A Database Query Language for Operations on Historical Data

Research Project

Presented as a requirement for the

Degree of Phd

in

Computer Science

by

Rubik Sadeghi

(Dec 1987)

Department of Mathematics & Computer Studies
Dundee College of Technology
Scotland

Table of Contents

Table of Figures

ACKNOWLEDGEMENTS

I would like first of all to thank all of the members of my family, my parents Gregor and Mahich Sadeghi, and my sisters and brothers who deserve all the thanks I can give for supporting me throughout this research with unlimited patience, love, and encouragement.

I would like to thank my supervisors Dr. Bill Samson and Professor Misbah Deen for providing me with excellent guidence, encouragement, and support.

Finally I would like to thank my colleagues Andy Wakelin and Kaizad Heerjee for many stimulating discussions. I also like to thank them for putting up with me through out the course of this research.

Abstract of the Dissertation

A Database Query Language for Operations on Historical Data

Rubik Sadeghi

Dundee College of Technology

1987

ABSTRACT

Information in most existing database management systems reflects only the current state of the data. However, many real world applications require the storage of and access to historical information. In this thesis a historical model is presented which integrates comprehensive time processing capabilities into the relational model. The role of time in this model is examined, as is the effect of time on relational algebra. An algebra for historical relations is defined.

Finally the Historical Query Language (HQL) is defined and implemented. HQL is designed to allow users to interrogate the historical database in a natural way without referring to the time domain explicitly. HQL has a user friendly syntax and supports user-defined functions and recursion.

CHAPTER 1

INTRODUCTION

For many existing Data Base Management Systems (DBMS) information in the database reflects the current state of the data i.e. the database is viewed as a snapshot of the enterprise which it models at a particular point in time. Although information in the database continues to change as new information is added, these changes are viewed as modifications of the state, with old, out of date data being deleted from the database. However this eliminates the possibility of accessing any information except that which involves facts that are currently true. The need to view data at different points in time leads to the concept of temporal or historical databases (HDB).

During the last seven years, there has been increasing interest in the area of HDBs. A quite extensive bibliography containing roughly 40 references to research into time and databases has been identified and annotated in [Bolo82], and a dozen more have since been reported. The underlying premise of the expanding body of research is the recognition that time is not merely another dimension, or another data item tagging along with each tuple, but rather something human users treat in very special ways. Time is a universal dimension of data management applications, and thus the focal point of the various attempts in this domain has been the conceptualisation and design of a general-purpose system

that deals with data dynamics in an explicit fashion.

The need for a DBMS with the functionality to support historical information can be seen in many applications. For example, in the storage of data relating to legislation, historical information is just as important as current data. Another example is in decision making applications where future strategies are planned by viewing changes that lead to a particular state. Historical databases have also been used in medical information systems where time is an important part of the data.

In the past, support for historical data was fulfilled by archiving or by the provision of audit trail files. The historical data are then obtained from the audit trail or archived files as needed. However, these methods provide no convenient or efficient mechanism for users to access the data.

In recent years a number of data models have been developed which support time varying information. A temporal extension should provide a mechanism for accessing historical data but should also make the time capabilities transparent to users who operate only with current data, at minimum cost.

Research into query languages for historical databases has also received a great deal of attention recently [Jone79,Ben-82,Snod84,Gadi85,Aria86], but so far more attention has been given to design and implementation issues of historical databases than to query languages for HDBs. Nevertheless with almost every new temporal database a new query language has also surfaced.

The problems inherent in the modeling of time are not unique to information processing; representing temporal knowledge and temporal reasoning arises in wide range of other disciplines. In artificial intelligence [Alle84,Kahn77,McDe82,Vila82], models of problem solving require sophisticated world models that can capture change. In planning the activities of a robot, for instance, one must model the effects of the robot's actions on the world to ensure that a plan will be effective. In natural language processing [Hirs81,McCa71,Mont73], researchers are concerned with extracting and capturing temporal and tense information in sentences. This knowledge is necessary to be able to answer queries about the sentences later. A significant literature also exists on related issues in logic [McAr76,Prio67], philosophy [Whit80] and physics [Tayl66].

Beyond the perplexing conceptual complexity involved, attempts to implement historically rich databases have suffered from the absence of adequate means to handle the huge volumes of secondary storage needed for maintaining "complete histories". However in recent years with development of optical mass-storage devices, which offer both immense and directly accessible data storage facilities [Fuji84], and also with the development of systems which address issues such as how to store history data in a compact form [Lum84], some of the storage problems have been solved.

For our underlying model, we adopt the relational model both because it is a well-studied and formalised database model, and because it is increasingly being used as a model for implemented systems.

The relational model, PRECI/C, the advanced query language DEAL, as well as earlier historical models and historical query languages are reviewed in chapter 2.

The basic concepts of time and time operations are described in chapter 3. The time relation is introduced along with a proposed normal form, as well as the changes to relational algebra operators required to handle time attributes. The new algebra is a straightforward extension of the conventional relational algebra which supports valid time, i.e. the time when an object or relationship in the enterprise being modeled is valid. Historical versions of the five relational operators union, difference, cartesian product, selection, and projection are defined and two new operators, time slice, and when, are introduced.

The historical query language (HQL) is described in chapter 4. HQL is based on DEAL, and is capable of supporting both conventional and historical queries. HQL has several new features, which provide extremely powerful query processing capabilities. These capabilities are described with examples of simple and complex queries.

The semantics of HQL is described in chapter 5 showing how HQL queries are mapped into expressions of relational algebra, for which a semantics has been defined. A three level architecture is described for translating HQL queries into relational algebra.

The implementation of HQL is described in chapter 6. The UNIX language development tools YACC and LEX are used to built a simple interpreter for HQL.

Implementation issues such as the handling of ever-growing storage size, representation of historical data in physical storage, and handling of structural changes are discussed in chapter 7.

A review of the major ideas of the dissertation is presented in chapter 8. Chapter 9 concludes with a discussion of directions for future research.

1.1. Research Objectives

The overall objective of this project is to design a database query language for operations on historical data.

This objective can be subdivided as follows:

(a) To produce a review of previous research in the area of temporal data models and historical query languages.

(b) To provide a technique for incorporating a semantics of time into relational databases. This includes close examination of the concept of time and how does time fit into the model. Two methods of time-stamping are examined and the view which treats time as a component of the tuple rather than the attribute is proposed.

(c) To extend the relational algebra consistently for dealing with temporal relations.

(d) To design a query language for operations on historical data.

CHAPTER 2

REVIEW

In this chapter an account is given of the relational model, the PRECI/C database management system and the query language DEAL. Previous research in the area of temporal data models and historical query languages is reviewed.

2.1. The Relational Model

The relational data model was first introduced by Codd in 1970 [Codd70]. Like any other data model, the relational model consist of three components:

(a) a set of objects

tuples, relations

(b) a set of operators

union, intersection, difference, product

select, project, join, divide and

relational assignment

(c) a set of integrity constraints

1. primary key values must not be null.

2. foreign key values must match primary key values or be

null.

The term "relational" is used in its accepted mathematical sense. Given sets A1,A2,...,An (not necessarily distinct), r is a relation on these n sets if it is a set of n-tuples each of which has its first element from A1, its second element from A2, and so on. In other words, r is defined as a subset of the cartesian product over n domains, A1XA2X...XAjX...An. Aj is referred to as the jth domain of r and r is said to have degree n. Relations of degree 1 are called unary, degree 2 binary, degree 3 ternary and degree n n-ary. An n-ary relation which can be represented as a table has the following properties:

- There are no duplicate tuples;

- Tuples are unordered (top to bottom);

- Attributes are unordered (left to right);

- All attribute values are atomic.

A domain is an abstract set of atomic data values. Domains are defined independently of relations. A relation may be viewed as a table, wherein each row of the table corresponds to a tuple of the relation. Entries in a column belong to the set of values constituting the underlying domain of that column, that is those data values which can appear as entries in that column. Each database relation is created by naming the relation and each constituent column (i.e. specifying the name of the underlying domain of each column). More than one column in a relation may have the same underlying domain.

The column names are also called attributes. The values of an attribute range over the domain associated with the column. For a given relation R with attributes A1,A2,...,An, R(A1,A2,...,An) will represent the relation scheme for relation R. It is common to say that relation r is an instance of the relation scheme. However, not all instances of a relation scheme have meaningful interpretations and so this leads to the introduction of a set of integrity constraints to restrict the valid states of a relation. An integrity constraint involving a set of attributes A1,A2,....An is a predicate on the collection of all relations that form such a set. A relation obeys the constraint whenever an evaluation of the predicate for that set returns the value "true".

Relational algebra (operators) possesses the important property of closure-that is, the result of applying any of those operations to any relations(s) is itself another relation.

It is worth mentioning that, of these operations, only five are primitive, namely selection, projection, product, union, and difference. The other three can be defined in terms of these five [Ullm83]. For example, the natural join is a projection of a selection of a product.

2.2. PRECI/C

PRECI/C [Deen85] is a small, simple, relational database system implemented in 'C' on PDP and VAX running UNIX. It was the result of a database design project carried out as part of the PRECI project at Aberdeen university. The PRECI/C system was designed and implemented with the following goals:

(a) to provide a test harness for further research.

(b) to apply ideas in software engineering to the design of a DBMS.

(c) to provide a teaching aid for database courses.

The software engineering concept of modularity and information hiding is employed in the design of PRECI/C which is founded on a layered system architecture with well defined modules. Each module performs defined tasks by providing a set of related functions which act as the interface to other modules.

Figure-2.1 shows the PRECI/C layered architecture. Each level provides an abstraction of the data and commands of the level below, with well defined interfaces between levels. Queries submitted at the top level are converted into relational and other high level commands. These high level commands are resolved into lower level functions which act on the virtual database provided by the level below. Functions provided at the top level are the relational operations such as project, join and so on. For example, the project function takes as parameters a relation identifier, the number of attributes to be projected out and the list of which those attributes are, then returns a new relation.

The SQL query

```
SELECT  snum,sname,city
FROM    supplier
```

would be mapped into

```
project(supplier_id,3,[snum,sname,city])
```

The main advantage of the system is that each module and level can be independently tested, alleviating some of the problems of debugging. Modules can be re-written or expanded with relative ease, allowing, for instance, the writing of a minimal system, followed by alteration and expansion as desired. Run-time efficiency, minimal memory usage, optimisation facility, data independence and ease in restructuring and reorganisation are the major features of the PRECI/C.

PRECI/C has also been used in developments which include:

(a) extensions to deductive data and queries [Deen85].

(b) the testing of concepts developed in the PRECI/* [Deen85] project.

(c) implementation issues in semantic data modelling [Edga86].

(d) graphical databases [Wake87].

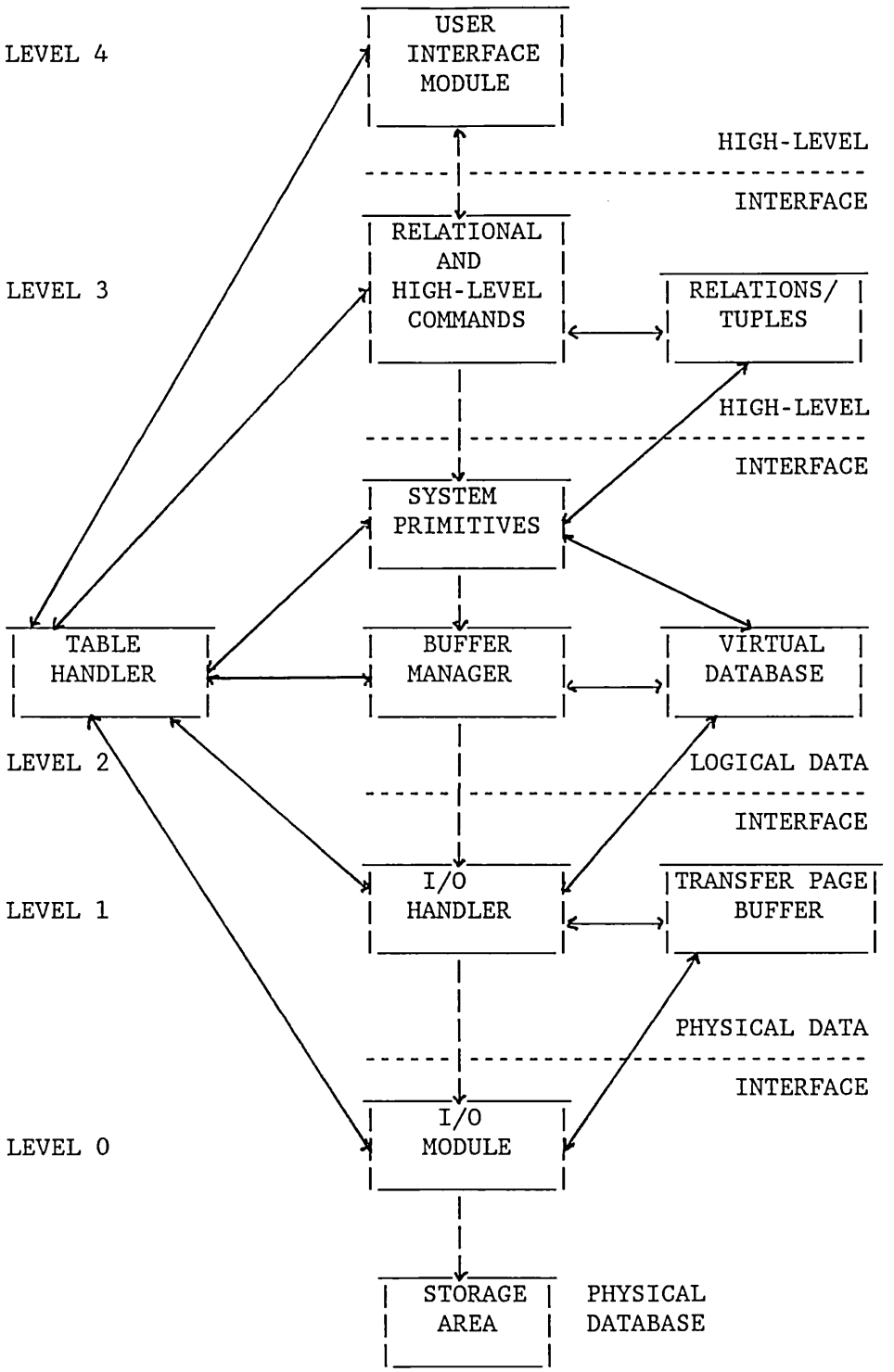


Figure 2.1: PRECI/C DBMS Overview

2.3. DEAL Query Language

DEAL (Deductive Algebra) [Deen85] is a proposed extension of a relational language, capable of supporting user-defined functions, recursions and deductions based on the full first-order predicate calculus. DEAL was intended to provide a unified framework for all database processing - conventional and deductive.

In the design of DEAL, special attention was given to the orthogonality of its constructs. The advantage of orthogonality is that it leads to a coherent language, a language that is simple, clean, and with consistent structure. It is based on the belief that orthogonality must be the guiding principle in language design. One general point that is worth stating is the following (taken from[Date84]):

> "Most languages are too big and intellectually unmanageable. The problems arise in part because the language is too restrictive; the number of rules needed to define a language increases when a general rule has additional rules attached to constrain its use in certain cases. Ironically, these additional rules make the language less powerful. Power through simplicity, simplicity through generality, should be the guiding principle".

2.3.1. Relational Operations

The structure of a DEAL query has three components

base-expression

attribute-spec

selection predicate

linked together as

```
base-expression [attribute-spec]
where selection predicate
```

The presence of both the attribute-spec and base-expression are not essential in a query; additionally, the selection predicate is optional. The base-expression can be a relational expression, or another DEAL query. As a relational expression, it may include any valid relational operations.

Figure-2.2 is an overview of the DEAL syntax. The complete syntax of DEAL is given in appendix-D.

```
query          ::=    query_expr
               |      func-defs

query-expr     ::=    query-block
               |      query-expr set-op query-block

query-block    ::=    rel-expr
                      [ '[' selection-list ']' ]
                      [ where condition ]

rel-expr       ::=    relation-name
               |      ( query-expr )
```

Figure 2.2: An Overview Syntax of DEAL

The syntax of DEAL compared with SQL is illustrated below.

DEAL	SQL
Emp[name,city] where enum > 2	select name,city from Emp where enum > 2
Emp	select * from Emp
Emp where city = 'london'	select * from Emp where city = 'london'

DEAL, unlike SQL, supports nesting at the external or user's level, for example the DEAL query

```
(Emp [enum,name,salary] where city = 'London')
where salary > 50k;
```

can not be directly translated into a SQL query without defining a view (for London employees first). The advantage of DEAL, in this instance, is that queries are expressed in a natural way and are therefore easier to understand.

A query to find the employee names and their department names for departments in Paris may be written as:

```
(Emp (edno , dno) Dept) [name,dname]
where loc = 'Paris';
```

The base-expression contains a join of Emp and Dept over the common domains edno and dno.

The principal operations allowed within a base-expression are

Cartesian product (**), Union (++)
Outer union (+?), and Difference (--)

The formal definitions of these relational operations is given in [Ullm83].

Retrieval involving a subquery

As an example, consider . Dates well-known supplier-parts-shipment database [Date86].

Get supplier numbers for suppliers who are located in the same city as supplier s1.

```
supplier [snum]
where city =
        ( supplier [city]
          where snum = s1 );
```

Get suppliers with above average status.

```
supplier
where status >
        ( average( supplier [status]) );
```

Query involving Set operations

Get part numbers for parts that either weigh more than 16 pounds or are supplied by supplier s2 (or both).

```
(parts [pnum] where weight > 16)
++
(shipment [pnum] where snum = s2);
```

Views

Views in DEAL are represented by functions (see the following section for more detailed discussion of functions). Like functions, views are not executed when they are created but they are merely stored in the system. For example, london suppliers could be defined as a view as follows:

```
london_supplier() {
        london_supplier := supplier where city = 'london';
}
```

where '{' and '}' are used to indicate the beginning and end of the function body.

Unlike SQL, any relational expression is permitted in the definition of a view in DEAL.

2.3.2. Functions and Recursion

The power of any query language is considerably enhanced by allowing constructs for looping and conditional execution. In the past this was achieved by embedding the query language into a high level programming language such as PL1 or PASCAL. In DEAL however these constructs and some others such as assignments and print statements are incorporated in the query language. Functions may also be defined, allowing recursive queries to be expressed. Given the built-in function first (first is function which takes a relation and returns a relation with a single tuple which is the first tuple of the original relation) we can define the function rest as follows:

```
rest(x:rel) {
     rest := x -- first(x) ;
}
```

where (--) is difference.

Another example which demonstrates how DEAL can be used for answering queries which cannot be handled by conventional query languages is illustrated below.

Given the relation parents, we can retrieve the paternal ancestors of a given person by using the following two functions which use recursion.

parents(dad,mum,pname)

dad	mum	pname
derek	jane	joe
fred	mary	jim
greg	alison	jane
jim	beth	derek
john	joan	beth

It is assumed that people's names are unique.

```
father(x:char) {
        father := parents [dad] where pname = x;
}

ancestor(x:char) {
        if (empty(father(x))) ancestor := null; else {
                ancestor := father(x) ++ ancestor(father(x));
        }
}
```

where (++) is union and (--) is set difference. null is a built-in relation with no tuples and arbitrary columns.

The call ancestor('joe') will then return a relation with three tuples which are paternal ancestors of joe i.e. derek, fred and jim.

Other complicated functions can also be defined with the help of built-in functions first and empty. For example there follows two versions of the aggregate function max. The first version is iterative and the second version is recursive.

```
max(x:rel) {
        MAX := 0;
        if (empty(x)) max := 0;
        while ( not(empty(x)) )
                f := first(x);          # is used to cast a relation
                if (#f > MAX) {         with one tuple and one attribute
                        MAX := #f;      to its value
                }
                x := x -- f;
        }
        max :=  MAX;
}

max(x:rel) {
        if (empty(x)) {
                max :=  0;
        } else {
                if (#first(x) > max(rest(x))) {
                        max :=  #first(x);
                } else max :=  max(rest(x));
        }
}
```

For example the query "find employees whose salary is greater than max salary of associate profs." is

```
employee where salary > max(employee [salary]
                            where position = 'associate');
```

2.3.3. Parts Explosion Problem

The parts explosion problem, which arises quite frequently in a

practical context, is well known as a problem that is beyond the capabilities of classical relational algebra. It is also well known that it is impossible to formulate a parts explosion as a single expression in the relational algebra or relational calculus. The current relational query languages such as SQL or QUEL therefore can not handle this problem without some comparatively major extensions [Date86].

In DEAL however, the parts explosion problem can be solved using a set of user-defined functions which use recursion.

The following two relations parts and link are assumed:

```
parts (pnum,cost)
link(supp,inf,qty)
```

A given part may contain any number of other parts as immediate components and may itself be an immediate component of any number of other parts. In other words, relation link represents a many-to-many relationship between parts and parts.

Figure-2.3 is an extension of relations parts and link.

parts

pnum	cost
1	10
2	15
3	20
4	12
5	8
6	15
7	20
8	30

link

sup	inf	qty
1	2	2
1	4	5
1	7	5
2	4	3
3	6	3
4	7	5
5	3	1
5	6	8
6	1	9

Figure 2.3: Part Explosion Database

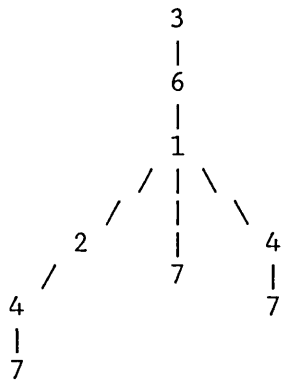
given the above relations, the parts explosion problem can now be stated as follows:

- Find the total cost of some given part to all levels

Or equivalently:

- Produce the bill of material for any given part.

The relation 'link' can be regarded as a collection of trees, for example part number 3 could be represented as:



To produce the bill of material for a given part, one would evaluate each branch of the tree completely and sum the cost over all branches.

The DEAL functions used to perform the task are as follows:

```
cost_of(pno:int) {
        cost_of := parts [cost] where pnum = pno
}

allparts(pno:int,z:rel) {
        allparts := z where sup = pno;
}
rest(pno:int,m:rel) {
        rest := m -- first((allparts(pno,m)));
}
quant(pno:int,z:rel) {
        quant := first((z [qty] where sup = pno));
}
infer(pno:int,z:rel) {
        infer := first((z [inf] where sup = pno));
}

link_cost(pno:int,z:rel) {
        if (empty(allparts(pno,z)) ) {
                 link_cost := cost_of(pno);
        } else {
                link_cost := quant(pno,z) * link_cost(infer(pno,z),z)
                                + link_cost(pno,(rest(pno,z)));
        }
}

total_cost(pno:int) {
        total_cost := link_cost(pno,link);
}
```

In [Date86] two approaches based on extending a relational language such as SQL or QUEL to deal with the parts explosion problem are given. These highlight the inadequacy of current programming languages to deal with the parts explosion problem.

2.4. Temporal Data Models

In recent years, a number of well-known data models have been extended to handle time [Wied75,Klop81,Ben-82,Clif83,Ande83,Schi85,Aria84,Gadi85,Tans85,Clif85] and [Aria86]. The majority of the models are relational, except TERM [Klop81] which extends the Entity-relationship model; EACL [Ande82,Ande83], which extends Abrial's binary relational model; TOD [Wied75], which is based on entity-attribute-value triples; CSL [Breu79], which is based on Falkenberg's object-role model, described in [Bube77]; and THM [Schi85], which extends a semantic data model based on entities, relationships, and events.

There are at least two possible approaches to the development of a model for HDBs. One is to extend the semantics of well established models like relational or entity-relationship or binary, to directly incorporate time. The other is to base HDBs on the static model, with time appearing as an additional domain type. The first method has been successfully applied by [Clif83].

The second method has been successfully applied by many researchers to the relational model [Ben-82,Snod84]. In these models, each relation is extended to contain additional temporal domain(s). This is normally referred to as time-stamping. Several benefits come from such an approach. The relational model is simple and is based on the well developed formalism of set theory and predicate calculus; database models directly incorporating time are significantly more complex, and are based on newer and less developed logics such as Montague, multiple

transition, and temporal logics. Finally, a temporal database based on the relational model can be implemented directly on conventional relational database management systems, utilising the significant results obtained in this area in the past decade.

There are some similarities among these temporal data models. They almost universally assume that time is linear and discrete (represented by dates or clock times), and that points are the primitive temporal entities; intervals being defined by their start and end-points.

The differences between the models are mainly in temporal realisation, i.e. how temporal information is associated with other components of the model (entities, tuples, events, etc.), and in whether one or both of the time of assertion and time of reference are supported. A taxonomy of temporal data models based upon whether they support only time of assertion, only time of reference or both, is given in [Snod85].

The Historical Database (HDB) model [Clif83] aims at providing a rigorous semantic basis for temporal databases. This is achieved through a linguistic view of databases and by exploiting the analogy or parallelism between HDB queries and high-order intensional logic operations.

Time is captured in the HDB through a set of time-stamped relation instances i.e., a database state, snapshot, or version. Each such relation instance is "completed" so that every object that ever existed is represented in it, which allows the alignment of parallel slices (states) to form a regular cube. The "historical relation," which is the

basic data construct in the HDB model, is defined as the flat union of all the completed instances.

HDB uses some designated internal attributes as the basis for incorporating explicit temporal semantics into the model, that is, relation schemes generically include two implicit attributes:

(1) a time-stamp attribute that marks the set of all tuples in a relation instance (called 'State')

(2) a boolean indicator for determining the existence of an object (called 'Exist').

HDB augments the set of explicitly defined database states with a function that explicates the mechanism by which data values between states are derived.

In models which are based on the relational model, time stamping may be done at two levels:

(a) tuple [Snod84,Ben-82,Jone79]

(b) attribute [Clif85,Tans85,Gadi85]

In [Clif85] time points are used for time-stamping attributes. This implies that the relations in this model are not in the First Normal Form (1NF), as the domain of time-varying attribute is non-simple. Many of the subtleties inherent in a temporally oriented database are discussed, in the context of developing extensions to the basic relational operations. Furthermore two new operations "time-slice" and

"when" which operate on time relations are defined.

In [Tans85] however, time intervals are used for time-stamping attributes. Each attribute value is stored along with a time interval over which it is valid. Relations in this model like relations in [Clif85] are not in 1NF, as the domain of the time-varying attribute is non-simple. The time values in this model are explicit; each time-varying attribute is really a triplet (A, L, U), where the value of A is valid over an interval whose start and end-points are given by the values of L and U respectively. The attributes can be atomic, set-valued, triplet-valued or set-triplet-valued. New operations are then defined to extract information from historical relations. These operations are pack, unpack, triplet-decomposition, triplet-formation, slice, and drop-time, which can convert a relation from non-1NF to 1NF and vice versa.

The homogeneous temporal model [Gadi85] is yet another model where time-intervals are used for time stamping attributes. The main difference between this model and that of Tansel's is that while Tansel's approach allows different attribute types to coexist in the same relation, in this model all attributes in a tuple are required to have the same time reference, i.e. the temporal domain within a tuple does not vary from one attribute to another, and also the use of time-stamps even for non time varying attributes. Unlike [Tans85] time values are implicit (hidden from the user) in this model.

A new algebra and calculus which includes temporal expressions is also developed and their equivalence is shown.

The Time Relational Model (TRM) [Ben-82] offers an elegant solution to some of the complexities involved in incorporating time aspects in a relational model. The Time-Relation (TR), the data construct in this model, is the unordered collection of an exhaustive set of objects' histories. TRM captures time in five mandatory temporal attributes, associating each tuple with various periods of time. These fixed time attributes are

(1) the recording time, the subject of the "as-of" statement

(2) the time when the statement included in the tuple starts to take effect

(3) the time when the tuple is marked as "physically" deleted

(4) the time the statement in the tuple ceases to be effective, that is, it is "no longer valid" and therefore logically deleted

(5) the time when the tuple is marked "void" for reasons of input error of a technical nature.

The last indicates that the statement contained in the tuple should be ignored in any database query as of a later point in time.

TRM allows users to view the database content as of a predetermined date. The primary access mechanism to data is a procedure that extracts a regular flat relation from a TR. This extracted time slice includes tuples that are "in effect" at a designated point in time, say p1, as of a possibly different point in time p2.

The model of [Aria84] focuses on a data model that provides a mechanism for capturing certain semantic qualities of time, specifically the history of objects is retained. Two different types of time attributes are identified and their implications to databases are examined. The Time Stamp Attribute (TSA) includes attributes like time of storing record in the database, the time an equipment malfunction happened, was reported, and so on. The Time Related Attributes (TRA) are attributes that contain some time data.

All tuples in the model have an intrinsic time attribute called Recording Time (RT). The intuitive semantics of RT is "when the database 'became aware' of the occurrence of the event", i.e., the time the information was stored in the database (transaction time). RT is a TSA.

To keep track of tuples as they change in time, each tuple is assumed to have a unique 'Temporal Invariant Key' (TIK). The TIK therefore extends Codd's surrogate [Codd79] by explicitly identifying objects over time.

To provide database users with a convenient way of dealing with time, a set of canonical views and operations on then are then developed.

The difficult problem of joins over time is also been addressed in this paper.

Both methods of time-stamping have their advantages and disadvantages. For example, time-stamping relations at attribute level is the more natural way of incorporating time into databases, but there are

other reasons:

(1) it is a closer abstraction of the real world.

(2) it eliminates data redundancy resulting from duplication of unchanged attributes which is inherent in the case of tuple time stamping.

(3) when a change occurs, it is generally to the value of an individual attribute, not to the values of all the attributes in a tuple;

(4) some attributes are inherently not time-varying;

(5) attributes vary over time in different ways;

(6) different attributes may be measured/recorded at different rates

The disadvantage of time-stamping at attribute level is that relations in the model are no longer in first normal form, since the domain for time-varying attributes is non-simple. Clearly this model can not be implemented directly on conventional relational database systems.

In all the models so far, topological relationships are not recorded independently, but are derived from the metric information inherent in the concrete representations. Anderson [Ande82] and Schiel [Schi85] however allow the definition of abstract temporal entities, ordered by a topological "before" relation. In Anderson's EACL model metric information is attached to the abstract entities via explicit relations that map the entities to positive real numbers. Scheil's THM model, however, permits both points and intervals as primitive. Inter-

vals may be derived from topological relations on points or on other intervals, e.g., if t is a point, then the following defines an interval:

$$BEF(t) = \{ s \mid s \text{ is before } t \}$$

Complex temporal entities can be built up from these entities by boolean operations. Anderson and Schiel also support periodic entities. A periodic entity is modelled as a set of intervals of equal length together with a "period" that specifies the separation between any two successive members of the set.

2.5. Historical Query Languages

A great deal of work has been done in the area of query languages for historical databases. A number of conventional query languages have been extended to cope with processing and retrieval of historical data. A historical query language should have the ability to retrieve time varying information; distinguish between queries which are applied to the historical database and the current database; and also allow a user to pose queries which reveal the changes of some data items over time ("walk through time" queries).

There have been two categories of query language for HDBs. Languages in the first category are extensions of static query languages with "time-view" constructs, i.e. constructs for extracting a static time slice (snapshot) of the database. Once the snapshot has been constructed, the user writes static queries against it. DATA [Aria82], Ben-Zvi's extended SQL [Ben-82] and Gemstone [Cope84] are examples of languages in the first category.

Languages in the second category support historical queries, i.e. queries that operate over entities (tuples, records) from different states. Queries like "List employees who earned more in 1984 than in 1985" can be supported by these historical query languages. LEGOL 2.0 [Jone79], HTQUEL [Gadi85], TQUEL [Snod84], TOSQL [Aria86] and Tansel's language (based on the relational algebra extended to handle non-1NF relations [Clif85]) are examples of such languages.

2.5.1. An Overview of Temporal Languages

In this section an overview of some of the languages mentioned above is given.

2.5.1.1. TQUEL

TQUEL [Snod84] is a superset of QUEL [Held75], the query language for INGRES [Ston76]. An important goal in the design of TQUEL was that it be a minimal extension, both syntactically and semantically, of QUEL.

TQUEL supports the expression of historical queries by augmenting the retrieve statement with a valid clause to specify how the implicit time domain is computed, and a when clause which is the temporal analogue to QUEL's where clause, to specify the temporal relationship of tuples participating in a derivation. These added constructs handle complex temporal relationships such as start of, end of, precede and overlap.

TQUEL enjoys the rigorous basis of QUEL and is constructed to conform to the tuple calculus [Date86], which provides the basis for the formal interpretation of its semantics. Nonetheless, TQUEL is not based on an explicit temporal data model, and the focus is mainly on the compiler aspects of a temporally oriented query language. As a result the temporal properties of data are only discussed in the context of valid expressions in the sequencing of events and periods of time.

The overlap operator specifies that the events and/or intervals overlap in time.

Example 1) What tournaments were won by Americans in 1984?

```
range of s is Nationality
range of a is Tour_results
retrieve into AmericanWon(player_name = s.player_name)
where s.player_name = a.player_name and
      s.country = "American"
when  a overlap "1984"
```

Unary operators start of and end of return the lower and upper bounds of a time interval. Sequentiality may be tested with the precede operator in TQUEL.

Example 2) Who has been No. 1 for the last two years?

```
range of s is Ranks
retrieve into NumberOne(player_name = s.player_name)
where s.rank = 1
when  (start of s) precede "jan,1,1984" and
      "jan,1,1986" precede (end of s)
```

When queries which return a time value in TQUEL are answered using the valid clause.

Example 3) When was McEnroe number 2 in ranking.

```
range of s is Ranks
retrieve MacTwo(player_name = s.player_name)
where s.player_name = 'McEnroe' and
      s.rank = 2
valid at start of s
```

Example 4) What was Becker's ranking when he won at Wimbledon in 1986?

```
range of s is Ranks
range of f is Tour_results
retrieve into Rank(rank = s.rank)
where f.player_name = s.player_name and
      f.player_name = 'Becker' and f.result = 1 and
when  f overlap s
```

TQUEL also provides a set of aggregate functions. These functions are analogues to QUELs aggregate operators max, min, avg, count, sum and any.

Example 5) How many times have Americans won at Wimbledon?

```
range of t is Tour_results
range of n is Nationality
retrieve into AmerWimWin(name = t.player_name)
where   n.player_name = t.player_name and
        n.country = 'American' and
        t.tour_name = 'Wimbledon' and
        r.result = 1
when t overlap n

range of w is AmerWimWin
retrieve NumAmerWimWin(number = count(w.name))
```

2.5.1.2. HTQUEL

HTQUEL [Gadi85] (Homogeneous Temporal Query Language) is another temporal query language which is based on QUEL [Held75]. The database model which HTQUEL is built on is the Homogenous data model (the model is an extended relation model and it is termed Homogenous because the temporal domain within a tuple does not vary from one attribute to another).

A retrieve statement in QUEL is a syntactically correct retrieve statement in HTQUEL.

In addition, a retrieve statement in HTQUEL has an optional "during v" clause, where v is a temporal domain. This is the time slice operator of HTQUEL.

Example 1) Who won at Wimbledon between 1980-1986?

```
range of m is Tour_results
retrieve (m.player_name)
        where m.result = 1 and m.tour_name = 'Wimbledon'
during [1980,1986]
```

There are also a number of temporal functions (Temporal Navigation is the termed used in HTQUEL) which operate on temporal domains in HTQUEL. For example the function FirstInterval, given a temporal domain, returns the first interval of that domain.

```
FirstInterval(v) = [1,1] where v = [1,1] U [5,7] U [9,10]
        ( 1,1,5,7,9,10 are instances)
```

To get an instant of a time domain, HTQUEL provides a set of functions which are analogous to interval functions. For example

```
FirstInstant(v) = 5 where v = [5,7] U [9,10]
```

The number of instant in a temporal domain could be calculated by the function length. For example

```
length(v) = 3 where v = [3,5]
```

this is because an instant represents an interval of unit length.

Example 2) What was Beckers ranking when he won Wimbledon for the first time?

```
range of m is Tour_results
retrieve into v(tdom(m))
        where m.player_name = 'Becker'
range of s is Ranks
retrieve (s.rank)
        t = FirstInstant(FirstInterval(v))
        print (s.rank(t))
```

tdom(r) denotes the union of temporal domains of all tuples of relation r.

HTQUEL also supports queries which return time-domains as their result i.e. when queries.

Example 3) When was the U.S.Open won by a non-American?

```
range of t is Tour_results
range of n is Nationality
retrieve tdom(t)
where   t.tour_name = 'U.S.Open' and
        t.result = 1 and
        n.country != 'American'
```

Temporal assignment is another feature of HTQUEL as illustrated in example 2. Set operations on time-domains are also permitted in HTQUEL.

2.5.1.3. TRM

In Ben-Zvis [Ben-82] TRM model, an extended SQL [Cham76] has been used as the query language for the temporal model. The query language uses a time view mechanism, i.e. providing a mapping from three-dimensional relations (time-relations) to two-dimensional relations. The extension on SQL is defined in such a way that it would not penalise users not interested in having access to the historical data.

Although not complete, this straightforward approach to extending SQL to include the notions of time and time-view demonstrates the simplicity of such an extension to a given query language.

The time-view has the following format

```
                                    Now                                  Now
TIME-VIEW [ E-Time =          Current time value    [ AS-OF =        Current time value  ] ]
```

AS-OF could be used in queries where the user might want to view the database content as of a predetermined date. The primary access mechanism to data is a procedure that extracts a regular flat relation from a time relation.

Example 1) List players who have won at Wimbledon since 1980.

```
TIME-VIEW     E-TIME = 1.1.1980
SELECT  player_name
FROM    Tour_results
WHERE   tour_name = 'Wimbledon' and
        result = 1
```

The query language also provides a set of time-aggregate functions. For example T-AVG, T-SUM, T-FIRST i.e. first effective time value, T-LAST and so on.

Example 2) Whose current rankings have been dropped with comparison to some previous ranking?

```
SELECT player_name
FROM   Ranks
WHERE  T-MAX(rank) != T-LAST(rank)
```

Temporal domain retrieval in this language is achieved by a special function called period. The result of the function is a set of time-periods.

Example 3) How long has Lendl been number 1 now?

```
periods(Ranks.name WHERE player_name = 'Lendl' and rank = 1)
```

Another feature of the language is that all the changes made in a given time-relation for a given time period can be retrieved with ease.

Example 4) List all the changes made to the ranks relation during 1986 and for each change list the player_name, rank and the associated effective-time-start.

```
SELECT  player_name,rank,E-start(rank)
CHANGES FROM 1.1.86 TO 31.12.86
FROM    Ranks
```

where E-start is the effective-time-start.

2.5.1.4. HQUEL

HQUEL [Tans87] (Historical Query Language) is a query language for an extended relational model which supports the time dimension at the attribute level. HQUEL, is designed as a minimal extension to QUEL. HQUEL includes new range declarations, and set theoretical expressions and conditions. Aggregate functions of QUEL are also extended to accommodate historical data. HQUEL is based on a modified relational model which uses attribute time stamping and non-first normal form relations.

Attributes of relations in HQUEL are tripled-valued. <[l,u],v> is a triplet that denotes the value v is valid over the interval [l,u]. Triplets may be used to represent the attributes which obtain a single value in the database history, like player_name, etc. Sets of triplets, on the other hand, are used to model the attributes which assume more

than one value in the database history.

For example the Ranks relation is represented in HQUEL as follows:

player_name	*$rank
Lendl	{<[1/6/84,1/9/85],2>, <[2/9/85,n],1>}
McEnroe	{<[1/6/84,1/9/85],1>, <[2/9/85,n],2>}
Becker	{<[15/7/84,1/1/85],4>, <[2/2/85,n],3>}

A set-valued attribute is prefixed with an '*'. Similarly, a tripled-valued attribute is prefixed with an '$'. The symbols '*$' precede a set tripled-valued attribute. $Al, $Au, and $Av represent lower bound, upper bound, and the value components of triplets, respectively for the attribute $A.

Example 1) What was Becker's ranking in June 1984?

```
Range of R is Ranks
Range of r is R.*$rank
Retrieve inot Beckerrank(r.$rank(V))
        where   R.player_name = 'Becker' and
                r.$rank(L) <= 6/84 and
                r.$rank(U) > 6/84
```

Set operations could also be performed on time-intervals as shown in the following example.

Example 2) What was Lendl's ranking when he won at the French open?

```
Range of t is Tour_results
Range of x is t.*$result
Range of R is Ranks
Range of r is R.*$rank
Retrieve into Lrank(r.$rank(V))
where t.player_name = R.player_name and
      t.player_name = 'Lendl' and
      t.tour_name = 'French open' and
      x.$result(V) = 1 and
      x.$result(T)   r.$rank(T) != 0
```

Aggregate function definitions are modified for historical data and a rich set of functions is proposed to handle the subtle semantics issues created by time.

Example 3) For each player, find the highest rank he has ever had.

```
Range of R is Ranks
Range of r is R.*$rank
Retrieve into Eranks(R.player_name,r.$rank)

Range of x is Eranks
Retrieve into highestrank
        (x.player_name,Hrank= max(x.$rank(V) by x.player_name)
```

2.5.1.5. TSQL

TSQL [Nava86] is a superset of the query language SQL, which has been extended by introducing several new semantic and syntactic components. All legal SQL statements are also valid in TSQL, and such statements have identical semantics in the absence of a reference to time.

Example 1) What was Becker's ranking in June 84 ?

```
SELECT  rank
FROM    Ranks
WHERE   player_name = 'Becker'
WHEN    '010684' DURING  Ranks.INTERVAL
```

The postfix operator INTERVAL is used to specify the time-interval of the tuple of the underlying relation.

Example 2) List all changes of ranks during 80-86 for all players having Jones as their manager.

```
SELECT  Rnaks.player_name,Ranks.rank
FROM    Ranks,Manager
WHERE   (Ranks.player_name = Mangers.player_name) AND
         (Mgr = 'Jones')
WHEN    Rnks.INTERVAL OVERLAP Manager.INTERVAL
TIME-SLICE year [1980,1986]
```

TSQL also supports retrieval by ordering.

Example 3) Get the fifth change of manager for Becker.

```
SELECT  player_name, 6th mgr
FROM    Managers
WHERE   player_name = 'Becker'
```

Example 4) Name players whose second manager was Smith.

```
SELECT  player_name
FROM    Managers
WHERE   SECOND mgr = 'Smith'
```

Moving Time-Window is a new concept in TSQL. It can be viewed as a moving time-interval (since only the length of the interval is specified), which applies to a temporally ordered relation to provide aggregate information referring to this interval, while the interval is attempted over the entire life span of a relation. These aggregate functions apply to the group formed by the tuples which fall within this interval length.

Example 5) Find the two-year time-period in which the rank of players increased the most.

```
SELECT  player_name,MAX(LAST rank - FIRST rank), WINDOW
FROM    Ranks
MOVING  WINDOW 2years
```

Example 6) List players who won more than 10 tournaments in any six months during period 1980-1986.

```
SELECT  player_name
FROM    Tour_result
WHERE   result = 1
MOVING  WINDOW  6months
GROUP BY player_name
HAVING  COUNT(*) > 10
TIME-SLICE year [1980,1986]
```

2.5.1.6. LEGOL 2.0

One of the early query languages which addressed the concept of time is LEGOL 2.0 [Jone79], which is based on relational algebra and used for writing rules which might appear in legislation. In particular it is intended for database applications where the correct handling of time is an important issue. LEGOL 2.0 is based on a relational database model in which each relation has three types of attributes : identifier attributes that name entities, a characteristic attribute that designates a property of an entity, and time attributes ("start-time" and "end-time") that designate a period over which an entity had a property.

The LEGOL 2.0 algebra also includes a set of time_related operators. They are:

(a) Time intersect (while)

(b) One sided time intersect (since, until)

(c) Time difference (while not)

(d) Time union (or while)

(e) Time set membership (during)

The most important of the time operators is 'time intersect' which is a special join operator. This specifies a matching condition on the non-temporal attributes of two relations, but also involves a implicit intersection of the period of time during which the condition is met by two matching tuples.

2.6. Summary

The literature surveyed in this chapter has certainly influenced this work. However, the following problems are apparent:

(1) Although the use of non-1NF data models can successfully capture the independent behaviour of attributes, it poses the major implementation problems inherent in all non-normalised relations; to our knowledge, no large commercial systems have been developed for this model. A redefinition of all existing relational algebra operations is required for non-1NF relations. It seems that every relation in non-1NF must be unnested (that is, converted into 1NF) before any query involving a join or comparison between two time varying attributes of the same relation can be answered. In the case of several time varying attributes with the same temporal behaviour, this scheme would associate redundant time-stamps with every time varying attribute.

(2) The operational semantics of the some of the proposed temporal query languages is not given.

(3) The temporal query languages proposed are limited in their scope. The majority are based on query languages which themselves have proved to be inadequate for conventional database systems.

(4) Except for TQUEL [Snod87], and LEGOL 2.0 [Jone79] where an implementation exists, there are no indications that the other proposed query languages are implementable in a practical sense.

In the next few chapters, we will try to deal with the above mentioned problems.

CHAPTER 3

THE TIME MODEL

This chapter demonstrates how the relational model and algebra can be extended to deal with the processing and retrieval of historical data. A new model for time and time relations is also introduced along with a proposed temporal normal form (T/NF). The changes to relational algebra operators required to handle time attributes are described. The new algebra, is a straightforward extension of the conventional relational algebra, which supports valid time; the time when an object or relationship in the enterprise being modeled is valid. Historical versions of the five relational operators union, difference, cartesian product, selection, and projection are defined and two new operators, temporal select (time slice), and when, are introduced.

3.1. Time

In the proposed model, time is considered as a set of consecutive, equidistant points.

Figure-3.1 shows the time axis, where t0,t1,t2,...tn are discrete time-points and NOW marks the current time. As time advances, the value of NOW also changes accordingly. The time unit is not specified. It may be days, hours, minutes, seconds, etc. Between two consecutive points there is a time duration which is equivalent to one time unit.

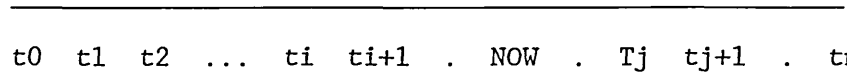


Figure 3.1: Time Axis

The time-points of time-relations can be categorised into three classes:

(1) Absolute is a fixed, one-off point on the time axis. Its time domain is a constant time. For example "1st of July 1985", "12:45 12/9/1986".

(2) Recurring time points are points which occur regularly on time axis. The granularity of these points may be different. For example first day of each week (day), Third week of each month (week)

(3) Relative time points are points which occur relative to an absolute time point. For example "A week from now", "Yesterday".

Time-interval is another concept in our time model. A time-interval is basically defined in terms of two time points start and end ([s,e] where $s \leq e$). When both these time-points are known the time-interval is said to be closed. If either the starting or ending time-point is unknown, the time-interval is said to be open. In our model time-intervals are only allowed to be open at the end.

The time interval between successive times may be arbitrarily

short, depending on the application area envisaged for the database.

In our model, a time-point is regarded as a special case of time-interval which has an infinitesimal duration which can be thought of as being negligible, or even zero valued.

The following assumptions have been made about the time domain in this model:

(a) Time is defined on a temporal axis and is discrete, i.e. each point of the temporal axis can be mapped into an integer number.

(b) Time is linear so

for two time points t1 and t2

t1 < t2 or t1 = t2 or t1 > t2

(c) The quantum of time is the second, which is considered to be a reasonable granularity for a wide range of applications.

(d) Time is represented as a natural number i.e. a time point is identified by a (positive or zero) integer that indicates the number of time points from the origin of the temporal axis. The choice of representing time this way is dictated by implementation requirements.

Similar assumptions has also been made about time in [Barb85].

In our model, we have chosen to represent time primarily though not exclusively in terms of intervals. In so doing we have followed the suggestions of James Allen [Alle83] that intervals are the most

computationally natural way of representing time. Also in terms of presentation i.e. display of time varying information, intervals are better.

An interval is equivalent to a set of time points. For example

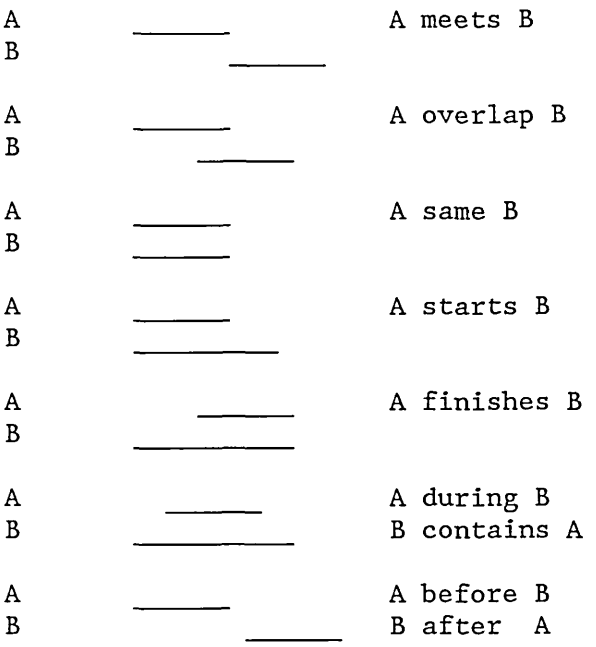
$$[\ P0\ ,\ Pn\ ] = \{\ t \mid P0 \leq t \leq Pn\ \}$$

where P0,P1,...Pn are set of consecutive time points

3.1.1. Time Operations

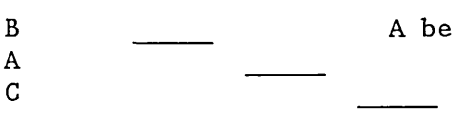
A new series of operations is developed which act upon time intervals. These operators play an important role in our temporal query language HQL which is designed to operate on our data model. The full definition of time operators is given in appendix-C.

Definition: Let A and B be two time intervals with beginning and end points. Then we can define the following binary ordering relations on A and B.



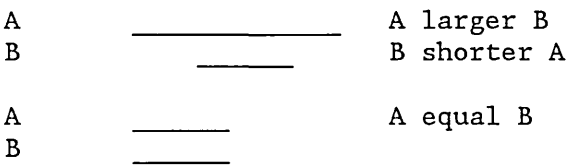
The operators before, after, during, contains, and meets are not commutative, whereas overlap, same, and finishes are.

Ordering relations of higher degree can easily be defined in terms of the above binary relations. For example



between(A,B,C) iff before(B,A) and before(A,C)

Definition: Let A and B be two time intervals, then we can define the following binary metric relations on A and B.



Definition: Let A and B be two time intervals, then the union of A and B denoted by $A \cup B$, is a set of time points belonging to A or B or both. In some cases the result may be represented as a single time interval; in other cases the result must be represented as two time intervals.

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

Example

A = [1/1/80 , 1/1/85]
B = [1/1/82 , 1/1/87]
then
$A \cup B$ = [1/1/80 , 1/1/87]

A = [1/1/75 , 1/1/79]
B = [1/1/80 , 1/1/85]
then
$A \cup B$ = [1/1/75 , 1/1/79] , [1/1/80 , 1/1/85]

Definition: Let A and B be two time intervals, then the intersection of A and B denoted by $A \cap B$, is a set of time points belonging to both A and B.

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

Example

A = [2/1/82 , 1/1/84]
B = [2/1/83 , 1/4/85]

then

$A \cap B$ = [2/1/83 , 1/1/84]

Definition: Let A and B be two time intervals, then the difference of A and B denoted by A - B, is a set of time points belonging to A and not to B.

$$A - B = \{ t \mid t \in A \wedge \neg(t \in B) \}$$

Example

A = [1/1/81 , 1/1/84]
B = [1/1/79 , 1/4/85]

then

A - B = [1/1/79 , 1/1/81] , [1/1/84 , 1/1/85]

Definition: Let A and B be two time intervals, then A and B are disjoint time intervals if

$$A \cap B = \emptyset$$

i.e., there is no overlap between A and B.

3.2. Temporal Relational Model

The relational time model consists of a collection of time relations, each one defined over some interval of time. Like any other data model, it consists of a collection of objects, operators and integrity constraints.

Before we proceed with the definition of the objects in our model, first we need to address the problem that is, where does time fit into the model?

In chapter 2 we saw that in models that are based on relational model, time could be introduced either at tuple or attribute level. We also discussed the advantages and disadvantages of both methods. In our model, for reasons of practicality and efficiency, we propose time as a component of the tuples rather than attributes.

3.2.1. Classification of Time Relations

(i) Static relations are those whose attributes' values are independent of time. These relations could be characterised by two time attributes which are valid for the life time of the entity they are representing.

(ii) Interval relations are characterised by two time attributes, start and end. These mark the limits of the effective time for which the information in the tuple may be considered to hold true.

(iii) Event relations which are characterised by a single time attribute for tuples representing instantaneous occurrences.

To unify all three classes of time relation we can treat Event relations like Interval where the end time attribute of the relation is the same as the start time.

3.2.2. Time Relations

A time relation in our model is characterised by two mandatory time attributes, start and end. Start and end attributes correspond to the lower and upper bounds of the time-interval. These mark the limits of

the effective time for which the information in the tuple may be considered to hold true.

Consider the relations Salaries and Managers shown below.

Salaries(name,salary)

name	salary	start	end
rubik	6000	1/3/80	15/1/81
rubik	7500	16/1/81	1/4/83
rubik	9000	2/4/83	19/11/87
clark	8000	1/6/82	1/1/84
jones	8500	1/1/83	1/3/85
jones	9500	2/3/85	19/11/87
smith	7000	1/1/80	12/5/81
samson	8500	1/1/83	30/1/85

Managers(dname,mgr)

dname	mgr	start	end
furniture	rubik	1/2/81	1/5/83
printing	rubik	2/5/83	19/11/87
shoe	clark	1/1/83	1/1/84
shoe	jones	1/2/84	1/4/85
clothing	jones	2/4/85	19/11/87
furniture	samson	1/4/84	30/1/85

A tuple < e, s, t1, t2 > in Salaries relation is interpreted as meaning that employee e earns a salary s, and this fact is effective from t1 to t2; that is, the salary s of e is continuously valid for the interval [t1,t2].

In the relation Salaries, the information being represented is the earning of a salary by a number of employees over a period of time. It does not show when these facts were recorded in the database, but it

shows when a particular salary becomes effective and when it ceases to be so.

When a tuple is first inserted, the end component of the interval is set to NOW, indicating that the tuple is currently valid. A delete operation on an existing tuple changes the end component of the tuple from NOW to some t1. The value of t1 is usually the current time, but it can be specified explicitly by the delete statement for a historical database. For a replace operation, a new version of the tuple augmented with an appropriate interval is inserted after a virtual delete operation is executed. For example, the above Salaries relation showing that rubik and jones received a series of salary rises.

With the definition of the time relation given above, one might still ask "why can't the user include additional columns in his relation and manage the appropriate time stamping by himself?" or "why do we need a time relation?".

The main objective of our time relational model is to provide the user with high degree of time-independence by automatically handling all time related issues, making all time stamps transparent to a database user. Thus the user need not to be concerned about access techniques or data organisation when operating on the data. Furthermore, by having the time mechanism built into the data model, many other advantages exist for the DBMS's own internal use.

3.2.3. Time Attributes

We recognise four types of attributes in our model:

(i) Constants — are attributes whose values stay constant throughout the life time of entity that they are part of. For example sex,date of birth.

(ii) User-defined time — is necessary when additional temporal information, not handled by time stamps, is stored in the database [Jone80]. The values of user-defined temporal attributes are not interpreted by the DBMS and are thus the easiest to support compared to time attributes.

(iii) Time Varying — as the name suggests, are attributes that change with time. For example salary, position.

(iv) Time Attributes — these attributes which define the interval for the effective time of an entity i.e. start and end.

3.2.4. Time Relation Normalisation

The design of a time-relational database introduces a number of problems concerning the normalisation of data.

In models, where relations are time stamped at attribute level [Clif85] a time relation might have more than one time-varying attribute. Each of these time-varying attributes might also have different start and end times. For example, the tuple

< e, s[t1,t2], m[t3,t4], p[t1,t4] >

where e is the primary key, has three time varying attributes each with different time intervals.

In our model however, where time-stamping is done at tuple level, only one time interval is used for the entire tuple. For example, the tuple

< e, s, m, [t1,t2] >

with two time-varying attributes s and m, means the attributes e, s, and m are valid throughout the interval [t1,t2], in other words the following is a valid description with attribute-level time stamping

< e, s[t1,t2], m[t1,t2] >

However, if the time-varying attributes s and m do not change at the same time then the interval [t1,t2] would not represent the correct life-span of its attributes. This would in turn create update and retrieval anomalies.

To overcome these problems, we propose time normalisation which involves dividing a relation which is normalised in the conventional way into a set of relations, each of which has attributes whose individual start and end times are the same.

3.2.5. The Need for Time Normalisation

One of the important properties of relations is that all tuples in relations are independent of each other. By definition, a relation is a set; its elements (tuples) must be independent of one another. However,

in an unnormalised time-relation, tuples might have incomplete information about the life-span of their attributes, therefore tuples become semantically dependent on other tuples for determination of complete information.

Consider the Sal-Dept relation which has two time-varying attributes Dno and Sal and a constant attribute Emp-name. The tuple

< Lloyd 50 5000 1/2/81 18/4/83 >

has the start and end values 1/2/81 and 18/4/83 which do not refer to the time-varying attribute Dno; they refer to the effective time-period of the time-varying attribute Sal with the value 5000. Therefore, this tuple has incomplete information regarding the life-span of the attribute value '50'. To find the complete set of information about the time-varying attribute Dno, one then has to look at the other tuples of the relation.

Sal-Dept

Emp-name	Dno	Sal	start	end
Lloyd	50	5000	1/2/81	18/4/83
Lloyd	50	7000	19/4/83	22/9/84
Lloyd	51	7000	23/9/84	1/1/85
Smith	55	6500	14/3/80	6/11/81
Smith	55	7500	7/11/81	-
Jones	51	7400	1/1/81	31/12/85
Jones	51	8000	1/1/86	-

The Sal-Dept relation can now be decomposed into two relations Salary and Department for time normalisation as follows:

Salary

Emp-name	Sal	start	end
Lloyd	5000	1/2/81	18/4/83
Lloyd	7000	19/4/83	1/1/85
Smith	6500	14/3/80	6/11/81
Smith	7500	7/11/81	-
Jones	7400	1/1/81	31/12/85
Jones	8000	1/1/86	-

Department

Emp-name	Dno	start	end
Lloyd	50	1/2/81	22/9/84
Lloyd	51	23/9/84	1/1/85
Smith	55	14/3/80	-
Jones	51	1/1/81	-

These decompositions represent information about the life-span of an attribute in a compact and concise manner. Time normalisation ensures that the life-spans of a tuple and its attribute-values are the same.

3.3. Time Relational Algebra

Time Relational Algebra (TRA) is an extension of the relational algebra. The TRA should possess a number of properties according to the following paradigm, which has also been proposed by Clifford [Clif85] :

(a) The algebra should be a consistent extension of normal relational algebra

(b) The algebra should, as far as possible, treat the time domain like any other domain and thus maintain the minimality of operations and

introduce new operations only where traditional relational operations cannot cope.

In section 2.1 brief but concise descriptions of relational model and relational operations were given. The five basic operations were: union, difference, cartesian product, selection and projection. In the next few sections, we will look closely at these operations and extensions required to handle time-relations.

Before discussing specific extensions to the relational algebra, we must consider what kinds of time the algebra supports. There are three kinds of time that a data model may support: valid time (or effective time), transaction time, and user-defined time.

The relational algebra already supports user-defined time in that user-defined time is simply another domain, such as integer or character string. The relational algebra, however, supports neither valid time nor transaction time.

Our algebra is similar to the algebra reviewed in [McKe87], that is it is an extension of relational algebra that supports valid time. Although several extensions supporting transaction time have been proposed [Ben-82,McKe87.] we consider only extensions of the relational algebra that support valid time. Because valid time and transaction time are orthogonal, support for each type of time can be studied in isolation.

An excellent review of nine historical algebras, is given in

[McKe87], all extensions of the relational algebra that support valid time.

3.4. Notation

3.4.1. Attributes

In conventional databases, each attribute of a relation has a domain of values it can assume, allowing null values under some conditions. For example dom(A) is domain of attribute A.

Temporal attributes (TA) are attributes that contain some time data. For example our time stamp attributes start and end are temporal attributes. For any TA, T, dom(T) is an interpreted domain whose underlying domain is the integer pairs where the first element is less than or equal to the second element.

3.4.2. Relation Schemes

A relation scheme is identified with a finite set of attributes. A scheme describes a relation. From the scheme, one can deduce the degree of the relation, i.e. the number of attributes it has; and what the attributes are. Thus if S is a finite set of attributes, say S = {A, B, C}, S also denotes a relation scheme. A database schema is a finite set of relation schemes. Sch = {S1, S2, ... Sk}.

3.4.3. Tuples

In our model a temporal tuple is defined as follows:

Let S be relation scheme, T a TA, $T \in S$ and B = S - T i.e. the set of non-time attributes of the scheme S, where dom(T) is defined as

dom(T) = { (start , end) | start ≤ end }

then a historical tuple, tup(S) is defined as a member of the set

$$\left(\underset{A \in B}{X} \; dom(A) \right) \; X \; dom(T)$$

where X denotes cartesian product, and S is the scheme of a relation as defined above.

Then for every attribute A ∈ B

value (A) ∈ dom(A)

The function time is also defined as

time : tup(S) -> dom(T)

We will be using value and time throughout this chapter to access values of non-time attributes and time attributes (start and end) of a tuple respectively.

3.4.4. Relations

A historical relation R is defined as a finite set of tuples (historical) with the restriction that no two tuples in the relation are value-equivalent.

Equivalence is defined as:

∀ P, P1 ∈ tup(S) (P equivalent P1

iff ∀ a : 1 ≤ a ≤ m (value(P(a)) = value(P1(a))))

where m is the degree of the relation (excluding the time attributes).

3.5. Time Relational Operations

In the next few sections we formally define the time relational algebra. We provide the formal definitions for the five extended relational operators time-select, time-project, time-union, time-difference and time-cartesian product, and two new operators temporal select (time slice) and when.

3.5.1. Time-Select

Let R be a relation and F be a logical formula concerning the attributes in R where, to evaluate F for a tuple p, p ∈ R, we substitute the value components of the attributes of p for all occurrences of their corresponding attribute names in F.

$$\sigma(R/F) = \{\ p\ \mid p \in R \wedge F(value(p1), \ldots\ , value(pm))\ \}$$

In words the above is "the set of tuples p such that p is in R, and F holds for all value components of the attributes of p. Thus, temporal select is identical to the normal relational select operator. It is simply the set of tuples in R for which F is true.

The select operator preserves the form of the model's relations. This means the result of a select is always another time relation for any possible F.

For example

σ(Salaries / salary > 8000)

name	salary	start	end
samson	8500	1/1/83	30/1/85
rubik	9000	2/4/83	19/11/87
jones	9500	2/3/85	19/11/87
jones	8500	1/1/83	1/3/85

3.5.2. Temporal Select

The temporal selection, best known as time slice, operates on the temporal dimension of the data. Like select, temporal selection preserves the basic relational form. This means that for any possible F the result is always a time relation, whether F extracts one continuous time range (e.g. "during the year of 1982"), or it specifies a set of disjoint intervals (e.g. "during the months of January in each of the years 1972-1985").

τ(R / overlap P) corresponds to the relation R as it looked at time P and

τ(R / overlap NOW) corresponds to the current version of the relation R.

For example

τ(Salaries / overlap 1981)

name	salary	start	end
rubik	6000	1/3/80	15/1/81
rubik	7500	16/1/81	1/4/83
smith	7000	1/1/80	12/5/81

3.5.3. Time-Projection

In relational algebra the projection of a relation over a scheme eliminates certain attributes from each tuple. Duplicate tuples in the result relation are then eliminated. The TRA projection, which always involves time domains, may result in tuples which are identical apart from their associated time intervals. Where the time intervals of two such tuples are contiguous, the two tuples may be rationalised into a single tuple with the earlier start time and later end time.

Rationalisation is, in a way, similar to removing duplicates in conventional databases. It is mainly used for efficiency of storage and for a more meaningful display of time relations.

Projection preserves the temporality of the resulting relation and maintains the property that the result of an operation on a time relation is another time relation.

In order to maintain compatibility with fundamental conventions of the relational model, if two projected tuples turn out to be identical, then one is eliminated from the representation of the result relation.

Projection of relation R over k (non-time) attributes:

$\Pi_{r1,r2,\ldots rk}(R)$ is expressed as follows:

$$\{\ t^{(k+2)} \mid (\exists\ u^{m}\ )(\ u \in R \wedge value(t[1]) = value(u[r1])$$
$$\wedge \ldots$$
$$\wedge\ value(t[k]) = value(u[rk])$$
$$\wedge\ time(t) = time(u)\ \}$$

(t^{i} indicates the t has arity i).

In other words, the above project is "the set of tuples t (which we

understand to be of length k+2) such that there exists u, with u in R and value component of the tuple t[i] = u[ri] for $1 \leq i \leq k$, and the time attributes of t i.e., start and end are the same as the time attributes of u.

For example,

Π_{name}(Salaries)

name	start	end
smith	1/1/80	12/5/81
samson	1/1/83	30/1/85
rubik	1/3/80	15/1/81
rubik	16/1/81	1/4/83
rubik	2/4/83	19/11/87
jones	1/1/83	1/3/85
jones	2/3/85	19/11/87
clark	1/6/82	1/1/84

which rationalises to

name	start	end
smith	1/1/80	12/5/81
samson	1/1/83	30/1/85
rubik	1/3/80	19/11/87
jones	1/1/83	19/11/87
clark	1/6/82	1/1/84

3.5.4. Time-Join

Join is another operator which needs modification in historical databases. The time-join extends the regular join operation to identify the overlapping period of time and the resulting relation implies a temporal coexistence of the attributes. The time-interval of the resulting relation is computed from that of the underlying relations. It is

assumed that the time-stamp attributes of joined tuples are defined on the same domain.

The time-join is defined as:

$$R1 * R2\ (xp, yq) =$$

$$\{\ t^{(r+s-2)} \mid (\exists\ u^{r})(\exists\ v^{s})(\ u \in R1 \wedge v \in R2\ )$$
$$\wedge\ time(u) \cap time(v) \neq \emptyset$$
$$\wedge\ value(u[xp]) = value(v[yq])$$
$$\wedge\ value(t[1]) = value(u[1]) \wedge \ldots$$
$$\wedge\ value(t[r-1]) = value(v[1])$$
$$\wedge\ value(t[r]) = value(v[1])$$
$$\wedge\ \ldots$$
$$\wedge\ time(t) = time(u) \cap time(v)\ \}$$

For example

Salaries * Managers (name , mgr)

To explain join we look at the intermediate result first. For example if Salaries and Managers relations were joined together in a normal relational algebra, we would end up with the following relation:

S.name	S.sal	S.start	S.end	M.dname	M.start	M.end
samson	8500	1/1/83	30/1/85	furniture	1/4/84	30/1/85
rubik	9000	2/4/83	19/11/87	printing	2/5/83	19/11/87
rubik	9000	2/4/83	19/11/87	furniture	1/2/81	1/5/83
rubik	7500	16/1/81	1/4/83	printing	2/5/83	19/11/87
rubik	7500	16/1/81	1/4/83	furniture	1/2/81	1/5/83
rubik	6000	1/3/80	15/1/81	printing	2/5/83	19/11/87
rubik	6000	1/3/80	15/1/81	furniture	1/2/81	1/5/83
jones	9500	2/3/85	19/11/87	shoe	1/2/84	1/4/85
jones	9500	2/3/85	19/11/87	clothing	2/4/85	19/11/87
jones	8500	1/1/83	1/3/85	shoe	1/2/84	1/4/85
jones	8500	1/1/83	1/3/85	clothing	2/4/85	19/11/87
clark	8000	1/6/82	1/1/84	shoe	1/1/83	1/1/84

Tuples where time intervals do not overlap are eliminated. We therefore end up with the following relation.

S.name	S.sal	S.start	S.end	M.dname	M.start	M.end
samson	8500	1/1/83	30/1/85	furniture	1/4/84	30/1/85
rubik	9000	2/4/83	19/11/87	printing	2/5/83	19/11/87
rubik	9000	2/4/83	19/11/87	furniture	1/2/81	1/5/83
rubik	7500	16/1/81	1/4/83	furniture	1/2/81	1/5/83
jones	9500	2/3/85	19/11/87	shoe	1/2/84	1/4/85
jones	9500	2/3/85	19/11/87	clothing	2/4/85	19/11/87
jones	8500	1/1/83	1/3/85	shoe	1/2/84	1/4/85
clark	8000	1/6/82	1/1/84	shoe	1/1/83	1/1/84

The time stamp of the final relation is the intersection of the two time intervals, i.e. the maximum of the two start times and the minimum of the two end times.

S.name	S.sal	M.dname	start	end
samson	8500	furniture	1/4/84	30/1/85
rubik	9000	printing	2/5/83	19/11/87
rubik	9000	furniture	2/4/83	1/5/83
rubik	7500	furniture	1/2/81	1/4/83
jones	9500	shoe	2/3/85	1/4/85
jones	9500	clothing	2/4/85	19/11/87
jones	8500	shoe	1/2/84	1/3/85
clark	8000	shoe	1/1/83	1/1/84

3.5.5. When

"When" is a special kind of a projection. It is a relational operator which, when applied to a time relation returns a set of time intervals. The time intervals are the implicit time attributes of the derived relation. The result of a When operator could also be used to form a temporal expression which can serve as a component of the time slice operator.

$\Omega(R)$ is expressed as:

$$\{\ t^{2} \quad |\ (\exists\ u^{m}\ )(\ u \in R\ ) \wedge time(t) = time(u)\ \}$$

For example,

$\Omega(\ \sigma(Managers\ /\ salary > 8000))$

start	end
1/1/83	30/1/85
2/4/83	19/11/87
1/1/83	1/3/85
2/3/85	19/11/87

which rationalises to a single tuple which includes all the above start and end times as follows:

start	end
1/1/83	19/11/87

3.5.6. Set Operations

The traditional set operations are union, difference, and cartesian product (intersect can be defined in terms of union and difference). Each operation takes two operands. For all except cartesian product, the two operand relations must be union-compatible that is, they must be of the same degree, n say, and the ith attribute of each (1,2,...n) should be based on the same domain. The union-compatibility rule is imposed to ensure that the result is still a relation.

3.5.6.1. Time-union

In the relational model, the union of two union-compatible relations R1 and R2, is the set of all tuples t belonging to either R1 or R2 or both. However, in time model, a time-union is defined as a union of three sets:

- The set of tuples which are in R1 but not in R2.
- The set of tuples which are in R2 but not in R1.
- The set of value-equivalent tuples in R1 and R2 with their time stamps unioned together.

The time-union is defined as

$$
\begin{aligned}
R1 \cup R2 = \; & \{ r1^m \mid r1 \in R1 \wedge \nexists\, r2 \; (r2 \in R2 \wedge \\
& \text{value}(r1[i]) = \text{value}(r2[i]) \;,\; 1 \le i \le m) \} \\
& \cup \\
& \{ r2^m \mid r2 \in R2 \wedge \nexists\, r1 \; (r1 \in R1 \wedge \\
& \text{value}(r2[i]) = \text{value}(r1[i]) \;,\; 1 \le i \le m) \} \\
& \cup \\
& \{ u^m \mid r1 \in R1 \wedge r2 \in R2 \wedge \\
& (\text{value}(u[i]) = \text{value}(r2[i]) = \text{value}(r1[i]) \;, \\
& 1 \le i \le m) \wedge \\
& \text{time}(u) = \text{time}(r2) \cup \text{time}(r1) \}
\end{aligned}
$$

For example

Let R1 = Π_{name}(Salaries) and

R2 = Π_{mgr}(Managers)

R1)

name	start	end
smith	1/1/80	12/5/81
samson	1/1/83	30/1/85
rubik	1/3/80	19/11/87
jones	1/1/83	19/11/87

	clark	1/6/82	1/1/84

R2)

name	start	end
samson	1/4/84	30/1/85
rubik	1/2/81	19/11/87
jones	1/2/84	19/11/87
clark	1/1/83	1/1/84

then, the time-union of R1 and R2 is:

set of tuples in R1 which are not in R2

smith	1/1/80	12/5/81

union

set of tuples in R2 which are not in R1

-

union

the set of value-equivalent tuples of R1 and R2 with their time stamps unioned

rubik	1/3/80	19/11/87
samson	1/1/83	30/1/85
jones	1/1/83	19/11/87
clark	1/6/82	1/1/84

Therefore, the result of time-union is:

name	start	end
smith	1/1/80	12/5/81
rubik	1/3/80	19/11/87
samson	1/1/83	30/1/85
jones	1/1/83	19/11/87
clark	1/6/82	1/1/84

3.5.6.2. Time-difference

In the relational model the difference of two union-compatible relations R1 and R2, is the set of all tuples t belonging to R1 but not R2. However, in time model, the time-difference is defined as union of

two sets:

- The set of tuples which are in R1 but not in R2

- The set of value-equivalent tuples in R1 and R2 with their time stamps equal to the set difference of the time stamps of the value-equivalent tuples.

The time-difference is defined as:

$$
\begin{aligned}
R1 - R2 = \\
&\{ r1^m \mid r1 \in R1 \wedge \nexists\, r2 \,(\, r2 \in R2 \wedge \\
&\quad value(r1[i]) = value(r2[i]) \;,\; 1 \le i \le m \,) \,\} \\
&\cup \\
&\{ u^m \mid r1 \in R1 \wedge r2 \in R2 \wedge \\
&\quad (\, value(u[i]) = value(r2[i]) = value(r1[i]) \;, \\
&\quad 1 \le i \le m \,) \wedge \\
&\quad time(u) = time(r2) - time(r1) \,\}
\end{aligned}
$$

For example,

Let R1 = Π_{name}(Salaries) and

R2 = Π_{mgr}(Managers)

then, the time-difference of R1 and R2 is:

set of tuples in R1 which are not in R2

smith	1/1/80	12/5/81

union

the set of value-equivalent tuples of R1 and R2 with set difference of their time stamps

samson	1/1/83	1/4/84
rubik	1/3/80	1/2/81
jones	1/1/83	1/2/84
clark	1/6/82	1/1/83

Therefore, the result of time-difference is:

name	start	end
smith	1/1/80	12/5/81
samson	1/1/83	1/4/84
rubik	1/3/80	1/2/81
jones	1/1/83	1/2/84
clark	1/6/82	1/1/83

3.5.6.3. Time-cartesian Product

In relational model, cartesian product(CP) of two relations R1 and R2 is the set of (r+s)-tuples whose first r components form a tuple in R1 and whose last s components form a tuple in R2. However, if CP was applied to a pair of time-relations, the result will not be a time-relation. Therefore to maintain the spirit of the relations of the time model, time-cartesian product extends the regular CP operator to identify the overlapping period of time and the time-interval of the resulting time-relation is computed from that of the underlying time-relations.

$$R1 \chi R2 =$$

$$\{ t^{(r+s-2)} \mid (\exists u^{r})(\exists v^{s})(u \in R1 \wedge v \in R2)$$

$$\wedge\ time(u) \cap time(v) \neq \emptyset$$
$$\wedge\ value(t[1]) = value(u[1])$$
$$\wedge\ \ldots$$
$$\wedge\ value(t[r-2]) = value(u[r-2])$$
$$\wedge\ value(t[r-1]) = value(v[1])$$
$$\wedge\ \ldots$$
$$\wedge\ value(t[r+s-2]) = value(v[r-2])$$
$$\wedge\ time(t) = time(u) \cap time(v)$$

For example

Salaries χ Managers =

S.name	S.salary	M.dname	M.mgr	start	end
smith	7000	furniture	rubik	1/2/81	12/5/81
samson	8500	shoe	jones	1/2/84	30/1/85
samson	8500	shoe	clark	1/1/83	1/1/84
samson	8500	printing	rubik	2/5/83	30/1/85
samson	8500	furniture	samson	1/4/84	30/1/85
samson	8500	furniture	rubik	1/1/83	1/5/83
rubik	9000	shoe	jones	1/2/84	1/4/85
rubik	9000	shoe	clark	2/4/83	1/1/84
rubik	9000	printing	rubik	2/5/83	19/11/87
rubik	9000	furniture	samson	1/4/84	30/1/85
rubik	9000	furniture	rubik	2/4/83	1/5/83
rubik	9000	clothing	jones	2/4/85	19/11/87
rubik	7500	shoe	clark	1/1/83	1/4/83
rubik	7500	furniture	rubik	1/2/81	1/4/83
jones	9500	shoe	jones	2/3/85	1/4/85
jones	9500	printing	rubik	2/3/85	19/11/87
jones	9500	clothing	jones	2/4/85	19/11/87
jones	8500	shoe	jones	1/2/84	1/3/85
jones	8500	shoe	clark	1/1/83	1/1/84
jones	8500	printing	rubik	2/5/83	1/3/85
jones	8500	furniture	samson	1/4/84	30/1/85
jones	8500	furniture	rubik	1/1/83	1/5/83
clark	8000	shoe	clark	1/1/83	1/1/84
clark	8000	printing	rubik	2/5/83	1/1/84
clark	8000	furniture	rubik	1/6/82	1/5/83

3.6. Null-Values and Granularity

In the real-world information is very frequently incomplete. Null-values are used in databases to facilitate the automatic support of situations where information supplied by the user is incomplete [Date82].

In historical databases, however, null-values are more complicated than null-values in conventional databases, due to the time-varying nature of the attributes of the time-relations. For this reason, the value of a time-varying attribute may not just be unknown but

(a) it may become unknown due to a break in the history of an object (NULL1).

(b) it may become non-existent due to a time-slice operation (NULL2) (see example below).

(c) it may be unknown for any point in time due to no information being recorded for that object (NULL3).

For example if we time slice the relation Tour-won (see appendix-F for relation Tour-won) for first of June 85 i.e.

τ(Tour-won / overlap 1/6/85)

we end up with the following relation:

player_name	No_won
McEnroe	NULL2
Lendl	NULL2
Edberg	NULL2
Wilander	NULL2
Becker	NULL2

Here we can see that the value for number of tournaments won is unknown. The value is unknown because no value exists for that point in time (b of above).

The time model has the capability of distinguishing between different kinds of null. It is partially done by specifying the granularity of the time-varying attributes in the relation schema. The granularity can then be checked before any time slice operation.

Null-values in historical database systems have largely been ignored by researchers in this field, except for [Clif85] where it has been discussed briefly with some examples.

3.7. Integrity Constraints

One of the most important functions of any DBMS is to prevent incorrect data from being recorded in the data base.

As in a conventional DBMS, there are a number of general integrity constraints for temporal databases. These are

(a) Start is always less than or equal to end (start ≤ end).

(b) Time attributes (start, end) can not be NULL at the same time.

(c) The time attributes (start and end) of two distinct tuples which represent the same entity can not overlap if any of the non-key time attributes have different values.

(d) The primary key of a tuple will always be a composite key which is a combination of normal attribute(s) and the time attributes.

Unlike entities in conventional database systems, entities in temporal databases may have different attribute values in the same relation, representing the values of these time-varying attributes at different times.

Tuple identifiers are used as entity identifiers in conventional database systems because there is one to one correspondence between entities and tuples. In temporal databases on the other hand, the correspondence between entities and tuples is one to many. Therefore entity identifiers will be different from tuple identifiers.

(e) For a given entity, at most one tuple which represents that entity

may have an unknown end time.

(f) A time varying attribute can not be part of the key.

3.8. Summary

The time relational model along with a proposed temporal normal form (T/NF) has been developed. The time relational algebra, which is a straightforward extension of the conventional relational algebra, is defined and formal definitions for the five extended relational operators time-select, time-project, time-union, time-difference and time-cartesian product, and two new operators temporal select (time slice) and when has been given The problem of null values and granularity in temporal databases was also considered.

In the next chapter the query language HQL is introduced.

CHAPTER 4

HISTORICAL QUERY LANGUAGE (HQL)

HQL is a relational query language based on DEAL [Deen85], capable of supporting both conventional and historical queries. HQL has a user friendly syntax in which complex queries may be formulated in a natural way. With the added power of recursion and high level constructs such as while loops, if statements and assignments, HQL is more expressive than any other temporal query language that has been published to date.

In chapter 2 an overview of current historical query languages was presented. Although some of the languages described there were well defined, the problem with most of these languages is that they tend to introduce new constructs and operators which make the language more complex and in some cases intellectually unmanageable.

A criteria for designing a historical query language are given in [Snod87]. They are :

- The language must be well defined, it should have a formal semantics. Without a formal semantics, the meaning of each construct, and the interaction between constructs, is unclear.

- The language must support valid time, by "supporting valid time", we mean specifically that queries can be formulated that derive information (i.e. tuples) valid at a point in time from information

in underlying relations valid at other point in time.

- The language must support transaction time, by "supporting transaction time", we mean that as-of queries can be supported.

- The language must be implementable, this property may be demonstrated formally through a semantics based on a well-defined algebra or, practically, through an implementation.

In addition to the above criteria, the following criterion was adopted in the work described in this thesis.

- A historical query language should not be based on current conventional query languages. This is because of the lack of expressive power of existing query languages. For example user-defined functions, recursion, and iteration are essential for transitive operations on relations.

HQL has been designed in accordance with the criteria described above. Although HQL does not support transaction time at the moment, it has been shown in [McKe87] that the transaction time can be added to the query language without any difficulty.

HQL's capabilities are illustrated with number of simple and complex queries in the following sections. Comparison is also made with TQUEL [Snod84] in some queries.

4.1. Overview

HQL is designed in such a way that it allows the users to interrogate the historical database in a natural way without referring to the

time domain explicitly. Time in HQL is encapsulated in the time-spec component of the query (figure-4.1 is an overview of the HQL syntax. The complete syntax of the HQL is given in the appendix-E). This includes a well defined set of predicates which operates on time intervals. The full definition of all these time predicates is given in appendix-C.

```
query           ::=     query_expr
                |       query_expr WHEN time_spec

time_spec       ::=     time_predicate

time_predicate  ::=     binary_ordering
                |       binary_metric

binary_ordering ::=     BEFORE          time_expr
                |       AFTER           time_expr
                |       MEETS           time_expr
                |       OVERLAP         time_expr
                |       STARTS          time_expr
                |       SAME            time_expr
                |       FINISHES        time_expr
                |       CONTAINS        time_expr
                |       BETWEEN '(' time_expr , time_expr ')'

binary_metric   ::=     LONGER          time_expr
                |       SHORTER         time_expr
                |       EQUAL           time_expr
```

Figure 4.1: An Overview of HQL

The default options in HQL are defined such that a query that omits the temporal portion (i.e. the time_spec) retains the standard meaning of the corresponding DEAL query on the current data only.

4.2. HQL Constants

In addition to DEAL's numeric and string constants HQL supports temporal constants. Strings appearing in the time clause of the HQL are interpreted as temporal constants denoting a particular time interval. For example, the string '1/1/1985' denotes an interval from midnight of

Jan 1, 1985, to midnight of Jan 2, 1985; Jan 1985 denotes the entire month; and 1985 denotes the entire year 1985.

In addition, HQL also has a set of time constants which are used for time manipulation. The time constant NOW is used to refer to the current time. Y(number), M(number) and D(number) are constant intervals which refer to an interval of number years, number months and number days long.

4.3. The Time Clause

The time component of HQL consists of the keyword when followed by a time-spec. This time clause is the temporal analogue to DEAL's where clause. It specifies the temporal relationships of the the participating tuples in the derivation; i.e., it evaluates temporal predicates by examining the relative chronological ordering of tuples' time-stamps.

The syntax of the time clause requires the keyword WHEN (figure-4.1) followed by a predefined temporal comparison operator and a time expression. The time expression is either a time constant i.e. a time-point or a time-interval, or a tuple of another time relation.

4.3.1. Time Predicates

There are two types of time predicates in HQL.

(1) Binary ordering — BEFORE, AFTER, MEETS, OVERLAP, STARTS, SAME, FINISHES, CONTAINS, and BETWEEN.

(2) Binary metric — LONGER, SHORTER, and EQUAL.

A time-point is treated as a degenerate time-interval in the time clause. Therefore all operations are defined in terms of time-intervals.

The tennis database given in appendix-F will be referred to in this chapter.

Example 1) What was Becker's ranking at the beginning of 1985?

```
Ranks [rank] where name = 'Becker'
when overlap '1/1/85';
```

The same query in TQUEL is:

```
range of a is Ranks
retrieve into Becksrank(name = a.name)
when a overlap "1/1/85"
```

Example 2) Who has been number 1 for the last two years ?

```
Ranks [name] where rank = 1
when contains ( [NOW-Y(2), NOW] );
```

The time that a player was number one must contain the time interval [20/4/85, 20/4/87].

The same query in TQUEL is:

```
range of a is Ranks
retrieve into rankone(name = a.name) where rank = 1
when (start of a) precede '29/4/85'
and  '29/4/87' precede (end of a)
```

Example 3) Name players who have been number 1 for less than a year.

We can use the time predicate shorter to answer the query as follows:

```
Ranks [name] where rank = 1
when shorter Y(1);
```

This is an example of a query which can not be handled by TQUEL.

Example 4) What is the nationality of players who have won at Wimbledon?

We can first define a function which will have all the Wimbledon winners as follows:

```
WimWin() {
        WimWin := Tournaments where name = 'Wimbledon'
                                  and result = 1;
}

( Nationality (player_name, name) WimWin ) [country];
```

Here join finds tuples in the Nationality relation which their time interval overlaps with time intervals of the WimWin. In TQUEL however, one has to explicitly specify the overlap in a when clause.

```
range of a is Nationality
range of s is WimWin
retrieve into NatWimWin(Nationality = a.nationality)
where s.player-name = a.name
when a overlap s
```

Example 5) What was Beckers ranking when he won at Wimbledon for the first time?

```
Ranks [rank] where player_name = 'Becker'
when overlap start_of(first( WimWin where
                                  player_name = 'Becker' ));
```

start of and end of are two built-in functions which return the upper and lower boundaries of an interval of a tuple. In HQL, the function first, when applied to a time-relation returns the first tuple in that relation with the lowest starting time.

Example 6) Which managers started the same time as Clark became manager of Becker?

```
Managers [mgr]
when starts Managers where player_name = 'Becker'
                    and   mgr = 'Clark';
```

4.4. WHEN Operator

WHEN is another operator which has been used by all the temporal query languages. WHEN is used for retrieval of temporal domains of relations. This unary operator on relations, unlike the other relational operators, yields as a result a set of times rather than a relation. It can also be used to form temporal expressions which can serve as a components of a time-slice or a selection operation. In general, the result of a WHEN may not always be an interval; it may well be a set of disconnected intervals.

In HQL Time-of is used for the retrieval of temporal domains of relations. But unlike the WHEN operator in [Clif85] it yields as a result a relation which contains a set of times rather than simply a set of times.

Example 1) When was the U.S. open won by a non-American ?

We can first define a function which will have all the U.S. winners as

follows:

```
USWin() {
        USWin := Tournaments where name = 'U.S.'
                                    and result = 1;
}

Time-of( USWin( player_name, player_name )Nationality );
```

4.5. User-defined Functions

In all of the query languages so far, users are provided with a set of built-in functions. Although these functions might be complete, they can not be changed, or be tailor made for a specific application. For instance in HTQUEL [Gadi85] a set of functions are defined for temporal navigation. These functions can not be guaranteed to be the only ones required by users.

A major strength of HQL, however, is that users are allowed to define their own functions. For instance to answer the query " Who was number 1 longest " we can define the function longest, which can also be used for other queries as well, as follows:

```
rest(x:rel) {
        rest := x -- first(x);
}

abs(x:int,y:int) {
        if ( x > y ) {
                abs := x - y;
        } else {
                abs := y - x;
        }
}

distance(x:rel) {
        distance := abs(start_of(x),end_of(x));
}

mainlongest(x:rel,y:rel) {
        if (empty(x))
                mainlongest := y;
        } else {
                tup := first(x);
                if (distance(tup) > distance(y) ) {
                        mainlongest := mainlongest(rest(x),tup);
                } else {
                         mainlongest := mainlongest(rest(x),y);
                }
        }
}


longest(x:rel) {
        longest := mainlongest(x,null);
}
```

where empty is a built-in function which when applied to a relation returns true if the relation is empty, false otherwise. null is a built-in relation with no tuples and arbitrary columns.

```
        longest( Ranks [player_name] where rank = 1 );
```

will return the player name who has been number 1 the longest.

Example 1) Who is the longest serving manager with a player now.

```
longest ( Managers [mgr] );
```

Duration is another example of a user-defined function, which demonstrates how easily functions which are usually built-in can be defined as user-defined functions in HQL.

```
duration(x:rel) {
        if (empty(x))
                duration := 0;
        else {
                v := distance (start_of(first(x)) , end_of(first(x)));
                duration := v + duration(rest(x));
        }
}
```

Example 2) How long was McEnroe number 1.

```
duration (Ranks where player_name = 'McEnroe' and rank = 1);
```

Another example which demonstrates how easily user-defined functions can be used to define functions where otherwise they would have to be built-in is the function 'nth'. For example to find Lendl's 3rd manager we can define the function 'nth' which takes a relation and a number then returns the nth tuple of the relation as follows:

```
nth(x:rel,n:int) {
        if (n = 1)
                nth := first(x);
        else
                nth := nth(rest(x),n-1);
}
```

and the call

```
nth(managers [mgr] where player-name = 'Lendl',3);
```

should return the right answer.

Other functions such as shortest or closest could also be defined quite easily in similar way.

4.6. Group By Type Operations

The group by operator logically rearranges a relation into partitions or groups, such that within any one group all rows have the same value for the group by field. However, to be able to access the individual groups, we have an operator called first-group. First-group can be user-defined, but the built-in version allows a more natural syntax.

For example, the query "Find players whose first manager was Smith" we can define a function and use the first-group operator as follows:

```
FManager(x:rel) {
        T := first-group x by [player-name];
        if (empty(T))
                FManager := null;
        else {
                if (first(T) [mgr] = 'Smith' )
                        FManager := first(T)[player_name] ++ FManager(x--T)
                else
                        FManager := FManager(x--T);
        }
}
```

The above function can easily be modified for nth manager, by replacing the condition in the if statement.

4.7. Let In

Let in is another high level construct which has been used in HQL. This is a construct which can be used directly at the query level (we do not need to define a function in order to use this construct). Let in is similar to the bottom-up approach of defining objects before they are used. A similar construct is also used in functional languages such as HOPE [Burs80] or ML [Miln85].

```
let y == a in b
```

Reads "==" as 'be defined as'. Here "a" is some arbitrary expression whose value is named "y". "b" is another expression which will use the identifier "y".

Example 1) example 5 of section 4 could also be written using let in construct as follows:

```
let v == start_of(first(WimWin where player_name = 'Becker'))
   in Ranks [rank] where player_name = 'Becker'
      when overlap v;
```

Example 2) Who was the first Wimbledon winner ?

```
let v == start_of(first(WimWin))
   in WimWin[player_name]
      when overlap v;
```

4.8. Defaults

The defaults assumed in any language is an important aspect of the language definition. The default for the retrieve statement in HQL is

```
<Query>
when overlap NOW
```

Here the default means that the query is to be applied to the current state.

For example the query

```
Ranks [name]
where rank = 1;
```

will be transformed to

```
Ranks [name]
where rank = 1
when overlap NOW
```

4.9. Summary

The query language HQL was presented in this chapter. The power of HQL lies in its ability to allow users to define functions. This gives the user the freedom to define any function with considerable ease using looping and conditional constructs and assignments. With the added power of recursion, HQL can be used to answer queries which can not be handled by conventional query languages.

HQL has a user friendly syntax with the ability to formulate complex queries in a natural way without referring to the time domain explicitly. With features such as recursion, iteration and user-defined functions, HQL is more expressive than any other temporal query language that has been developed so far.

The semantics of HQL is described in the next chapter. We show how HQL queries are mapped into expressions of relational algebra, for which

a semantics has been defined.

CHAPTER 5

SEMANTICS

A problem with most current query languages is that no formal semantics of the language exists. Languages are normally presented in terms of an informal syntax together with a few selected examples of their use, and often these examples fail to highlight the real problem areas of the language.

For the definition of the semantics of HQL, we have followed the principles in [Ceri85]; that is the translation of any query language into expressions of relational algebra can be regarded as the definition of semantics of the language.

Unlike [Snod87] which uses relational calculus as the target language, we will show how HQL queries can be mapped to expressions of relational algebra, for which a semantics has been defined [Klug82]. We chose relational algebra rather than relational calculus for the target language for the following reasons:

(a) It is more procedural than relational calculus; algebraic expressions give the order of application of operations in the computation of the query. It is therefore a more appropriate model for query optimisation and system implementation.

(b) Many existing approaches to query optimisation and equivalence are (or can easily be) expressed in relational algebra.

(c) Relational algebra uses relations (or sets) as operands, while calculus is based on tuple variables. For the optimisation of queries, particularly with distributed environments or special-purpose database machines, set-oriented models are more appropriate than tuple-oriented models.

The formulation of the HQL semantics as relational algebra expressions offers a straightforward means of implementing HQL. An HQL query is mapped into the relational algebra, which is then applied to the underlying DBMS.

The architecture of our processor is similar to the architecture described in [Ceri85] (figure-5.1). Although the subset of HQL we have chosen for translating into relational algebra is not as complicated as the SQL subset used in [Ceri85] the architecture is well suited to our purpose and it is simple enough to be used for any other query language.

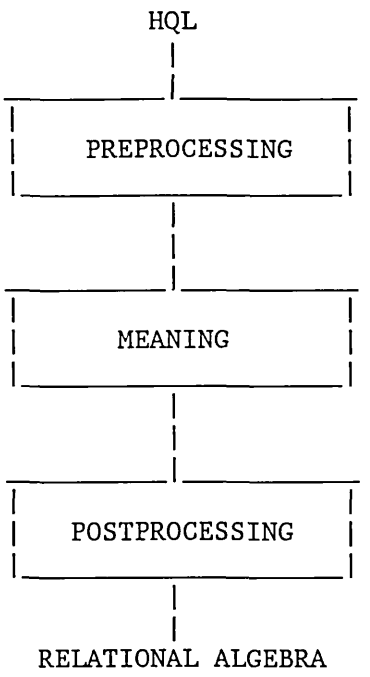


Figure 5.1: Overall Architecture of the Translator

The preprocessing and meaning are at the heart of this approach. The last transformation, called postprocessing, eliminates some operations which have no effect and performs some trivial query optimisation.

5.1. Notation

We have used the traditional operations of relational algebra [Ullm83] with the notation of [Ceri85] :

PJ[A]	(projection over a set A of attributes)
SL[p]	(selection of tuples satisfying predicate p)
CP	(Cartesian product)
UN	(union)
DF	(difference)
INTER	(intersection)
JN[jp]	(join with join predicate jp)

Table-1 is the summary of notation, which will be used in meaning.

car(set of relational expressions) indicates the cartesian product of all the relational expressions; if the argument is a singleton, car returns the unique relational expression.

st(relational expression) is the start time stamp attribute occurring in a relational expression.

en(relational expression) is the end time stamp attribute occurring in a relational expression.

Table-1: Summary of Notation

5.2. Preprocessor

The preprocessor is a syntax-directed transformer. It takes a user query and transforms it into a complete HQL query. i.e. by adding the time predicate and expanding the query where necessary.

For example, the query

```
Ranks [player_name]
where rank = 1
```

is transformed into

```
Ranks [player_name]
where rank = 1
when overlap NOW
```

and also, the query

```
Tour_results
```

is transformed into

```
Tour_results [player_name,tour_name,result]
when overlap NOW
```

5.3. Meaning

The meaning is a syntax-directed transformation of HQL queries into relational algebra expressions. The meaning is similar in style to the definition of the semantics of programming languages described in [Bjor82], although with a more informal approach. In this section we will be using the extended syntax of HQL given in figure-5.2. The complete syntax of HQL is given in appendix-E.

Two types of queries are accepted in this level, simple time queries and complex time queries.

5.3.1. Simple Time Queries

A simple time query in HQL has the following form

```
<relation-list> '[' <selector> ']'
WHERE   <predicate>
WHEN    <simple-time-predicate>
```

However, the <predicate> is optional, and might be missing in the query. The meaning is specified by the following transformation, which considers two cases: the case in which the predicate is missing, and the case in which it is present.

```
meaning(Q: Q is a simple time query) =
if empty(Q.<predicate>)
then {case 1}

        PJ [ Q.<selector> ]
        SL [ tr(Q.<time-op>) ]
        car(Q.<relation-list>)

else {case 2}

        PJ [ Q.<selector> ]
        SL [ tr(Q.<time-op>), Q.<predicate> ]
        car(Q.<relation-list>)

with tr(OVERLAP)  =  other1(Q)
     tr(CONTAIN)  =  other2(Q)
     tr(BEFORE)   =  other3(Q)
     tr(AFTER)    =  other4(Q)
     tr(MEETS)    =  other5(Q)
     tr(STARTS)   =  other6(Q)
     tr(FINISHES) =  other7(Q)
     tr(SAME)     =  other8(Q)


other1(Q) = st(Q.<relation-list>) <= en(Q.<time-const>)
     and en(Q.<relation-list>) > st(Q.<time-const>)

other2(Q) = st(Q.<relation-list>) <= st(Q.<time-const>)
     and en(Q.<relation-list>) >= en(Q.<time-const>)

other3(Q) = en(Q.<relation-list>) < st(Q.<time-const>)

other4(Q) = st(Q.<relation-list>) > en(Q.<time-const>)

other5(Q) = en(Q.<relation-list>) = st(Q.<time-const>)

other6(Q) = st(Q.<relation-list>) = st(Q.<time-const>)

other7(Q) = en(Q.<relation-list>) = en(Q.<time-const>)

other8(Q) = st(Q.<relation-list>) = st(Q.<time-const>)
        and en(Q.<relation-list>) = en(Q.<time-const>)
```

For example, the query

```
        Ranks [ player-name ]
        where rank = 1
        when contains 86
```

is translated as follows:

```
        PJ [ player-name ]
        SL [ start <= 1/1/86 , end >= 31/12/86 , rank = 1 ]
        Ranks
```

```
simple_time_query       ::=     <relation_list> '[' <selector> ']'
                                [ WHERE <predicate> ]
                                [ WHEN <simple_time_predicate> ]

complex_time_query      ::=     <relation_list> '[' <selector> ']'
                                [ WHERE <predicate> ]
                                [ WHEN <complex_time_predicate> ]

simple_time_predicate   ::=     time_op         time_const

complex_time_predicate  ::=     time_op         sub_query

sub_query               ::=     simple_time_query
                        |       complex_time_query

time_op                 ::=     BEFORE
                        |       AFTER
                        |       MEETS
                        |       OVERLAP
                        |       STARTS
                        |       SAME
                        |       FINISHES
                        |       CONTAINS
                        |       BETWEEN

time_const              ::=     NOW
                        |       time_points
                        |       time_interval

time_points             ::=     DAY '/' MONTH '/' YEAR

time_interval           ::=     '[' time_expr , time_expr ']'
```

Figure 5.2: A Subset of Extended Syntax of HQL

5.3.2. Complex Time Queries

A complex time query in HQL has the following form

```
<relation-list> '[' <selector> ']'
WHERE   <predicate>
WHEN    <complex-time-predicate>
```

```
meaning(Q: Q is a complex time query) =

PJ [ Q.<selector> ]
SL [ tr(Q.<time-op>),Q.<predicate> ]
car(Q.<relation-list> U meaning(Q.<sub-query>))

with tr(OVERLAP)  =  c_other1(Q)
     tr(CONTAIN)  =  c_other2(Q)
     tr(BEFORE)   =  c_other3(Q)
     tr(AFTER)    =  c_other4(Q)
     tr(MEETS)    =  c_other5(Q)
     tr(STARTS)   =  c_other6(Q)
     tr(FINISHES) =  c_other7(Q)
     tr(SAME)     =  c_other8(Q)
```

```
c_other1(Q) = st(Q.<relation-list>) <= en(Q.<subquery>.relation-list)
     and en(Q.<relation-list>) > st(Q.<subquery>.relation-list)

c_other2(Q) = st(Q.<relation-list>) <= st(Q.<subquery>.relation-list)
     and en(Q.<relation-list>) >= en(Q.<subquery>.relation-list)

c_other3(Q) = en(Q.<relation-list>) < st(Q.<subquery>.relation-list)

c_other4(Q) = st(Q.<relation-list>) > en(Q.<subquery>.relation-list)

c_other5(Q) = en(Q.<relation-list>) = st(Q.<subquery>.relation-list)

c_other6(Q) = st(Q.<relation-list>) = st(Q.<subquery>.relation-list)

c_other7(Q) = en(Q.<relation-list>) = en(Q.<subquery>.relation-list)

c_other8(Q) = st(Q.<relation-list>) = st(Q.<subquery>.relation-list)
       and en(Q.<relation-list>) = en(Q.<subquery>.relation-list)
```

For example, the query

```
Ranks [ rank ] where player_name = 'Navratilova'
when overlap Nationality where player_name = 'Navratilova'
                         and   country = 'U.S';
```

is transformed first into

```
Ranks [ rank ] where player_name = 'Navratilova'
when overlap Nationality [ player_name,country ]
                       where player_name = 'Navratilova'
                       and   country = 'U.S'
```

and then into

```
PJ [ Ranks.country ]
SL [ Ranks.start <= Nationality.end , Ranks.end >= Nationality.start ,
     Nationality.player_name = 'Navratilova' ]
( Ranks CP meaning ( Nationality [ player_name, country ]
                       where player_name = 'Navratilova'
                       and   country = 'U.S' ))
```

5.4. Postprocessing

Postprocessing is applied to the relational expression produced by the evaluation of the meaning. The postprocessing transformations are purely syntactic.

(1) All the selections on cartesian products of relations are either transformed into joins, or directly applied to relations; for instance,

```
SL [ R.A = S.A ] SL [ S.A > 7 ] (R CP S)
```

becomes

```
R JN [ R.A = S.A ] SL [ S.A > 7 ] S
```

(2) All projections which have no effect are eliminated. Thus, a projection over the complete attribute schema of one relation is eliminated. In a sequence of projections which are not interleaved with other operations, just the first projection is maintained. For

example,

PJ [R.A] PJ [R.A,R.B] PJ [R.A,R.B,R.C] SL [R.D > 8] R

becomes

PJ [R.A] SL [R.D > 8] R

5.5. Summary

It was shown how HQL queries are mapped into expressions of relational algebra. A three level architecture was described for translating HQL queries into relational algebra.

This approach suggests a methodology in the design of new relational query languages:

(1) it provides an unambiguous specification of the language;

(2) it indicates a method for the fast implementation of a translator, which can be used as the first part of a compiler (or interpreter) for the language.

The implementation of HQL is described in the next chapter. The formulation of HQL semantics as relational algebra expressions offers a straightforward means of implementing HQL.

CHAPTER 6

HQL IMPLEMENTATION

For many years, compilers and interpreters were considered to be impossibly large pieces of software. It sometimes took years to implement one. In recent years, however, with development of software tools and techniques, it is now a much simpler and straightforward task to construct compilers or interpreters.

This section shows how a simple interpreter can be easily built with the help of the UNIX tools YACC and LEX.

The principles discussed in [Kern84] have been followed for the HQL implementation. The Unix Programming Environment [Kern84] gives an insight into the UNIX operating system and also how YACC and LEX can be used as program development tools for implementing a simple interpreter for a language with functions, procedures and statements.

6.1. Overview of YACC and LEX

YACC+ is a parser generator, that is, a program for converting a grammatical specification of a language into a parser that will parse statements in the language [Kern84]. The YACC user specifies the structures of his input, together with code to be invoked as each such structure is recognised. YACC turns such a specification into a subroutine called yyparse, that handles the input process.

+ YACC stands for "yet another compiler-compiler"

LEX [Lesk75] is another UNIX tool, used to generate lexical analysers in a manner analogous to the way YACC creates parsers. LEX accepts a high-level, problem oriented specification for character string matching, and produces a program in a general purpose language, e.g., C or Ratfor, that recognises regular expressions. The program generated by LEX is called yylex. Figure-6.1 is a diagrammatic representation of the way YACC and LEX interact with one another.

Figure-6.1 is taken from [Kern79].

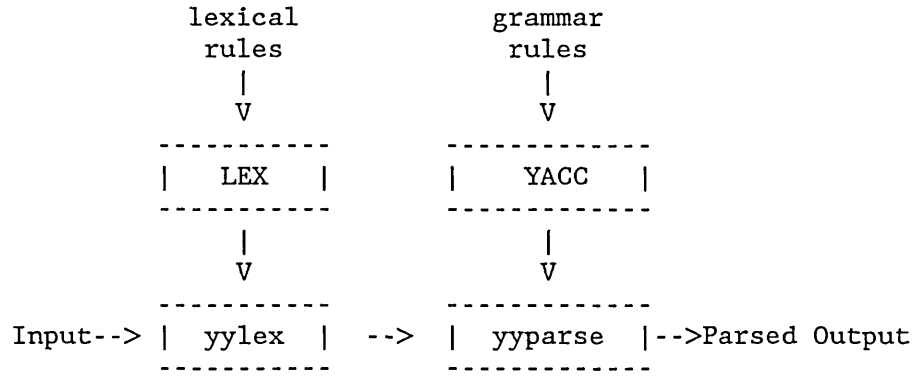


Figure 6.1: LEX with YACC

Appendix-A is a full working example of a simple YACC and LEX program.

6.2. A Basic Interpreter

An interpreter is a program which executes intermediate code generated by a translator for a programming language. The intermediate code may be thought of as the machine language of an abstract computer designed to execute the source code i.e. it mimics or simulates the operations which a machine would carry out if it were capable of processing programs written in that language. Figure-6.2 is an overview of

an interpreter.

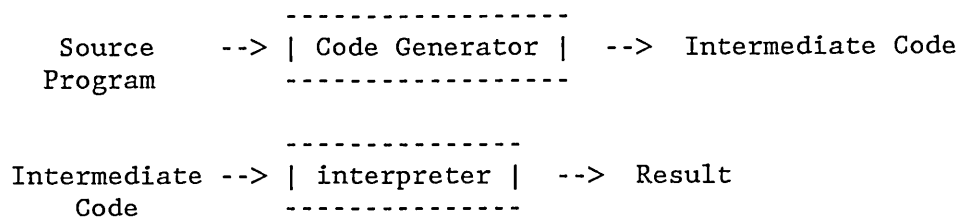


Figure 6.2: An Overview of an Interpreter

Interpreters are often smaller than compilers and facilitate the implementation of complex programming language constructs. However, the main disadvantage of interpreters is that the execution time of an interpreted program is usually slower than that of a corresponding compiled object program [Aho79].

Stack machines are usually used for constructing simple interpreters. A stack machine is an array containing operators and operands. The machine is generated during parsing by calls to the function code which puts an instruction into the next free spot in the program array. When an operand is encountered, it is pushed onto a stack; most operators operate on items on the top of the stack. The operators are machine instructions; each is a function call following the instruction. Once the end of a statement is reached, the generated code is executed to compute the desired result.

For example, to handle the arithmetic expression

```
2 * 3 + 5
```

The following code is generated

```
constpush                       push a constant onto stack
    2                               ... the constant 2
```

```
constpush                       push a constant onto stack
    3                               ... the constant 3
   mult                 multiply top two items; product replaces them
constpush                       push a constant onto stack
    5                               ... the constant 5
   add                    add top two items; product replaces them
  print               pop the value at the top of the stack and print it
   STOP                         end of instruction sequence
```

When this code is executed, the expression is evaluated and result is printed out.

The main part of the YACC program to generate the above code is as follows:

```
main:           expr
                        { code(print); code(STOP);      }
        ;

expr:           expr '*' expr
                        { code(mult);                   }
        |       expr '+' expr
                        { code(add);                    }
        |       NUMBER
                        { code(constpush); code($1);    }
        ;
```

The stack is manipulated by calls to the functions push and pop which push an element onto the stack and return the top element from the stack respectively. Execution of the machine is straightforward. Each cycle executes the function pointed to by the program counter, which is incremented after each cycle. An instruction with opcode STOP terminates the cycle. Figure-6.3 illustrates stack evaluation during execution.

For example constpush pushes the next element in the program array onto the stack.

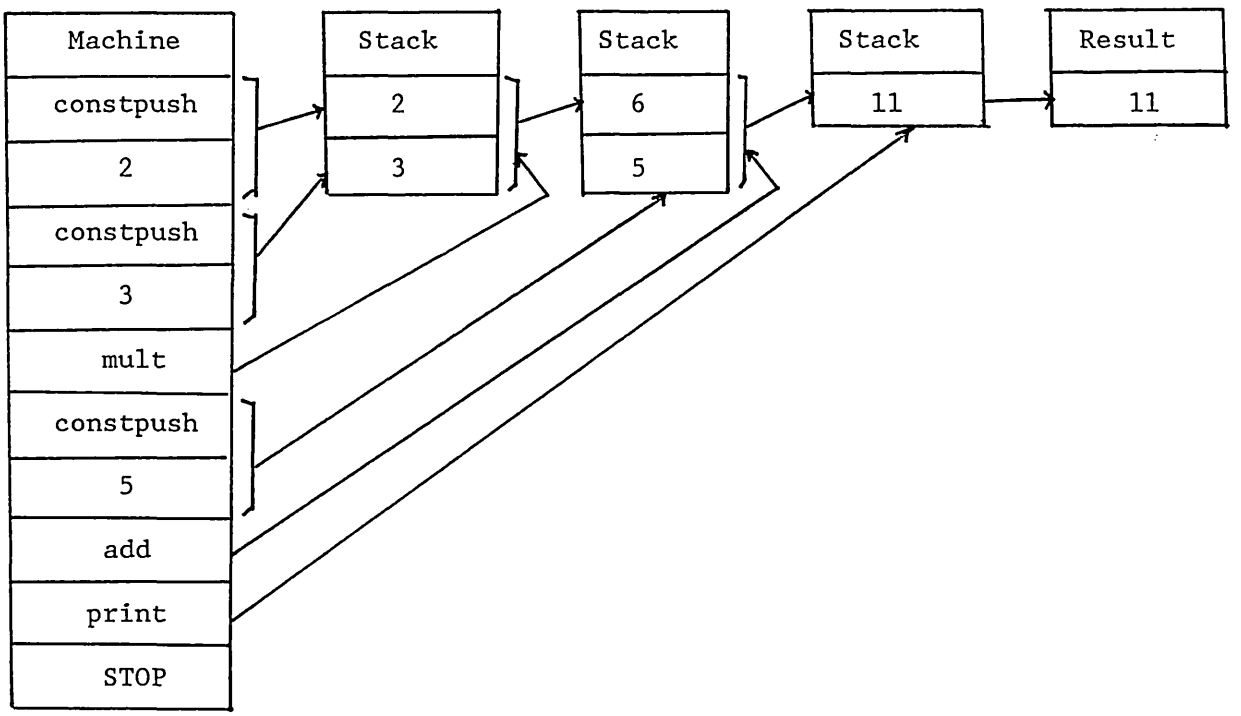


Figure 6.3: Stack Evaluation for an Arithmetic Expression

The function add pops the top two elements of the stack and after adding it pushes the result back onto the stack. The function print simply pops the stack and prints the value of it.

6.3. Compilation of HQL into a Machine

In the previous sections an overview of YACC and LEX was given. Now follows a demonstration of how a simple HQL query is handled by YACC and LEX, and actions taken when a YACC rule has been recognised.

The following is a subset of an HQL data manipulation structure

```
relation
[ '[' selection-list ']'  ]
[ where condition ]
```

For compiling HQL queries, we have followed the principles discussed in the previous section. A new set of functions is defined as operators for HQL.

For example, the query "find names of all suppliers and their locations" can be expressed in HQL as

```
supplier [sname,city]
```

the following intermediate code is generated:

```
 relpush                    push a relation onto stack
supplier                    ... the relation supplier
attrpush                   push an attribute onto stack
  sname                       ... the attribute sname
attrpush                   push an attribute onto stack
  city                        ... the attribute city
  proj     project out a number of attributes from relation on the stack
    2                 ... the number of attributes for project
 display       pops relation on the top of the stack and displays it
  STOP                      end of instruction sequence
```

The YACC program used to parse and generate code for the above is as follows:

```
query:          query_block
                        { code(display); code(STOP);    }
        ;

query_block:    relation '[' selection_list ']'
                        { code(proj); code($1);  }
        ;

selection_list: attr_name
                        { $$ = 1;       }
        |       attr_list ',' attr_name
                        { $$ = $1 + 1;  }
        ;

attr_name:      name
                        { code(attrpush); code($1); }
        ;

relation:       name
                        { code(relpush); code($1); }
        ;
```

Once the code has been generated, the execution of the code is similar to the execution described in the previous section for the arithmetic expression(figure-6.3).

Figure-6.4 illustrates stack evaluation during the execution of the code.

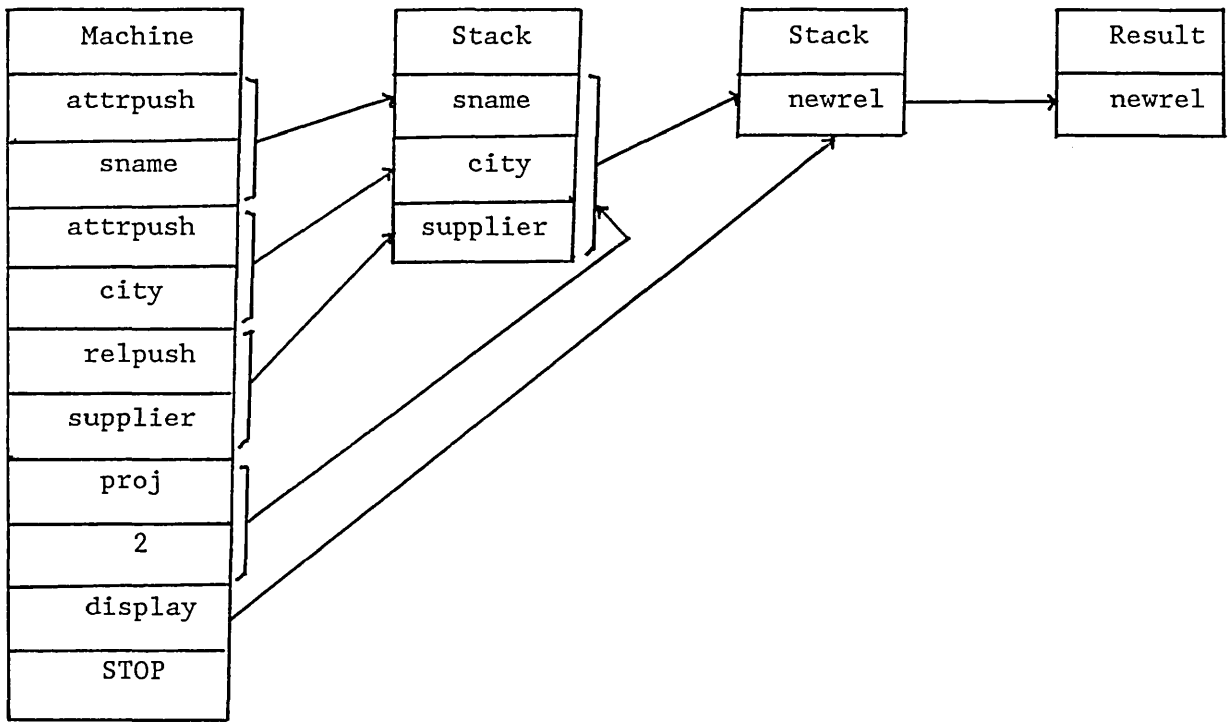


Figure 6.4: Stack Evaluation for a Simple Query

Another example which highlights how HQL queries are handled by the system is the following ("what was Becker's rank in 1986 " ?):

```
Ranks
where   player_name = 'Becker'
when    overlap 86
```

The above query is first converted into one which is free of time operators (i.e. overlap, contains, ...) in accordance with the rules discussed in semantics of HQL in chapter 5. Thus the following query:

```
Ranks [ player_name, rank ]
where   player_name = 'Becker'
   and  start <= 1/1/86
   and  end >= 31/12/86
```

Then the following code is generated which is as follows:

```
relpush                    push a relation onto stack
```

```
   Ranks                    ... the relation Ranks
  attrpush                 push an attribute onto stack
player_name               ... the attribute player_name
  strpush                   push an string onto stack
   Becker                     ... the string Becker
     eq                       ... the operator equal
  attrpush                 push an attribute onto stack
   start                      ... the attribute start
  timepush                push time constant onto stack
   1/1/86                    ... the time constant 1/1/86
     le                  ... the operator less than or equal
  attrpush                 push an attribute onto stack
    end                       ... the attribute end
  31/12/86                 ... the time constant 31/12/86
     ge                ... the operator greater than or equal
    sel                        selection operator
  attrpush                 push an attribute onto stack
player_name               ... the attribute player_name
  attrpush                 push an attribute onto stack
    rank                      ... the attribute rank
    proj   project out a number of attributes from relation on the stack
     2                ... the number of attributes for project
  display      pops relation on the top of the stack and displays it
    STOP                   end of instruction sequence
```

Once the code has been generated the execution is similar to the previous example. Figure-6.5 illustrates stack evaluation during the execution of the code.

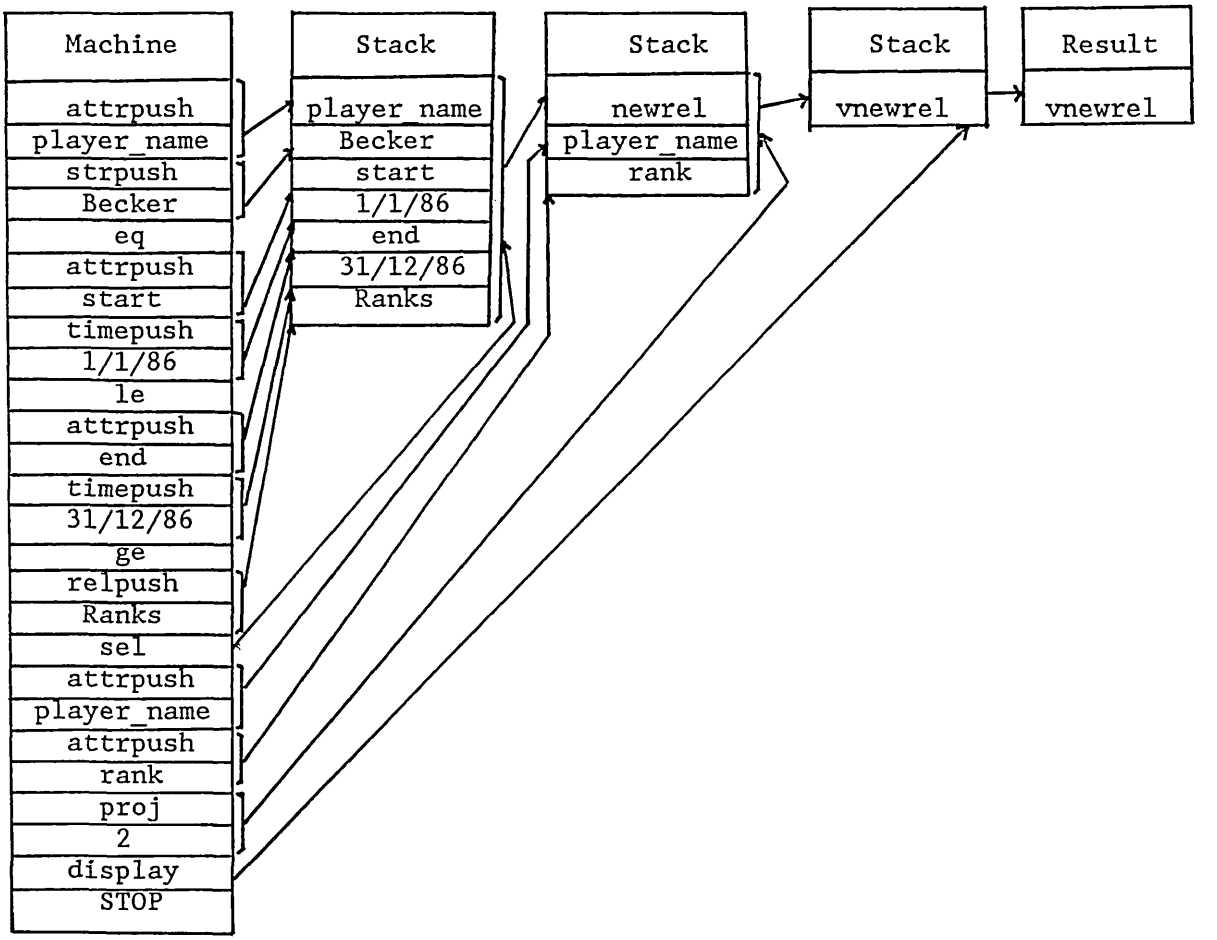


Figure 6.5: Stack Evaluation for a Time Query

Now consider some of the HQL operations in detail. Each element on the stack is either an integer or a pointer to a symbol table entry. The interpreter stack type is therefore defined as a union of the two as shown below.

```
typedef union Datum {
        int     val;
        Symbol  *Sym;
} Datum;
```

(see appendix-B for data type Symbol).

The operator proj collects all the data required by the PRECI/C project function i.e. relation-name, number of attributes to be projected and the projected attributes, then it calls the function project. Once the project is done it then pushes the relation back onto stack as shown in figure-6.5.

```
proj()
{
        Datum rel,d,dl;
        int   at[MAXNUMATTR],number_of_attr,i;

        number_of_attr = (int)(*pc++);  /* pc is program counter */

        /* get relation id from the stack */

        rel = pop();

        /* now get attribute names */

        for (i=0; i < number_of_attr; i++) {
                d = pop();
                at[number_of_attr-1-i] = d.val;
        }
        /* call PRECI/C function project */

        dl.val = project(rel.val,number_of_attr,at);

        /* push the new relation back onto stack */

        push(dl);
}
```

The operator display is straightforward, it pops the stack to return the relation we want to display and then calls the PRECI/C function display_rel to display it.

```
display()
{
        Datum d;

        d = pop();                          /* pop the stack */
        display_rel(d.val);      /* display the relation */
}
```

The lexical analyser for HQL is given in appendix-B.

6.4. Views and Functions

Views and functions are distinct in HQL, although they are defined by the same mechanism. The distinction is that views always return relations, but functions return all other types. This helps with run-time checking of type correctness.

Views and functions may take arguments, separated by commas, when invoked. Arguments are passed by value and within views or functions are semantically equivalent to variables.

Views and functions may be recursive, but the stack implemented has limited depth (about a hundred calls).

Implementation of views (or functions) is rather neat. A second stack is required to keep track of the nested view and function calls. The second stack which is called a Frame (see appendix-B) is similar to an activation record [Aho79]. There are four items that appear on the second stack.

(1) The symbol table entry.

(2) The return address.

(3) The n-th argument on stack.

(4) The count of the number of arguments.

During compilation, a function is entered into the symbol table by storing its origin in the table and updating the next free location after the generated code, if the compilation is successful.

When a view or a function is called during execution, all arguments will already been been computed and pushed onto the stack (the first argument is the deepest). The information for the second stack, i.e. the frame, is created by the function call, which then goes on to execute the code of the routine.

This structure is illustrated in figure-6.6.

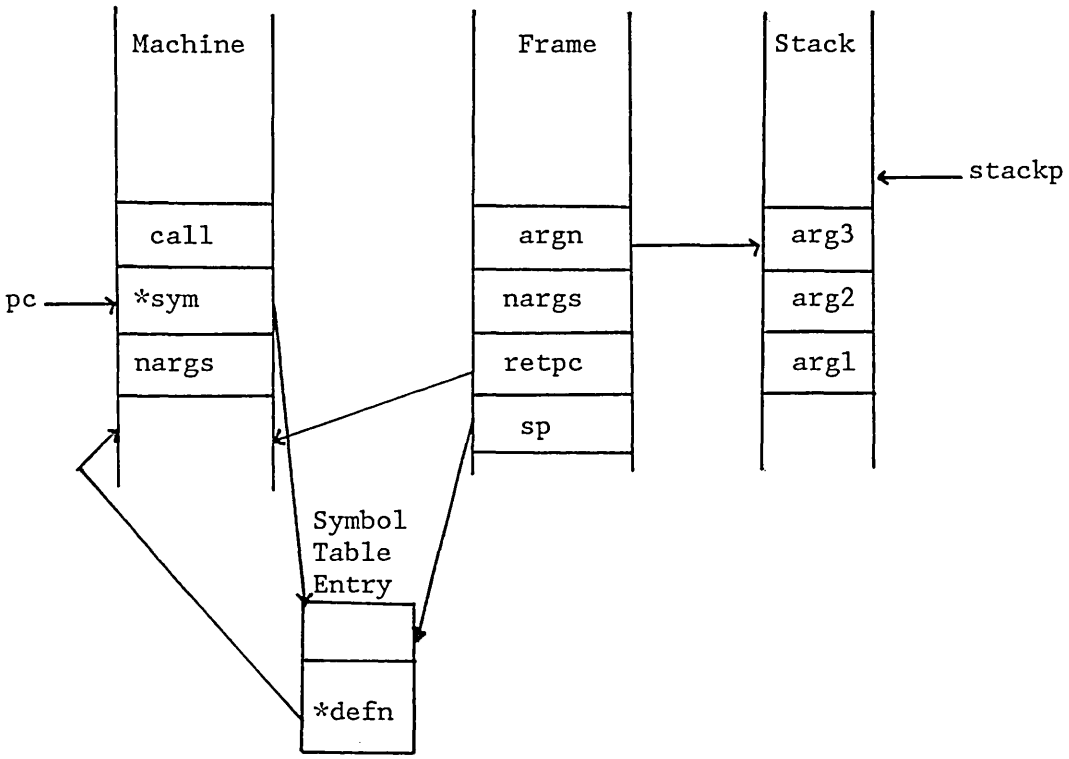


Figure 6.6: Data Structures for View or Function Call

Eventually the called routine will return by executing a routine which returns from a view or a function.

6.5. Built-in Views and Functions

There are number of built-in views and functions in HQL. Because HQL allows user-defined functions, the number of built-in operators is small. The built-in operators are installed in the symbol table by the function init (see appendix-B), which is called by the main program.

---

pc is program counter during execution.

The built-in operators are implemented rather easily. We look at two of them in detail.

The built-in view first takes a relation and returns another relation which has got one tuple only, which is the first tuple of the original relation.

The built-in function card takes a relation and returns the cardinality of that relation.

The function card is defined as follows:

```
double card(relid)
Datum relid;
{
        int ntups;

        if (relid.vtype != RELATION)    /* if not valid */
                execerror("invalid relation in card",(char *)0);
        ntups = getcardof((int) relid.v.val);
        return(ntups);
}
```

where function getcardof is a PRECI/C function which given a relation returns the cardinality of it.

The view first is more complicated than the built-in card, but it is similar in structure to card.

```
double first(relid)
double relid;
{
        int ntups, err;
        int *tuple;

        ntups = getcardof((int)relid);
        if (ntups == 0)                 /* relation empty */
                return((int)relid);     /* return the relation */
```

```
        /* get the first tuple of the relation */

        tuple = firstup((int)relid);

        /* create a new relation */

           ...

           ...


        /* define temporary relation */

        newrelid = deftrel(...);

        /* insert the tuple into the newly created relation */

        err = insertup(newrelid,tuple);
        if (err < 0)
                execerror("error at first (insertup)", (char *)0);

        /* if no errors return the new relation */

        return(newrelid);
}
```

where functions firstup, deftrel, insertup are PRECI/C function.

6.6. Summary

In this chapter we described how a simple interpreter with the ability to handle recursion has been developed for HQL with the help of the UNIX language development tools YACC and LEX. We have show how an HQL query is mapped into expressions of relational algebra, and then mapped into PRECI/C statements.

The next chapter investigates the implementation issues of our proposed temporal relational model.

CHAPTER 7

IMPLEMENTATION APPROACHES AND CONSIDERATIONS

There are several issues to be investigated for the implementation of database management systems with temporal support.

One of the major problems is the handling of ever-growing storage size. It is impractical to store all the states of a database while it evolves over time. This will cause the relation to grow very fast, resulting in performance and storage problems. Perhaps, these were the reasons that prompted Bubenko [Bube77] to say that "... we can not - for practical reasons - realise (implement) information models with the full temporal generality ..." and Clifford [Clif83] to say it is "prohibitively costly" to have a "direct implementation".

There are also other problems which need to be addressed such as structural changes and efficient access methods.

There are two approaches to the design and implementation of a historical database system.

(i) In one approach additional layers are built on top of an existing DBMS to support different applications using abstract data types or domain level operations. The shell is an interpretive interface which presents a database user with a time or graphical system.

Every database request is interpreted by the shell and implemented via a series of internal DBMS calls. This underlying process is fully

transparent to the database users. This normally referred to as an interpretive approach (figure-7.1).

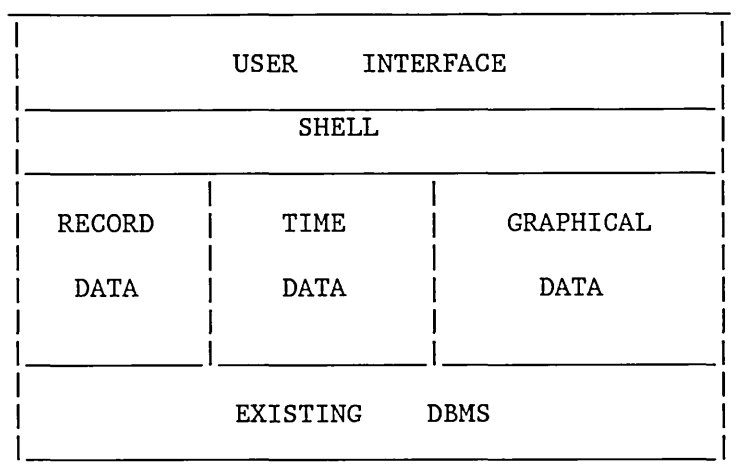


Figure 7.1: The Interpretive Approach

(ii) In the second approach the DBMS itself is extended to deal with all types of data. in other words the notions of time and graphics are part of the overall DBMS i.e. they are built-in. This is normally referred to as a built-in approach (figure-7.2).

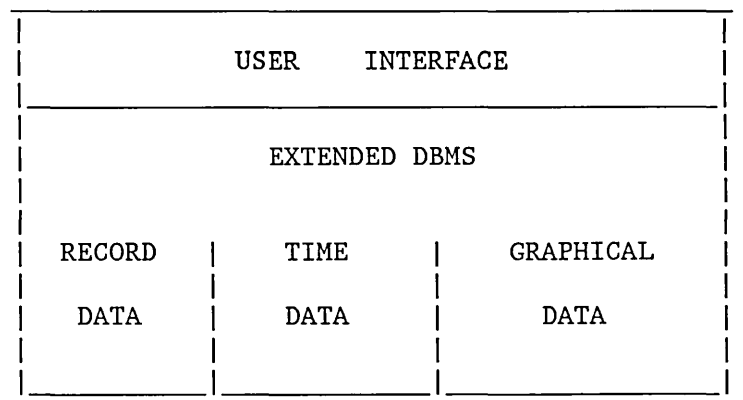


Figure 7.2: The Built-in Approach

Both methods have advantages and disadvantages. For example, the main advantages which an interpretive approach has over a built-in

approach are:

- It requires less effort to implement
- It can implemented on an existing database system without having to change it.

The main advantages which a built-in approach have over an interpretive are:

- It allows more efficient implementation.
- Better performance is possible.
- Efficient storage is possible.
- It supports structural changes.

7.1. Basic Implementation

A basic design for the implementation of our model is presented here. The design can handle time support efficiently and it is straightforward to implement.

In our model time stamps start and end are stored with each tuple. The time stamps are protected and are normally transparent to users. Included with each current tuple is a pointer that points to the first history tuple when there is one.

The methods for creating and storing history data works as follows: All history information belonging to one tuple is chained in reverse time order. The beginning of the chain is always in the current tuple,

except in the case where tuples have been deleted. When that happens, it will be moved to the history chain. In its place, a small delete indicator is kept along with the pointer to show the beginning of the chain.

With this basic structure, we can process all the current and history data in a relation. In choosing to store data this way, it has been assumed that recent data is most frequently accessed and the older the data the less likely it is to be accessed.

7.2. Storage Structure

There are a number of issues which need to be considered when designing a storage structure for historical databases. These are as follows:

- The storage requirements for history data may be potentially enormous, while the size of current data is relatively static.

- Unlike current data, history data need never be updated except when recording errors are corrected.

- History data, in general, is not accessed as frequently as current data.

These differences make it natural to separate the current data from the historical data and process them separately.

Therefore, a two level storage structure would seem to be ideal. Similar storage structures have also been used by other researchers in this field notably [Ben-82,Lum84,Ahn86].

This scheme to separate current data from the bulk of history data can minimise the overhead for non-temporal queries, and at the same time provide a fast access path for temporal queries. It is possible to use different access methods for each of them. The primary store may utilise any conventional access method suitable for a static relation, e.g., hashing, ISAM, B-tree etc. It is even possible to use different types of storage medium for the two. For example, history data may be stored on optical disks, while current data is kept on magnetic disks [Ahn86].

7.3. Physical Storage

There are a number of ways in which current and historical data can be stored. For the model described in this thesis a method has been chosen which is both simple and efficient (figure-7.3). History information belonging to one tuple is chained in reverse order of time and is kept in one page corresponding to that tuple. The advantage of this method is that it significantly reduces the number of disk accesses for retrieving history data thereby improving the performance of temporal queries. The disadvantage of this method is that it requires a sophisticated update mechanism to maintain clustering while achieving a high degree of storage utilisation.

A variation of the above method is one in which, all history tuples are kept together in one place in number of pages (figure-7.4). When overflow occurs, i.e. a page gets too full, a new page is allocated from a pool of free pages and the data on the old page is split into two, moving all history of some selected tuples to the new page. If all the data in the overflowed page belong to one tuple, a new page is added as

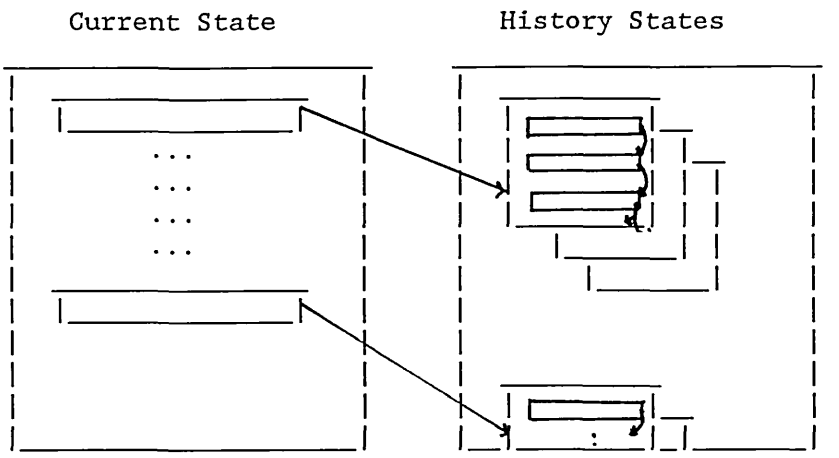


Figure 7.3: The Single Page History Chain

a successor.

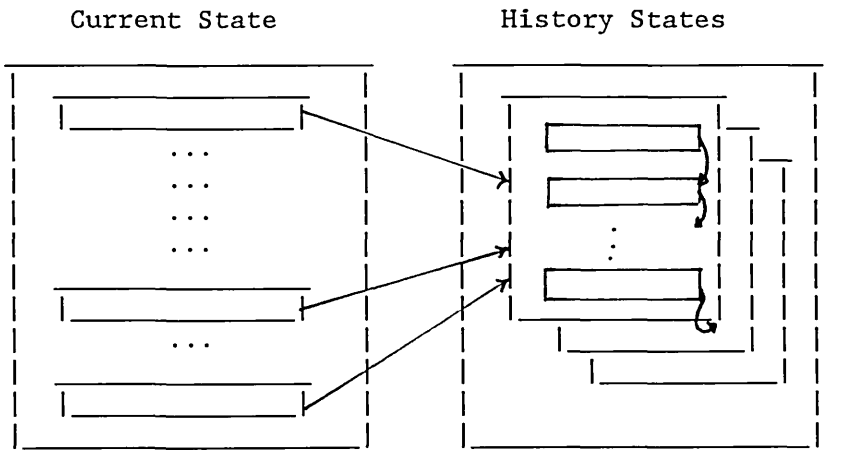


Figure 7.4: The Multi Page History Chain

Implementation issues concerning the storage structure for DBMSs with temporal support has been discussed thoroughly by Ahn in [Ahn86].

Storage space will always be a big problem in historical databases. There are a number of ways in which the amount of data stored in the HDB might be reduced.

Selective update and a Summary mechanism are two methods which could be used to reduce the amount of history data, without any major side effects.

7.4. Selective Updates

In some applications where one does not need to keep the entire history of a relation, selective update could be used to record only certain tuples as history data. In other words selective update can specify the time period for which history data for a relation will be kept, if at all.

Selective update is a special case of general update in Historical Database systems, but it should be done in such a way that it does not affect the history data in that it will not lead to incomplete data being stored in the data base.

7.5. Summary Mechanism

The summary mechanism (figure-7.5) could serve two purposes

(i) We may not need to keep all of the old data, but only some parts of it e.g. a company with a very large payroll may not want to keep names and addresses of previous employees but it may be sufficient to retain just the salary distribution for each year, or even just the median salary in each several grades for each year.

If the company then wishes to compare present day salaries with those of 10 years ago, the figures are still available.

(ii) If certain classes of queries are anticipated on a regular basis,

the summary mechanism can be organised to deal with these comparatively efficiently. E.g. if we want to know whether employees' salaries are rising in real terms, then the median for each grade, produced annually alongside the cost of living index would provide this information.

The summary mechanism can also be applied to the current data.

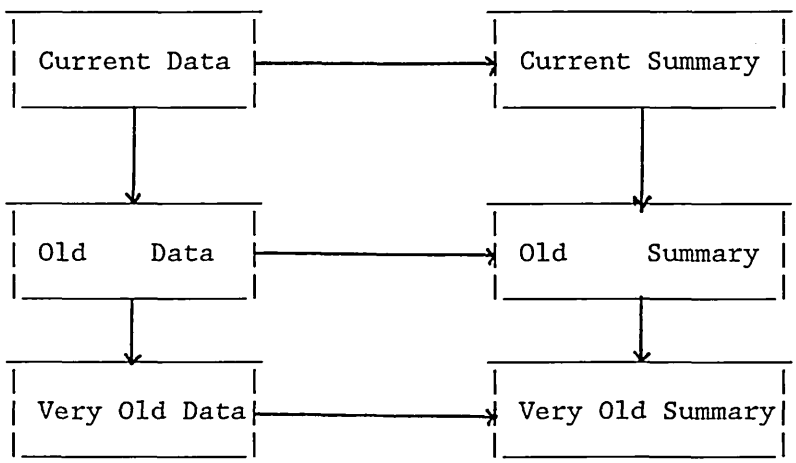


Figure 7.5: Summary Mechanism

7.6. Structural Changes

It is a relatively easy task to change the structure of an existing relation, like adding a new column or changing a data type with out affecting the users who operate on these relations. System R [Astr76], permits the addition of fields or attributes at the end of the current structure. Deletion of attributes is not allowed.

The "view" mechanism can solve a limited number of problems which can arise after a relation has been changed. In this case view is a special query which extracts the old data out of the new data, every time a

reference is made to the old version [Ben-82].

Mapping from old to new data will not generally raise problems but mapping from current (new) to old may raise problems. E.g. to get Employee and Department from views R1,R2,R3 which were defined in terms of base relations may not be feasible because of missing values.

One way to keep track of changes made to the database is by scheme evolution.

7.6.1. Scheme Evolution

A database scheme in a relational database system describes the structure of the database; the contents of the database must adhere to that structure [Date82,Ullm83]. Scheme evolution refers to changes to the scheme of a database over time. Conventional databases allow only one scheme to be in force at a time, requiring restructuring when the scheme is modified.

There has been significant interest in the area of database restructuring in the past decade [Nava80,Gerr76]. A great deal of work has also been done in the area of database reorganisation and restructuring for historical databases [Aria86,Woel86,Clif87,Bane87].

In [Ben-82] a time data dictionary (TDD) was used to keep track of structural changes. TDD is a system database which contains information concerning various objects that are of interest to the system. It keeps a record of all current and history objects which are in the database i.e. relations and attributes. On creation of a relation it will record

the begin time, and end time of the relation and its attributes. Since TDD itself consists of relations just like ordinary user relations, it can also be queried by the systems query language.

In [McKe87] a relation is defined as a sequence of schemes and states. Scheme modification commands are introduced to alter the scheme sequence. A semantic type system is required to identify semantically incorrect expressions and to enforce consistency constraints between the scheme and the contents of the database.

7.7. Summary

A basic design which can handle time efficiently and is straightforward to implement has been presented. It was shown how history data can be maintained for tuples and two methods were proposed for storing current and history data. Two methods were presented for reducing the amount of data stored in the database, and the problem of structural changes was addressed.

CHAPTER 8

CONCLUSION

In this thesis we discussed the need for a database management system with history data support. An extension of the relational data model which includes the temporal semantics of the real world into a database has been presented.

A model which integrates the time concept directly into the design of database management systems has been developed. It is argued that to allow modification of the data structures in relations, to enhance performance and to minimise space would require an approach similar to the one set out in this model for the resulting system to be effective.

A new model for time and time relations is introduced along with a proposed temporal normal form (TNF). The changes to relational algebra operators required to handle time attributes are described. The new algebra is a straightforward extension of the conventional relational algebra, which supports valid time. Historical versions of the five relational operators union, difference, cartesian product, selection, and projection are defined and two new operators, time slice, and when, are introduced.

Tuple level time stamping is chosen, rather than attribute level time-stamping, for its simplicity and for its ability to be implemented directly on conventional relational database systems.

Storage will always be a big problem in the design of historical databases for a foreseeable future. Storing all history data would cause the storage space to expand very rapidly. To allow us to minimise storage requirements, we present two methods which can be used for storing history data efficiently and effectively.

A detailed syntax and semantics of the temporal query language HQL, which is superset of DEAL is also presented. HQL has a user friendly syntax with the ability to formulate complex queries in a natural way without referring to time explicitly. With features such as recursion, iteration and user-defined functions, HQL is more expressive than any other temporal query language that have been developed so far.

For implementation, it has been shown that the formulation of HQL semantics as relational algebra expressions offers a straightforward means of implementing HQL. It has been shown how an HQL query maps into expressions of relational algebra, and then mapped into PRECI/C statements. There are great advantages in developing software under the UNIX operating system. UNIX is simple but rich in facilities and provides an attractive programming environment. The C programming language [Kern78] together with numerous utility programs and software tools [Kern84] provides a productive programming environment.

It is therefore believed that the time model presented here, in conjunction with the query language HQL, will serve as a basis for practical temporal database management systems which can provide sufficient power for defining, storing, extracting and managing temporal informa-

tion.

CHAPTER 9

FUTURE WORK AND DIRECTIONS FOR FUTURE RESEARCH

In this thesis, we presented an overview of a time model, and defined a temporal query language (HQL) for dealing with temporal information. However, much more research is necessary before a viable temporal database management system can be developed.

Many additions are possible to our query language HQL. The operations available for temporal predicates are certainly not exhaustive, and new ones could be added easily to both the language and its semantics.

Although aggregate functions are not defined formally in HQL, the user-defined mechanism of HQL makes it possible to provide both conventional and time-varying aggregates.

There are host of other issues which need to be considered in the design of a temporal query language. For example, how should changes to the schema be incorporated into the language? How should indeterminacy be incorporated? Should the valid time be linear or branching? These are some of the issues which have also been raised by other researchers in this field notably [Snod87].

The reason databases with temporal support have not been put into practice despite their benefits is partly due to the lack of research on implementation issues, such as the handling of ever-growing storage

size, and efficient access methods for both temporal and non-temporal queries.

9.1. Performance and Optimisation

The large amount of data to be maintained in HDBs causes performance problems. New storage structures and access methods need to be developed to achieve reasonable performance for a variety of temporal queries, without penalising more frequent non-temporal queries. We also need to develop query processing algorithms and optimisation techniques to work with the new access methods exploiting the unique characteristics of a database with temporal support.

9.2. Recovery and Concurrency

Some work still needs to be done in this area. It seems possible that there may be some way of taking advantage of the time stamps in the tuples to provide a better concurrency control strategy.

9.3. Null Values and Constraints

Although a great deal of work has been done in the area of null values for conventional relational database management systems, the problem of null values is far from being solved. According to Date [Date86], any attempt to incorporate support for null values into an implemented system should be considered premature at this time. Although an alternative approach based on the concept of default values is being introduced, a great deal still needs to be done in this area.

It is also necessary to define a set of semantic constraints for historical databases. A classification and specification of the

constraints needs to be attempted.

These are but some of the problems remaining to be solved; we expect to come across many more as we continue to build a system with time support.

The nature of time in computer-based information systems and its handling in such systems have recently caught the attention of the database community, both researchers and practitioners. The recent intensifying attention to the role of time in information systems is a manifestation of the never ending quest, to use Codd's words [Codd79],

> "to capture more meaning and enhance the semantic capacity of data models and systems for the management of data".

Time touches almost every aspect of information system design, implementation, and operation; and there is still much to be understood.

APPENDIX A

A SIMPLE YACC AND LEX EXAMPLE

Given the Backus-Naur Form for a simple expression in figure-A.1, we demonstrate how a simple program could be written using YACC and LEX which will accept input and perform all the operations specified in the grammar.

```
list:   expr
        list expr
expr:   NUMBER
        expr + expr
        expr - expr
        expr * expr
        expr / expr
        ( expr )
```

Figure A.1: BNF of a Simple Expression

The general format for a YACC specification is:

```
{ declarations }
%%
{ rules }
%%
{ programs }
```

where one or more sections can be omitted, while the first %% is essential. Therefore, the simplest YACC program is:

```
%%
{ rules }
```

The rules section contains lines of the type

```
name : body ;
```

where the colon and semicolon are YACC punctuation; name is a non-terminal symbol and body is a sequence of zero or more names and literals*.

Non-terminal "names" are declared in the declaration section as:

```
%token name1 name2 ...
```

The start rule for a grammar can be explicitly set. If not explicitly set, the left hand side of the first grammar rule is taken as the start symbol, by default.

The YACC program for the above BNF is as follows:

---

* A literal consists of characters enclosed in single quotes "'".

```
%{
#include <stdio.h>
%}
%token  NUMBER
%left   '+' '-'         /* left associative, same precedence */
%left   '*' '/'         /* left associative, higher precedence */
%%
list:           /* nothing */
        |       list '\n'
        |       list expr '\n'
                        { printf("\t%.8g\n", $2);       }
        ;

expr:           NUMBER
                        { $$ = $1;      }
        |       expr '+' expr
                        { $$ = $1 + $3; }
        |       expr '-' expr
                        { $$ = $1 - $3; }
        |       expr '*' expr
                        { $$ = $1 * $3; }
        |       expr '/' expr
                        { $$ = $1 / $3; }
        |       '(' expr ')'
                        { $$ = $2;      }
        ;
%%
        /*  end of grammar */
```

Alternative rules in YACC are separated by '|'. Any grammar rule can have an associated action, which will be performed when an instance of that rule is recognised in the input. An action is a sequence of C statements enclosed in braces { and }. Within an action, $n (that is, $1, $2, etc.) refers to the value returned by the n-th component of the rule, and $$ is the value to be returned as the value of the whole rule. So, for example, in the rule

```
expr:   NUMBER
           { $$ = $1;    }
```

$1 is the value returned by recognising NUMBER, that value is to be

returned as the value of the expr.

At the next level, when the rule is

```
expr:   expr '+' expr
                { $$ = $1 + $3; }
```

the value of the result expr is the sum of the values from the two components expr's. Notice that '+' is $2; every component is numbered.

At the next level i.e. the top level, when the rule

```
list:   list expr '0
                { printf("      %.8g0, $2);    }
```

is recognised the value associated with expr is printed [Kern84].

Like YACC, the general format for a LEX program is

```
{ definitions }
%%
{ rules }
%%
{ user defined subroutines }
```

where the definition and user defined subroutines are optional. Therefore the minimum LEX program is

```
%%
```

that merely copies the input to the output, unchanged.

Each regular expression represents the user's control decision. It is written in the form of a table, with the regular expressions on the left hand column and LEX actions to its right. The normal C escapes, like \t, \n, are recognised. Also, the backslash can be used to escape the meanings of operators.

The LEX program for the above example is as follows:

```
%{
#include        "y.tab.h"
%}

%%

[ \t\n]         { ; }   /* ignore space, tab and newline characters */

[0-9]+          {
                        int d;

                        sscanf(yytext,"%f",&d);
                        yylval = d;
                        return (NUMBER);
                }

.               {       return (yytext[0]);     }

%%
```

The variable yylval is used for communication between the parser and the lexical analyser. yylex returns the type of a token as its function value, and sets yylval to the value of the token (if there is one).

For YACC and LEX to be able to work together the user must provide a certain amount of environment. For example, as with every C program, a program called main must be defined, that eventually calls the YACC generated subroutine yyparse.

```
main()
{
        yyparse();
}
```

yyparse repeatedly calls yylex the lexical analyser supplied by the user to obtain input tokens. Once a rule is satisfied then an action takes place. In this case yyparse returns the value 0. If an error occurs, in which case yyparse returns the value 1.

APPENDIX B

HQL LEX PROGRAM

In the program the keywords along with other information are stored in a symbol table which is a linked list. The structure of the symbol table is shown in in figure-B.1. Access to symbol table is through the function lookup and install The former searches the list for a particular name and returns a pointer to it's location in the linked list, while install puts a variable with its associated type and value at the head of the list.

```
typedef struct Symbol {  /* symbol table entry */
        char    *name;
        int     type;    /* VAR, UNDEF */
        union {
                double  val;            /* if VAR     */
                double  (*ptr)();       /* BLTIN      */
                int     (*defn)();      /* FUNCTION   */
                int     *str;           /* STRING     */
        } u;
        struct Symbol *next;    /* to link to another */
} Symbol;
```

Figure B.1: Symbol Table Structure

Whenever a user inputs a rule, LEX calls lookup to check if the token exists in the symbol table. If negative, it calls install to install it into the linked list (see LEX program which follows). In either case, the token address is passed to the YACC generated function, yyparse(), by LEX. On the completion of a particular rule, the linked list of keywords is not deleted, the symbol table being available for further input.

In this way, the user initialises the symbol table only once at the start of the program. Thee initialisation is done using the function init shown in figure-B.2.

HQL keywords are installed in the symbol table by the function init as follows:

```
static keywords[] = {
        char    *name;
        double  key;
} keywords[] = {
        "where",        WHERE,
        "before",       BEFORE,
        "after",        AFTER,
        "meets",        MEETS,
        "overlap",      OVERLAP,
        "shorter",      SHORTER,
        "longer",       LONGER,
        "equal",        EQUAL,
                ...
                ...
                ...
        0,              0
};

static struct {
        char    *name;
        double  (*views)();
} builtins[] = {
        "first",        first,
        "card",         card,
        "empty",        empty,
               ....
               ....
        0,              0
};
```

```
init()   /* install keywords in symbol table */
{
        int     i;
        Symbol  *s;

        for (i = 0; keywords[i].name; i++)
                install(keywords[i].name, UNDEF, keywords[i].key);
        for (i=0; builtins[i].name; i++) {
                s = install(builtins[i].name, BLTINS,0.0);
                s->u.ptr = builtins[i].func;
        }
}
```

Figure B.2: Init Function

Figure-B.3 is the structure of a frame for function and view implementation.

```
typedef struct Frame {          /* view/func call stack         */
        Symbol  *sp;            /* symbol table entry           */
        Inst    *retpc;         /* where to resume after return */
        Datum   *argn;          /* n-the argument on stack      */
        int     nargs;          /* number of argument           */
} Frame;
```

Figure B.3: Frame Structure

```
%S      AA BB
%{
int     i;
%}

%%
<AA,BB,INITIAL>[ \t]    { ; }

<INITIAL>[0-9]+         {
                        int d;
                                sscanf(yytext,"%f", &d);
                                yylval.sym = install("",NUMBER,d);
                                return(NUMBER);
                        }

<INITIAL>[%a-zA-Z][.a-zA-Z_]*   {
```

```
                        Symbol  *s;
                        int     stp;

                        if (strlen(yytext) > 21)
                                execerror("name too long",yytext);

                        /* check the parameter list first */
                        stp = param_lookup(yytext);
                        if (stp) {
                        yylval.sym = (Symbol *)emalloc(sizeof(Symbol));
                        yylval.sym->name = (char *)emalloc(strlen(yytext)+1
                                strcpy(yylval.sym->name,yytext);
                                yylval.sym->u.val = stp;
                                return ARG;
                        }
                        if ((s = lookup(yytext)) == (Symbol *) 0)
                                s = install(yytext,UNDEF,0.0);
                        yylval.sym = s;
                        return s->type == UNDEF ? VAR : s->type;
                        }

<INITIAL>["|']          { BEGIN AA; }

<AA>([\et\en]*)?([a-z ][a-z0-9 ]*)([\et\en]*)["|']      {
                        char    buf[100],*p;
                        int     c;

                        yytext[--yyleng] = '\0';
                        for (p=buf,i=0;i<yyleng,c=yytext[i]; i++,p++)
                                *p = backslash(c);
                        *p = 0;
                        yylval.sym = (Symbol *)emalloc(sizeof(Symbol));
                        yylval.sym->name = (char *)emalloc(strlen(buf)+1);
                        strcpy(yylval.sym->name,buf);
                        BEGIN INITIAL;
                        return(QSTRING);
                }

<INITIAL>">"            { return(GT);           }
<INITIAL>"<"            { return(LT);           }
<INITIAL>"!="           { return(NE);           }
<INITIAL>">="           { return(GE);           }
<INITIAL>"<="           { return(LE);           }
<INITIAL>"="            { return(EQ);           }
<INITIAL>"&&"           { return(C_AND);        }
<INITIAL>"||"           { return(C_OR);         }
<INITIAL>"="\=          { return(EQEQ);         }
<INITIAL>":"\=          { return(BEQ);          }
<INITIAL>"*"\*          { return(CART);         }
<INITIAL>"+"\+          { return(UNION);        }
<INITIAL>"-"\-          { return(DIFF);         }
<INITIAL>"*"\?          { return(INTER);        }
```

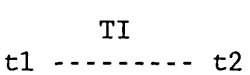
```
<INITIAL>"?"\?          { return(JOIN);          }

<INITIAL>\n             { return('0');    }

<INITIAL>.              { return(yytext[0]);     }

%%
```

APPENDIX C

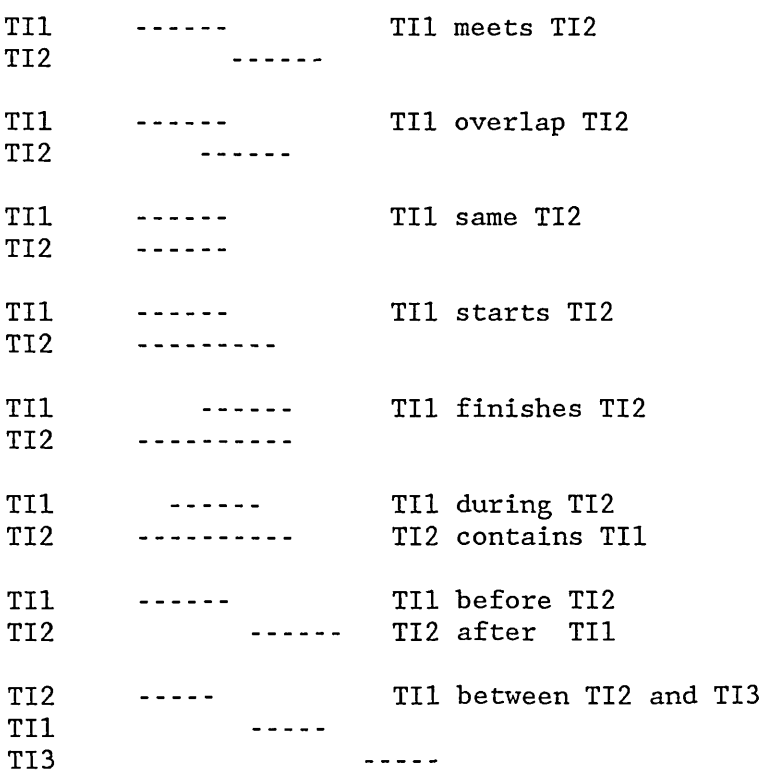
FORMAL DEFINITION OF TIME OPERATORS

In the model an interval is formed by a pair of time points.

TI = (t1,t2)



The following predicates are defined on the time intervals (TI1 = (t1,t2), TI2 = (t3,t4), TI3 = (t5,t6)).



meets

```
meets : Interval X Interval -> Boolean

meets(TI1,TI2) <= if t2 = t3 then true
                  else if t2 != t3 then false
                       else unknown
```

overlap

```
overlap : Interval X Interval -> Boolean

overlap(TI1,TI2) <= if t1 <= t4 and t3 >= t4 then true
                    else if not t1 <= t4 and t3 >= t4 then false
                         else unknown
```

same

```
same : Interval X Interval -> Boolean

same(TI1,TI2) <= if t1 = t3 and t2 = t4 then true
                 else if t1 != t3 or t2 != t4 then false
                      else unknown
```

starts

```
starts : Interval X Interval -> Boolean

starts(TI1,TI2) <= if t1 = t3 then true
                   else if t1 != t3 then false
                        else unknown
```

finishes

```
finishes : Interval X Interval -> Boolean

finishes(TI1,TI2) <= if t2 = t4 then true
                     else if t2 != t4 then false
                          else unknown
```

between

```
between : Interval X Interval X Interval -> Boolean

between(TI1,TI2,TI3) <= if t4 < t1 and t2 < t5 then true
                        else if not t4 < t1 or t2 < t5 then false
                             else unknown
```

before

```
before : Interval X Interval -> Boolean

before(TI1,TI2) <= if t2 < t3 then true
                   else if t3 <= t2 then false
                        else unknown
```

during

```
during : Interval X Interval -> Boolean

during(TI1,TI2) <= if t1 >= t3 and t2 <= t4 then true
                   else false
```

contains

```
during : Interval X Interval -> Boolean

during(TI1,TI2) <= if t3 >= t1 and t4 <= t2 then true
                   else false
```

distance

```
distance : Time_point X Time_point -> time_const

distance(t1,t2) <= abs(t1-t2)
```

larger

```
larger : Interval X Interval -> Boolean

larger(TI1,TI2) <= if distance(t1,t2) > distance(t3,t4)
                                                    then true
                   else false
```

shorter

```
shorter : Interval X Interval -> Boolean

shorter(TI1,TI2) <= if distance(t1,t2) < distance(t3,t4)
                                                    then true
                    else false
```

equal

```
equal : Interval X Interval -> Boolean

equal(TI1,TI2) <= if distance(t1,t2) = distance(t3,t4)
                                                    then true
                  else false
```

APPENDIX D

DEAL BNF

```
query           ::=   query-expr
                |     function-defs
                |     ddl
                |     dml

query-expr      ::=   query-block
                |     query-expr set-op  query-block
                |     query-expr ( attr-name , attr_name ) query-block
                |     ( query-expr )

query-block     ::=   relation
                      [ '[' selection-list ']' ]
                      [ where condition ]

set-op          ::=   INTERSECT
                |     UNION
                |     DIFFERENCE
                |     CART

condition       ::=   expression

selection-list  ::=   expression
                |     selection-list , expression

function-defs   ::=   function-id ( param-list )  statement

param-list      ::=   identifier ':' type
                |     param-list , identifier ':' type

type            ::=   INT | REL | ATTR | STRING

statement       ::=   assignment
                |     if-statement
                |     while-statement
                |     function-defs
                |     print-expr
                |     '{' compound-st '}'
                |     function-id ':=' expression

assignment      ::=   identifier ':=' expression
                |     identifier ':=' table-spec

if-statement    ::=   if cond statement
                |     if cond statement else statement
```

```
while-statement ::=     while cond statement

compound-st     ::=     empty
                |       compound-st
                |       compound-st statement

print-expr      ::=     PRINT expression

cond            ::=     ( expression )

expression      ::=     primary
                |       expression binop expression
                |       expression binop table-spec

primary         ::=     identifier
                |       constant
                |       filed-spec
                |       ( expression )
                |       functions

constant        ::=     NUMBER
                |       STRING

binop           ::=     * | / | + | -
                |       < | > | <= | >= | != | =
                |       and | or
                |       && | ||

table-spec      ::=     query-block
                |       ( query-expr )

functions       ::=     function-id ( arg-list )
                |       builtin ( arg-list )

arg-list        ::=     empty
                |       table-spec
                |       arg-list , expression

ddl             ::=     create-table
                |       drop

create-table    ::=     CREATE TABLE name ( field-defn-list )

field-defn-list ::=     field-defn
                |       field-defn-list , field-defn

field-defn      ::=     field-name ( ftype [ NONULL ] )

ftype           ::=     INT
                |       CHAR ( NUMBER )
                |       DATE
```

```
drop            ::=     DROP relation

dml             ::=     insert
                |       delete

insert          ::=     INSERT INTO rel-name ':' insert-spec

insert-spec     ::=     query-expr
                |       lit-tuple

lit-tuple       ::=     ( entry-list )

entry-list      ::=     entry
                |       entry-list , entry

entry           ::=     constant

delete          ::=     DELETE  query-expr

relation        ::=     rel-name
                |       functions

function-id     ::=     name

built-in        ::=     card
                |       first

rel-name        ::=     name

field-name      ::=     name

name            ::=     identifier
```

The ddl statement of DEAL is quite similar to SQL's ddl, except that DEAL has a new data type DATE.

APPENDIX E

HQL BNF

The following nonterminals are not included in the syntax description of HQL because they are identical to their DEAL counterparts: <query-expr>, <function-defs>, <expression>, <ddl>, <delete>, and <insert-spec>.

```
hql             ::=     query
                |       function-defs
                |       ddl
                |       dml
                |       let-in

query           ::=     query-expr
                |       query-expr WHEN time-spec

time-spec       ::=     time-predicate

time-predicate  ::=     binary-ordering
                |       binary-metric

binary-ordering ::=     BEFORE          time-expr
                |       AFTER           time-expr
                |       MEETS           time-expr
                |       OVERLAP         time-expr
                |       STARTS          time-expr
                |       SAME            time-expr
                |       FINISHES        time-expr
                |       CONTAINS        time-expr
                |       BETWEEN ( time-expr , time-expr )

binary-metric   ::=     LONGER          time-expr
                |       SHORTER         time-expr
                |       EQUAL           time-expr


time-expr       ::=     time-primary
                |       time-expr   '+'   time-expr
                |       time-expr   '-'   time-expr

time-primary    ::=     time-const
                |       query-expr
                |       time-points
                |       time-interval
```

```
                |       ( time-expr )
                |       built-in ( time-expr )

time-const      ::=     NOW
                |       D ( integer )
                |       W ( integer )
                |       M ( integer )
                |       Y ( integer )

time-points     ::=     DAY '/' MONTH '/' YEAR

time-interval   ::=     '[' time-expr , time-expr ']'

built-in        ::=     START-OF
                |       END-OF
                |       FIRST

attr-spec       ::=     attr-name
                |       agg-fn ( expression )

agg-fn          ::=     MAX
                |       MIN
                |       AVG
                |       COUNT
                |       SUM

function-call   ::=     TIME-OF ( query-expr )

dml             ::=     insert
                |       delete

insert          ::=     APPEND  rel-name : insert-spec

rel-name        ::=     name

attr-name       ::=     name

integer         ::=     number

name            ::=     identifier
```

APPENDIX F

TENNIS DATABASE

i) A relation called "Tour_results" with attributes (columns) : "player_name", "tour_name", "start", "end". With pkey (primary key) values of "player_name","start" and "end".

For example McEnroe finished 8th i.e. reached the quarter finals, of Wimbledon in 1985.

Tour results

player_name	tour_name	result	start	end
McEnroe	Wimbledon	8	1985	1986
McEnroe	Wimbledon	1	1984	1985
Becker	Wimbledon	1	1985	1986
Becker	Wimbledon	1	1986	1987
Lendl	U.S.Open	2	1983	1984
Lendl	U.S.Open	2	1984	1985
Lendl	U.S.Open	1	1985	1986
Lendl	U.S.Open	1	1986	1987
Lendl	Wimbledon	8	1985	1986
Lendl	Wimbledon	2	1986	1987

ii) A relation called "Nationality" with attributes (columns) : "player_name", "country", "start", "end". With pkey (primary key) values of "player_name","start" and "end".

For example Navratilova was Czeck national from 1956 till 1980, but since 1975 she has been a U.S. national.

Nationality

player_name	country	start	end
McEnroe	U.S.	1959	-
Wilander	Sweden	1963	-
Borg	Sweden	1956	-
Lendl	Czeck.	1960	-
Navratilova	Czeck.	1956	1980
Navratilova	U.S.	1980	-
Becker	Germany	1967	-

iii) A relation called "Ranks" with attributes (columns) : "player_name", "rank", "start" and "end". With Pkey (primary key) values of "player_name", "start" and "end".

For example McEnroe was ranked 1 from 1/6/1984 till 1/9/85 and he has been no 2 since then.

Ranks

player_name	rank	start	end
Wilander	3	11/12/85	-
Becker	4	11/12/85	-
Edberg	5	11/12/85	-
McEnroe	1	1/6/84	1/9/85
Lendl	2	1/6/84	1/9/85
Lendl	1	2/9/85	-
McEnroe	2	2/9/85	-

iv) A relation called "Tour_won" with attributes (columns) : "player_name", "No_won" (no of tourn. won), "start" and "end". With Pkey (primary key) values of "player_name", "start" and "end".

For example Lendl won a total of 5 championships in 1985.

Tour won

player_name	No_won	start	end
McEnroe	2	1985	1986
Lendl	5	1985	1986
Edberg	2	1985	1986
Wilander	3	1985	1986
Becker	3	1985	1986

v) A relation called "Managers" with attributes (columns) : "player_name", "mgr", "start", "end". With pkey (primary key) values of "player_name","start" and "end".

For example Smith was manager of Lendl from June 82 till Nov 85.

Managers

player_name	mgr	start	end
McEnroe	Jones	2/79	10/82
McEnroe	Smith	11/82	-
Lendl	John	1/81	1/82
Lendl	Null1	2/82	6/82
Lendl	Smith	7/82	12/84
Lendl	Jones	1/85	-
Becker	Clark	1/85	-

References

Aho79.A.V. Aho and J.D. Ullman, Principles of Compiler Design, Addison-Wesley, 1979.

Alle83.

J.F. Allen, "Maintaining Knowledge about Temporal Intervals," Communications of the ACM, vol. 26, no. 11, Nov 1983.

Alle84.

J.F. Allen, "Towards a General Theory of Action and Time," A.I., vol. 23, pp. 123-154, North-Holland, 1984.

Ande83.

T.L. Anderson, "Modelling Events and Processes at the Conceptual Level," Proc. of the 2nd Int. Computer Aided Design Conf., pp. 151-168, 1983.

Aria84.

G. Ariav, A. Beller, and H. L. Morgan, "A Temporal Data Model," Technical Report 84-12-03, Dept of DS, University of Pennsylvania, Dec 1984.

Aria86.

G. Ariav, "A Temporally Oriented Data Model," ACM TODS, vol. 11, pp. 499-527, Dec 1986.

Bane87.

J. Banerjee, W. Kim, H.J. Kim, and H.F. Korth, "Semantics and

Implementation of Schema Evolution in Object-oriented Databases," Proc of SIGMOD, ACM, San Francisco, 1987.

Ben-82.

J. Ben-Zvi, "The Time Relational Model," Ph.D. Diss., 1982. University of California

Bolo82.

A. Bolour, T.L. Anderson, L.J. Dekeyser, and H.K.T. Wong, "The Role Time in Information Processing: A survey," ACM SIGMOD Record, vol. 12, pp. 27-50, Apr 1982.

Bube77.

J. Bubenko, "The Temporal Dimension In Information Modeling," in Architecture And Models in Data Base Management Systems,, ed. G.M. Nijssen, North Holland, 1977.

Ceri85.

S. Ceri and G. Gottlob, "Translating SQL into Relational Algebra: Optimisation, Semantics, and Equivalence of SQL Queries," IEEE Trans. Softw. Eng., vol. 11, pp. 324-345, April 1985.

Clif85.

J. Clifford, "On An Algebra for Historical Relational Databases: Two Views," Proc. of the ACM SIGMOD Conference, pp. 247-265, 1985.

Clif87.

J. Clifford and A. Croker, "The Historical Data Model (HRDM) and Algebra Based on Lifespans," Proc of the Int. Conf. on Data

Engineering, IEEE, Los Angeles, Feb 1987.

Codd70.

E.F. Codd, "A Relational Model of Data for Large Shared Data Banks," CACM, vol. 13, June 1970.

Date86.

C.J. Date, Relational Database: Selected Writings, Addison-Wesley, 1986.

Deen85.

S. M. Deen, R. Carrick, and D. M. Kennedy, "A Flexible DBMS for Research and Teaching (PRECI/C)," British National Conference on Databases 85, p. 11, 1985.

Deen85.

S.M. Deen, "DEAL: A Relational Language with Deductions, Functions and Recursions," Data and Knowledge Engineering, vol. 1, 1985.

Deen85.

S.M. Deen, R.R. Amin, and M.C. Taylor, "PRECI* project for distributed databases," Computer Journal, May 1985.

Edga86.

J. Edgar, "Semantics Data Modelling," PhD Thesis, University of Aberdeen, 1986.

Fuji84.

L. Fujitani, "Laser optical disk: The coming revolution in on-line

storage," CACM, vol. 27, pp. 546-554, June 1984.

Gadi85.

S.K. Gadia and J.H. Vaishnav, "A Query Language for a Homogenous Temporal Database," ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, 1985.

Gerr76.

R. Gerritsen and H.L. Morgan, "Dynamic Restructuring of Database with Generation Data Structures," Proc of the ACM Annual Conf., pp. 281-286, ACM, Houston, Oct 1976.

Hirs81.

C. Hirschman and G. Story, "Representing implicit and explicit time relations in narrative," Proc. IJCAI, pp. 289-295, Vancouver, 1981.

Jone79.

S. Jones, P. Mason, and R. Stamper, "LEGOL 2.0: A Relational Specification Language for Complex Rules," Informatin System, vol. 4, pp. 293-305, 1979.

Jone80.

S. Jones and P.J. Mason, "Handling the time dimension in a data base.," Proc of the Int. Conf. on Data Bases, pp. 65-83, July 1980.

Kahn77.

K. Kahn and G.A. Gorry, "Mechanizing Temporal Knowledge," Artificial Intelligence, vol. 9, pp. 87-108, North-Holland, 1977.

Kern78.

B.W. Kernighan and D.M. Ritchie, The C Programming Language, Prentice-Hall, Englewood Cliffs., New Jersey, 1978.

Kern79.

B.W. Kernighan and M.D. McIlroy, "LEX - A Lexical Analyser Generator," The UNIX Programmer's Manual, vol. 2B, Bell Laboratories, Murray Hill, New Jersey, 1979.

Kern84.

Brian W. Kernighan and Rob Pike, The UNIX Programming Environment, Prentice-Hall Inc., Englewood Cliffs., New Jersey 07632, 1984.

Klop81.

M.R. Klopprogge, "TERM: An Approach to include the Time Dimension in the Entity-Relationship Model," in Entity-Relationship Approach to Information Modelling and Analysis, pp. 477-512, 1981.

Lum84.V. Lum, P. Dadam, R. Erbe, J. Guenauer, P. Pistor, G. Walch, H. Werner, and J. Woodfill, "Designing DBMS Support for the Temporal Dimension," in ACM SIGMOD Conf., 1984.

McAr76.

R.P. McArthur, Tense Logic, Reidel, Hingham, Mass., 1976.

McCa71.

J. McCawley, Tense and Time Reference in English, Holt, Reinhardt and Winston, New York, 1971.

McDe82.

D. McDermott, "A Temporal Logic for Reasoning About Processes and Plans," Cognitive Science, vol. 6, no. 2, April-June, 1982.

Miln85.

R. Milner, "The Standard ML Core Language," Polymorphism, vol. 2, pp. 1-28, Oct 1985.

Mont73.

R. Montague, "The Proper Treatment of Quantification in Ordinary English," In Approaches to Natural Language, Reidel, Hingham, Mass, 1973.

Nava86.

S.B. Navathe and R. Ahmed, "A Temporal Relational Model and a Query Language," UF-CIS Technical Report TR-85-16, Florida, April 1986.

Prio67.

A. Prior, Past, Present, Future, Oxford University Press, New York, 1967.

Schi85.

U. Schiel, "The Time Dimension in Information Systems," Information Systems, pp. 67-76, North-Holland, 1985.

Snod84.

R. Snodgrass, "The Temporal Query Language TQuel," ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, pp. 204-213, 1984.

Ston76.

M. Stonebraker, E. Wong, and P. Kreps, "The Design and Implementation of INGRES," ACM TODS, vol. 1, September 1976.

Tans87.

A.U. Tansel and M.E. Arkun, "HQUEL, A Query Language for Historical Relational Databases," Proc. of the 3rd Int. Workshop on Statistical and Scientific Database Management, 1987.

Tayl66.

E.F. Taylor and J.A. Wheeler, Space-Time Physics, Freeman, San Francisco, 1966.

Ullm83.

J.D. Ullman, Principles of Database Systems, MD: Computer Science,, Rockville, 1983.

Vila82.

M.B. Vilain, "A System For Reasoning About Time," Proc. of the National Conf. on A.I., pp. 197-201, 1982.

Wake87.

A.W. Wakelin, A Database Query Language for Operations on Graphical Data, Dundee, 1987. Ph.D Thesis, Dundee College of Technology

Whit80.

G.J. Whitrow, The Natural Philosophy of Time, Oxford University Press, New York, 1980.

Astr76.

M.M. Astrahan, M.W. Blasgen, D.D. Chamberlin, K.P. Eswaran, J.N. Gray, P.P. Griffiths, W.F. King, R.A. Lorie, P.R. McJones, J.W. Mehl, G.R. Putzolu, I.L. Triager, B.W. Wade, and V. Watson, "System R: A Relational Approach to Database Management," ACM TODS, vol. 1, June 1976.

Ahn86.I. Ahn, "Towards an Implementation of Database Management Systems With Temporal Support," Proc. of the 2nd Int'l Conf. on Database Engineering, pp. 374-381, 1986.

Aria82.

G. Ariav and H.L. Morgan, "MDM: Embedding the Time Dimension in Information Systems," Technical Report 82-03-01, 1982. Department of Decision Sciences, University of Pennsylvania

Breu79.

B. Breutmann, E. Falkenberg, and R. Mauer, "CSL: A Language for Defining Conceptual Schemas," Database Architecture, pp. 237-256, North-Holland, 1979. G. Bracchi and G. M. Nijssen, eds.

Woel86.

D. Woelk, W. Kim, and W. Luther, "An Object-Oriented Approach to Multimedia Databases," Proc. of the ACM SIGMOD Conf. on Management of Data, pp. 311-325, Washington, DC, May 1986.

Ande82.

T.L. Anderson, "Modelling Time at the Conceptual Level," in Improving Database Utility and Responsiveness, Academic Press, New York,

1982. P. Scheuermann editor

Clif83.

J. Clifford and D. Warren, "Formal Semantics for Time in Databases," Transactions on Database systems, vol. 8, no. 2, pp. 214-254, June 1983.

Cope84.

G. Copeland and D. Maier, "Making Smalltalk a Database System," Proc. of the ACM SIGMOD conf., pp. 316-325, 1984.

McKe87.

E. McKenzie and R. Snodgrass, "Extending the Relational Algebra to Support Transaction Time," Proc. of ACM SIGMOD Conf. on Management of Data, pp. 467-478, Association of Computing Machinery, San Francisco, May 1987.

McKe87.

E. McKenzie and R. Snodgrass, "Scheme Evolution and the Relational Algebra," Technical Report TR87-003, May 1987. Department of Computer Science, University of North Carolina

Nava80.

S.B. Navathe, "Scheme Analysis for Database Restructuring," ACM TODS, vol. 5, pp. 157-184, June 1980.

Bjor82.

D. Bjorner and C.B. Jones, Formal Specification and Software Development, Prentice-Hall, Englewood Cliffs, NJ, 1982.

Codd79.

E.F. Codd, "Extending the Database Relational Model to Capture More Meaning," ACM TODS, vol. 4, pp. 394-434, Dec 1979.

McKe87.

E. McKenzie and R. Snodgrass, "An Evaluation of Historical Algebras," Technical Report TR87-020, The University of North Carolina, Dept. of Computer Science, October 1987.

Burs80.

R.M. Burstall, D.B. McQueen, and D.T. Sannella, "HOPE: An Expreri-mental Applicative Language," Proc. LISP Conference, pp. 136-143, 1980.

Wied75.

G. Wiederhold, F. Fries, and S. Weyl, "Structured Organisation of Clinical Databases," Proc. of the NCC, 1975.

Cham76.

D.D. Chamberlain, M.M. Astrahan, K.P. Eswaran, P. Griffiths, R.A. Lorie, J.W. Mehl, P. Reisner, and B.W. Wade, "SEQUEL 2: A Unified Approach to Data Definition, Manipulation, and Control," IBM J. Research and Development, pp. 560-575, November 1976.

Date84.

C.J. Date, "Some Principles of Good Language Design," SIGMOD RECORDS, pp. 1-7, 1984.

Tans85.

A. Tansel, "On An Algebra for Historical Relational Databases: Two Views," Proc. of the ACM SIGMOD Conf., pp. 247-265, 1985.

Barb85.

F. Barbic and B. Pernici, "Time Modeling In Office Information Systems," Proc. of ACM SIGMOD Inten. Conf. on Management of Data, pp. 51-62, May 1985.

Snod87.

R. Snodgrass, "The Temporal Query Language TQUEL," ACM TODS, vol. 12, pp. 247-298, June 1987.

Held75.

G.D. Held, M. Stonebraker, and E. Wong, "INGRES - A Relational Data Base System," Proc. NCC, May 1975.

Lesk75.

M.E. Lesk, "Lex - A Lexical Analyser Generator," Comuter Science Technical Report, Murray Hill, New Jersey, October 1975. Bell Laboratories

Date82.

C.J. Date, An Introduction to Database Systems, II, Addison-Wesley, Reading, Mass., 1982.

Snod85.

R. Snodgrass and I. Ahn, "A Taxonomy of Time in Databases," Proc of the ACM SIGMOD conf., pp. 236-246, 1985.

Klug82.

A. Klug, "Equivalence of Relational Algebra and Relational Calculus Query Languages Having Aggregate Functions," J. ACM, vol. 29, pp. 699-717, July 1982.

Date86.

C. J. Date, An Introduction to Database Systems, Volume I, Fourth Edition, Addison-Wesley, Reading, Mass., 1986.