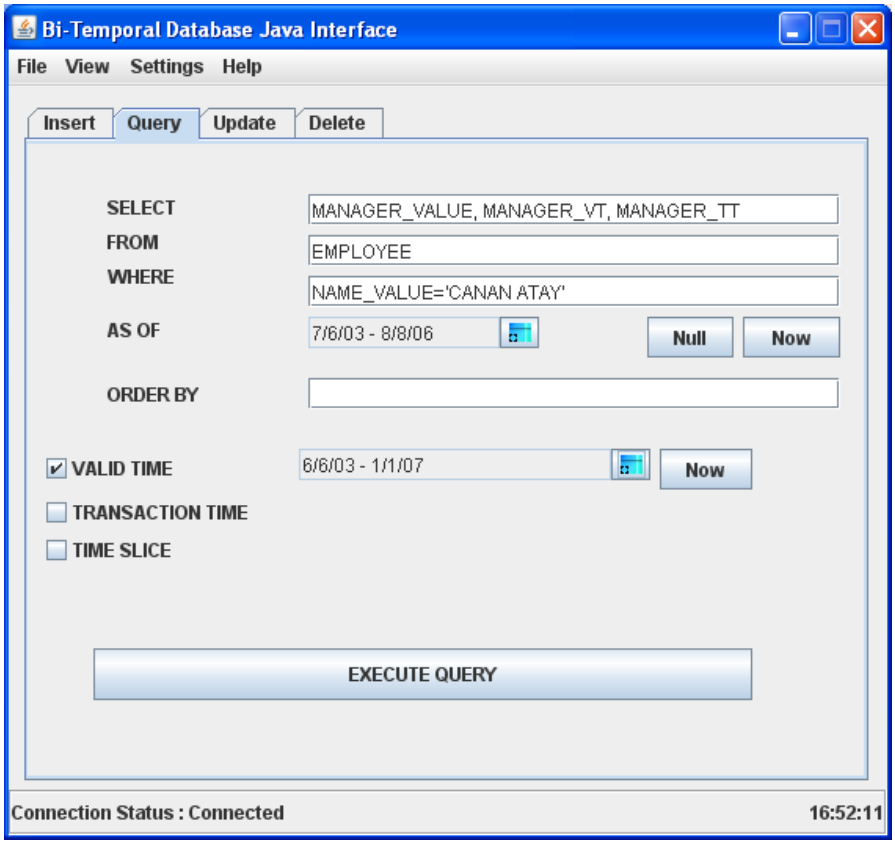


Figure 5.26-a: Query5 with BtSQL.

```
SELECT SUBSTR(MAN.BTA_TYPE, 45) AS MANAGER,
SUBSTR(MAN.BTA_TYPE, 23,10) AS VT_LOWERBOUND,
SUBSTR(MAN.BTA_TYPE, 34,10) AS VT_UPPERBOUND,
SUBSTR(MAN.BTA_TYPE, 1,10) AS TT_LOWERBOUND,
SUBSTR(MAN.BTA_TYPE, 12,10) AS TT_UPPERBOUND
FROM EMPLOYEE E, TABLE(E.NAME) NAM,
      TABLE(E.DEPT_MNG) DEP_MAN,
      TABLE(DEP_MAN.MANAGER_HISTORY) MAN
WHERE SUBSTR(NAM.BTA_TYPE, 45) = 'CANAN EREN'
AND MY_PKG.AS_OF(DEP.BTA_TYPE, '06.06.2003',
'08.08.2006')=1
AND SUBSTR(MAN.BTA_TYPE, 33,11) BETWEEN '06.06.2003' AND
'01.01.2007'
```

Figure 5.26-b: Query5 with BTA_String_SQL type.

```
SELECT MAN.VALUE AS MANAGER,
MAN.VALID_TIME_LOWER_BOUND AS VT_LOWERBOUND,
MAN.VALID_TIME_UPPER_BOUND AS VT_UPPERBOUND,
MAN.TRANSACTION_TIME_LOWER_BOUND AS TT_LOWERBOUND,
MAN.TRANSACTION_TIME_UPPER_BOUND AS TT_UPPERBOUND
FROM EMPLOYEE E, TABLE(E.NAME) NAM,
      TABLE(E.DEPT_MNG) DEP_MAN,
      TABLE(DEP_MAN.MANAGER_HISTORY) MAN
WHERE NAM.VALUE = 'CANAN EREN'
AND MY_PKG.AS_OF(VALUE(MAN),'06.06.2003', '08.08.2006')=1
AND MAN.VALID_TIME_UPPER_BOUND BETWEEN '06.06.2003' AND
'01.01.2007'
```

Figure 5.26-c: Query5 with BTA_ADT_SQL type.

Query 6: What are the EMP# and managers of employees who were employed by the company between '01.01.2006' and '03.03.2006', as known by the database on '06.06.2006'?

This is a historical context query with time point. The AS_OF operation rolls back the manager attribute to time point '06.06.2006'. The selection operation picks manager tuples where valid time components are between '01.01.2006' and '03.03.2006. The projection operation retains the EMP# and manager attributes' all five components.

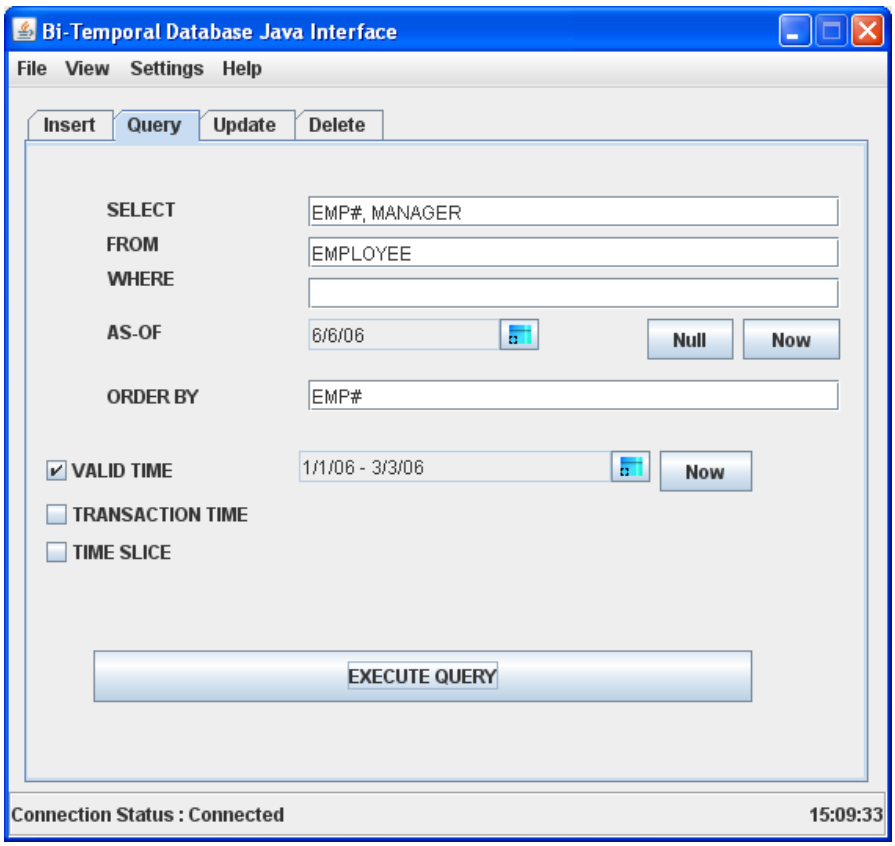


Figure 5.27-a: Query6 with BtSQL.

```
SELECT E.EMP#, SUBSTR(MAN.BTA TYPE, 45) AS MANAGER,
SUBSTR(MAN.BTA_TYPE, 23,10) AS VT_LOWERBOUND,
SUBSTR(MAN.BTA_TYPE, 34,10) AS VT_UPPERBOUND,
SUBSTR(MAN.BTA_TYPE, 1, 10) AS TT_LOWERBOUND,
SUBSTR(MAN.BTA_TYPE, 12,10) AS TT_UPPERBOUND
FROM EMPLOYEE E, TABLE(E.DEPT_MNG) DEP_MAN,
      TABLE(DEP_MAN.MANAGER_HISTORY) MAN,
      TABLE(DEP_MAN.DEPARTMENT_HISTORY) DEP
WHERE MY_PKG.AS_OF(DEP.BTA_TYPE, '06.06.2006')=1 AND
SUBSTR(MAN.BTA_TYPE, 23,10) BETWEEN '01.01.2006' AND
'03.03.2006'
```

Figure 5.27-b: Query6 with BTA_String_SQL type.

```
SELECT E.EMP#, MAN.VALUE AS MANAGER,
MAN.VALID_TIME_LOWER_BOUND AS VT_LOWERBOUND,
MAN.VALID_TIME_UPPER_BOUND AS VT_UPPERBOUND,
MAN.TRAN_TIME_LOWER_BOUND AS TT_LOWERBOUND,
MAN.TRAN_TIME_UPPER_BOUND AS TT_UPPERBOUND
FROM EMPLOYEE E, TABLE(E.DEPT_MNG) DEP_MAN,
      TABLE(DEP_MAN.MANAGER_HISTORY) MAN,
      TABLE(DEP_MAN.DEPARTMENT_HISTORY) DEP
WHERE MY_PKG.AS_OF(VALUE(DEP), '06.06.2006')=1 AND
MAN.VALID_TIME_LOWER_BOUND BETWEEN '01.01.2006'AND
'03.03.2006'
```

Figure 5.27-c: Query6 with BTA_ADT_SQL type.

Comparing these queries to the SpyTime benchmark queries in [SZ01] results as follows. While SpyTime do not have any current queries, we have an example of "now" query in section 5.3.4. Both SpyTime and NBRM queries have examples of valid/transaction time point, valid time interval related bitemporal queries. While SpyTime does not support transaction time interval type queries, NBRM queries do. NBRM queries query given time point or time interval in the past (historical context), but SpyTime does not have such an example. Both SpyTime and NBRM queries have auditing purposes type bitemporal context queries. Because BtSQL is powerful enough to support all the semantics of the queries listed in [SZ01], NBRM queries satisfy more than the requirements of SpyTime benchmark queries.

5.4.5 Results of Update Operations and Queries

This section reports the results and findings from the testing of the Nested Bitemporal Relational Model, and also discusses the strengths and weaknesses of the individual modules implemented. Table 5.4 lists implementation methods and corresponding numbers used for testing.

Table 5.4: List of table names.

Table Name	Implementation Method
ADT_SQL_Table	BTA: Abstract data type written with SQL SofBTA_Table: Implemented on nested table type
ADT_Java_Table	BTA: Abstract data type written with Java SofBTA_Table: Implemented on nested table type
ADT_SQL_Array	BTA: Abstract data type written with SQL SofBTA_Array: Implemented on array table type
ADT_Java_Array	BTA: Abstract data type written with Java, SofBTA_Array: Implemented on array table type
String_SQL_Table	BTA: String implementation type written with SQL SofBTA_Table: Implemented on nested table type
String_Java_Table	BTA: String implementation type written with Java SofBTA_Table: Implemented on nested table type
String_SQL_Array	BTA: String implementation type written with SQL SofBTA_Array: Implemented on array table type
String_Java_Array	BTA: String implementation type written with Java SofBTA_Array: Implemented on array table type

Figure 5.28 shows the run times for the 10,000 initial insert for all tables. Insert time is approximately the same for the BTA_String and the BTA_ADT, written with SQL or in Java. As Figure 5.28 indicates, inserting takes much less time with SofBTA_Array tables than the SofBTA_Table.

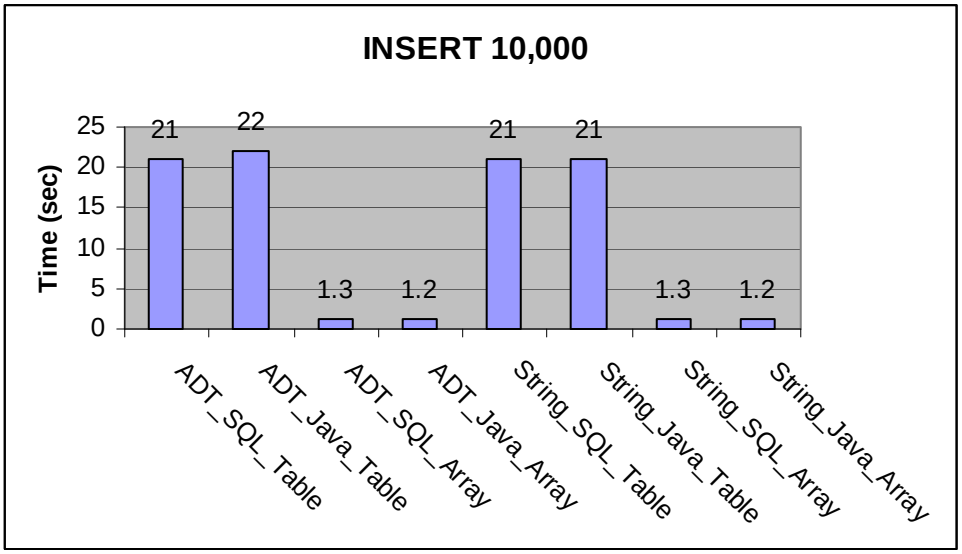


Figure 5.28: Insert times for 10,000 initial tuples.

The execution times from update_1 to update_4 are presented in Figure 5.29. BTA_ADT performs significantly better than the BTA_String. In term of collection tables, SofBTA_Array type with BTA_ADT combination performs better than all other three combinations.

Update_5 and update_6 update a set of tuples resulting from a selection condition applied to a table. BTA_ADT performs significantly better than the ADT_String. SofBTA_Table performs the same with both BTA_ADT and BTA_String. SofBTA_Array with BTA_ADT outperforms greatly BTA_String. Figure 5.30 depicts the results of these two updates.

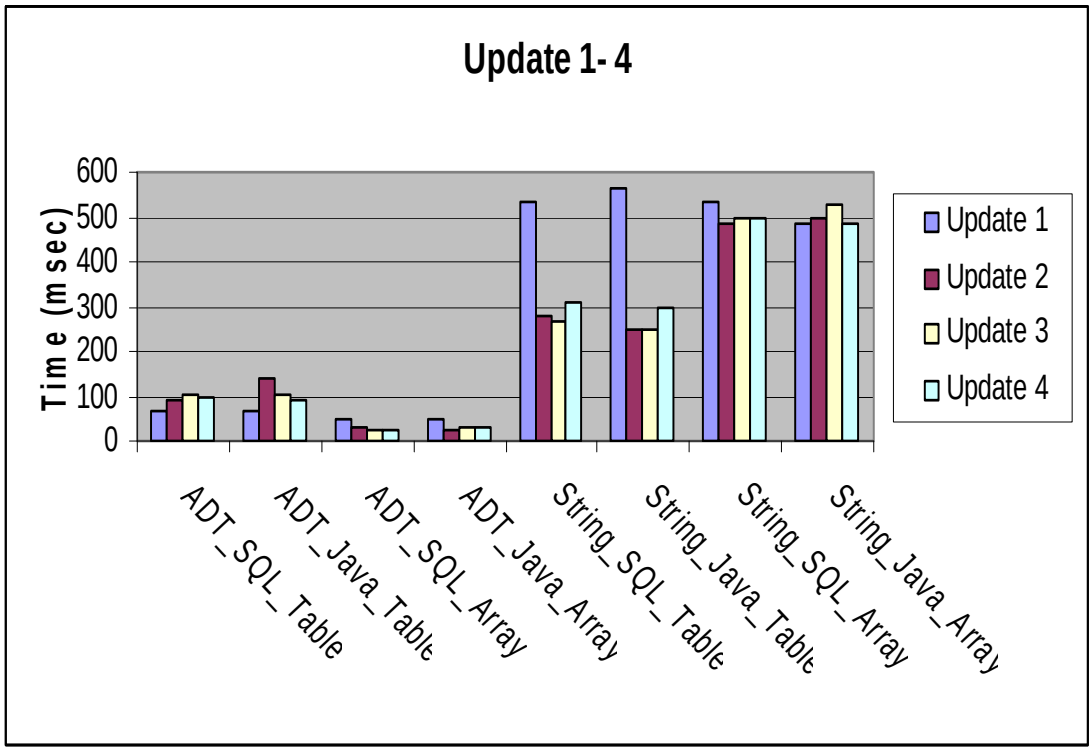


Figure 5.29: Updating a single tuple times.

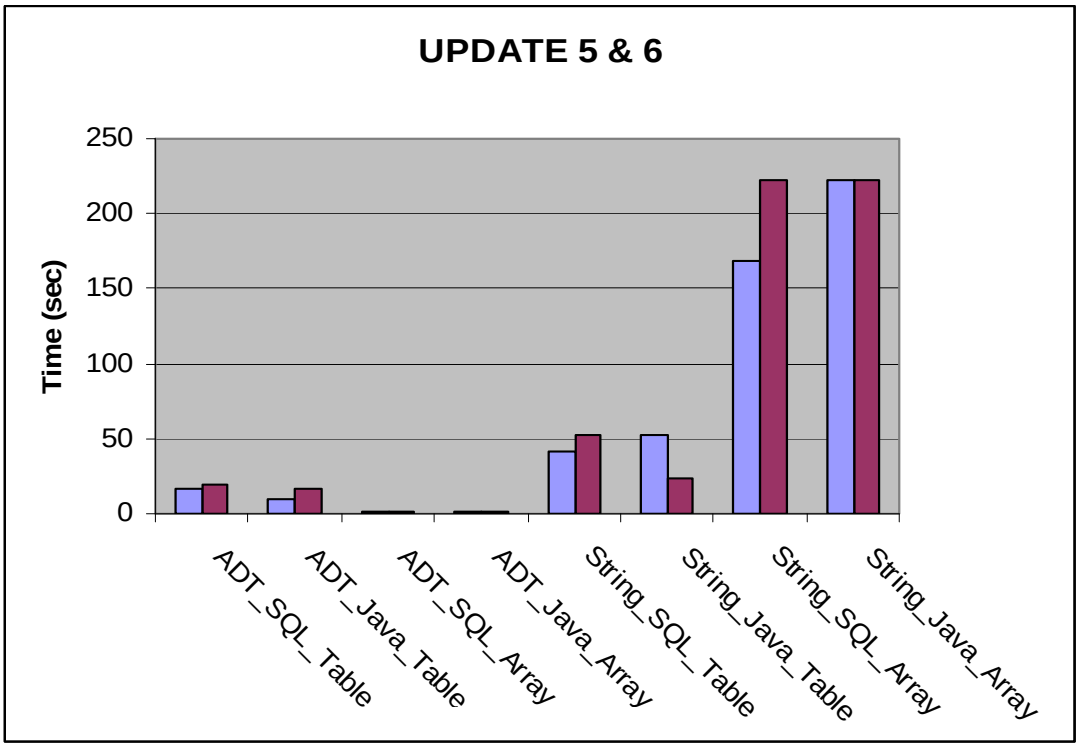


Figure 5.30: Updating a group of tuples times.

Figure 5.31 shows the results of updating the time-related attributes for all tuples. Same as with other updates, BTA_ADT performs to a great extent than the BTA_String. In term of SofBTA's, SofBTA_Array with BTA_ADT outperforms greatly BTA_String. For all type of updates, BTA_ADT and BTA_String defined with SQL or Java performs the same.

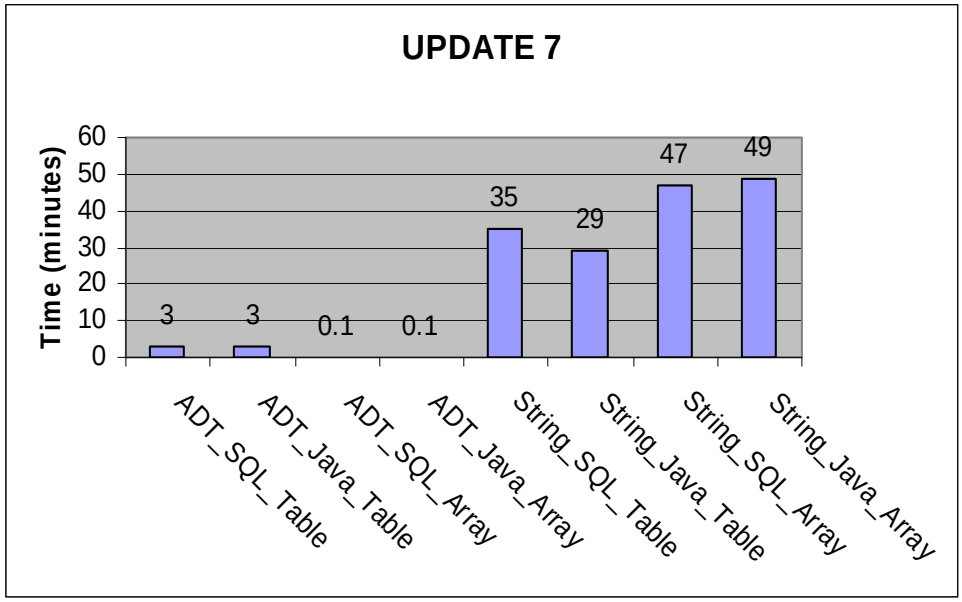


Figure 5.31: Updating all tuples times.

Query1 goes through the salary of employees, and returns the salary values along with their associated timestamps between the given valid time ranges. All tested methods returned the selected tuples almost at the same run time. BTA_ADT and BTA_String time components are extracted and used in the expression successfully on bitemporal context. Figure 5.32 is the result of query times for query1.

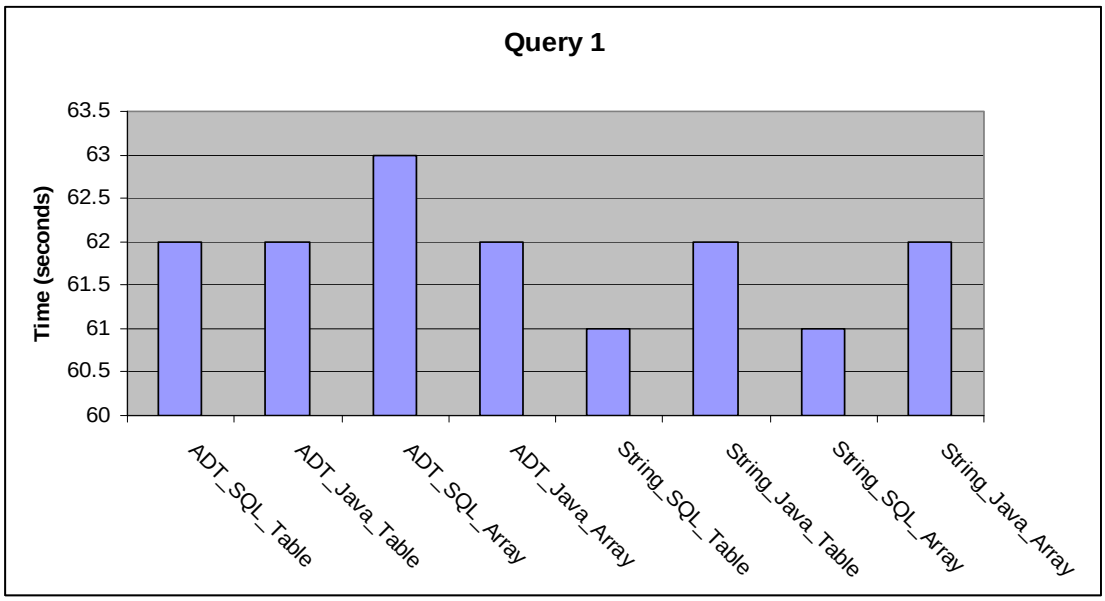


Figure 5.32: Query1 times for different implementation methods

Query2 joins the table with itself, and then uses the time slice operation. As figure 5.33 indicates, SofBTA_Table with BTA_String outperforms all other tested methods.

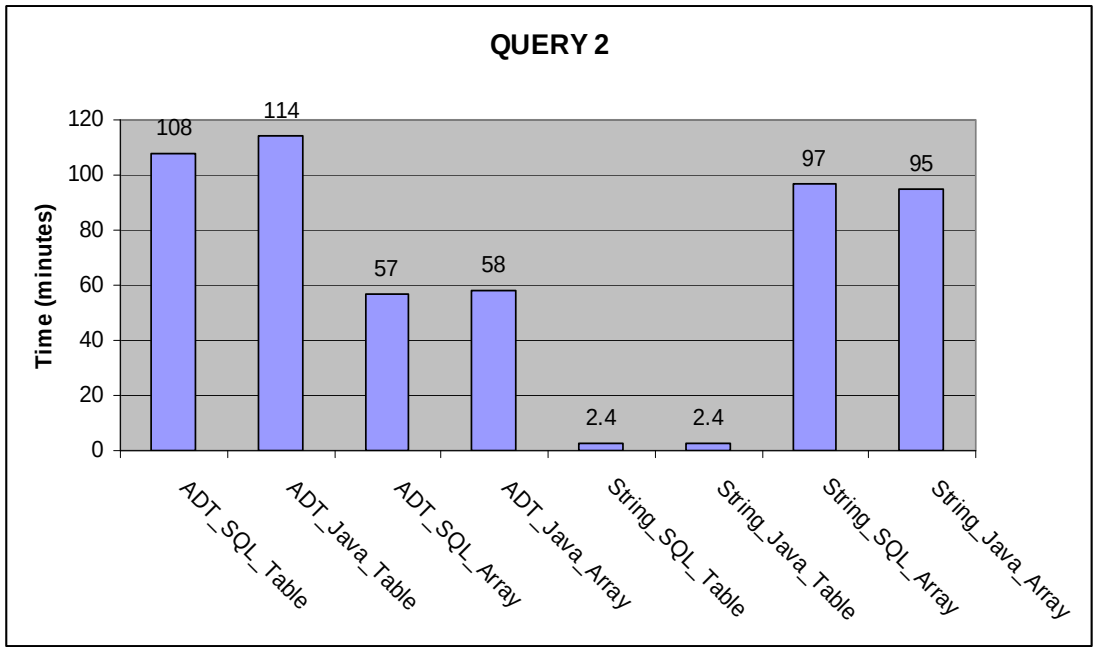


Figure 5.33: Query2 times for different implementation methods

Query3 selects one tuple i.e., the department of a particular employee from rollbacked attribute. Rollbacked bitemporal attribute is on two level of nesting. BTA_String and BTA_ADT perform almost the same. In terms of SofBTA's, SofBTA_Array outperforms SofBTA_Table. Query3 run times are depicted in Figure 5.34.

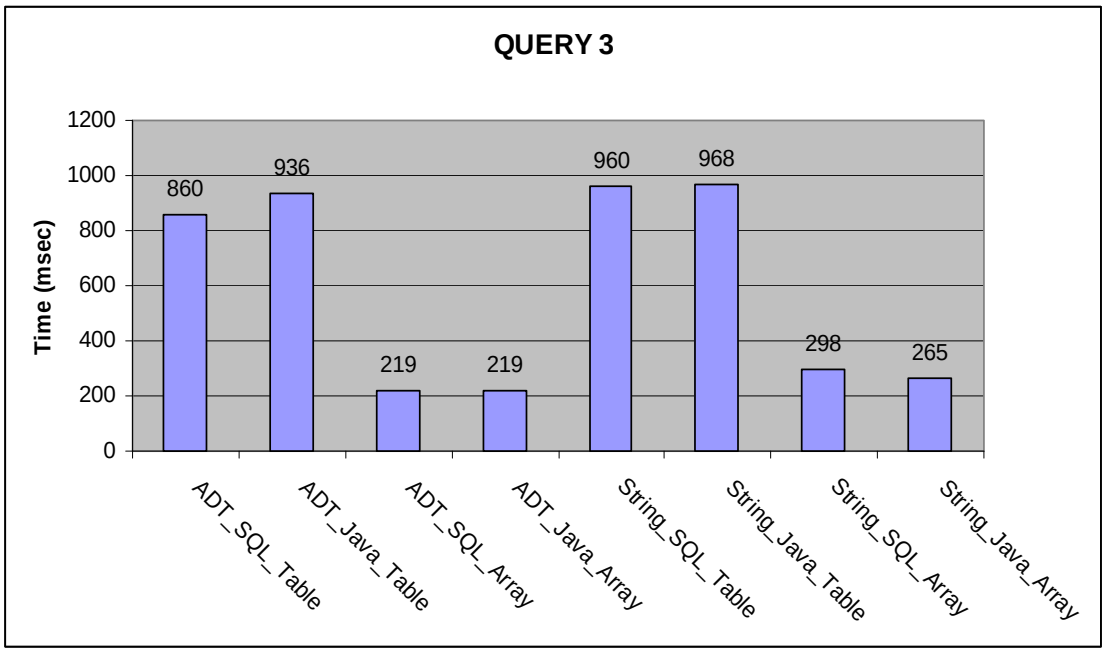


Figure 5.34: Query3 times for different implementation methods

Query4 first rolls back the two level-nested ‘DEPARTMENT’ bitemporal attribute. Then, it goes through every tuple and returns the name of employees who is affiliated with given department id. In this query type, BTA_ADT and BTA_String perform the same. SofBTA_Table performs slightly better than the SofBTA_Array. Running times for query4 are presented in Figure 5.35.

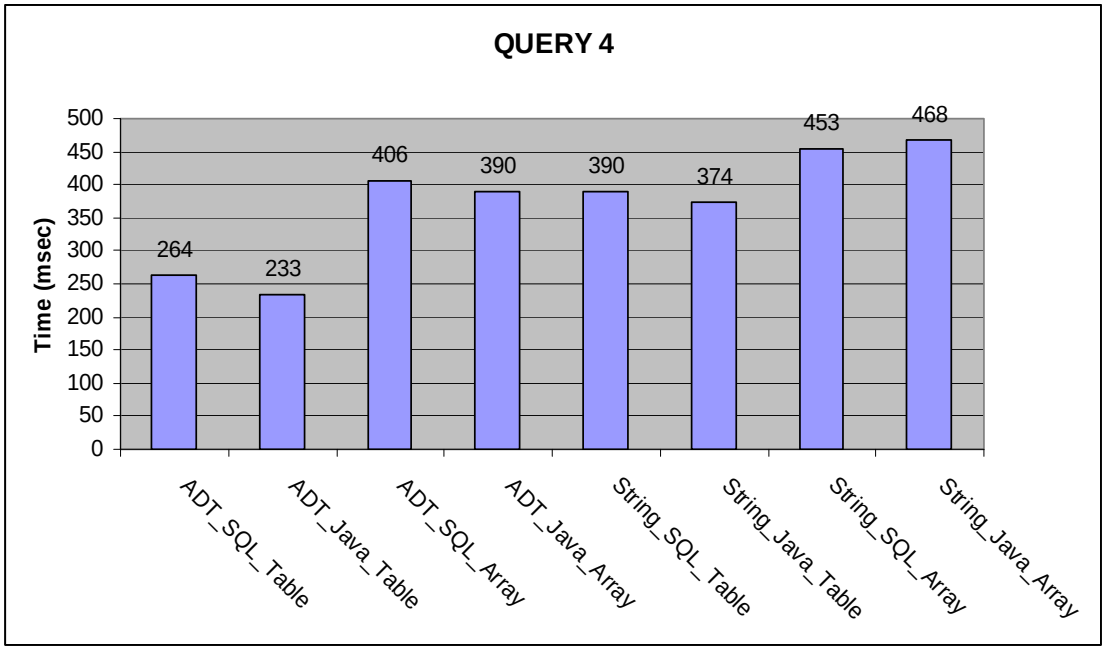


Figure 5.35: Query4 times for different implementation methods

Query5 first rolls back the two level-nested ‘MANAGER’ bitemporal attribute. Then, it returns the manager values along with its associated timestamps between the given valid time ranges. In terms of BTA’s, BTA_String performs better than the BTA_ADT. SofBTA_Array with BTA_String performs much better than the other combinations for this query. Figure 5.36 is the result of query times for query5.

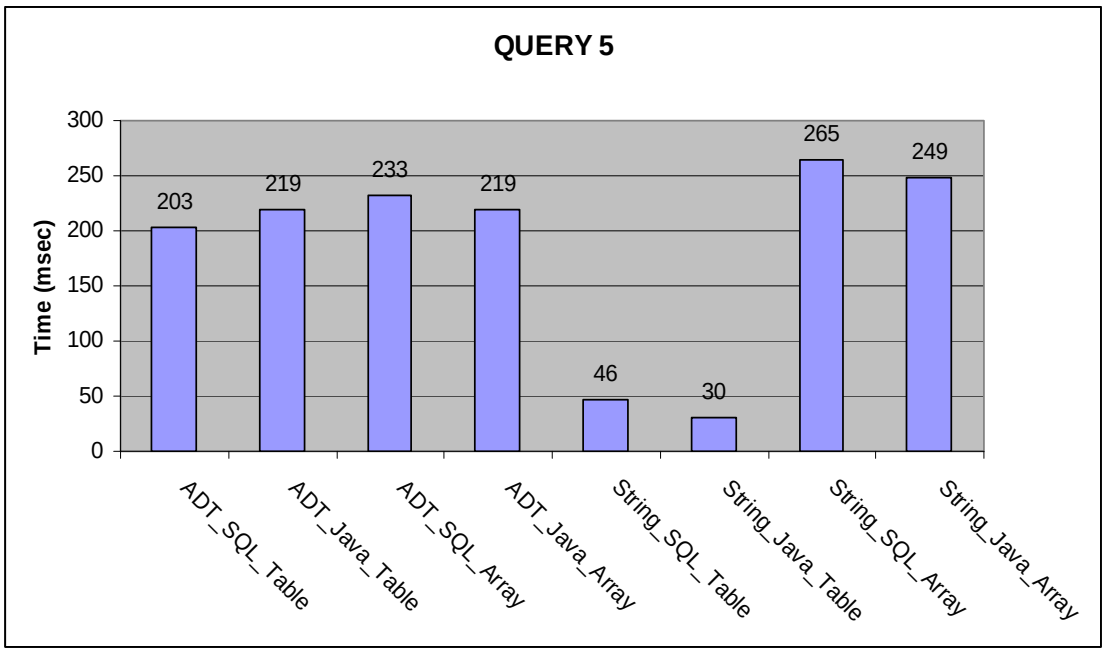


Figure 5.36: Query5 times for different implementation methods

Query6 first rolls back the two level-nested ‘MANAGER’ bitemporal attribute. Then, it goes through manager history of employees and returns the manager names between the given valid time ranges. BTA_ADT slightly outperforms the BTA_String. SofBTA_Array is better than SofBTA_Table. Query6 run times are depicted in Figure 5.37.

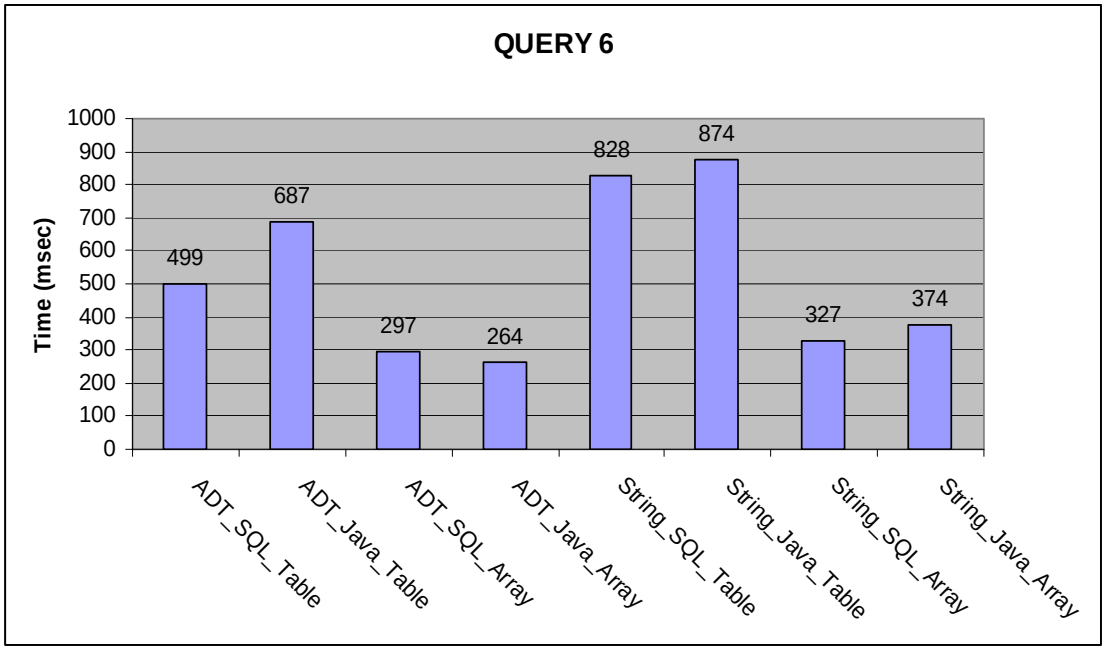


Figure 5.37: Query6 times for different implementation methods

There is not much difference query 3 in the performance of query 6 since the multiple attributes are retrieved from a single qualifying tuple from a rollbacked attribute. For all type of queries, BTA_ADT and BTA_String defined with SQL or Java perform the same.

After these statistics were collected, 100,000 new tuples were inserted into all eight tables. Figure 5.38 shows the insertion run times.

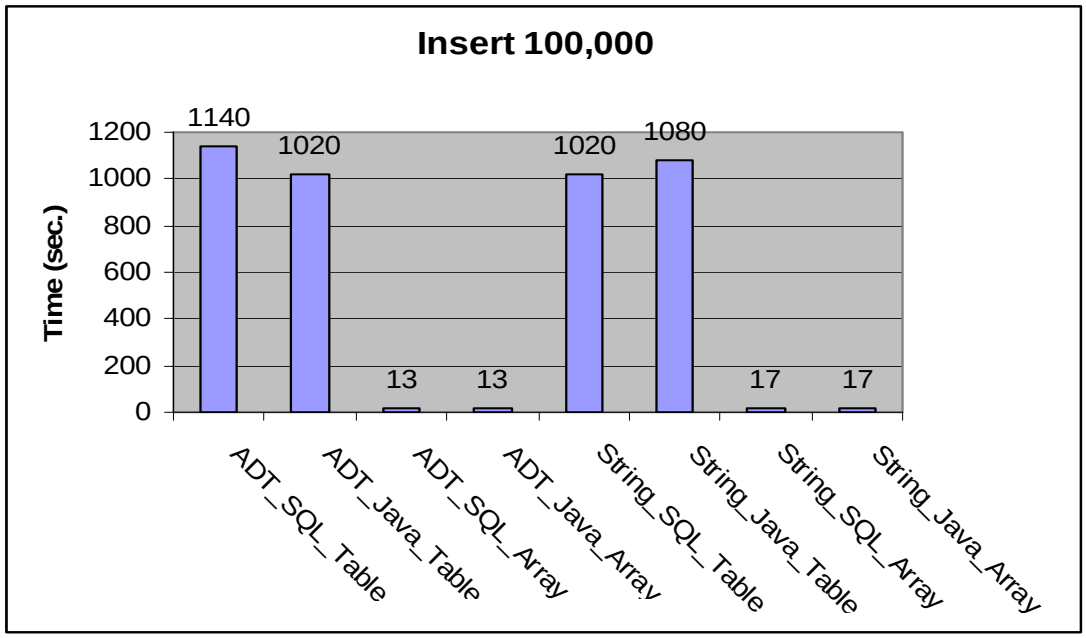


Figure 5.38: Insert times for 100,000 additional tuples.

5.5 Bitemporal Database Design

In this implementation, we have an entity relation, EMP, and the relationship relation, DEPARTMENT. EMP relation has five attributes where three of them are bitemporal. DEPARTMENT relation has three attributes where two of them are bitemporal. Table 5.5 and Table 5.6 depict EMP and DEPARTMENT tables, respectively. The implementation was done with BTA_ADT_SQL and BTA_String_SQL bitemporal atom types on nested tables. Table definitions are listed in appendix A.4 for EMP and DEPARTMENT relations.

Table 5.5: A nested bitemporal EMP relation.

EMP #	NAME	**ADDRESS-H**	BIRTH - DATE	**DEPT_ID-H**	**SALARY-H**
		ADDRESS		DEPT_ID	SALARY

Table 5.6: A nested bitemporal DEPARTMENT relation.

DEPT_ID	**BUDGET-H**	**MANAGER-H**
	BUDGET	MANAGER

We have applied the data generation method that we used for EMPLOYEE relation in section 5.4.3 for populating the EMP relation. For the DEPARTMENT relation, there are 30 departments from DEPARTMENT_ID1 through DEPARTMENT_ID30. Department's initial budget value was assigned randomly using a normal distribution where mean = 300,000 and standard deviation = 100,000 for each tuple. Every department had a 10% budget increase four times. Each department was assigned to one manager at a time. Manager numbers were chosen randomly by using a normal distribution where mean = 15 and standard deviation = 4.5. Departments changed their managers 3 times on average.

DEPARTMENT table contains 30 tuples one for each department, 60 set of bitemporal atom tables (SofBTA_Table), and 230 bitemporal atoms (BTA_ADT_SQL/ BTA_String_SQL). The same update operations that were applied to EMPLOYEE table in section 5.4.4 also performed to EMP table. For the update operations, run time results were similar.

The main goal of this implementation is to show that the nested bitemporal relational model allows the formulation of useful queries by joining two bitemporal nested tables. To demonstrate the NBRM functionality, we illustrated this point with the two set of queries. Queries were run, and the required time was measured over two different databases.

The first group of queries, (QA, QB and QC) used bitemporal context, the second group of queries, (QD and QE) used rollback context to test the 'AS_OF' clause. QB and QE queries are on one relation and others require joining two relations. Figures 5.39(a-c) through 5.44(a-c) shows how these queries were written with BtSQL and SQL codes.

Query A: What are the current budget of departments whose employees' current salary is more than 70K?

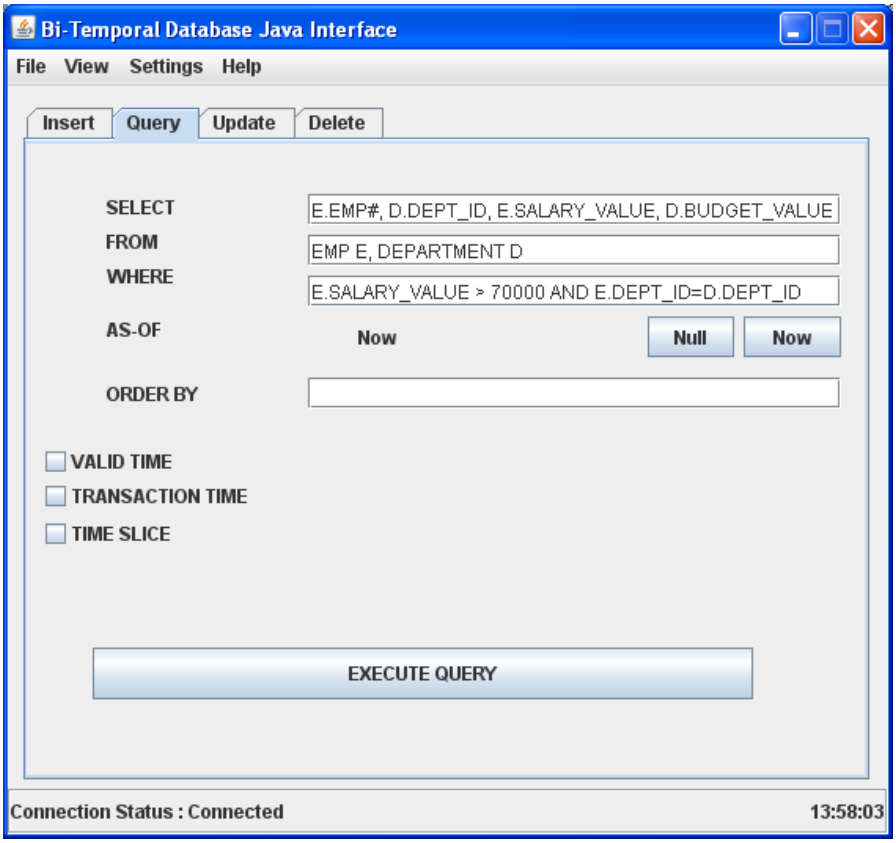


Figure 5.39-a: QueryA with BtSQL.

```
SELECT DECODE(ROW NUMBER() OVER (PARTITION BY E.EMP# ORDER BY
E.EMP#),1,E.EMP#,NULL) AS EMP#, D.DEPT_ID,
SUBSTR(SAL.BTA_TYPE, 45) AS SALARY_VALUE,
SUBSTR(BUD.BTA_TYPE, 45) AS BUDGET_VALUE
FROM EMP E, DEPARTMENT D,
      TABLE(E.DEPT_ID) DEP,
      TABLE(E.SALARY) SAL,
      TABLE(D.BUDGET) BUD
WHERE  SUBSTR(DEP.BTA_TYPE, 45) = D.DEPT_ID
AND SUBSTR(SAL.BTA_TYPE, 45) > 70000
AND SUBSTR(SAL.BTA_TYPE, 34,10) = '09.09.9999'
AND SUBSTR(BUD.BTA_TYPE, 34,10) = '09.09.9999'
AND SUBSTR(DEP.BTA_TYPE, 34,10) = '09.09.9999'
```

Figure 5.39-b: QueryA with BTA_String_SQL type.

```
SELECT DECODE(ROW NUMBER() OVER (PARTITION BY E.EMP# ORDER BY
E.EMP#),1,E.EMP#,NULL) AS EMP#, D.DEPT_ID,
SAL.VALUE_PART AS SALARY,
BUD.VALUE_PART AS BUDGET
FROM EMP E, DEPARTMENT D,
      TABLE(E.DEPT_ID) DEP,
      TABLE(E.SALARY) SAL,
      TABLE(D.BUDGET) BUD
WHERE  DEP.VALUE_PART = D.DEPT_ID  AND SAL.VALUE_PART > 70000
AND SAL.VALID_TIME_UPPER_BOUND = '09.09.9999'
AND BUD.VALID_TIME_UPPER_BOUND= '09.09.9999'
AND DEP.VALID_TIME_UPPER_BOUND= '09.09.9999'
```

Figure 5.39-c: QueryA with BTA_ADT_SQL type.

Query B: List the employee number of current employees and the history of departments they work for.

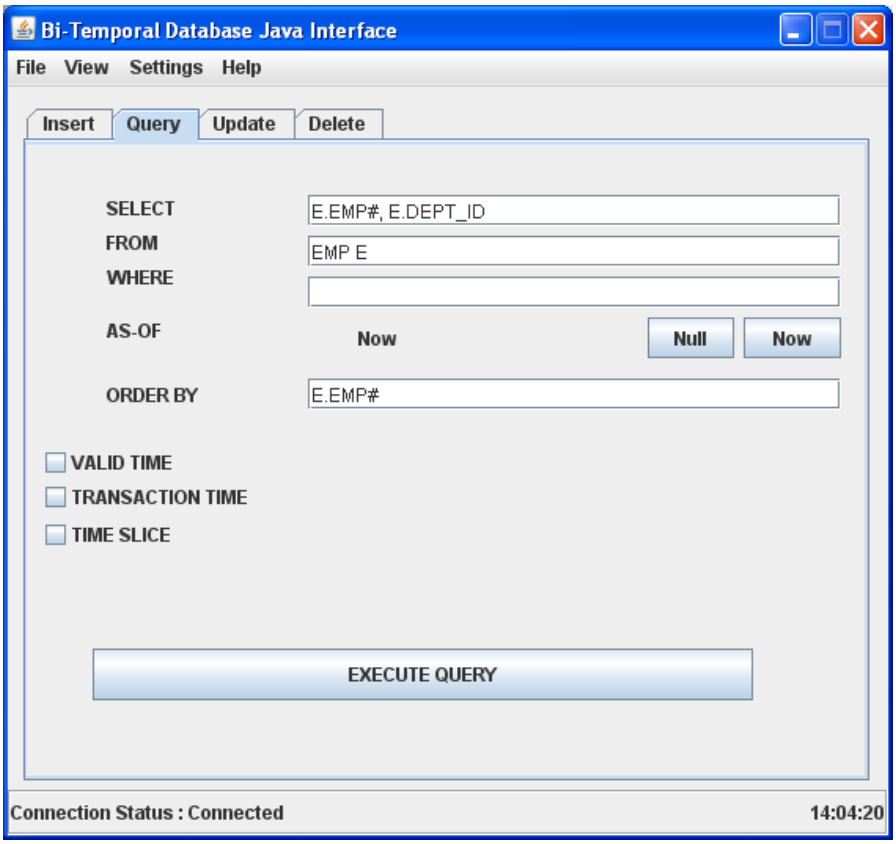


Figure 5.40-a: QueryB with BtSQL.

```
SELECT E.EMP# AS EMP#,
SUBSTR(DEP.BTA_TYPE, 45) AS DEPT_ID,
SUBSTR(DEP.BTA_TYPE, 23,10) AS DEPT_LB,
SUBSTR(DEP.BTA_TYPE, 34,10) AS DEPT_UB
FROM EMP E,TABLE(E.DEPT_ID) DEP
WHERE E.EMP# in (SELECT E.EMP#
                 FROM EMP EM,
                   TABLE(E.DEPT_ID) DP
                 WHERE SUBSTR(DEP.BTA_TYPE, 34,10)='09.09.9999')
ORDER BY E.EMP#, SUBSTR(DEP.BTA_TYPE, 22,11)
```

Figure 5.40-b: QueryB, with BTA_String_SQL type.

```
SELECT E.EMP# AS EMP#, DEP.VALUE PART AS DEPT VALUE,
       DEP.VALID_TIME_LOWER_BOUND AS DEPT_LB,
       DEP.VALID_TIME_UPPER_BOUND AS DEPT_UB
FROM EMP E, TABLE(E.DEPT_ID) DEP
WHERE E.EMP# in (SELECT E.EMP#
                 FROM EMP EM,
                      TABLE(E.DEPT_ID) DP
                 WHERE DP.VALID_TIME_UPPER_BOUND='09.09.9999')
ORDER BY E.EMP#, DEP.VALID TIME LOWER BOUND
```

Figure 5.40-c: QueryB, with BTA_ADT_SQL type.

Query C: What are the names of employees whose salary was ever larger than the budget of their departments at any time?

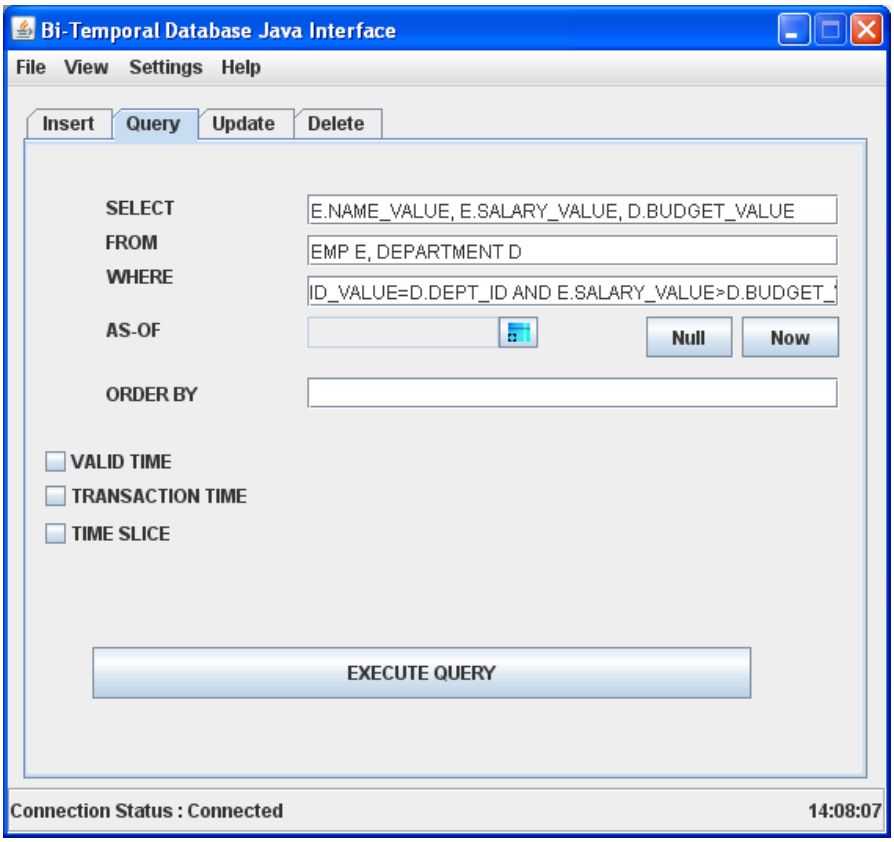


Figure 5.41-a: QueryC with BtSQL.

```
SELECT DECODE(ROW NUMBER() OVER (PARTITION BY TABLE A.EMP# ORDER
BY TABLE_A.EMP#),1, TABLE_A.EMP#,NULL) AS EMP#,
TABLE_A.SALARY_VALUE AS SALARY,
TABLE_A.DEPARTMENT_VALUE AS DEPARTMENT,
SUBSTR(BUD.BTA_TYPE, 45) AS DEPARTMENT_BUDGET,
       CASE WHEN (TABLE_A.STARTTIME> SUBSTR(BUD.BTA_TYPE, 23,10) )
            THEN TABLE_A.STARTTIME
            ELSE SUBSTR(BUD.BTA_TYPE, 23,10)
       END AS STARTTIME,
       CASE WHEN (TABLE_A.ENDTIME< SUBSTR(BUD.BTA_TYPE, 34,10))
            THEN TABLE_A.ENDTIME
            ELSE SUBSTR(BUD.BTA_TYPE, 34,10)
       END AS ENDTIME
FROM
(SELECT E.EMP#,
    TO_NUMBER(TO_CHAR(SUBSTR( A.BTA_TYPE, 45))) AS SALARY_VALUE,
    TO_CHAR(SUBSTR(B.BTA_TYPE, 45)) AS DEPARTMENT_VALUE,
    CASE WHEN (SUBSTR(A.BTA_TYPE, 23,10) > SUBSTR(B.BTA_TYPE,
23,10))
             THEN  SUBSTR(A.BTA_TYPE, 23,10)
             ELSE  SUBSTR(B.BTA_TYPE, 23,10)
    END AS STARTTIME,
    CASE WHEN (SUBSTR(A.BTA_TYPE, 34,10) < SUBSTR(B.BTA_TYPE,
34,10))
             THEN  SUBSTR(A.BTA_TYPE, 34,10)
             ELSE  SUBSTR( B.BTA_TYPE, 34,10)
    END AS ENDTIME
FROM EMP E, TABLE(E.SALARY)(+) A, TABLE(E.DEPT_ID)(+) B
WHERE
(
(TO_DATE(SUBSTR(A.BTA_TYPE, 23,10),'MM.DD.YYYY') BETWEEN
(TO_DATE(SUBSTR(B.BTA_TYPE , 23,10),'MM.DD.YYYY')+1) AND
(TO_DATE(SUBSTR(B.BTA_TYPE, 34,10),'MM.DD.YYYY')-1)
)
OR
(TO_DATE(SUBSTR(A.BTA_TYPE, 34,10) ,'MM.DD.YYYY') BETWEEN
(TO_DATE(SUBSTR(B.BTA_TYPE, 23,10),'MM.DD.YYYY ')+1) AND
(TO_DATE(SUBSTR(B.BTA_TYPE, 34,10),'MM.DD.YYYY')-1))
OR
(
TO_DATE(SUBSTR(A.BTA_TYPE, 23,10),'MM.DD.YYYY') <=
TO_DATE(SUBSTR(B.BTA_TYPE, 23,10),'MM.DD.YYYY')
AND
TO_DATE(SUBSTR( B.BTA_TYPE, 34,10),'MM.DD.YYYY') <=
TO_DATE(SUBSTR(A.BTA_TYPE, 34,10),'MM.DD.YYYY'))
)
)TABLE_A, DEPARTMENT D,TABLE(D.BUDGET)(+) BUD
WHERE TABLE_A.DEPARTMENT_VALUE=D.DEPT_ID
AND TABLE_A.SALARY_VALUE> TO_NUMBER(SUBSTR(BUD.BTA_TYPE, 45))
AND
(
(TO_DATE(TABLE_A.STARTTIME,'MM.DD.YYYY') BETWEEN
(TO_DATE(SUBSTR(BUD.BTA_TYPE, 23,10),'MM.DD.YYYY')+1) AND
(TO_DATE(SUBSTR(BUD.BTA_TYPE, 34,10),'MM.DD.YYYY')-1))
OR
(TO_DATE(TABLE_A.ENDTIME,'MM.DD.YYYY') BETWEEN
(TO_DATE(SUBSTR(BUD.BTA_TYPE, 23,10),'MM.DD.YYYY')+1) AND
(TO_DATE(SUBSTR(BUD.BTA_TYPE, 34,10),'MM.DD.YYYY')-1))
OR
(TO_DATE(TABLE_A.STARTTIME,' MM.DD.YYYY') <=
TO_DATE(SUBSTR(BUD.BTA_TYPE, 23,10),'MM.DD.YYYY') AND
TO_DATE(SUBSTR(BUD.BTA_TYPE, 34,10),'MM.DD.YYYY') <=
```

```
TO DATE(TABLE A.ENDTIME,'MM.DD.YYYY'))
)
```

Figure 5.41-b: QueryC with BTA_String_SQL type.

```
SELECT DECODE(ROW NUMBER() OVER (PARTITION BY TABLE A.EMP# ORDER BY
TABLE_A.EMP#),1,TABLE_A.EMP#, NULL) AS EMP#, TABLE_A.SALARY_VALUE
AS SALARY,
TABLE_A.DEPARTMENT_VALUE AS DEPARTMENT,
B.VALUE_PART AS DEPARTMENT_BUDGET,
       CASE WHEN (TABLE_A.STARTTIME>B.VALID_TIME_LOWER_BOUND )
            THEN TABLE_A.STARTTIME
            ELSE B.VALID_TIME_LOWER_BOUND
       END AS STARTTIME,
       CASE WHEN (TABLE_A.ENDTIME<B.VALID_TIME_UPPER_BOUND)
            THEN TABLE_A.ENDTIME
            ELSE B.VALID_TIME_UPPER_BOUND
       END AS ENDTIME
FROM (SELECT E.EMP#,
    TO_CHAR(A.VALUE_PART) AS SALARY_VALUE,
    TO_CHAR(B.VALUE_PART) AS DEPARTMENT_VALUE,
    CASE WHEN (A.VALID_TIME_LOWER_BOUND > B.VALID_TIME_LOWER_BOUND)
            THEN A.VALID_TIME_LOWER_BOUND
            ELSE B.VALID_TIME_LOWER_BOUND
       END AS STARTTIME,
     CASE WHEN (A.VALID_TIME_UPPER_BOUND<B.VALID_TIME_UPPER_BOUND)
            THEN A.VALID_TIME_UPPER_BOUND
            ELSE B.VALID_TIME_UPPER_BOUND
       END AS ENDTIME
FROM EMP E,TABLE(E.SALARY)(+) A, TABLE(E.DEPT_ID)(+) B
WHERE  (
(A.VALID_TIME_LOWER_BOUND BETWEEN B.VALID_TIME_LOWER_BOUND+1 AND
 B.VALID_TIME_UPPER_BOUND-1)
OR
(A.VALID_TIME_UPPER_BOUND BETWEEN B.VALID_TIME_LOWER_BOUND+1 AND
 B.VALID_TIME_UPPER_BOUND-1)
OR
(A.VALID_TIME_LOWER_BOUND <= B.VALID_TIME_LOWER_BOUND AND
 B.VALID_TIME_UPPER_BOUND <= A.VALID_TIME_UPPER_BOUND)
)
) TABLE_A, DEPARTMENT D, TABLE(D.BUDGET)(+) B
WHERE TABLE_A.DEPARTMENT_VALUE=D.DEPT_ID
AND TABLE_A.SALARY_VALUE> B.VALUE_PART
AND
(
(TABLE_A.STARTTIME BETWEEN B.VALID_TIME_LOWER_BOUND+1 AND
                           B.VALID_TIME_UPPER_BOUND-1)
OR
(TABLE_A.ENDTIME BETWEEN B.VALID_TIME_LOWER_BOUND+1 AND
                         B.VALID_TIME_UPPER_BOUND-1)
OR
(TABLE_A.STARTTIME <= B.VALID_TIME_LOWER_BOUND AND
                      B.VALID_TIME_UPPER_BOUND <= TABLE_A.ENDTIME)
)
```

Figure 5.41-c: QueryC with BTA_ADT_SQL type.

Query D: List the employee number, department history they work for and department's budget history that is stored in the database during the time range ['01.01.2003', '01.01.2004'].

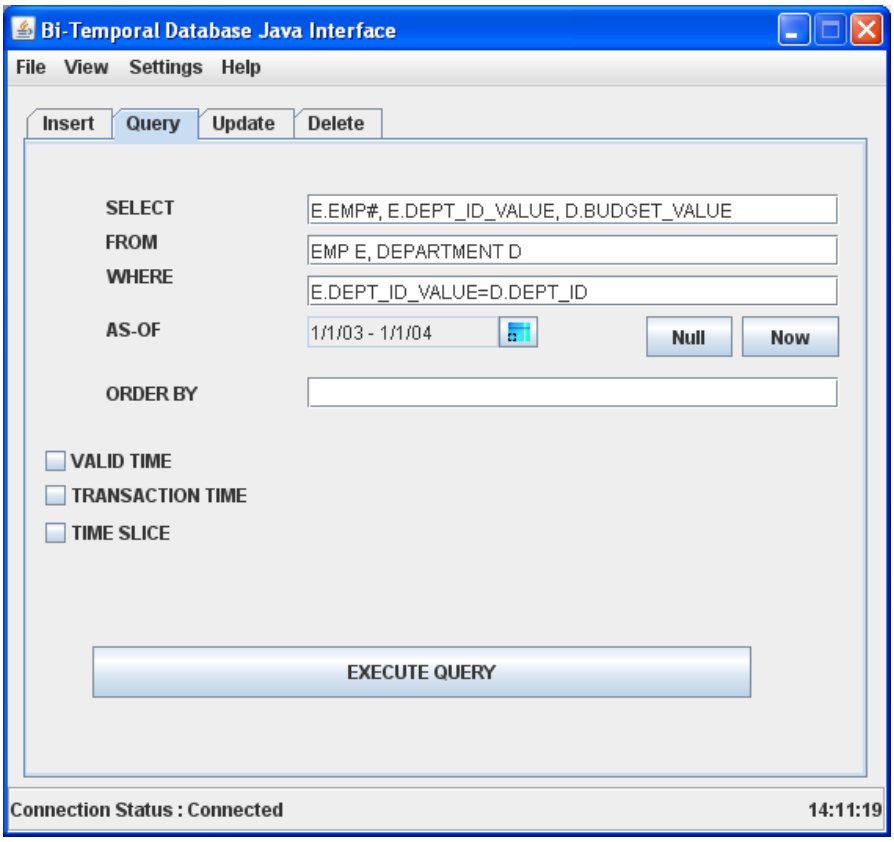


Figure 5.42-a: QueryD with BtSQL.

```
SELECT DECODE(ROW NUMBER() OVER (PARTITION BY E.EMP# ORDER BY
E.EMP#),1,E.EMP#,NULL) AS EMP#,
SUBSTR(DEP.BTA_TYPE, 45) AS DEPT_ID,
SUBSTR(BUD.BTA_TYPE, 45) AS BUDGET
FROM EMP E, DEPARTMENT D,
      TABLE(E.DEPT_ID) DEP,
      TABLE(D.BUDGET) BUD
WHERE SUBSTR(DEP.BTA_TYPE, 45) = D.DEPT_ID AND
MY PKG.AS OF(DEP.BTA TYPE, '01.01.2003','01.01.2004')=1
```

Figure 5.42-b: QueryD with BTA_String_SQL type.

```
SELECT DECODE(ROW NUMBER() OVER (PARTITION BY E.EMP# ORDER BY
E.EMP#),1,E.EMP#,NULL) AS EMP#,
DEP.VALUE_PART  AS DEPT,
BUD.VALUE_PART AS BUDGET
FROM EMP E, DEPARTMENT D,
      TABLE(E.DEPT_ID) DEP,
      TABLE(D.BUDGET) BUD
WHERE DEP.VALUE_PART = D.DEPT_ID AND MY_PKG.AS_OF(VALUE(DEP),
'01.01.2003','01.01.2004')=1
```

Figure 5.42-c: QueryD with BTA_ADT_SQL type.

Query E: As of 01.01.2005, who was working in the DEPARTMENT_ID2?

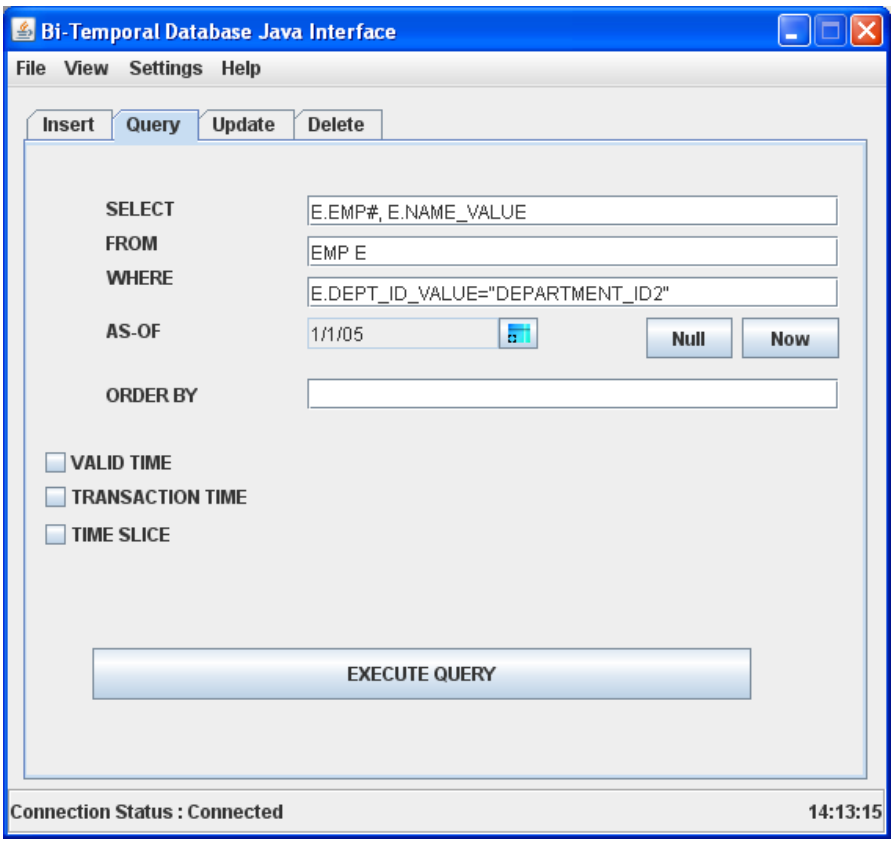


Figure 5.43-a: QueryE with BtSQL.

```
SELECT DECODE(ROW NUMBER() OVER (PARTITION BY E.EMP# ORDER BY
E.EMP#),1,E.EMP#,NULL) AS EMP#,
E.NAME AS NAME,
SUBSTR(DEP.BTA_TYPE, 45) AS DEPT_ID,
SUBSTR(DEP.BTA_TYPE, 23,10) AS VT_LOWERBOUND,
SUBSTR(DEP.BTA_TYPE, 34,10)  AS VT_UPPERBOUND
FROM EMP E,
      TABLE(E.DEPT_ID) DEP
WHERE SUBSTR(DEP.BTA_TYPE, 45) = 'DEPARTMENT_ID2' AND
MY_PKG.AS_OF(DEP.BTA_TYPE, '01.01.2005')=1
```

Figure 5.43-b: QueryE with BTA_String_SQL type.

```
SELECT DECODE(ROW NUMBER() OVER (PARTITION BY E.EMP# ORDER BY
E.EMP#),1,E.EMP#,NULL) AS EMP#,
E.NAME AS NAME, DEP.VALUE_PART AS DEPARTMENT
FROM EMP E,
     TABLE(E.DEPT_ID) DEP
WHERE DEP.VALUE_PART = 'DEPARTMENT_ID2' AND
MY_PKG.AS_OF(VALUE(DEP), '01.01.2005')=1
```

Figure 5.43-c: QueryE with BTA_ADT_SQL type.

QueryA joins EMP and DEPARTMENT tables on DEPT_ID and for the salary values > 70K. Then, projects the employee number, current salary and budget values along with their lower bound timestamps. BTA_ADT_SQL performs slightly better than the BTA_String_SQL type in terms of run time. QueryA run times are depicted in Figure 5.44. QueryB lists each employee's department history using EMP relation. As figure 5.44 indicates, BTA_ADT_SQL performs slightly better than the BTA_String_SQL type.

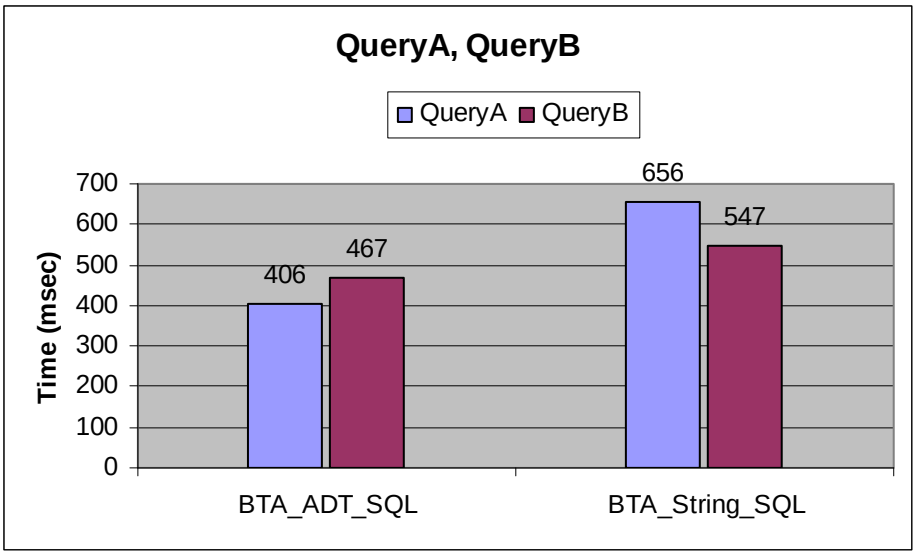


Figure 5.44: QueryA and QueryB times for different implementation methods.

QueryC first specifies the intersection time between salary and department bitemporal attributes on the EMP table. Then, it specifies the intersection time with budget bitemporal attribute by joining EMP and DEPARTMENT relations on the condition salary value greater than the budget value. Finally, it projects employee number, employees' salary, department and budget values for these conditions along with intersect lower and upper bounds dates. BTA_ADT_SQL performs much better than the BTA_String_SQL type for this query type. QueryC run times are depicted in Figure 5.45.

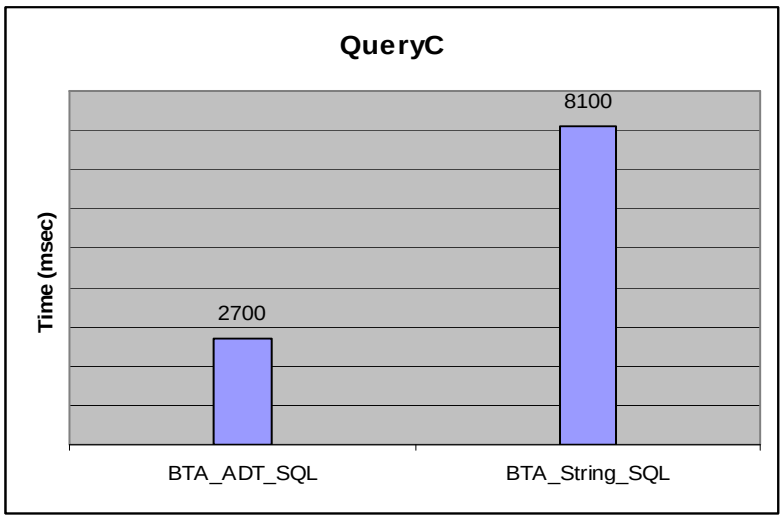


Figure 5.45: QueryC times for different implementation methods.

QueryD first joins EMP and DEPARTMENT relations on DEPT_ID and roll backed 'DEPT_ID' attribute. Then, it projects the employee number of employees and employee's affiliated department history and budget history of that department for the given time interval. In this query type, BTA_ADT_SQL is better than the BTA_String_SQL type. Running times for queryD is presented in Figure 5.46. While queryD is a 'time interval' type rollback context query, queryE is a 'time point' type. QueryE first rolls back the 'DEPT_ID' attribute. Then, it returns the employee number and name of employees who used to work at given department_id for the given time point. A BTA_ADT_SQL and BTA_String_SQL implementation method performs almost the same for this query. Figure 5.46 is the result of query times for queryE.

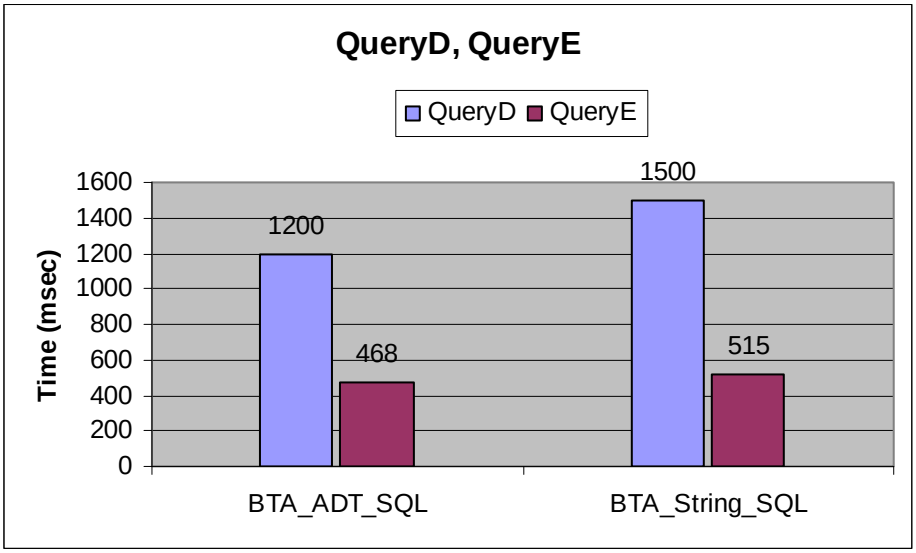


Figure 5.46: QueryD and Query times for different implementation methods.

Both BTA_ADT_SQL and BTA_String_SQL time components as well as value parts are extracted and used in the expression in bitemporal context and in rollback context, respectively. This implementation shows that nested bitemporal relational model can successfully be utilized for more than one nested bitemporal relations.

5.6 Conclusion

This chapter has presented how a proposed nested bitemporal relational model can be built on a commercially available object-relational database system. BtSQL, a GUI bitemporal preprocessor, translates temporal statements into standard SQL statements. We have implemented successfully the time slice and rollback bitemporal relational algebraic operations defined in chapter 4. We have constructed eight nested bitemporal relational databases with different type of bitemporal atoms representations. Different types of queries and updates are considered to measure the performance of the proposed implementation methodologies. For each type of update and retrieval queries, the resulting run times is given in a graph as well as a brief explanation to justify why this particular update and retrieval query has been chosen. This implementation focused mainly on the feasibility of nested bitemporal relational databases; therefore, specific tuning recommendations were not deeply investigated. Instead, implementation of an attribute time-stamping approach on a conventionally available database and bitemporal relational operations were given priority. It has been shown that the proposed model is used successfully with bitemporal, current and historical context. Therefore, the full expressive power of NBRM presented in this thesis has been demonstrated. The experimental results show that implementation of the NBRM on an object-relational database is quite attainable.

The conclusions of these tests can be summarized as follows:

- Since the bitemporal atom's five components are stored in BTA_String and in BTA_ADT, extracting and manipulating any one of them in the expressions is possible.
- BTA_ADT and BTA_String perform the same whether defined with SQL or in Java. Either one could be used, depending on the application's needs. Although this test is only on text data, Java might perform better for binary types of data, such as picture, video, or voice.
- While BTA_ADT bitemporal atom is better with updates, BTA_String is better with querying.

- SofBTA_Table or SofBTA_Array can be used to keep the history of an object being modeled. However, a temporal attribute is a function of time, and the size of its value is unfixed. The use of SofBTA_Array requires increasing table size manually, as needed, whereas the size of the SofBTA_Table increases dynamically.
- SofBTA_Table are better with updates; however, SofBTA_Array type outperforms the SofBTA_Table type in terms of queries as measured in run time.
- The bitemporal relational algebraic operators', time slice and rollback, implemented and tests have shown that both new operators functionally performed well.
- It is shown that support for bitemporal atom types and query language is accommodated for both implementations.
- Queries listed in this thesis might be used as a basis comparing the performance of different bitemporal database management systems.
- We have successfully managed to hide tedious bitemporal query specifications from the user with BtSQL.

Table 5.7 summarizes insert, update and query times of the implementation tests results with 5 being the fastest and 1 being the slower. Each column is graded individually.

Table 5.7: Implementation test results

Table Name	**Insert**	**Update**	**Q1**	**Q2**	**Q3**	**Q4**	**Q5**	**Q6**
ADT_SQL_Table	3	4	4	1	4	5	4	4
ADT_Java_Table	3	4	4	1	4	5	4	4
ADT_SQL_Array	5	5	4	3	5	4	4	5
ADT_Java_Array	5	5	4	3	5	4	4	5
String_SQL_Table	3	2	4	5	4	4	5	4
String_Java_Table	3	2	4	5	4	4	5	4
String_SQL_Array	5	2	4	2	5	3	3	5
String_Java_Array	5	2	4	2	5	3	3	5

1- Slower, 2- Slow, 3- Fast, 4- Faster, 5- Fastest

Chapter 6

6. CONCLUSIONS

6.1 Summary and Conclusions

In general, the semantics of time domain are well understood in the realms of temporal databases, artificial intelligence, and especially in temporal reasoning. Valid time and transaction time are used to represent an object's history in a database. Moreover, valid time and transaction time are orthogonal. Time stamps are attached to the tuples (temporally ungrouped) or attributes (temporally grouped) where 1NF and N1NF (nested) relations are used, respectively. This thesis proposes a temporally grouped bitemporal relational data model using nested relations.

We propose to use nested relations to manage temporal data. We refer to the data model as a *Nested Bitemporal Data Model*, and this model includes both valid time and transaction time. Valid time and transaction time are attached to attributes. Each attribute value is either an atomic value or a bitemporal atom, which consists of five components: a value, its validity period and the time this data is recorded in the database.

We define the concept of a *context* for a bitemporal database. The database can be viewed within bitemporal, current or a historical context. The bitemporal context is for auditing purposes, whereas the historical context is for querying past states of a bitemporal database. The current context is for querying the snapshot state of a bitemporal database. Retroactive and postactive changes are also available in the bitemporal context.

We define a Nested Bitemporal Relational Algebra for the proposed model. Operations are defined at the attribute level, thus keeping the structure of a tuple intact.

We define a rollback operator that allows access to a database state at any time point or time interval. We extend SQL with bitemporal querying constructs and develop a GUI user interface. We also define a nested bitemporal relational calculus for the proposed model. We implement as a test bed a nested bitemporal database on a commercially available object-relational database system. We define nested bitemporal relations with string implementation and abstract data types. Bitemporal atoms are defined in SQL3 and Java. Bitemporal attributes are stored in different types of collection table types. We use the test bed to demonstrate the feasibility of our proposed solutions and to evaluate their performance.

Object-relational database systems have richer semantics and data types – such as abstract data types – than relational DBMSs. Moreover, they have the capability to define temporal semantics through these abstract data types. The standard query language SQL3 includes object-relational features that can serve as built-in temporal semantics, and which therefore provide a robust platform for implementing temporal databases. Commercial object-relational database systems implement some features of SQL3 and provide readily available platforms to test the concepts developed in this thesis.

We believe that temporal data management will eventually become an integral part of many application domains, and knowledge discovery systems. Our bitemporal database system can be used as a test bed to demonstrate the feasibility of bitemporal databases in many application domains, since it supports the essential constructs needed in bitemporal databases.

6.2 Future Research Directions

XML (Extensible Markup Language) has emerged as the new standard for exchanging structured data over the Internet. XML's main characteristic is that it is naturally nested to arbitrary depths. Additionally, XQuery can express powerful queries. XML provides support for attribute time-stamped document structures, to which our work is similar. Temporal and bitemporal atomic data types – and, consequently,

temporal databases – can be naturally represented in XML, using attribute time-stamping. Therefore, a promising direction of research is to combine nested bitemporal relational model in XML.

Spatial databases have been an active area of research over the last two decades, in parallel with temporal databases. Another possible research direction would be the incorporation of spatial data into nested bitemporal relational model, which would effectively create a spatio-bitemporal database. Such spatio-bitemporal databases would have built-in support for both space and time(s), and consequently would enable new database applications.

Multimedia databases provide features that allow users to store and query different types of multimedia information, such as images, video clips, audio clips, and documents. Multimedia is one of the fastest growing areas of business today – especially over the Internet. Its objects have temporal relationships that must be maintained in their presentation. Our current work can be further extended for multimedia databases. The most straightforward direction is to implement the bitemporal behavior of multimedia objects.

Data warehouses store historical data, and therefore could clearly benefit from the research on temporal databases. Data warehouses need to incorporate bitemporal data representation. NBRM can be used in conjunction with a data warehouse, and it also could feed data to a data warehouse.

Comparison of nested bitemporal relational model and tuple time stamped bitemporal models is another future research direction that we plan to take. We will use same updates and run same queries on the same data to carry out a performance evaluation of our proposed model against tuple timestamped bitemporal models.

Temporal functional dependencies and integrity constraints have been studied by several researchers ([TG89, Ta04, JSS94]). Another research direction would be the

study of functional dependencies and integrity constraints for the nested bitemporal relational model. Design of bitemporal databases is also another research direction that could be pursued.

APPENDIX A

A NESTED BITEMPORAL RELATIONAL DATABASE

IMPLEMENTATION STEPS ON ORACLE 9i DBMS

A.1 - Bitemporal Data Types

ABSTRACT DATA TYPE USING PL/SQL:

An object type can be used like any other type in further declarations of object-types or table-types. The BTA_ADT_SQL object type is used to define *bitemporal atom*, which has TRAN_TIME_LOWER_BOUND, TRAN_TIME_UPPER_BOUND, VALID_TIME_LOWER_BOUND, VALID_TIME_UPPER_BOUND and an atomic value of type VARCHAR.

```
CREATE TYPE BTA_ADT_SQL AS OBJECT (
      TRAN_TIME_LOWER_BOUND   DATE,
      TRAN_TIME_UPPER_BOUND   DATE,
      VALID_TIME_LOWER_BOUND  DATE,
      VALID_TIME_UPPER_BOUND  DATE,
      VALUE                VARCHAR2(50)
);
```

ABSTRACT DATA TYPE USING HOST LANGUAGE JAVA: BTA_ADT__Java types are created in Java and loaded into database.

```
public class BTA_ADT_Java {
      public Date TRAN_TIME_LOWER_BOUND;
      public Date TRAN_TIME_UPPER_BOUND;
      public Date VALID_TIME_LOWER_BOUND;
      public Date VALID_TIME_UPPER_BOUND;
      public String VALUE;
}

loadjava -user thesis_oracle/password@orcd BTA_ADT_Java.class
```

```
CREATE OR REPLACE TYPE BTA_ADT_Java AS OBJECT
  EXTERNAL NAME 'BTA_ADT_Java' LANGUAGE JAVA
  USING SQLData (
       TRAN_TIME_LOWER_BOUND date external name 'TRAN_TIME_LOWER_BOUND',
       TRAN_TIME_UPPER_BOUND date external name 'TRAN_TIME_UPPER_BOUND',
       VALID_TIME_LOWER_BOUND date external name
'VALID_TIME_LOWER_BOUND',
       VALID_TIME_UPPER_BOUND date external name
'VALID_TIME_UPPER_BOUND',
       VALUE varchar2(50) external name 'VALUE'
)
```

A.2 - Object Table Types

OBJECT TABLE TYPES WITH NESTED TABLES: SALARY object table type is defined to make possible usage of SALARY bitemporal atoms in the SALARY nested bitemporal relation. Other object table types are defined similarly.

```
CREATE TYPE NAME        AS SofBTA_Table OF BTA_ADT_SQL;
CREATE TYPE ADDRESS     AS SofBTA_Table OF BTA_ADT_SQL;
CREATE TYPE DEPARTMENT  AS SofBTA_Table OF BTA_ADT_SQL;
CREATE TYPE MANAGER     AS SofBTA_Table OF BTA_ADT_SQL;
CREATE TYPE SALARY      AS SofBTA_Table OF BTA_ADT_SQL;
```

The TYPE_DEPT_MNG object type is used to define DEPARTMENT and MANAGER nested bitemporal relation.

```
CREATE TYPE TYPE_DEPT_MNG AS OBJECT (
      DEPARTMENT_HISTORYDEPARTMENT,
      MANAGER_HISTORY         MANAGER
);

CREATE TYPE DEPT_MNG AS SofBTA_Table OF TYPE_DEPT_MNG;
```

OBJECT TABLE TYPES WITH ARRAY OF TABLES:

```
CREATE TYPE NAME        AS SofBTA_Array(5) OF BTA_ADT_SQL;
CREATE TYPE ADDRESS     AS SofBTA_Array(10) OF BTA_ADT_SQL;
CREATE TYPE DEPARTMENT  AS SofBTA_Array(10) OF BTA_ADT_SQL;
CREATE TYPE MANAGER     AS SofBTA_Array(10) OF BTA_ADT_SQL;
CREATE TYPE SALARY      AS SofBTA_Array(20) OF BTA_ADT_SQL;

CREATE TYPE TYPE_DEPT_MNG AS OBJECT (
      DEPARTMENT_HISTORYDEPARTMENT,
      MANAGER_HISTORY   MANAGER
);

CREATE TYPE DEPT_MNG AS SofBTA_Array(10) OF TYPE_DEPT_MNG;
```

A.3 - Nested Bitemporal Table Creation

NESTED BITEMPORAL TABLE CREATION WITH NESTED TABLES: NAME, ADDRESS, DEPT_MNG and SALARY are nested bitemporal relations in the

EMPLOYEE table. A nested table type column requires a storage table where rows for all nested tables in the column are stored. Similarly, with a multilevel nested table collection of nested tables: the inner set of nested tables requires a storage table just as the outer set does. It must be specified one by appending a second nested-table storage clause [ORA01].

```
CREATE TABLE EMPLOYEE (
  EMP# NUMBER primary key,
  NAME NAME,
  ADDRESS  ADDRESS ,
  BIRTH_DATE DATE,
  DEPT_MNG DEPT_MNG,
  SALARY SALARY
)
SofBTA_Table NAME STORE AS NAME_TABLE,
SofBTA_Table ADDRESS STORE AS ADDRESS_TABLE,
SofBTA_Table DEPT_MNG STORE AS DEPT_MNG_TABLE
(
  SofBTA_Table MANAGER_HISTORY STORE AS MANAGER_TABLE,
  SofBTA_Table DEPARTMENT_HISTORY STORE AS DEPARTMENT_TABLE
),
SofBTA_Table SALARY STORE AS SALARY_TABLE
;
```

NESTED BITEMPORAL TABLE CREATION WITH ARRAY TABLES:

```
CREATE TABLE EMPLOYEE (
  EMP# NUMBER primary key,
  NAME NAME,
  ADDRESS  ADDRESS ,
  BIRTH_DATE DATE,
  DEPT_MNG DEPT_MNG,
  SALARY SALARY
);
```

A.4 – Nested Bitemporal Relational Model for Database Design

```
CREATE TABLE EMPLOYEE (
  SSN NUMBER primary key,
  NAME VARCHAR2(50),
  ADDRESS  ADDRESS ,
  BIRTH_DATE DATE,
  DEPT_ID DEPT_ID,
  SALARY SALARY
)
SofBTA_Table ADDRESS AS SofBTA_Table OF ADDRESS_TABLE,
SofBTA_Table DEPT_ID AS SofBTA_Table OF DEPT_ID_TABLE,
SofBTA_Table SALARY  AS SofBTA_Table OF SALARY_TABLE
;

CREATE TABLE DEPARTMENT (
  DEPT_ID VARCHAR2(20) primary key,
  BUDGET BUDGET,
  MANAGER MANAGER
)
SofBTA_Table BUDGET  AS SofBTA_Table OF BUDGET_TABLE,
SofBTA_Table MANAGER AS SofBTA_Table OF MANAGER_TABLE;
```

A.5 - Nested Bitemporal Relational Model with Time Points

```
CREATE TYPE BTA_ADT_TIME_POINT AS OBJECT (
      TRAN_TIME_POINT   DATE,
      VALID_TIME_POINT  DATE,
      VALUE             VARCHAR2(50)
);


CREATE TYPE NAME        AS SofBTA_Table OF BTA_ADT_TIME_POINT;
CREATE TYPE ADDRESS     AS SofBTA_Table OF BTA_ADT_TIME_POINT;
CREATE TYPE DEPARTMENT  AS SofBTA_Table OF BTA_ADT_TIME_POINT;
CREATE TYPE MANAGER     AS SofBTA_Table OF BTA_ADT_TIME_POINT;
CREATE TYPE SALARY      AS SofBTA_Table OF BTA_ADT_TIME_POINT;

CREATE TYPE TYPE_DEPT_MNG AS OBJECT (
      DEPARTMENT_HISTORYDEPARTMENT,
      MANAGER_HISTORY   MANAGER
);

CREATE TYPE DEPT_MNG AS SofBTA_Table OF TYPE_DEPT_MNG;
```

```
CREATE TABLE EMPLOYEE (
  EMP# NUMBER primary key,
  NAME NAME,
  ADDRESS ADDRESS,
  BIRTH_DATE DATE,
  DEPT_MNG DEPT_MNG,
  SALARY SALARY
)
SofBTA_Table NAME STORE AS NAME_TABLE,
SofBTA_Table ADDRESS STORE AS ADDRESS_TABLE,
SofBTA_Table DEPT_MNG STORE AS DEPT_MNG_TABLE
(
  SofBTA_Table MANAGER_HISTORY STORE AS MANAGER_TABLE,
  SofBTA_Table DEPARTMENT_HISTORY STORE AS DEPARTMENT_TABLE
),
SofBTA_Table SALARY STORE AS SALARY_TABLE
;
```

A.6 - Nested Historical Relational Model

Model defined in this thesis can be used as Historical Database by using only one time dimension, valid time.

```
CREATE TYPE TA_ADT_SQL AS OBJECT (
      VALID_TIME_LOWER_BOUND  DATE,
      VALID_TIME_UPPER_BOUND  DATE,
      VALUE                   VARCHAR2(50)
);


CREATE TYPE NAME        AS SofBTA_Table OF TA_ADT_SQL;
CREATE TYPE ADDRESS     AS SofBTA_Table OF TA_ADT_SQL;
CREATE TYPE DEPARTMENT  AS SofBTA_Table OF TA_ADT_SQL;
CREATE TYPE MANAGER     AS SofBTA_Table OF TA_ADT_SQL;
CREATE TYPE SALARY      AS SofBTA_Table OF TA_ADT_SQL;

CREATE TYPE TYPE_DEPT_MNG AS OBJECT (
      DEPARTMENT_HISTORYDEPARTMENT,
      MANAGER_HISTORY         MANAGER
);

CREATE TYPE DEPT_MNG AS SofBTA_Table OF TYPE_DEPT_MNG;
```

```
CREATE TABLE EMPLOYEE (
  EMP# NUMBER primary key,
  NAME NAME,
  ADDRESS ADDRESS,
  BIRTH_DATE DATE,
  DEPT_MNG DEPT_MNG,
  SALARY SALARY
)
SofBTA_Table NAME STORE AS NAME_TABLE,
SofBTA_Table ADDRESS STORE AS ADDRESS_TABLE,
SofBTA_Table DEPT_MNG STORE AS DEPT_MNG_TABLE,
(
  SofBTA_Table MANAGER_HISTORY STORE AS MANAGER_TABLE,
  SofBTA_Table DEPARTMENT_HISTORY STORE AS DEPARTMENT_TABLE
),
SofBTA_Table SALARY STORE AS SALARY_TABLE
;
```

A.7 - Nested Rollback Relational Model

```
CREATE TYPE ROLLBACK_TA_ADT_SQL AS OBJECT (
      TRAN_TIME_LOWER_BOUND DATE,
      TRAN_TIME_UPPER_BOUND DATE,
      VALUE       VARCHAR2(50)
);

CREATE TYPE NAME        AS SofBTA_Table OF ROLLBACK_TEMPORAL_VARCHAR;
CREATE TYPE ADDRESS     AS SofBTA_Table OF ROLLBACK_TEMPORAL_VARCHAR;
CREATE TYPE DEPARTMENT  AS SofBTA_Table OF ROLLBACK_TEMPORAL_VARCHAR;
CREATE TYPE MANAGER     AS SofBTA_Table OF ROLLBACK_TEMPORAL_VARCHAR;
CREATE TYPE SALARY      AS SofBTA_Table OF ROLLBACK_TEMPORAL_NUMBER;

CREATE TYPE TYPE_DEPT_MNG AS OBJECT (
      DEPARTMENT_HISTORYDEPARTMENT,
      MANAGER_HISTORY   MANAGER
);

CREATE TYPE DEPT_MNG AS SofBTA_Table OF TYPE_DEPT_MNG;


CREATE TABLE EMPLOYEE (
  EMP# NUMBER primary key,
  NAME NAME,
  ADDRESS ADDRESS,
  BIRTH_DATE DATE,
  DEPT_MNG DEPT_MNG,
  SALARY SALARY
)
SofBTA_Table NAME STORE AS NAME_TABLE,
SofBTA_Table ADDRESS STORE AS ADDRESS_TABLE,
SofBTA_Table DEPT_MNG STORE AS DEPT_MNG_TABLE
(
  SofBTA_Table MANAGER_HISTORY STORE AS MANAGER_TABLE,
  SofBTA_Table DEPARTMENT_HISTORY STORE AS DEPARTMENT_TABLE
),
SofBTA_Table SALARY STORE AS SALARY_TABLE;
```

APPENDIX B

BtSQL Bi-Temporal SQL

BtSQL, a graphical user interface bitemporal preprocessor, runs on WindowsNT. BtSQL is written in Java programming language using NetBeans development environment. It uses JDBC to communicate with the underlying DBMS, Oracle 9i. This appendix briefly explains how BtSQL preprocessor works.

BtSQL control menu box commands are Move, Minimize, and Close. Click the minimize button to shrink the document to an icon in the taskbar. Click the icon on the Taskbar to restore the application window to its previous size. To close a palette, click its close box.

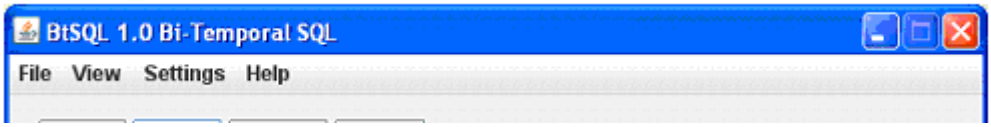


Figure B.1: BtSQL Control menu box and menu bar

BtSQL Control menu bar commands are File, View, Settings, and Help. Press any menu heading to access dialog boxes, submenus, and commands.

Choosing “Settings” from menu bar, Connection Settings window appears. This window asks Server IP number, Port number that Oracle uses and SID number. After verifying schema name, user name and user password verification, ‘Connection Settings’ window connects to the database. This window is closed by clicking close button. To close the connection, re-open the ‘Connection Settings’ window and click on the ‘Disconnect’ button.

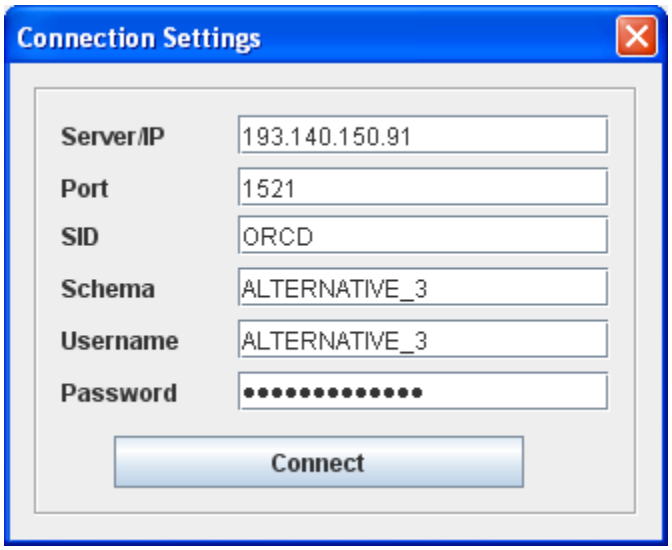


Figure B.2: BtSQL Connection Settings page

Once the connection is established, connection status is shown at the bottom of the page along with current time, as seen in figure B.3.

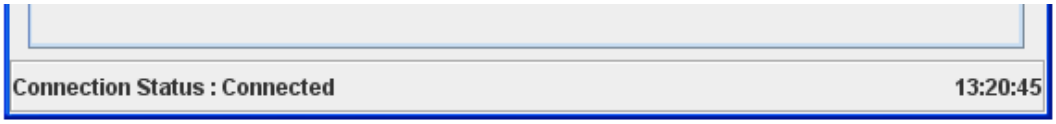


Figure B.3: BtSQL Connection status

BtSQL has four palettes: Insert, Query, Update, and Delete as shown in figure B.4. Update palette has there palettes; Single Tuple, Group of Tuples, and All of Tuples, as depicted and B.5.

Figure B.4: BtSQL palettes

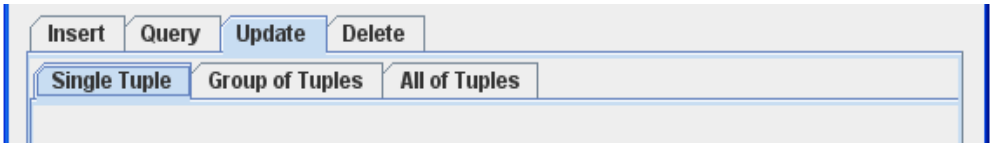


Figure B.5: BtSQL Update palettes

INSERT in BtSQL: The process of inserting 'CANAN EREN' with EMP# 32001, birth date '10.03.1966', address 'West 34th Street NY NY 10292', into the DEP_ID23 department is illustrated in Figure B.6. 'MAN_ID12' is assigned as his manager, and her salary is 40,000, starting on April 29, 2008. BtSQL receives these data from user and calls INSERT_EMPLOYEE procedure, which transforms to SQL standard to insert an entity.

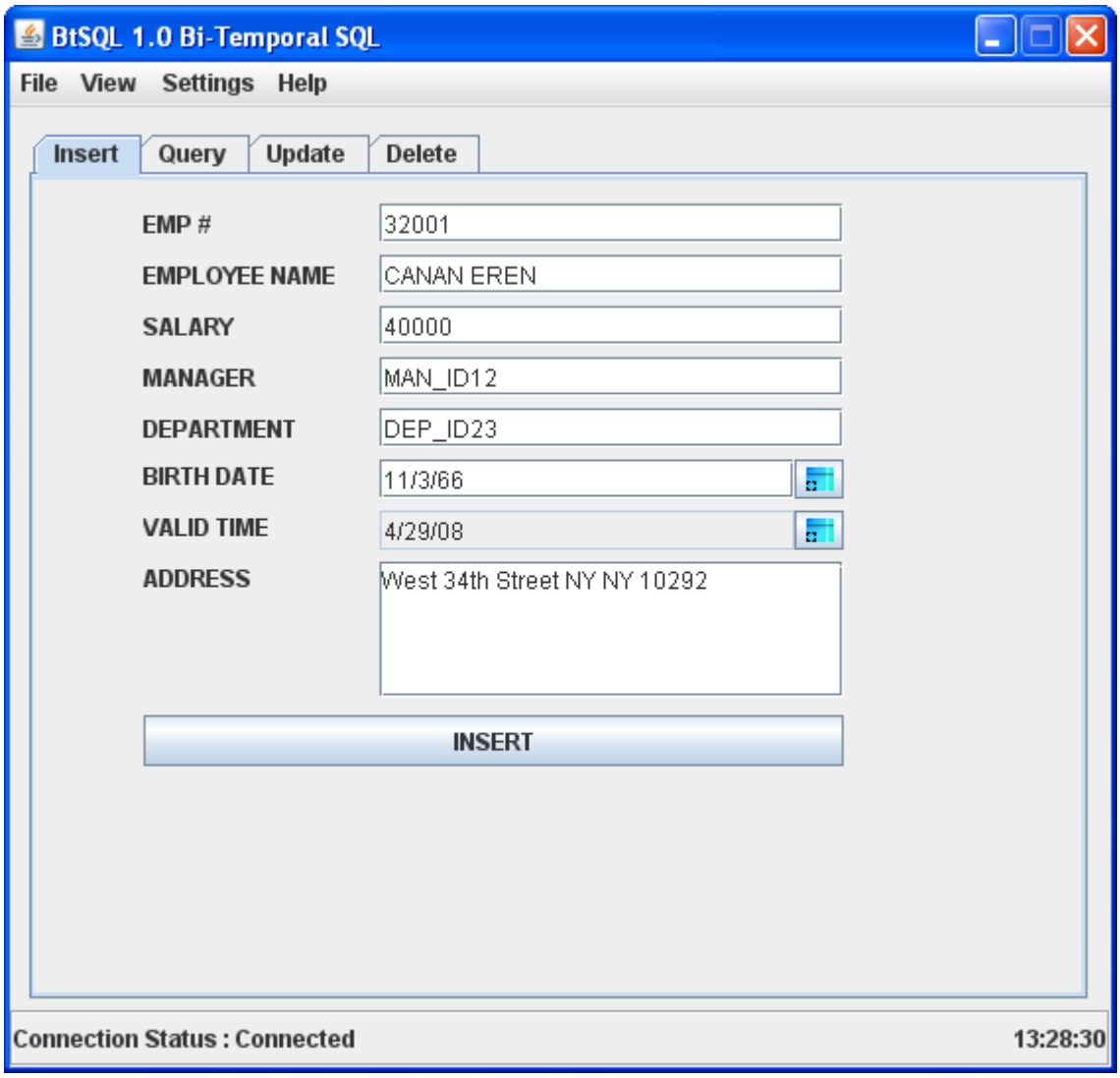


Figure B.6: INSERT page in BtSQL

Update in BtSQL: The UPDATE page has these choices displayed in different pages. If an update is on a single tuple, BtSQL displays the allowed bitemporal attribute names. After choosing the bitemporal attribute upon which the update is to be performed, BtSQL asks for the key attribute value and the new value. The bitemporal atom's value part is replaced with the provided new value.

For example, Figure B.7, shows how the employee's salary data is updated with EMP# = 32001 to 44,000 effective June 2, 2008. If a data error is discovered, a compensating update operation has to be performed to correct the error. The erroneous data are kept; the correct value part and valid time are updated by using the correct date.

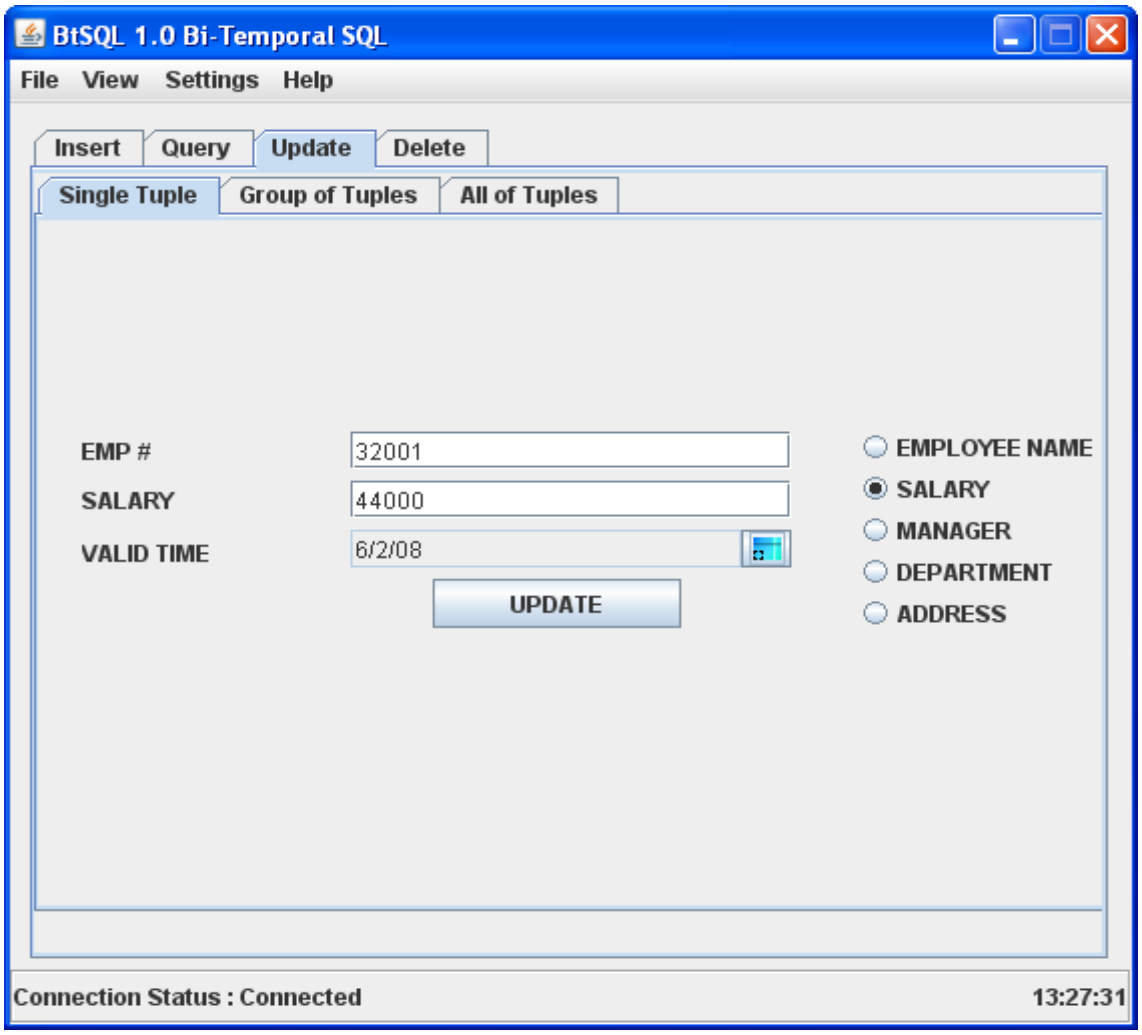


Figure B.7: Update a single tuple in BtSQL

If a group of tuples is to be updated, the required bitemporal attributes are selected. Then, BtSQL asks for a condition and the new values, as well as a valid time. After receiving the new value(s) and valid time from the user, the function updates all the tuples with the new value and the valid time, as seen in figure B.8 that replaces MAN_ID5 with MAN_ID31 valid from May 1, 2008.

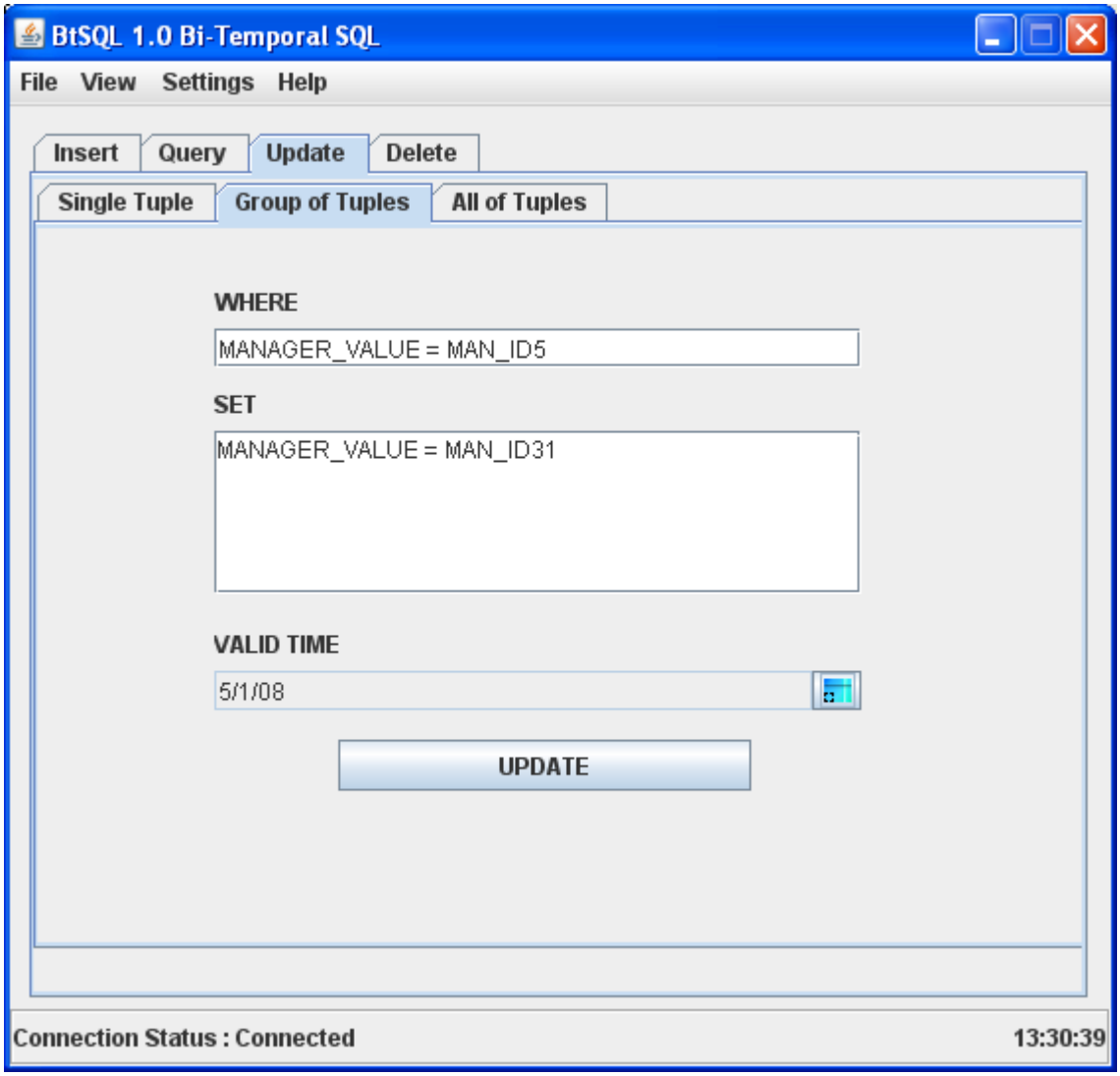


Figure B.8: Update a group of tuples in BtSQL

All of tuples are updated only in SALARY attribute in BtSQL. Figure B.9 depicts this page where salary increase percentage and valid time that this increase was/is/will be effective are asked from the user.

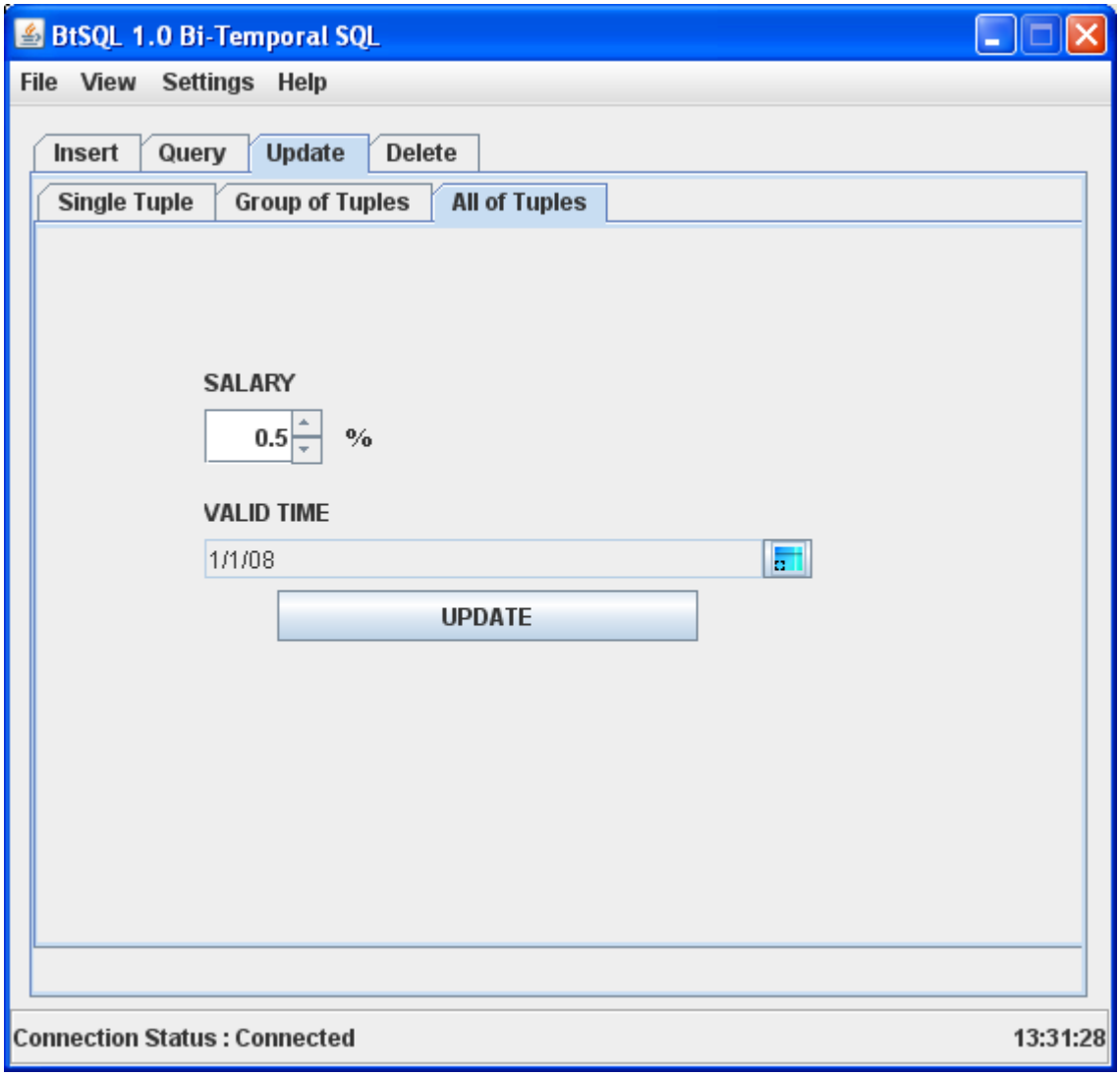


Figure B.9: Update all tuples in BtSQL

Delete in BtSQL: An object or a tuple is typically never deleted from the bitemporal database. The procedure receives the EMP# of the employee and the valid time when the employee leaves the company. Figure B.10 shows the BtSQL's DELETE page, which indicates that EMP# 13456 will not be working beginning on May 1, 2008.

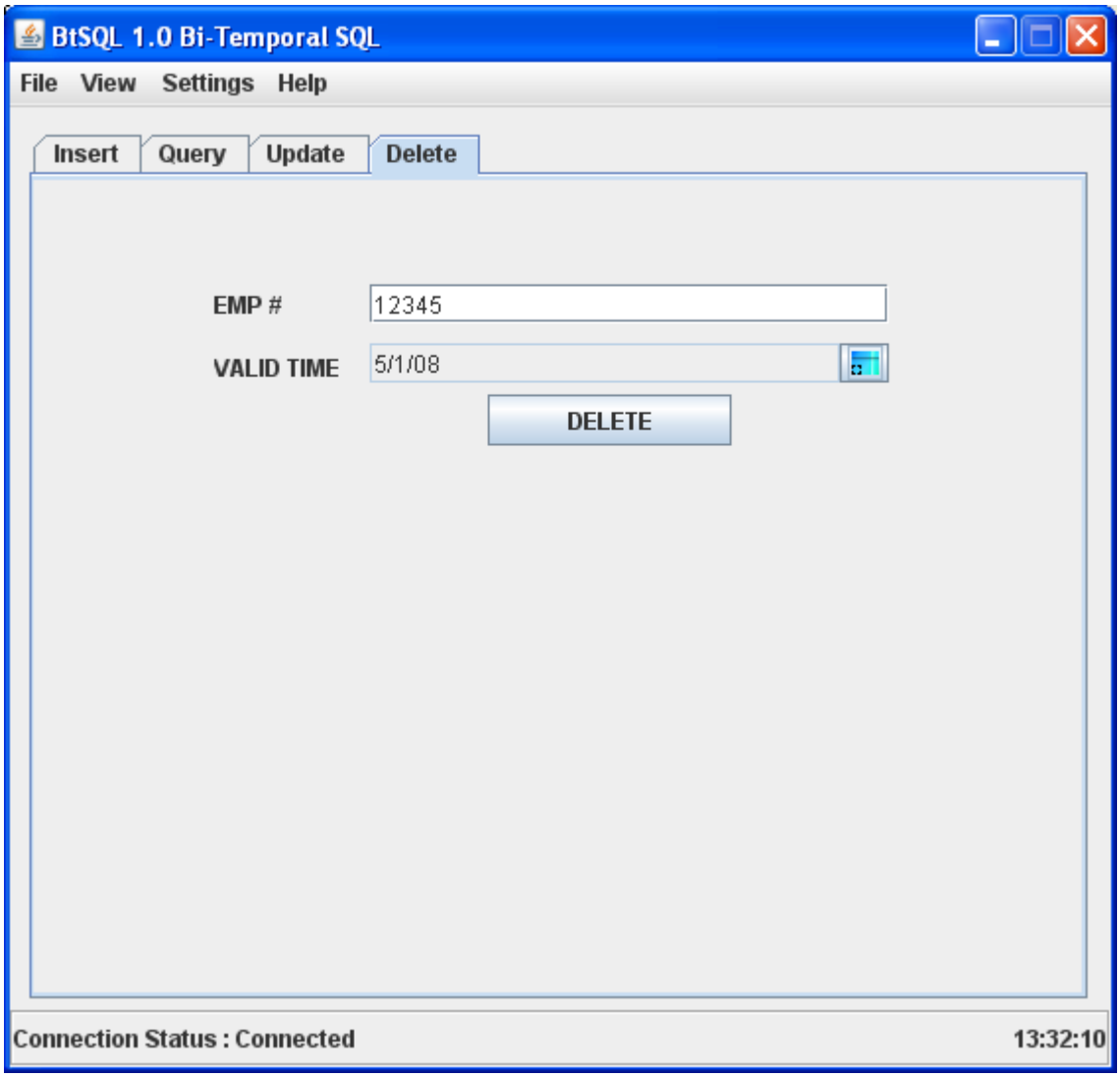


Figure B.10: Delete page in BtSQL

Queries in BtSQL: BtSQL accepts queries in a bitemporal context, in a historical context or in a current context. Queries involve the relation name listed in the FROM clause. The query selects tuples that satisfy the condition(s) of the WHERE clause, and then projects the result to the attributes listed in the SELECT clause. All the options and flavors of the SELECT statement in SQL can also be used in BtSQL. The preprocessor BtSQL also works with any missing WHERE clause – the same as in regular SQL. It also allows the user to employ the ORDER BY clause, to order the tuples in the result of a query by the values of one or more attributes.

BtSQL reads from SELECT-FROM-WHERE-ORDER BY text fields and parses them. If a bitemporal attribute's valid time and/or transaction time interval needs to be displayed, they should be specified as VT and/or TT after the bitemporal attribute's name. BTA's valid time and/or transaction time point data can be displayed by concatenating LB and/or UB into bitemporal attribute name. For each attribute name and/or bitemporal component, corresponding list shown in Table B.1 is used for parsing. For instance, if Name is read from 'SELECT' text field, it is replaced with NAM.VALUE_PART, NAM.TRAN_TIME_LOWER_BOUND, NAM.TRAN_-TIME_UPPER-BOUND, NAM.VALID_TIME_LOWER_BOUND, NAM.VALID-_TIME_UPPER- _BOUND. If SALARY_VTLB is read, then it is replaced by SALARY.VALID_TIME-_LOWER_BOUND bitemporal component. Same parsing algorithm is used for 'WHERE' and 'ORDER BY' text field. For each bitemporal attribute name in 'SELECT' or 'WHERE' text field, TABLE (attribute name) is appended for the 'FROM' text field.

Valid Time and/or Transaction Time check box might be used for a time point or time interval instead of listing them in the 'WHERE' text field. Valid Time and/or Transaction Time are appended into 'WHERE' statement for all bitemporal attributes listed in 'SELECT' text field. Valid Time and/or Transaction Time check box allows us to use these time components in a bitemporal, in a current or in a historical context by choosing calendar (time point or time interval) or 'Now' button in BtSQL query page.

If two bitemporal attributes' common time intervals – or “when” – need to be queried, then the Time Slice check box should be selected. Slice is used in queries, as any other clauses, independent of the bitemporal context, current context or historical context.

Bitemporal context queries in BtSQL: Figure B.11 displays an example of bitemporal context query where whole database is queried. Name and address change of all employee’s whose numbers are between 15000 and 16000 are displayed with all bitemporal components. The result set is ordered in ascending order by employee number and valid time lower bound of name bitemporal attribute. Valid time, transaction time and time slice combination is also possible with bitemporal context.

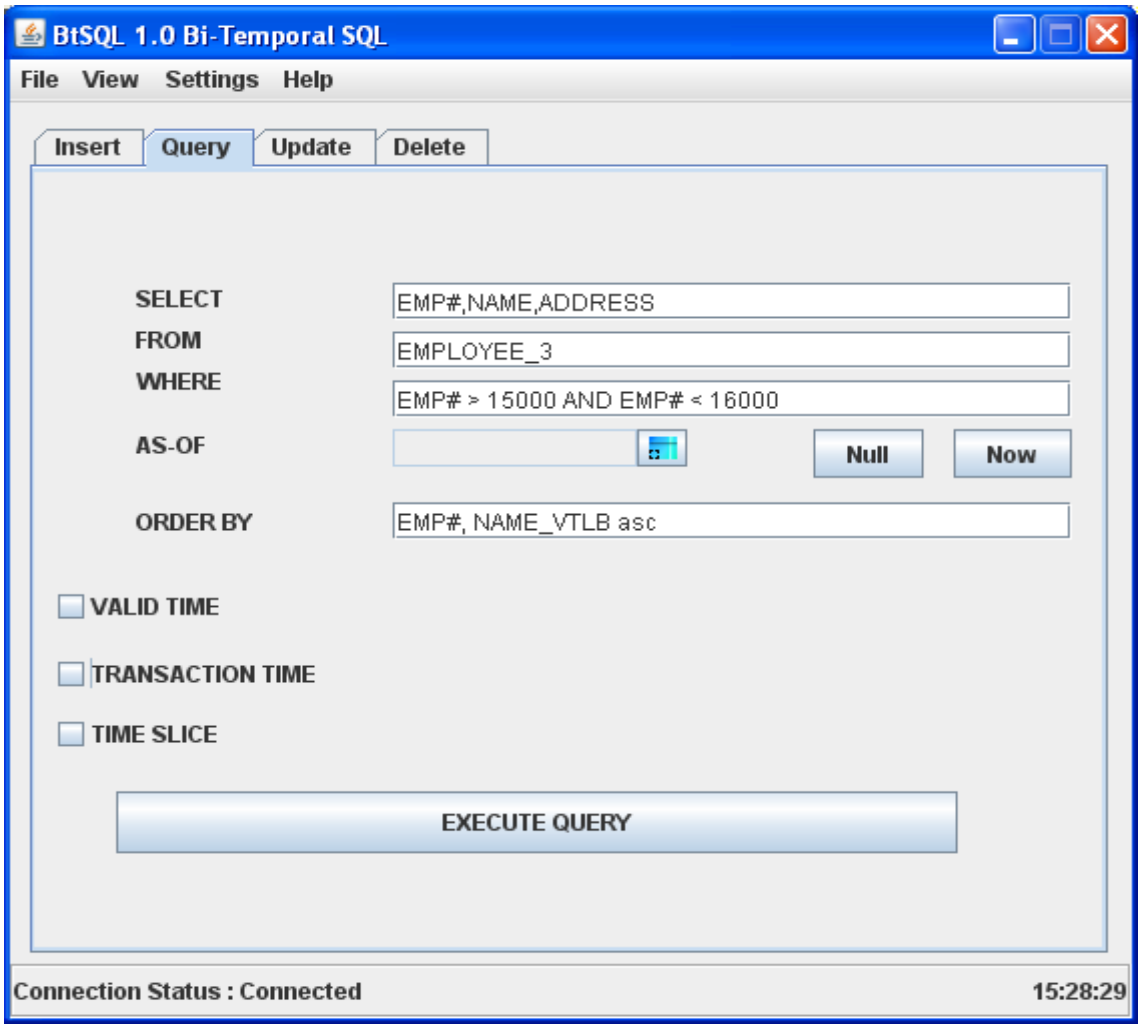


Figure B.11: Bitemporal context query in BtSQL

Current context queries in BtSQL: If 'Now' button is selected in AS OF line, it defaults to a current context. Figure B.12 displays a query that lists employee numbers and names that currently managed by Manager Id 15 and earn more than 80K. The query selects the employee numbers and names that satisfy the conditions MANAGER = 'MAN_ID15' and SALARY>800000, and whose BTA's valid times are equal to '09.09.9999', which is used for *now*. It then passes the result to the EMP# and NAME attributes listed in the SELECT clause. Clicking on 'Null' button allows us to use BtSQL back in a bitemporal context.

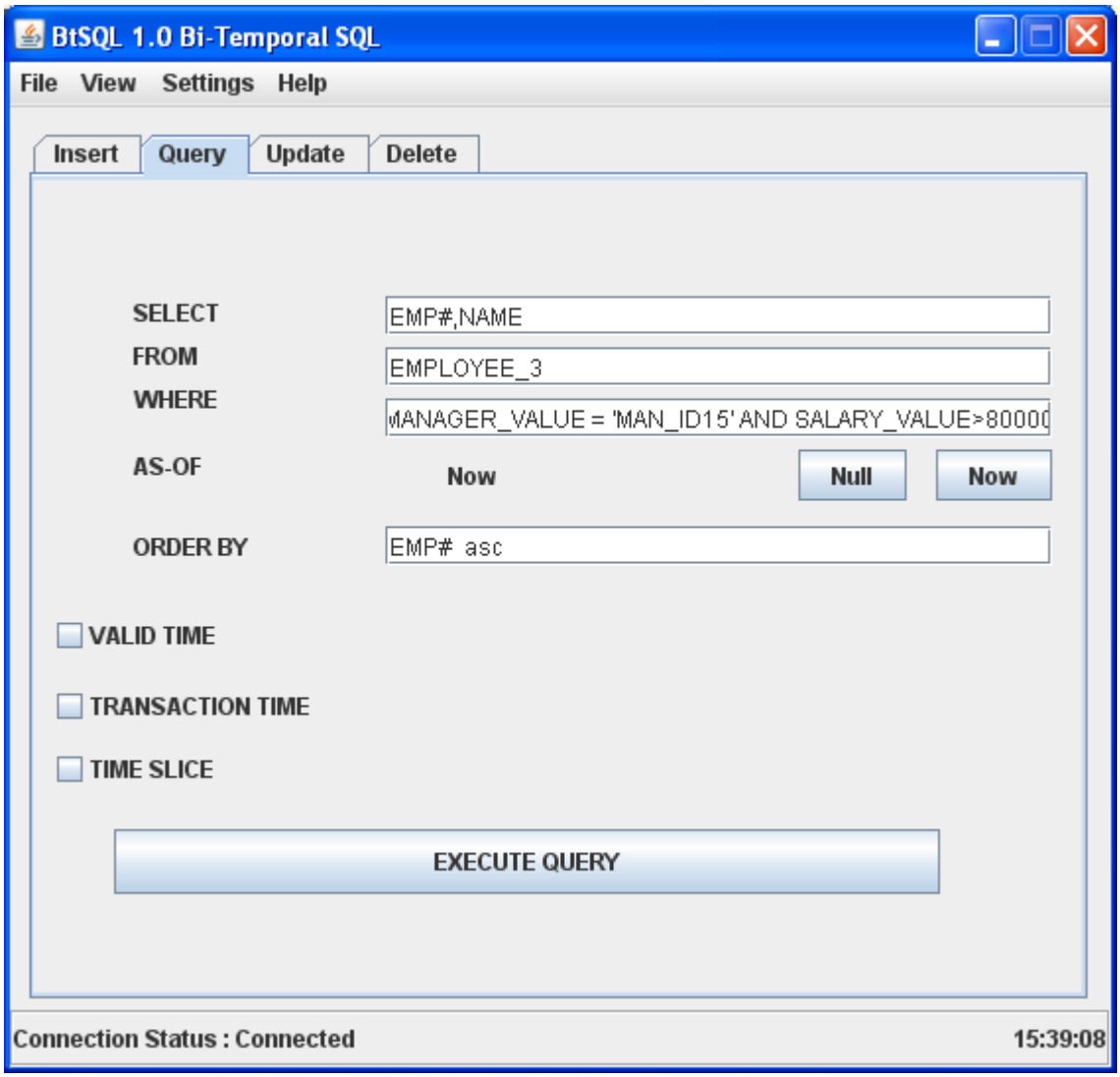


Figure B.12: Current context query in BtSQL

Historical context queries in BtSQL: If any time point or time interval is chosen from the calendar on the AS OF line, BtSQL is used in a historical context. All other restrictions as well as capabilities for bitemporal context apply as well. Figure B.13 is an example of a historical context where name and address change of all employee's between 01/01/97 and 01/01/2001 whose numbers are between 15000 and 16000 are displayed with name and address bitemporal components.

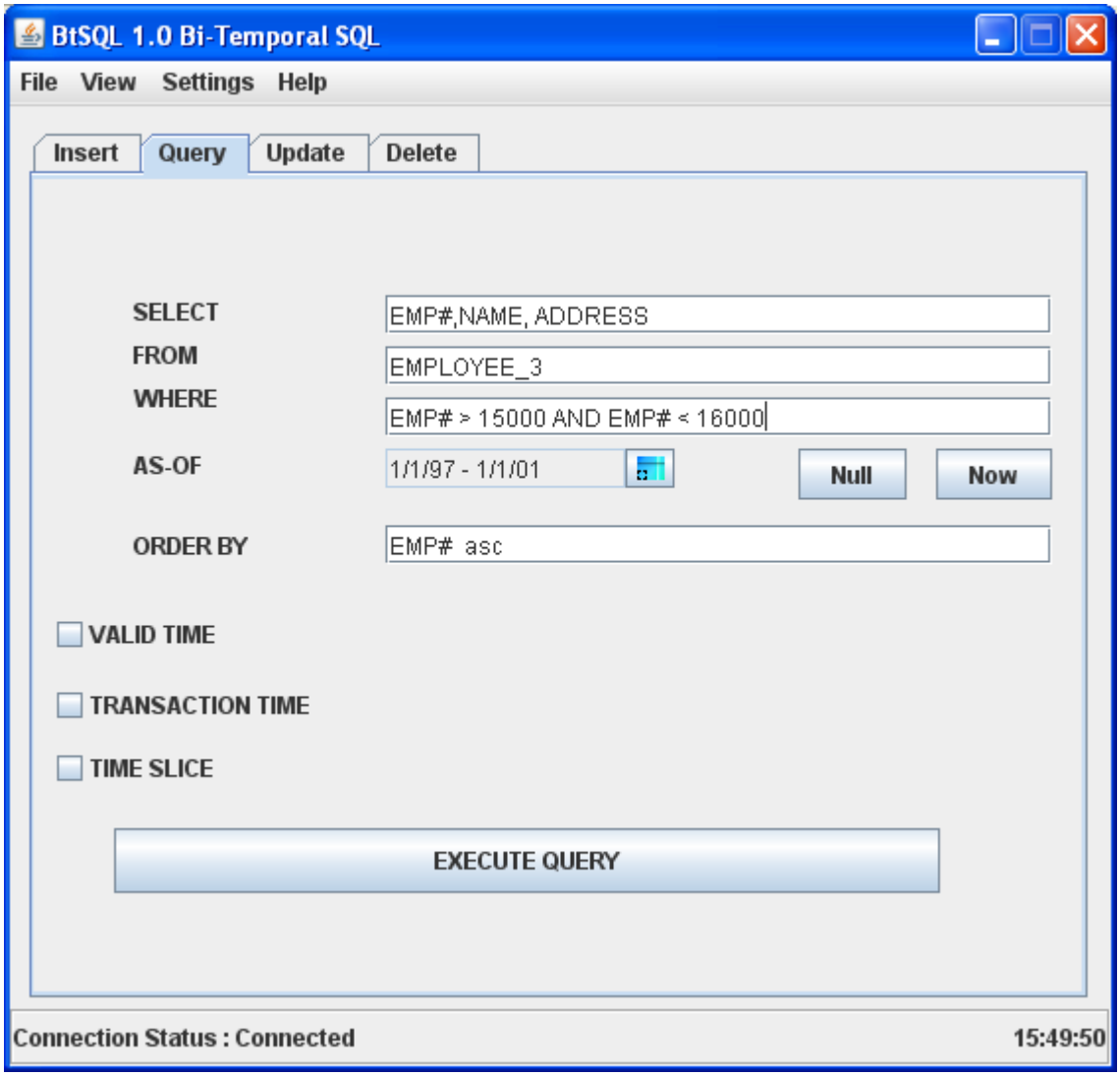


Figure B.13: Historical context query in BtSQL

Table B.1: List of attribute and bitemporal components that are parsed in BtSQL.

NAME_VALUE	NAM.VALUE_PART
NAME_TTLB	NAM.TRAN_TIME_LOWER_BOUND
NAME_TTUB	NAM.TRAN_TIME_UPPER_BOUND
NAME_VTLB	NAM.VALID_TIME_LOWER_BOUND
NAME_VTUB	NAM.VALID_TIME_UPPER_BOUND
NAME_TT	NAM.TRAN_TIME_LOWER_BOUND, NAM.TRAN_TIME_UPPER_BOUND
NAME_VT	NAM.VALID_TIME_LOWER_BOUND, NAM.VALID_TIME_UPPER_BOUND
NAME	NAM.VALUE_PART, NAM.TRAN_TIME_LOWER_BOUND, NAM.TRAN_TIME_UPPER_BOUND, NAM.VALID_TIME_LOWER_BOUND, NAM.VALID_TIME_UPPER_BOUND
ADDRESS_VALUE	ADR.VALUE_PART
ADDRESS_TTLB	ADR.TRAN_TIME_LOWER_BOUND
ADDRESS_TTUB	ADR.TRAN_TIME_UPPER_BOUND
ADDRESS_VTLB	ADR.VALID_TIME_LOWER_BOUND
ADDRESS_VTUB	ADR.VALID_TIME_UPPER_BOUND
ADDRESS_TT	ADR.TRAN_TIME_LOWER_BOUND, ADR.TRAN_TIME_UPPER_BOUND
ADDRESS_VT	ADR.VALID_TIME_LOWER_BOUND, ADR.VALID_TIME_UPPER_BOUND
ADDRESS	ADR.VALUE_PART, ADR.VALID_TIME_LOWER_BOUND, ADR.VALID_TIME_UPPER_BOUND, ADR.TRAN_TIME_LOWER_BOUND, ADR.TRAN_TIME_UPPER_BOUND
DEPARTMENT_VALUE	DPTH.VALUE_PART
DEPARTMENT_TTLB	DPTH.TRAN_TIME_LOWER_BOUND
DEPARTMENT_TTUB	DPTH.TRAN_TIME_UPPER_BOUND
DEPARTMENT_VTLB	DPTH.VALID_TIME_LOWER_BOUND
DEPARTMENT_VTUB	DPTH.VALID_TIME_UPPER_BOUND
DEPARTMENT_TT	DPTH.TRAN_TIME_LOWER_BOUND, DPTH.TRAN_TIME_UPPER_BOUND
DEPARTMENT_VT	DPTH.VALID_TIME_LOWER_BOUND, DPTH.VALID_TIME_UPPER_BOUND

DEPARTMENT	DPTH.VALUE_PART, DPTH.TRAN_TIME_LOWER_BOUND, DPTH.TRAN_TIME_UPPER_BOUND, DPTH.VALID_TIME_LOWER_BOUND, DPTH.VALID_TIME_UPPER_BOUND
MANAGER_VALUE	MNGH.VALUE_PART
MANAGER_TTLB	MNGH.TRAN_TIME_LOWER_BOUND
MANAGER_TTUB	MNGH.TRAN_TIME_UPPER_BOUND
MANAGER_VTLB	MNGH.VALID_TIME_LOWER_BOUND
MANAGER_VTUB	MNGH.VALID_TIME_UPPER_BOUND
MANAGER_TT	MNGH.TRAN_TIME_LOWER_BOUND, MNGH.TRAN_TIME_UPPER_BOUND
MANAGER_VT	MNGH.VALID_TIME_LOWER_BOUND, MNGH.VALID_TIME_UPPER_BOUND
MANAGER	MNGH.VALUE_PART, MNGH.TRAN_TIME_LOWER_BOUND, MNGH.TRAN_TIME_UPPER_BOUND, MNGH.VALID_TIME_LOWER_BOUND, MNGH.VALID_TIME_UPPER_BOUND
SALARY_VALUE	SAL.VALUE_PART
SALARY_TTLB	SAL.TRAN_TIME_LOWER_BOUND
SALARY_TTUB	SAL.TRAN_TIME_UPPER_BOUND
SALARY_VTLB	SAL.VALID_TIME_LOWER_BOUND
SALARY_VTUB	SAL.VALID_TIME_UPPER_BOUND
SALARY_TT	SAL.TRAN_TIME_LOWER_BOUND, SAL.TRAN_TIME_UPPER_BOUND
SALARY_VT	SAL.VALID_TIME_LOWER_BOUND, SAL.VALID_TIME_UPPER_BOUND
SALARY	SAL.VALUE_PART, SAL.TRAN_TIME_LOWER_BOUND, SAL.TRAN_TIME_UPPER_BOUND, SAL.VALID_TIME_LOWER_BOUND, SAL.VALID_TIME_UPPER_BOUND

BIBLIOGRAPHY

[Al83] Allen J., *Maintaining Knowledge about Temporal Intervals*. Communications of the ACM, 1983, 16(11), pages 832-843.

[Ar86] Ariav G., *A Temporally Oriented Data Model.* ACM Transactions on Database Systems, 11, No. 4, Dec 1986, pp 4499-527.

[BG93] Bhargava. G., Gadia. S., *Relational Database Systems with Zero Information Loss*. IEEE Transactions on Knowledge and Data engineering Vol.5, No.1, 1993.

[BJW00] Bettini C., Jajodia S., Wang S. X., Time Granularities in Databases, Data Mining and Temporal Reasoning, Springer 2000.

[BKTT04] P. Buneman, S. Khanna, K. Tajima, and W. Tan. Archiving scientific data. *ACM Trans. Database Syst.*, 29(1):2–42, 2004.

[BZ93] Gadia S. K., *Ben-Zvi's Pioneering Work in Relational Temporal Databases*. Chapter 8, pp. 202 – 207. In A. Tansel et al., editors, *Temporal Databases,* Benjamin/Cummings (1993).

[Ca94] Cattell R.G.G., editor, *The Object Database Standard: ODMG-93*, Release 1.1, Morgan Kaufmann Publishers, San Francisco, 1994.

[CC87] Clifford J., Croker A., *The Historical Relational Data Model (HRDM) and Algebra Based on Lifespans*, in Proceedings of the International Conference on Data Engineering. (Los Angeles, Calif.). IEEE Computer Society Press, 1987.

[CR87] Clifford J., Rao A., *A Simple, General Structure for Temporal Domains*, In Proc. Temporal Aspects in Information Systems, Sophia-Antipolos, France, May 1987.

[CCT94] Clifford J., Croker A. and Tuzhilin A. On Completeness of Historical Relational Query Languages. ACM Transactions on Database Systems, 19(1), 64-116 (1994).

[Co70] Codd E. F., *A Relational Model of Data for Large Shared Data Banks*. CACM, 1970, 13:6.

[CT85] Clifford J., Tansel A. U., *On an algebra for Historical Relational Databases: Two views.* ACM SIGMOD International Conference on Management of Data, 1985, pages 247 – 265.

[CW83] Clifford J., Warren D. S., *Formal Semantics for Time in Databases*. ACM Transactions on Database Systems, 1983, 8(2), pages 214 - 254.

[EN04] Elmasri R., Navathe S., *Fundamentals of Database Systems, Fourth Edition*. Addison Wesley Publishing Company, 2004.

[EW90] Elmasri R., Wuu G., *A Temporal Model and Query Language for ER Databases*, In Proceedings of the Sixth International Conference on Data Engineering, pp 76-83, 1990.

[EWH85] Elmasri R., Weeldreyer J., Hevner A., *The category concept: An extension to the ER model*. Data and Knowledge Engineering, 1985.

[FT83] Fisher, P., Thomas, S., 'Operators for Non-First Normal Formal Relations', Proc. of 7th Intl. Computer Software applications Conf., 1983.

[Ga86] Gadia S.K. *Weak Temporal Relations*. Proceedings of the 5th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, Cambridge, Massachusetts, 70-77 (1986).

[Ga88] Gadia S. K*., A Homogeneous Relational Model and Query Languages for Temporal Databases,* ACM Transactions on Database Systems, December 1988.

[Ga03] Garani G., *A Temporal Database Model Using Nested Relations.* PhD Thesis, School of Computer Science and Information Systems, Birkbeck College, 2003.

[Go98] Gorawalla, I.A., *Temporality in Object Database Management Systems*. PhD Thesis, University of Alberta, Canada, 1998.

[GB89] Gadia S.K., Bhargava G., *A Formal Treatment of Updates and Errors In A Relational Database*, Technical Report TR97-14, Department of Computer Science, Iowa State University.

[GM00] F. Grandi and F. Mandreoli. The Valid Web: An XML/XSL Infrastructure for Temporal Management of Web Documents. In ADVIS, 2000.

[GN93] Gadia S. K., Nair S., *Temporal Databases: A Produce to Parametric Data*. Chapter 2, pp. 28–66. In A. Tansel et al., editors, *Temporal Databases,* Benjamin/Cummings (1993).

[GÖ93] Goralwalla I. A., Özsu M. T., *Temporal Extensions to a Uniform Behavioral Object Model*. In Proceedings of the 10th International Conference on the ER Approach, 1993, pages 110 - 121.

[GS03] M. Gergatsoulis and Y. Stavrakas. Representing Changes in XML Documents using Dimensions. In *Xsym*, 2003.

[GSn03] D. Gao and R. T. Snodgrass. Temporal Slicing in the Evaluation of XML Queries. In VLDB, 2003.

[GT91] Garnett L., Tansel A.U., "Equivalence of the Relational Algebra and Calculus Languages for Nested Relations," Math. And Computers with Applications, vol. 23, no. 10, pp. 3-25, 1991.

[GV85] Gadia S. K., Vaishnav J. H., *A Query Language for a Homogeneous Temporal Database.* In Proceedings of the International Conference on Principles of Database Systems, 1985, pages 51-56.

[GY88] Gadia S.K., Yeung C.A., *Generalized Model for a Relational Temporal Database*. In Proceedings of the ACM SIGMOD Conference, pages 251_259, 1988.

[ISO03] Information technology - Database languages – SQL Part 14: XML Related Specifications. 2003

[JDB+98] Jensen C.S., Dyreson C.E., Böhlen M, Clifford J., Elmasri R., Gadia S.K., Grandi F., Hayes P., Jajodia S., Käfer W., Kline N., Lorentzos N.A., Mitsopoulos Y., Montanari A., Nonen D., Peressi E., Pernici B., Roddick J.F., Sarda N.L., Scalas M.R., Segev A., Snodgrass R.T., Soo M.D., Tansel A., Tiberio P. and Wiederhold G. The Consensus Glossary of Temporal Database Concepts-February 1998 Version. In [EJS98], 367-405 (1998).

[JMR91] Jensen, C. S., Mark L., and Roussopoulos, N. *Incremental Implementation Model for Relational Databases with Transaction Time.* IEEE Transactions on Knowledge and Data Engineering, 3, No. 4, Dec. 1991, pp. 461–473.

[JS82] Jaeschke G., Schek H., *Remarks on the Algebra of Non First Normal Form Relations*, Proc. ACM SIGACT SIGMID Symp. Principles of Database Systems, pp. 124-138, 1982.

[JSS94] Jensen, C. S., Soo, M. D., Snodgrass, R. T.: Unifying Temporal Data Models Via a Conceptual Model, Information Systems, (1994) 19(7):513–547.

[JSST01] Jensen C.S., Schneider M., Seeger B. and Tsotras V.J. (Eds.), *Advances in Spatial and Temporal Databases: Proceedings of the 7th International Symposium (SSTD)*, Redondo Beach, CA, USA. (2001).

[KS92] Käfer W., Schöning H., *Realizing a temporal complex-object data model*, ACM SIGMOD Record, pages 266 - 275 , 1992.

[Lo88] Lorentzos, N. A., *A formal extension of the relational model for the representation and manipulation of generic intervals*. Ph.D. Thesis, Birkbeck College. University of London., 1988.

[Ma77] Makinouchi A. *A consideration on Normal Form of Not-Necessarily-Normalized Relations in the Relational Data Model*. Proceedings of the 3rd International Conference on Very LargeData Bases, Tokyo, Japan, 447-453 (1977).

[MS02] Melton J., Simon, A. R., SQL:1999 Understanding Relational Language Components, Morgan Kaufmann Publishers, 2002.

[NA87] Navathe, S., Ahmed, R., *TSQL–A Language Interface for History Databases,* in Proceedings of the Conference on Temporal Aspects in Information Systems, 1987.

[NA89] Navathe S., Ahmed R., *A Temporal Relational Model and a Query Language*. Information Sciences, 49, 147-175, 1989.

[NG06] Noh S,. Gadia S., *A comparison of two approaches to utilizing XML in parametric databases for temporal data,* Information and Software Technology, Volume 48, Issue 9, September 2006, Pages 807-819.

[OO83] Ozsoyoglu, M. Z., Ozsoyoglu, G.: An Extension of relational Algebra for Summary Tables, Proceedings of the 2nd Intl Workshop on SDB Management, (1983).

[OOM87] Ozsoyoglu G., Ozsoyoglu M.Z., Matos V., *Extending Relational Algebra and Relational Calculus with Set-Valued Attributes and Aggregate Functions*, ACM Trans. Database Systems, vol. 12, no. 4, pp. 566-592, 1987.

[ORA01] www.oracle.com

[OS95] Ozsoyoglu G., Snodgrass, R., *Temporal and Real-TimeDatabases:A Survey*, *IEEE* Transactions on Knowledge and Data Engineering, 7, No.4, Aug. 1995, pp. 513–532.

[RKS88] Roth M.A., Korth H.F., Silberschatz A. *Extended Algebra and Calculus for Nested Relational Databases*. ACM Transactions on Database Systems, 13(4), 389-417 (1988).

[S+94] Snodgrass, R. T., I. Ahn, G. Ariav, D.S. Batory, J. Clifford, C.E. Dyreson, R. Elmasri, F. Grandi, C.S. Jensen, W. Käfer, N. Kline, K. Kulkarni, T.Y.C. Leung, N. Lorentzos, J.F. Roddick, A. Segev, M.D. Soo and S.M. Sripada. *The TSQL2 Temporal Query Language*. *ACM SIGMOD Record*, 23, No. 1, Mar. 1994.

[SA86] Snodgrass R., Ahn I., *Performance Evaluation of a Temporal Database Management System*, ACM 1986.

[Sa90] Sarda N.L., *Extensions to SQL for Historical Databases*, IEEE Trans. Knowledge and Data Eng., vol. 2, no. 2, pp. 220–230, 1990.

[SC91] Su S., Chen H., *A Temporal Knowledge Representation Model OSAM/T and Its Query Language OQL/T*, Proceedings of the 17th International Conference on Very Large Data Bases, September. 1991.

[Sh81] Shipman D. W. *The Functional Data Model and Data Language DAPLEX*. ACM Transactions on Database System, 1981, 6(1), pages 140-173.

[Sn00] Snodgrass R., *Developing Time-Oriented Database Applications in SQL*, Morgan Kaufmann Publishers, 2000.

[Sn87] Snodgrass R., *The Temporal Query Language TQUEL*, ACM Transactions on Database Systems, June 1987.

[Sn93] Snodgrass R., *An Overview of Tquel*, Chapter 6, pp. 141–182. In A. Tansel et al., editors, *Temporal Databases,* Benjamin/Cummings (1993).

[St98] Steiner A., *A Generalisation Approach to Temporal Data Models and Their Implementations.*, PhD Thesis, Swiss Federal Institute of Technology, Zurich, 1998.

[SS86] Schek H. J., Scholl S., *An Algebra for the Relational Model with Relation-Valued Attributes*, Information Systems, 11 (2), (1986).

[SZ01] Shasha D., Zhu Y., *SpyTime – a Performance Benchmark for Bitemporal Database*, www.cs.nyu.edu/shasha/spytime/spytime.html

[T+93] *Temporal Databases: Theory, Design, and Implementation*, Tansel A. U., J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass, eds., Benjamin/Cummings, 1993.

[Ta86] Tansel, A. U. *Adding Time Dimension to Relational Model and Extending Relational Algebra*, Information Systems, 11, No. 4 (1986), pp. 343–355.

[Ta90] Tansel A. U., *Modelling Temporal Data*. Information and Software Technology, 1990, 32, pages 514-520.

[Ta97] Tansel A. U., *Temporal Relational Data Model*, IEEE Transactions on Knowledge and Data Engineering, IEEE Transactions on Knowledge and Data Engineering, Vol. 9, No. 3, June 1997.

[Ta04] Tansel A. U., *Integrity Constraints in Temporal Relational Databases*, International Conference on Information Technology: Coding and Computing, Volume 2, page 460, 2004.

[To97] Toman D., *A Point_based Temporal Extension of SQL*, Proc. 5th International Conference on Deductive and Object-Oriented Databases, LNCS 1341, 103-121, 1997.

[TA86] Tansel A. U., Arkun M. E., *HQuel, A Query Language for Historical Relational Databases,* Proceedings of the Third International Work shop on Statistical and Scientific Databases. July 1986.

[TA06] Tansel A. U., Atay C. E., Nested Bitemporal Relational Algebra, ISCIS 2006, 622-633.

[TG89] Tansel A. U., Garnett L., *Nested Temporal Relations*, Proc. ACM SIGMOD Int'l Conf. Management Data, pp. 284–293, 1989.

[TT97] Tansel A. U., Tin E., *Expressive Power of Temporal Relational Query Languages*, IEEE Transactions on Knowledge and Data Engineering, Vol. 9, No. 1, Jan. 1997.

[Ul95] Ullman J.D. Principles of Database and Knowledge-Base Systems. Computer Science Press (1995).

[WD93] Wuu G., Dayal U., *A Uniform Model for Temporal and Versioned Object-oriented Databases*. Chapter 10, pp. 230–247. In A. Tansel et al., editors, *Temporal Databases,* Benjamin/Cummings (1993).

[WJS93] X.S. Wang, S. Jajodia, and V. Subrahmanian. Temporal Modules: An approach Toward Temporal Databases. In Proc. ACM SIGMOD Int. Conf. on Management of Data, pages 227-236, 1993.

[WZ03-a] F. Wang and C. Zaniolo. Temporal Queries in XML Document Archives and Web Warehouses. In TIME-ICTL, 2003.

[WZ03-b] F. Wang and C. Zaniolo. Publishing and Querying the Histories of Archived Relational Databases in XML. In WISE, 2003.

[WZ04] Fusheng Wang, Carlo Zaniolo. XBiT: An XML-Based Bitemporal Data Model. ER 2004: 810-824.

[WZZ05] Using XML to Build Efficient Transaction-Time Temporal Database Systems on Relational Databases, Technical Report 81, TimeCenter, 2005.

[XMLQ] W3C, XML query, Website: http://www.w3.org/XML/Query