A parametric prototype for spatiotemporal databases

Seo-Young Noh
Iowa State University

Follow this and additional works at: http://lib.dr.iastate.edu/cs_techreports
Part of the Databases and Information Systems Commons

Recommended Citation
http://lib.dr.iastate.edu/cs_techreports/268

This Article is brought to you for free and open access by the Computer Science at Digital Repository @ Iowa State University. It has been accepted for inclusion in Computer Science Technical Reports by an authorized administrator of Digital Repository @ Iowa State University. For more information, please contact digirep@iastate.edu.
A parametric prototype for spatiotemporal databases

Seo-Young Noh

Department of Computer Science
Iowa State University
rsyoung@cs.iastate.edu

April 5, 2004
Chapter 1

Project Summary

1.1 Objectives

The goal of this project is to design and implement the Parametric Databases (ParaDB) for spatiotemporal data. ParaDB consists of two conceptual level descriptions—Parametric Data Model (ParaDM) and Parametric Structured Query Language (ParaSQL).

1.2 ParaDM and ParaSQL

ParaDM has been actively researched since the mid 1980s and models how to store information at abstract level. This model can be used either in a relational data model or in an object-oriented data model. This project conceptually adapts the relational data model to represent the ParaDM. But, there exist some differences compared to the classical relational data model. The main difference is that it has to manage multi-dimensions—space and time, but the classical relational data model does not provide concepts to support these dimensions at user or storage level. Some approaches [9, 14, 15] extend classical relational databases to multidimensional databases as adding attribute fields for time and space, but these approaches have many limitations. One of drawbacks is that information on an object might be stored in separate tuples. It causes the increase of the complexity of query languages in expressing queries because it has to gather the separate tuples for the object in order to query on it.

In ParaDM, the information on an object is described in a single tuple, not multiple tuples. Therefore, each tuple maintains all information on the object over time and space. In order to achieve this, ParaDM considers an attribute value as a function from a parametric element. Parametric element in spatiotemporal databases consists of temporal element, spatial element, and spatiotemporal element. The set of all parametric elements is closed under union ($\cup$), intersection ($\cap$), and complementation ($\neg$). Because of the nature of parametric element, the operations are counterparts of “or,” “and,” and “not” in natural languages, and the closure properties makes substantial simplification in expressing spatiotemporal queries. A temporal element is defined as a finite union of intervals, a spatial element as a finite union of regions\(^1\), and a spatiotemporal element as a finite union of pairs of spatial and temporal elements, respectively.

ParaSQL is a SQL style query language to manipulate and define data over the ParaDM. The essential skeleton of the ParaSQL has been introduced in [33, 34, 35, 38] with many examples, but

---

\(^1\)Arbitrary space for regions will not be considered, instead regions are defined as finite unions of spatiotemporal rectangles in this project.
there is no formal syntax for the language. Therefore, the formal syntax definition will be defined in this project.

All examples provided in the literature are only for navigating tuples, not the internal navigation of tuples. Because an object is stored in a single tuple in the parametric data model, tuple level comparisons are well suited to the model. However, it is important and necessary to provide a method for internal navigations over parametric elements in tuples to extract qualified dimensions for specific conditions. Therefore, the sublanguage for internal navigation over parametric elements should be studied as well as formalized. The grammar for ParaSQL will be described by using IOS/IEC 14977 Extended BNF (Standard EBNF).

1.3 The Scope of Implementation

Based on the conceptual level design of ParaDB, we can divide the ParaDB to be implemented into three main systems—Query Processing System (QPS), Query Execution System (QES), and Storage Managing System (SMS).

The QPS processes ParaSQL queries posed by users and generates physical query plans for the queries. QPS consists of a parser, a logical query planner, and physical query planner. In this project, the implementation of an optimizer is not included because the optimizer itself is a full fledged topic.

The QES executes user queries based on physical query plans generated by QPS. This system contains iterators and a buffer manager to access tuples and improve disk I/Os. Main system of QES is iterators and they are implemented by using HDF5\textsuperscript{2} library.

SMS maintains a system catalog and database files. In this project, Hierarchical Data Format (HDF) which is developed by the National Center for Supercomputing Applications (NCSA) at the University of Illinois, Urbana-Champaign will be adapted as a storage.

The main reasons that HDF5 is chosen as a storage format in this project can be explained into four aspects. Firstly, it supports not only unlimited size of files, but unlimited number of objects in a file. Since information on an object should be stored in a single tuple in the ParaDB, the size of a file is a critical aspect. Secondly, it is reliable. HDF5 has been developed and evolved since the late 1980s to facilitate data exchange between NCSA scientists and is used by many applications such as NASA-Earth Observing System (EOS), DOE’s Advanced Simulation and Computing Program, and so on. Thirdly, it is an open source project. It is valuable when there are needs to modify source code. Lastly, it provides the library tuned and adapted to read and write data efficiently on parallel computing systems. Since the amount of information on objects is very large, supporting parallelization is an important aspect. Even though third and fourth are not directly related to the project, they should be considered as future work.

Table 1.1 shows the categories and tools which will be implemented and used in this project. For the implementation of ParaSQL, Java will be used. Since the HDF5 library is implemented by C, there should exist a system handling between the query processing system and execution system implemented by different programming languages. In order to connect two systems, Java Native Interface (JNI) will be used in this project. For the test of the ParaDB, meteorology and geographical datasets will be used.

\textsuperscript{2}HDF has different products and HDF5 is the latest one.
Table 1.1: Implementation categories and tools

<table>
<thead>
<tr>
<th>Categories</th>
<th>Tools</th>
</tr>
</thead>
<tbody>
<tr>
<td>Query Language</td>
<td>Grammar for Tuple Navigator</td>
</tr>
<tr>
<td></td>
<td>Grammar for Internal Navigator</td>
</tr>
<tr>
<td></td>
<td>ISO/IES EBNF</td>
</tr>
<tr>
<td></td>
<td>ISO/IES EBNF</td>
</tr>
<tr>
<td>Query Processing</td>
<td>ParaSQL Parser</td>
</tr>
<tr>
<td></td>
<td>Logical Query Planner</td>
</tr>
<tr>
<td></td>
<td>Physical Query Planner</td>
</tr>
<tr>
<td></td>
<td>Java and XML</td>
</tr>
<tr>
<td></td>
<td>Java and XML</td>
</tr>
<tr>
<td></td>
<td>Java and XML</td>
</tr>
<tr>
<td>Query Execution</td>
<td>Query Executor</td>
</tr>
<tr>
<td></td>
<td>Iterators</td>
</tr>
<tr>
<td></td>
<td>Buffer Manager</td>
</tr>
<tr>
<td></td>
<td>JNI (Java + C)</td>
</tr>
<tr>
<td></td>
<td>C and HDF5 library</td>
</tr>
<tr>
<td></td>
<td>C</td>
</tr>
<tr>
<td>Storage Manager</td>
<td>Catalog</td>
</tr>
<tr>
<td></td>
<td>Database Files</td>
</tr>
<tr>
<td></td>
<td>HDF5 File Format</td>
</tr>
<tr>
<td></td>
<td>HDF5 File Format</td>
</tr>
</tbody>
</table>

1.4 Contributions

The outcomes and significance of this project can be summarized into two points. Firstly, this project will demonstrate the advantage and appropriateness of the parametric data model for spatiotemporal data for the uniform treatment of space and time. Secondly, benchmarking suits for usability of query languages and performance will be created and it is hoped that these benchmarks will become useful to database community at large.

1.5 Organization

The organization of this proposal is as follows:

- Chapter 2 explains the parametric data model including temporal, spatial, spatiotemporal, and multilevel security databases which can be constructed based on the data model.
- Chapter 3 discusses the ParaSQL in the viewpoint of data manipulation languages (DML) and data definition languages (DDL).
- Chapter 4 discusses the implementation issues and partially implemented ParaSQL and the logical query planner.
- Chapter 5 discusses some temporal data models and temporal query languages and compares them with ParaSQL.
- Chapter 6 discusses some spatial data models and spatial query languages and compares them with ParaSQL.
- Chapter 7 discusses some spatiotemporal data models and spatiotemporal query languages, and compares them with ParaSQL. In this chapter, we will also discuss the needs for developing an internal navigation mechanism for handling specific queries.
- Chapter 8 briefly introduces the hierarchical data format and the programming model.
- Appendix A describes the syntax for the ParaSQL in EBNF.
1.6 Reading Direction

Since reading around 120-page proposal is time consuming work, we recommend the following reading guidelines.

The most important chapters are Chapter 1, 2, 3, and 4 because they are introducing the basic ideas, backgrounds, and the current status of this project. Therefore it is highly recommended to read these chapters.

Chapter 5 through Chapter 7 are the literature reviews for other data models and query languages. Even though the goal of this project is to design and implement only a spatiotemporal database based on the parametric data model, the spatiotemporal database should include spatial, temporal, and spatiotemporal features. This is the main reason that the literature reviews have been allocated to many pages in order to validate that our data model and query language are more suitable for representing and querying spatiotemporal data. Therefore, it is recommended to read these chapters if readers want to look at more detailed explanations that why the ParaDM and ParaSQL provide more flexible expression mechanisms compared to other models and query languages; otherwise, readers can just skip these chapters.

HDF5 is introduced in Chapter 8, even though it is mentioned in Chapter 4 as one of component in a proposed system architecture. Understanding the HDF as a storage model to be used in the proposed system will be sufficient to understand Chapter 4 without reading Chapter 8.
Chapter 2

Parametric Data Model

In this chapter, we will discuss the parametric data model. This model has been studied since the mid 1980s by Gadia [20, 32, 33, 34, 35, 38, 79]. Merits of the parametric model are that it provides a generic model for multi-dimensional databases and the query language for the data model is reflected from natural language queries including “or,” “and,” and “not” without the increase of complexities to express the queries.

2.1 Introduction

In many applications, space, time, and belief dimensions play an elemental role that is revealed in data arising in such applications. Whereas time and space dimensions consist of instants and points in time and space, respectively, the belief domain consists of subjects\(^1\) with varying views of reality. The parametric data model encompasses temporal, spatial, and belief data. Such data is called parametric data. In the parametric data model, there is an underlying parametric space that is simply a set of points. Parametric values are functions from subsets of the parametric space. In temporal, spatial, and belief data, the parametric space consists of a set of instants of time, a set of spatial points, and a set of subjects, respectively.

In order to define the parametric data model in formal, let \(PS\) be the parametric space, \(DB\) be the set of databases, and \(D\) be the set of dimensions. We can define three sets as follows:

\[
PS = \{p_1, p_2, \cdots, p_m\} \\
DB = \{db_1, db_2, \cdots, db_n\} \\
D = \{D_1, D_2, \cdots, D_m\}, \text{ where } m, n > 0
\]

Parametric space \(PS\) can be seen as a set of point \(p_i\). Therefore, point \(p_i\) is defined as a set of meaningful granules of dimension \(D_i\) as below:

\[
p_i = \{x | x \text{ is a meaningful granule of dimension } D_i\}
\]

If a domain is a temporal domain, \(x\) will be an instant of time. If the domain is a spatial domain, it will be a partial point. If the domain is a belief domain, it will be a subject. Intuitively, \(p_i\) and \(D_i\) are identical, but \(p_i\) is seen as a point in the parametric space. Therefore, the different notations are necessary to distinguish between them.

\(^1\)A subject is an active entity, such as a process that can request access to objects, whereas an object is a passive entity, such as a record, a file, or a field within a record. The term, object, used in Bell and LaPadula model, is not the same as an object-oriented DBMS (OODBMS) where objects are active containers of information [58]
Elements in $DB$ represent specific databases corresponding to elements of the parametric space. It is defined as follows:

\[ db_i = \text{a concrete database dealing with data integrated with a subset of } PS \]

Based on the definitions, the parametric data model (ParaDM) can be defined as a function from $PS$ into the domain of $DB$. Therefore, it will be defined as follows:

\[ \text{ParaDM} : PS \rightarrow DB \]

If parametric space elements, $p_t$, $p_s$, and $p_b$ are temporal-, spatial-, and belief- dimension, respectively, concrete databases generated by the parametric data model will be a temporal database, a spatial database, a spatiotemporal database (by combining two elements $p_t$ and $p_s$), and a belief database as shown below:

\[
\begin{align*}
\text{ParaDM}(p_t) &= \text{Temporal Database} \\
\text{ParaDM}(p_s) &= \text{Spatial Database} \\
\text{ParaDM}(p_s, p_t) &= \text{Spatiotemporal Database} \\
\text{ParaDM}(p_b) &= \text{Belief Database}
\end{align*}
\]

Therefore, the parametric data model can be seen as a unified data model for any combination of parametric space elements. Even though the specific features for each dimension are different from a dimension to a dimension, they can be handled with the same manners in an abstract level. This concept makes the parametric data model powerful and expressive without using separate query languages for dimensions.

### 2.2 Parametric Domain

A finite set of named dimensions is postulated. Examples are Longitude (Lo), Latitude (La), Height (He), Time (Ti), and (scientific) Theory (Th). Associated with each dimension is a parametric domain, a universe, that is a set of points together with a structure. It is not required that all dimensions are known a priori, but it is assumed that every dimension has a fixed, unique, universal identity and description. Thus although Longitude and Latitude are spatial dimensions, they are not interchangeable. However, dimensions can be combined. For example, Longitude–Time is a spatiotemporal dimension, Lo×Ti, consisting of set of Longitude–Time ordered pairs.

#### 2.2.1 Closure Properties

Objects span certain domains that are subsets of parametric spaces. In other words, such domains are foot prints of parametric objects (values). Parametric domains must be allowed to have flexible shapes and properties to facilitate object representation and computations required by queries. One important requirement is that to achieve symmetry among “or,” “and,” and “not” of natural language in a query language, the chosen concept of domains must ensure closure with respect to unions, intersections, and complementations. An immediate consequence of this is, for example, that one cannot insist on only having regions that are connected\(^2\).

\(^2\)The United States consists of 50 states that are not connected to each other, for example, Hawaii and Alaska.
2.2.2 Dimension Alignment

Let’s consider following query: “When was the temperature in Ames below freezing?” This natural language sentence treats spatial and temporal domains seamlessly without any boundaries. But in a database, City\(^3\) may be a spatial relation over Lo×La and Temperature could be another relation that contains temperature values on a spatiotemporal grid around earth having Lo×La×He×Ti dimension. In a hypothetical query language, the question reduces to \([[\text{Ames}}] \cap [[\text{Temperature} < 0]]\), where \([[\cdot]]\) denotes the parametric domain of a property, event, or an object. How does one compute this intersection of a spatial operand with a spatiotemporal operand? In the parametric model, that is done by invoking the concept of \textit{dimension alignment}.

Suppose \(\mathcal{R}\) is the universe of space and \(\mathcal{T}\) is the universe of time, respectively. Then the above query can be represented as \(\text{reg}_i \cap (\mu \times \text{reg}_j)\) and it will be treated as \((\mathcal{T} \times \text{reg}_i) \cap (\mu \times \text{reg}_j)\), where \(\text{reg}_i, \text{reg}_j \in \mathcal{R}\) and \(\mu \in \mathcal{T}\) [19].

The idea of the dimension alignment is that all parametric domains are part of a single multi-dimensional universe, even though the whole universe may not be known ahead of time. To carry on a computation one needs to bring only the necessary missing dimensions so that two operands have the same dimension. Once this is done, the set operation (an intersection in this case) can be computed.

2.2.3 Parametric Values

Defining the connection between objects and their domains is the fundamental step to guarantee that the concept of parametric domains will be the base to a good model and a query language. Therefore, it is important to define a \textit{parametric (attribute) value}. In the parametric model, a parametric value is a function from its parametric domain. In spatial, temporal, and belief databases there has been an unfortunate trend to separate the domain and value of a function as two different attributes in a relation. The separation between the domain and the value makes it difficult to include multiple independent values in a single tuple (and relation). For example, in the parametric approach, weather parameters such as temperature, pressure, and humidity can all reside in a single tuple.

The concept of a parametric value poses the biggest challenge in the implementation of parametric databases. The challenge is the largeness, for example, the representation of temperature in a climatic model could run into gigabytes (even terabytes in the future when higher resolutions and longer simulations are used). This raises the issue of scalability and efficiency.

2.3 Parametric Tuples and Homogeneity

Just as in classical relation, a \textit{parametric tuple} is a concatenation of values. The main difference here is that values are parametric values, and they can be very large.

All attributes in a tuple have the same parametric domain. This requirement is called \textit{homogeneity} and it ensures us that an instantaneous snapshot of the tuple does not contain any nulls [33]. The concept of homogeneity is best motivated by considering a temporal (historical) relation. If we consider the state of a temporal tuple at a single instant of time, one essentially obtains a classical relation. If the temporal relation is homogeneous, no snapshot will have nulls. On the other hand, if the temporal relation is not homogeneous, then at some instant of time the snapshot will be a classical relation with null(s). Homogeneity is not a requirement, it is an assumption. One may or may not make this assumption.

\(^{3}\)We assume that City and Temperature relations exist in a parametric database.
2.4 Parametric Relations, Keys, Empty Keys and Restructuring

2.4.1 Parametric Relations and Keys

Every set of tuples is not considered a relation in the parametric model. A key has to be designated for a relation. The key identifies an object uniquely by values that remain invariant in the parametric domain of the object. Formally, a relation $r$ over a scheme $R$, with $K \subseteq R$ as the key of $r$, is a finite set of non-void homogeneous tuples such that no key attribute value in a tuple changes from one point in its domain to another, and no two tuples assume the same key value. Such keys are termed parametrically invariant. There is one object per key and the parametric model preserves the one-to-one correspondence between the tuples in the database and objects in the real world [79].

![MANAGEMENT relation](image)

Let’s consider a database consisting of MANAGEMENT relation. The dependencies Dept $\rightarrow$ Manager, and Manager $\rightarrow$ Dept in MANAGEMENT relation are assumed to hold at every instant of time independently of other instants. The classical methodology for schema design applies, and the same scheme(s) work in the parametric model. The MANAGEMENT temporal relation is shown in Figure 2.1. Dept is designated as the key for the MANAGEMENT relation. The domain of $r$, denoted $[[r]]$, is defined as the union of domains of all tuples in $r$, i.e. $[[r]] = \bigcup_{\tau \in r}[[\tau]]$. $[[\text{MANAGEMENT}]]$, the domain of MANAGEMENT relation of Figure 2.1, is $[11, 60] \cup [71, \text{NOW}]$. The domain of a relation is a parametric element. The restriction of $r$ to parametric element $\mu$, denoted $r \downarrow \mu$, is defined in a natural manner. The snapshot of $r$ at a point $p$, denoted $r(p)$, is defined to be $r \downarrow \{p\}$ [79].

2.4.2 Empty Keys

A key of a relation is a set of attributes. In the classical case one often assumes that the key is nonempty, that is, a key consists of one or more attributes. In the case of spatial data, often only one object exists at a spatial point. In that case there is no need to have any key attributes to identify different objects. Organizing geographical information, such as the United States, with an empty key is akin to having a map in a atlas. Alternatively, such a relation could be organized by using any set of attributes as the key. This may seem paradoxical at first, but minimization of attributes is not necessary in this context. For example, the information about the United States could be organized by cities and states, or by counties and states, or by regions.

It needs to be noted that the whole spatiotemporal climatic cube, consisting of several megabytes (even terabytes) can be realized as a single tuple relation without any key. That a climatic data cube
does not have a key is quite intuitive. After all, it only contains values of climatic parameters such as temperature and pressure at points in space and time dimensions. It does not seem attractive to organize it by temperature or pressure [35].

2.4.3 Restructuring

Because of data dependencies, sometimes more than one choice of a key may be possible for a relation. For example, due to Dept $\rightarrow$ Manager and Manager $\rightarrow$ Dept, either Dept or Manager can be chosen as the key for MANAGEMENT relation. In a classical relation with ordinary data, the contents of a relation do not depend upon the choice of a key, but they do in parametric relations. In Figure 2.1, attribute Dept was chosen as the key of MANAGEMENT relation. What if the choice of the key was to be changed to attribute Manager? Figure 2.2 shows the MANAGEMENT relation with this key. Whereas the former relation represented department objects, the latter represents manager objects.

<table>
<thead>
<tr>
<th>MANAGEMENT_1</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Dept</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[45,60] Shoes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[45,49] Toys</td>
<td></td>
<td>[45,49] Leu</td>
</tr>
<tr>
<td>[41,47] Clothing</td>
<td></td>
<td>[41,47] Tom</td>
</tr>
<tr>
<td>[71,NOW] Clothing</td>
<td></td>
<td>[71,NOW] Inga</td>
</tr>
</tbody>
</table>

Figure 2.2: MANAGEMENT_1 with Manager as the key [35]

What is the relationship between MANAGEMENT and MANAGEMENT_1 relation? Obviously the two relations are not equal; they do not even have the same number of tuples. Formally, two temporal relations $r$ and $s$ over the same scheme are said to be weakly equal if they have the same snapshots at all points in the parametric space. Although MANAGEMENT and MANAGEMENT_1 are not equal, they have the same snapshots at every instant of time. Therefore, they are weakly equal. It is possible to go from one to another by applying the restructuring operator $I_K$. The key of a relation $r$ over a scheme $R$ can be changed to $K$ by computing $I_K(r)$, provided that $K \rightarrow R$ in every snapshot of $r$ [35].

2.5 Algebra for Parametric Model

The algebra in the parametric model is composed of three expressions as follows:

1. A relational expression returns a relation.
2. A domain expression returns a parametric domain.
3. A boolean expression returns a boolean value (TRUE, FALSE).

Some domain and boolean expressions are strictly nonterminal and some are terminal. Relational expressions are always terminal and can be nested inside of the other expressions. The
expression $[[r]]$ that returns the combined parametric domain of tuples in a relational expression $r$. This domain expression is pivotal in making the three types of expressions mutually recursive, user friendly, and efficient compared to other languages where joins (instead of selection) become necessary [35, 38].

2.5.1 Domain Expression

Whereas a classical boolean expression such as “Temperature $>$ 32” returns TRUE or FALSE, in the parametric database it is not a boolean expression because it is true at some points in the parametric space and false at others. In the parametric model, it takes the syntactic form $[[\text{Temperature} > 32]]$. It evaluates to a parametric element, that is, the set of all points $p$ where temperature is greater than 32 degrees. Thus, a boolean in classical becomes a domain in the parametric case.

Domain expressions are the syntactic counterparts of parametric elements. They are formed using constant parametric elements, $[[A]]$, $[[A \theta B]]$, $[[A \theta b]]$, $[[e]]$, $\cup$, $\cap$, and $\neg$, where $A$ and $B$ are attributes, $b$ is a constant, $\theta$ is an operator such as “$>$,” and $e$ is a relational expression [35, 79].

2.5.2 Boolean Expression

Boolean expressions are syntactic counterparts of boolean values TRUE and FALSE. They are formed using $\mu \subseteq \nu$, where $\mu$ and $\nu$ are domain expressions. More complex expressions are formed using $\land$, $\lor$, and $\neg$. Note that expressions of the form $\mu = \nu, \mu \neq \nu$, etc., can be derived using the above constructs. If $t$ is an instant of time, $\{t\} \subseteq \nu$ can be written as $t \in \nu$. In the parametric model, the syntactic form $A \theta B$ is considered an abbreviation for the boolean expression “$[[A \theta B]] \neq \emptyset$.” Because of this convention, the classical SQL queries can be literally embedded in the parametric model.

Boolean expressions in parametric databases are new in the sense that they are not a syntactic counterpart of any classical concept. The operator $\sigma(r, f, \phi)$ returns a tuple of $r$ based upon whether the boolean condition $f$ is satisfied. If it is satisfied, the whole tuple is not returned, but only the part restricted to the domain $\phi$ is.

2.5.3 Relational Expression

1. Union, intersection, and difference

If $r$ and $s$ are relations over the scheme $R$ and with the key $K$, then $r \cup s$, $r - s$, and $r \cap s$ also have the same scheme and key. In the case of union, the tuples of $r$ and $s$ that agree on all key attributes are collapsed together; other tuples of $r$ and $s$ remain unaltered in $r \cup s$. In the case of intersection, if a tuple of $\tau$ of $r$ and $\tau'$ of $s$ agree on all their key attributes, then instants where $t$ and $\tau'$ agree on all attributes are remain in the domain of $\tau$; and other tuples of $r$ removed from the domain of $r$. In the case of difference, if a tuple $\tau$ of $r$ and a tuple $\tau'$ of $s$ agree on all their key attributes, then instants where $t$ and $\tau'$ agree on all attributes are removed from the domain of $\tau$; and other tuples of $r$ remain unchanged in $r - s$ [35].

2. Projection

In the parametric model, a user thinks in terms of relations that have keys. Operator $\pi_X(r)$ defined by $\pi_X(r) = \{r[X] : \tau \in r\}$ is not a user operator because the set of tuples yields by $\pi_X(r)$ lacks a key. However, $\pi_X(r)$, called the internal projection, viewed as an operator on arbitrary sets of tuples, is interesting on its own merit. On one hand, the internal projection
helps in understanding the nature of the projection suitable for users, and on the other, the internal projection can be used to enhance optimization of relational expressions [35].

Now let’s introduce the projection operator suitable for the users of the parametric model. Suppose \( r \) is a relation over the scheme \( R \), \( K \) is the key of \( R \), and \( X \) is a subset of \( R \). There are two cases: either the user has a key in mind for the result of the projection or the user wishes to rely on the system to designate a key.

- If the user has a key \( K' \) in mind (the functional dependency \( K' \rightarrow X \) must be satisfied by \( \pi_X(r) \)), then the syntactic form \( \Pi_{X,K'}(r) \) can be used, which evaluates the relation \( I_{K'}(\pi_X(r)) \), with \( K' \) as its key.

- Alternatively, the determination of a key can be left to the system. In this simple case when the key \( K \) of the input relation is a subset of the projected attributes \( X \), \( \Pi_X(r) \) is defined to be the relation \( \pi_X(r) \), with \( X \) as its key. If \( K \) is not a subset of \( X \), the \( \Pi_X(r) \) is defined to be the relation \( I_X(\pi_X(r)) \), with \( X \) as its key.

The internal projection is inexpensive compared to the user projection. Therefore, during evaluation of expressions the user projection can be substituted by the internal projection [35].

3. Selection

The selection operator of the classical database \( \sigma(r, \phi) \), where \( \phi \) is a boolean expression, also carries over to the parametric model. One interesting twist is that the boolean expression \( \phi \) of classical database embeds as a domain expression in the parametric model. For example, the counterpart of the boolean expression “Dept=Toys” that returns TRUE or FALSE in the classical model is the domain expression \([\text{Dept}=\text{Toys}]\), which returns a parametric element in the parametric model. In addition to the above twist, the selection operator in the parametric model can optionally include a third argument, \( f \), a boolean expression.

Select operators in the parametric data model is of the form \( \sigma(e, f, \phi) \), where \( e \) is a relational expression, \( f \) a boolean expression, and \( \phi \) a domain expression. It is evaluated as below:

\[ \{ \tau \downarrow \phi(\tau) : \tau \in r, f(\tau) \text{ holds, and } \tau \downarrow \phi(\tau) \text{ is not empty} \} \]

If \( f \) evaluates to TRUE for a tuple, \( \sigma \) allows one to select only a relevant part of it, which is specified by \( \phi \). The key of \( \sigma(r, f, \phi) \) is the same as the key of \( r \) [79].

4. Natural Join

Suppose \( r \) and \( s \) are relations with schemas \( R \) and \( S \), respectively. A tuple in the natural join \( r \bowtie s \) of \( r \) and \( s \) is obtained by concatenating a tuple in \( r \) and a tuple in \( s \), and only preserving the instants where both the tuples are defined and agree on their common attributes. Formally, suppose \( \tau_1 \) is a tuple in \( r \) and \( \tau_2 \) is a tuple in \( s \). Then \( \tau_1 \bowtie \tau_2 \) may be defined as the largest homogeneous tuple \( \tau \) over \( RS \) such that \( \tau \) agrees with \( \tau_1 \) on \( R \) and with \( \tau_2 \) on \( S \). Now \( r \bowtie s \) is defined as follows:

\[ \{ \tau_1 \bowtie \tau_2 : \tau_1 \in R, \tau_2 \in S \text{ and } \tau_1 \bowtie \tau_2 \text{ is not null} \} \]

The key of \( r \bowtie s \) is the concatenation of the keys of \( r \) and \( s \) [35].

5. Cross Product

In a tuple of \( r \times s \)–the literal cross product \( r \times s \) of two homogeneous relations \( r \) and \( s \), the attributes of \( r \) may not have the same time domain as the attributes of \( s \). Therefore,
the definition of homogeneity is to stringent to admit \( r \times s \) as a relation. Here it seems appropriate to be independent of each other. This leads to multihomogeneity: A tuple is said to be multihomogeneous if the snapshots of the tuple at vectors of given instants do not contain nulls. Thus, homogeneity has two tiers: unihomogeneity and multihomogeneity. Unihomogeneity is when the parameterization of classical databases is with respect to a single time line; multihomogeneity is when the parameterization is along more than one time lines. In this project, we only consider unihomogeneity [35].

2.6 Parametric Model for Temporal Databases

In this section, we will discuss the usability of the parametric data model for temporal databases.

2.6.1 Temporal Elements

We assume that we are given a universe of time that consists of an interval \([0, \text{NOW}]\) of instants with a linear order \(\prec\) on it. Here NOW denotes the current instant of time. For simplicity, it is assumed that \([0, \text{NOW}]\) is the discrete set \(\{0, 1, \cdots, \text{NOW}\}\) [35, 79].

Time intervals are not adequate to model the history of an object in a single tuple, and they make query languages difficult to use. To obtain timestamps that are closed under the set theoretic operations, the concept of temporal elements is introduced. A temporal element is a finite union of time intervals. A time interval is a temporal element. An instant \(t\) may be identified with the interval \([t, t]\); thus, it may be regarded as a temporal element. Examples of temporal elements are \([11,20]\) \(\cup\) \([31,40]\), or NOW. As expected, the set of all temporal elements is closed under \(\cup, \cap, \\neg\) (complementation with respect to \([0,\text{NOW}]\)) [35].

2.6.2 Temporal Attribute Values

To capture the changing value of an attribute, a temporal value of an attribute \(A\) is defined to be a function from a temporal element into the domain of \(A\). An example of a temporal value of attribute COLOR is \(\langle [25,32] \text{ red}, [33,\text{NOW}] \text{ blue} \rangle\). If \(\xi\) is an temporal value, \([\xi]\) denotes its domain. Thus \([[[[25,30] \text{ red}, [33,\text{NOW}] \text{ blue}]]] = [25,\text{NOW}]\). \(\xi \downarrow \mu\) denotes the restriction of \(\xi\) to the temporal element \(\mu\). A temporal value is also called an attribute value or simply a value [35, 79].

2.6.3 Associative Navigation

Binary operations of the type \(A \theta B\), \(a \theta B\) and \(A \theta b\), where \(A, B\) are attributes and \(a, b\) are constants, are used in selection operations in classical query languages. On tuple substitution, a comparison evaluates to TRUE or FALSE. In the temporal case, we need a way of comparing temporal assignments which are functions of time. Clearly such a comparison does not make sense at instants when either one or both of the assignments are not defined. Also, it may happen that the comparison is TRUE for some instants and FALSE for other instants [79].

The counterpart of the construct \(A \theta B\) of the relational model is \([A \theta B]\), which captures the time when \(A\) is \(\theta\) relationship to \(B\). This is introduced as below:

\([\xi_1 \theta \xi_2] = \{t|\xi_1 \text{ at } t \text{ and } \xi_2 \text{ are defined at } t, \text{ and } \xi_1(t) \theta \xi_2(t) \text{ is } \text{TRUE}\}\)

For example, let \(\xi_1\) and \(\xi_2\) be \(\langle [25,32] \text{ red}, [33,\text{NOW}] \text{ blue} \rangle\) and \(\langle 0,\text{NOW} \rangle \text{ blue} \rangle\), respectively. Then \([\xi_1 = \xi_2] = [33,\text{NOW}]\). The construct \([A \theta b]\) is also allowed, where \(b\) is a constant, which is evaluated by identifying the constant \(b\) with the value \(\langle 0,\text{NOW} \rangle \) b \rangle [35].
2.6.4 Homogeneity and Temporal Tuples

A **homogeneous tuple** $\tau$ over a schema $R$ is a function from $R$ such that for every attribute $A$ in $R$, $\tau(A)$ is a temporal value of $A$ and all the temporal values in the tuple have the same domain. Informally, a tuple is a concatenation of temporal values whose temporal domains are the same. The assumption that all temporal values in a tuple have the same domain makes the tuples homogeneous. Suppose $\tau$ is a tuple. Then the temporal domain of $\tau$ is the temporal domain of any attribute and is denoted by $[[\tau]]$. The tuple is said to be **void** if its domain is empty. A void tuple represents absence of information and such a tuple will not be allowed to occur in a relation. If $\mu$ is a temporal element, $\tau \downarrow \mu$ is obtained by restricting each value in $\tau$ to the temporal element $\mu$.

2.6.5 Temporal Relations

In parametric model, the entire history of a real world object is accumulated in a single tuple. A temporal relation $r$ over $R$, with $K \subseteq R$ as its key, is a finite set of nonempty tuples such that no key attribute value of a tuple changes with time, and no two tuples agree on all their key attributes. A key in parametric model is not required to be minimal or nonempty. Because a relation is required to have a key, every set of tuples over a scheme $R$ is not considered a relation. Figure 2.1 showed relation MANAGEMENT with Dept as its key. In case of MANAGEMENT relation, the Dept attribute also satisfies both the requirements for being a key. Collectively, these two relations form a temporal database [79].

2.7 Parametric Model for Spatial Databases

In this section, we will discuss the usability of the parametric data model for spatial databases.

2.7.1 Spatial Elements

We assume an underlying universal region $\mathcal{R}$. The user views it as a set of points. Let define $\mathcal{REG}$ be a set of subsets of $\mathcal{R}$ which is of interest to users, and that $\mathcal{REG}$ is closed under unions, intersections, subtractions, and complementations. A spatial element is an element of $\mathcal{REG}$. There are no specific assumptions about the constitution of $\mathcal{R}$. $\mathcal{R}$ can be an $n$-dimensional Euclidean space, surface of a sphere, portion of a plane, a curve and so on. We do not assume that $\mathcal{R}$ is discrete or continuous. Main hypothesis is that the regions in $\mathcal{REG}$ should have some reasonable description. A set may be infinite, but its description may be finite. For example suppose $\mathcal{R}$ is the 2-dimensional Euclidean plane $\{(x, y) : x$ and $y$ are real numbers}$}. The upper half plan can be described as $y \geq 0$, and the right half plane as $x \geq 0$. Although the two half planes are infinite, their intersection is easily computed as $x \geq 0 \land y \geq 0$. The union of the two half planes is described as $x \geq 0 \lor y \geq 0$. Complement of this union may simply be described as $\neg(x \geq 0 \lor y \geq 0)$, or $x < 0 \land y < 0$. In the notation, there is no need to describe $\mathcal{R}$ itself in a complicated manner; it is described by the constant predicate TRUE [33].

2.7.2 Spatial Attribute Values

A spatial assignment which would be a function from a spatial element into a domain of an attribute. We are not interested in allowing an arbitrary function to be considered a spatial assignment. A spatial assignment (or simple assignment) $\xi$ to an attribute $A$ is a function from spatial element $\mathcal{REG}$ into the domain of $A$, such that $\xi$ takes only finitely many values, and inverse image of every value taken by $\xi$ is a spatial element of $\mathcal{REG}$. This allows us to represent a spatial assignment
as \( \langle \text{reg}_1 a_1, \text{reg}_2 a_2, \ldots, \text{reg}_m a_m \rangle \), where \( \text{reg}_1, \text{reg}_2, \ldots, \text{reg}_m \) are spatial elements in \( \text{REG} \), and for each \( i, 1 \leq i \leq m, a_i \) is the value of the assignment at every point in \( \text{reg}_i \). Figure 2.3 is an example of a spatial assignment to the attribute Crop in CROP relation, which can also be represented as \( \langle \text{reg}_1 \text{wheat}, \text{reg}_2 \text{corn} \rangle \).

The domain of an assignment is called \textit{spatial domain}. The operator \( [\cdot] \) denotes the spatial domain of a spatial assignment. Thus, if \( \xi = \langle \text{reg}_1 \text{wheat}, \text{reg}_2 \text{corn} \rangle \), then \( [\xi] = \text{reg}_1 \cup \text{reg}_2 \). The restriction of a spatial assignment \( \xi \) as a function to the spatial region \( \text{reg} \) is denoted \( \xi \downarrow \text{reg} \).[33]

2.7.3 Spatial Value Navigation

In the spatial context, we need a way of comparing spatial assignments, i.e., compare functions of space. Clearly, such a comparison does not make sense at points where one or both of the assignments are not defined. Also it may happen that at some points the comparison returns TRUE and at other points it returns FALSE. If \( \xi_1 \) and \( \xi_2 \) are spatial assignments then we define \( [\xi_1 \theta \xi_2] = \{ x : \xi_1 \text{ and } \xi_2 \text{ are defined at } x \text{ and } \xi_1(x) \theta \xi_2(x) \text{ is TRUE } \} \). The construct \( [\xi_1 \theta \xi_2] \) is of fundamental importance in spatial databases. It evaluates the set of points where \( \xi_1 \) is in \( \theta \) relationship with \( \xi_2 \) and its value lies between \( \emptyset \) and \( [\xi_1 \cap \xi_2] \). The value is \( \emptyset \) if the spatial assignments are never related. It is natural to expect \( [\xi_1 \theta \xi_2] \) to be a spatial element. To see this, let \( \xi_1 \) and \( \xi_2 \) be as follows:

\[
\begin{align*}
\xi_1 &= \langle \text{reg}_1 a_1, \text{reg}_2 a_2, \ldots, \text{reg}_m a_m \rangle \\
\xi_2 &= \langle \text{reg}'_1 b_1, \text{reg}'_2 b_2, \ldots, \text{reg}'_n b_n \rangle
\end{align*}
\]

where \( \text{reg}_i \) and \( \text{reg}_j \) are spatial elements in \( \text{REG} \).

Therefore, \( [\xi_1 \theta \xi_2] = \{ \text{reg}_i \cap \text{reg}'_j : 1 \leq i \leq m, 1 \leq j \leq n \text{ and } a_i \theta b_j \text{ holds} \} \) which is a spatial element in \( \text{REG} \) because \( \text{REG} \) is closed under \( \cap \).[33]

2.7.4 Spatial Tuples

A \textit{spatial tuple} is a concatenation of spatial assignments whose spatial domains are the same. The spatial domain of a tuple \( \tau \), denoted \( [\tau] \), is simply the spatial domain of any of its spatial assignments. The assumption that all spatial assignments in a tuple have the same domain is called the \textit{homogeneity} assumption. The \textit{restriction} of a tuple \( \tau \) to a spatial element \( \text{reg} \), denoted \( \tau \downarrow \text{reg} \), is the tuple obtained by restricting every assignment in \( \tau \) to \( \text{reg} \).[33]

2.7.5 Spatial Relations

A \textit{spatial relation} \( r \) over \( R \), with \( K \subseteq R \) as its key, is a finite set of nonempty tuples, such that no key attribute value of a tuple changes from one spatial point to another, and no two tuples agree
on all their key attributes. Figure 2.4 shows COUNTY relation with the schema CName Crop. CName is designated as key; this satisfies both the requirements of a key: within the same county the CName of a county does not change from one place (point) to another, and no two counties have the same CName. The figure also shows STATE relation with the schema SName CName, with SName as its key [33].

<table>
<thead>
<tr>
<th>COUNTY</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CName</td>
<td>Crop</td>
<td></td>
</tr>
<tr>
<td>creg_1 ∪ creg_2</td>
<td>story</td>
<td></td>
</tr>
<tr>
<td></td>
<td>creg_1 wheat</td>
<td></td>
</tr>
<tr>
<td></td>
<td>creg_2 corn</td>
<td></td>
</tr>
<tr>
<td>creg_3 ∪ creg_4 ∪ creg_5</td>
<td>orange</td>
<td></td>
</tr>
<tr>
<td></td>
<td>creg_3 wheat</td>
<td></td>
</tr>
<tr>
<td></td>
<td>creg_4 barley</td>
<td></td>
</tr>
<tr>
<td></td>
<td>creg_5 rice</td>
<td></td>
</tr>
<tr>
<td>creg_6</td>
<td>polk</td>
<td></td>
</tr>
<tr>
<td></td>
<td>creg_6 wheat</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>STATE</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SName</td>
<td>CName</td>
<td></td>
</tr>
<tr>
<td>creg_1 ∪ creg_2</td>
<td>IA</td>
<td></td>
</tr>
<tr>
<td></td>
<td>creg_1 ∪ creg_2 ∪ creg_6</td>
<td>story</td>
</tr>
<tr>
<td></td>
<td>creg_6 polk</td>
<td></td>
</tr>
<tr>
<td>creg_3 ∪ creg_4 ∪ creg_5</td>
<td>CA</td>
<td></td>
</tr>
<tr>
<td></td>
<td>creg_3 ∪ creg_4 ∪ creg_5</td>
<td>orange</td>
</tr>
</tbody>
</table>

Figure 2.4: COUNTY and STATE tables in the parametric database [33]

If \( r \) is a spatial relation over \( R \), then the spatial domain of \( r \), denoted \([r]\), is the union of the spatial domains of all its tuples. From closure properties of \( REG \), \([r]\) is clearly seen to be a spatial element. For example, for the spatial domain of COUNTY relation shown in Figure 2.4, \([COUNTY]\) will be as follows:

\[
[[COUNTY]] = creg_1 ∪ creg_2 ∪ creg_3 ∪ creg_4 ∪ creg_5 ∪ creg_6
= \bigcup_{n=1}^{6} creg_i
\]

2.8 Parametric Model for Spatiotemporal Databases

In this section, we will discuss the usability of the parametric data model for spatiotemporal databases.

2.8.1 Spatiotemporal Elements

We assume that we are given some universal region \( \mathcal{R} \). To this we add the universe of time and obtain the spatiotemporal universe \( \mathcal{R} \times \mathcal{I} = \mathcal{R} \times [0, \text{NOW}] \).
In case of space, we are interested in spatial elements which are closed under unions, intersections and complementations. We have seen in the previous sections that the temporal elements are also closed under these three set operations. These closure properties are essential for seamless querying. In order to maintain this seamlessness, we define a spatiotemporal element to be of the form

\[ reg_1 \times \mu_1 \cup reg_2 \times \mu_2 \cup \cdots \cup reg_n \times \mu_n \]

where for each \( i, 1 \leq i \leq n \), \( reg_i \) is a spatial element and \( \mu_i \) is a temporal element. Clearly, spatiotemporal elements are closed under the three set operations and form a boolean algebra [34].

### 2.8.2 Spatiotemporal Attribute Values

To capture the space and time varying value of an attribute, we introduce a notion of a spatiotemporal assignment. A spatiotemporal assignment to an attribute \( A \) is a function from a spatiotemporal element into the domain of \( A \). For example, \( \langle reg_1 \times [0, 5] \text{ red}, reg_2 \times [0, \text{NOW}] \text{ blue} \rangle \) is an example of spatiotemporal assignment to attribute COLOR. The semantics of this data is “from time 0 to 5 the value of COLOR over the spatial element \( reg_1 \) was red and from time 0 to NOW it was (is) blue over the spatial element \( reg_2 \).”

Note that all the notions introduced in section 2.6 and 2.7 readily extend to the spatiotemporal case. For example, \( [[\cdot]] \) now denotes the spatiotemporal domain of a spatiotemporal assignment. The notions of spatiotemporal tuples and spatiotemporal relations are a natural extension of the definitions given in section 2.6.5 and 2.7.5 [34].

### 2.8.3 Dynamic Alignment of Dimensions

In the spatiotemporal databases, the data can be space independent, time independent, or space and time independent. Now we show how the dimensional seams are removed at the query level. We allow operators to be overloaded. For example, a user can write \( reg \cap \mu \), where \( reg \) is purely a spatial element, and \( \mu \) is a temporal element. The system does the dimension alignment automatically as needed, by padding the whole spaces in the missing dimensions of operands. In this case, \( reg \cap \mu \) will be treated as \( (reg \times T) \cap (R \times \mu) \), where \( T \) is the universe of time, and \( R \) is the universe of space. Such alignment applies to all data types, e.g. attribute values, relations. Another interesting corollary of this phenomenon is that \( [[\cdot]] \) operator can be applied to ordinary data. Thus, in the context of spatial alignment \( [[a]] \) evaluates to \( R \), and in spatiotemporal context it evaluates to \( R \times T \) [19, 34].

### 2.8.4 Spatiotemporal Relations

In this subsection, an application for agriculture environment management will be introduced and the relations for the application will be constructed. The application to be managed is a mix of spatial information, spatiotemporal information and ordinary data. The purpose of the application is to make decisions about the environmental consequences of the concentration of various agricultural chemicals used. Figure 2.5 shows the spatial maps where the application experiment is conducted [19, 34].

**Description of an Application**

We are given a fixed spatial region, which can be assumed as a bounded portion of a plane. In this region, varying soil textures prevail. In the same region, several different crops are being
grown with different tillage methods. Because of various reasons, varying from increasing crop production to pest-control, some chemicals are uniformly applied to the whole region. Some of these chemicals seep through the soil and contaminate ground water. In this application, it is assumed that the seepage depends only on the following: the chemical being applied, the crop type, the tillage method and the soil texture. There are some wells in the region, where the readings of the concentration of various chemicals are taken from time to time. The time is assumed to be acyclic. We are given the (US) Environmental Protection Agency (EPA) data about chemicals. This data specifies the maximum contaminant level and minimum detectable limit allowable concentrations of the chemicals in ground water.

For this application we pair up the wells such that each well belongs to one and only one pair. It is customary to treat the pair of wells as a single entity and classify the wells in the pair as either up-gradient (U/G) or down-gradient (D/G) depending on the direction of ground water flow. The direction of ground water flow is from the up-gradient well to the down-gradient well. The concentration of the chemicals in the down-gradient well is affected by the dilution effect due to the up-gradient well and hence there is a need to classify the wells as up-gradient (U/G) and down-gradient (D/G) to take this effect into account [34].

Figure 2.6 shows how it is modeled as a spatiotemporal database, called RelAgriDB. RelAgriDB consists of four relations as follows:

1. **SOIL**: This relation is spatial, but it is time independent. The key is Texture. This relation
contains information about the texture of soil.

2. CROP: This relation is also spatial, but it is time independent. The key is Crop-Name. This relation contains information about the crops grown in various regions along with the tillage method used.

3. WELL: This relation contains information about the concentration of chemicals in the wells taken at different times. This relation is space and time dependent. The key is Chem-Name.

4. CHEMICAL: This relation shows the environmentally acceptable range of chemicals in the soil, and it is space and time independent. The key of this relation is Chem-Name.

(a) SOIL and CROP relations

<table>
<thead>
<tr>
<th>SOIL</th>
<th>Texture</th>
</tr>
</thead>
<tbody>
<tr>
<td>sreg₁ ∪ sreg₅</td>
<td>Sandy loam</td>
</tr>
<tr>
<td>sreg₂ ∪ sreg₃</td>
<td>Loamy sand</td>
</tr>
<tr>
<td>sreg₄</td>
<td>Clay loam</td>
</tr>
<tr>
<td>sreg₆</td>
<td>Silty clay loam</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CROP</th>
<th>Tillage</th>
</tr>
</thead>
<tbody>
<tr>
<td>creg₁ ∪ creg₅</td>
<td>corn</td>
</tr>
<tr>
<td>creg₁ no till</td>
<td></td>
</tr>
<tr>
<td>creg₅ min till</td>
<td></td>
</tr>
<tr>
<td>creg₂ ∪ creg₃</td>
<td>wheat</td>
</tr>
<tr>
<td>creg₂ conven till</td>
<td></td>
</tr>
<tr>
<td>creg₃ no till</td>
<td></td>
</tr>
<tr>
<td>creg₄ soybean</td>
<td></td>
</tr>
<tr>
<td>creg₄ no till</td>
<td></td>
</tr>
</tbody>
</table>

(b) WELL relation with concentration in parts per billion (ppb)

<table>
<thead>
<tr>
<th>WELL</th>
<th>Chem-Name</th>
<th>U/G Conc</th>
<th>D/G Conc</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>atrazine</td>
<td>p₁ × [0, NOW] 1.0</td>
<td>p₁ × [0, NOW] 0.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>p₂ × [0, 5] 1.5</td>
<td>p₂ × [0, 10] 1.4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>p₂ × [6, NOW] 3.5</td>
<td>p₂ × [11, NOW] 2.9</td>
</tr>
<tr>
<td></td>
<td>simazine</td>
<td>p₁ × [0, 9] 10.0</td>
<td>p₁ × [0, NOW] 9.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>p₂ × [10, NOW] 12.2</td>
<td></td>
</tr>
</tbody>
</table>

(c) CHEMICAL relation with concentration in parts per billion (ppb)

<table>
<thead>
<tr>
<th>CHEMICAL</th>
<th>Max</th>
<th>Min</th>
</tr>
</thead>
<tbody>
<tr>
<td>atrazine</td>
<td>3.00</td>
<td>0.05</td>
</tr>
<tr>
<td>simazine</td>
<td>35.0</td>
<td>0.05</td>
</tr>
</tbody>
</table>

Figure 2.6: A spatiotemporal database for the application [33]
2.9 Parametric Model for Multilevel Security Databases

In this section, we will discuss the usability of the parametric data model for multilevel security databases.

2.9.1 User Levels and User Elements

The parametric model for temporal data discussed in section 2.6 can easily be adapted to multilevel security. The term instant and temporal element are changed to user level and user element, respectively. Corresponding to the universe of time \( \{t_1, t_2, \ldots, t_n\} \) in the temporal case is the universe of user levels \( \{u_1, u_2, \ldots, u_n\} \) in multilevel security. Figure 2.7 shows the relation corresponding to a multilevel security relation [32].

![Figure 2.7: EMP relation for a multilevel security in the parametric data model [32]](image)

2.9.2 The User Hierarchy in Multilevel Security

The primitive \( \subseteq \) on parametric elements leads to a user hierarchy. The user hierarchy gives different users access to different portions of the database. In the parametric model, a user \( u_1 \) is below \( u_2 \) in the user hierarchy if and only if \( \llbracket u_1 \rrbracket \subseteq \llbracket u_2 \rrbracket \), where \( \llbracket u_1 \rrbracket \) and \( \llbracket u_2 \rrbracket \) are the domains assigned by the system to the users \( u_1 \) and \( u_2 \).

In multilevel security, one encounters a special (less general) case of the user hierarchy. The difference is that in multilevel security, the domains are more rigidly determined by the system. A partial order \( \preceq \) among the user levels is postulated and \( \llbracket u \rrbracket \) is defined as \( \{u' : u' \prec u\} \). The following property holds in the user hierarchy:

If \( u_1 \) and \( u_2 \) are user levels, then \( u_1 \preceq u_2 \) if and only if \( \llbracket u_1 \rrbracket \subseteq \llbracket u_2 \rrbracket \)

Note that the primitive \( \llbracket \rrbracket \) of parametric databases can be used to induce a partial order \( \preceq \) in a multilevel security. To understand this, suppose we choose to use \( \llbracket \rrbracket \) as the primitive. When a new user \( u \) enrolls to use the database, the \( \llbracket u \rrbracket \) must be determined for that user. One choice is to let \( \llbracket u \rrbracket \) be one of the existing user domains. In such a case, the user \( u \) is enrolled at an existing user level and no new user level is created. The alternative would be to choose \( \llbracket u \rrbracket \) as a union of some of existing user domains. This is simply a way of saying that the new user is enrolled at a level that is immediately above the users whose domains have been unioned. One more condition should be added to complete the requirements for the case of multilevel security: \( \llbracket u \rrbracket \) must contain the level assigned to \( u \), allowing a user to access his or her own data [32].
In order to present a simple, but intuitive example, assume that the relational algebra contains a relational expression of the form “r,” where r is a relation in the stored database. Let’s consider EMP relation shown in Figure 2.7 is given to us. We assume that the set of users \{u_1, u_2, u_3\} such that \(u_1 \leq u_2\) and \(u_2 \leq u_3\). Suppose that the user \(u_2\) wants to see the current state of the EMP relation. To do this, he or she executes a query for retrieving information about employees. The query retrieves the result shown in Figure 2.8 [32].

<table>
<thead>
<tr>
<th>EMP</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>{u_2} John {u_2}     80K  {u_2}    Toys</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>{u_2} Tom {u_2}      50K  {u_2}    Shoes</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 2.8: The result of the query posed by user \(u_2\) [32]

2.10 Summary

In this chapter, we have discussed the parametric data model. First we discussed the general concept of the model such as parametric elements, domains, values, tuples and homogeneity, and so on. The parametric elements satisfy the closure properties for union, intersection, and complementation. Following the closure properties provides two main advantages: 1) all information on an object can be stored in a single tuple, and 2) union, intersection, and complementation correspond to “or,” “and,” and “not” in natural languages, thus ParaSQL that will be discussed in the next chapter helps users capture “and,” “or,” and “not” of natural languages seamlessly in user queries.

We also discussed the applicability of the parametric model to concrete databases such as temporal databases, spatial databases, spatiotemporal databases, and multilevel security databases. In each section dealing with those databases, we applied the parametric model to those databases, for example, we have defined temporal elements, spatial elements, spatiotemporal elements, and user elements for temporal-, spatial-, spatiotemporal-, and multilevel security databases corresponding to parametric elements. For spatiotemporal databases, the parametric data model simply constructed the concepts for spatiotemporal databases as adapting concepts from temporal and spatial databases That is because all those databases are dealing with dimensions at the abstract level which is the main concept of the parametric data model.

Finally, we can conclude from this chapter that the main advantage of the parametric data model is that it leads to a seamless integration of ordinary, temporal, spatial, spatiotemporal and belief data.
Chapter 3

Parametric Structured Query Language

In this chapter we will discuss the formal syntax and semantics of the parametric structured query language (ParaSQL). In order to describe the syntax of ParaSQL, we follow the standard Extended Backus-Naur Form (EBNF)\(^1\) standardized by IOS (the International Organization for Standardization) and IEC (the International Electronical Commission) [44]. In the following sections, we will discuss the overview of EBNF, the skeleton of ParaSQL, the data definition language, and the data manipulation language in ParaSQL.

3.1 Overview of IOS/IEC 14977 Extended BNF

IOS/IEC 14977 Extended BNF defines a standard syntactic metalanguage\(^2\) based on BNF. It includes the most widely adopted extensions together with additional features which are often required when providing a formal definition. The features of EBNF are as follows:

- **Terminal symbols:** Terminal symbols of the language can be denoted by enclosing them in either double or single quotes, for example, '"x"' or 'x'. This enables any character to be a terminal symbol of the language.

- **Definitions for an explicit number of item:** Fortran contains a rule that a label field contains exactly five characters; an identifier in PL/I or COBOL has up to 32 characters. It is very difficult for a pure BNF to express these rules, but not in EBNF:
  \[
  \text{Fortran label = 5 * character;}
  \]

- **Definition specifying the exceptional cases:** An Algol comment ends at the first semicolon. A rule like this cannot be expressed concisely or clearly in BNF, but in EBNF:
  \[
  \text{comment character = character - ";"}
  \]

\(^1\)According to [44], since the definition of the programming language Algol 60 the custom has been to define the syntax of a programming language formally. Algol 60 was defined with a notation now known as BNF or Backus-Naur From. This notation has proved a suitable basis for subsequent languages but has frequently been extended or slightly altered. The many different notations are confusing and have prevented the advantages of formal unambiguous definitions from being widely appreciated. The syntactic metalanguage Extended BNF is based on Backus-Naur Form and includes the most widely adopted extensions.

\(^2\)A syntactic metalanguage is a formal symbolic language used to describe and reason upon constructs of another language. Therefore, it is a notion for defining the syntax of a language by a number of rules.
• **Comments:** Programming languages and other structures with a complicated syntax need many rules to define them. A syntax will be clear if explanations and cross-references can be provided; accordingly the standard metalanguage contains a comment facility so that ordinary text can be added to a syntax for the benefit of a human reader without affecting the formal meaning of the syntax.

• **Multi-word meta-identifiers:** A meta-identifier\(^3\) need not be a single word or enclosed in brackets because there is an explicit concatenate symbol. This also ensures that the layout of a syntax (except in a terminal symbol) does not affect the language being defined.

Table 3.1 shows the summary of EBNF. The middle column of the table indicates, when appropriate, whether the metalanguage symbol is a prefix operator, or an infix operator, or a postfix operator [71].

<table>
<thead>
<tr>
<th>Extended BNF</th>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>unquoted word</td>
<td></td>
<td>Nonterminal symbol</td>
</tr>
<tr>
<td>&quot;...&quot;</td>
<td></td>
<td>Terminal symbol</td>
</tr>
<tr>
<td>'...'</td>
<td></td>
<td>Terminal symbol</td>
</tr>
<tr>
<td>(...)</td>
<td></td>
<td>Brackets</td>
</tr>
<tr>
<td>[...]</td>
<td></td>
<td>Optional symbols</td>
</tr>
<tr>
<td>{...}</td>
<td></td>
<td>Symbols repeated zero or more times</td>
</tr>
<tr>
<td>{...}-</td>
<td></td>
<td>Symbols repeated one or more times</td>
</tr>
<tr>
<td>=</td>
<td>in</td>
<td>Defining symbol</td>
</tr>
<tr>
<td>;</td>
<td>post</td>
<td>Rule terminator symbol</td>
</tr>
<tr>
<td></td>
<td></td>
<td>in</td>
</tr>
<tr>
<td>,</td>
<td>in</td>
<td>Concatenation symbol</td>
</tr>
<tr>
<td>-</td>
<td>in</td>
<td>Exception symbol</td>
</tr>
<tr>
<td>*</td>
<td>in</td>
<td>Repetition symbol</td>
</tr>
<tr>
<td>(<em>...</em>)</td>
<td></td>
<td>Comment</td>
</tr>
</tbody>
</table>

### 3.2 The Skeleton of ParaSQL

ParaSQL consists of three main expressions in its skeleton—**query expression**, **modification expression**, and **create expression** as shown in Figure 3.1.

**query expression** and **modification expression** are involved in the data manipulation languages (DML) and **create expression** involved in the data definition languages (DDL). Note that all ParaSQL queries should end with symbol ;\(^4\). The most important expression is **query expression** consisting of **relational expression**, **domain expression**, and **boolean expression**. These three expressions are mutually recursive and we discussed them in Chapter 2.

---

\(^3\)Another name of a non-terminal symbol in the language.

\(^4\)In our query examples, we omit the semicolon in order to follow the general way in expressing queries with SQL style query languages, but the semicolon should be added at the end of queries in real query executions.
parametric sql = expression list, ";";
expression list = query expression
| modification expression
| create expression;

Figure 3.1: The syntax of “expression list” in ParaSQL

3.3 Data Definition Language

In this section, we will look at create expression. create expression consists of two meta-identifiers - create database and create table as shown below:

create expression = create database | create table;

These two meta-identifiers basically follow the standard SQL for creating a database and a table. But there are some additional meta-identifiers to express the feature of parametric data model.

3.3.1 create database

The syntax of create database meta-identifier is depicted in Figure 3.2.

create database = "CREATE DATABASE", database name, database parameter list;
database parameter list = database parameter,
{",", database parameter};
database parameter = "DATABASE ID", database id
| "DATABASE SIZE", database size

Figure 3.2: The syntax of “create database” in ParaSQL

CREATE DATABASE is the keyword for creating a parametric database and followed by two meta-identifiers sequentially. database name describes the name of the database to be created and it is an alpha numeric string which is a combination of letters and digits\(^5\). The next meta-identifier, database parameter list, provides the information about an identification and a size of the database. The following example shows how to use CREATE DATABASE statement to create a parametric database.

---

\(^5\)In the description of semantics on meta-identifiers in ParaSQL, only high level meta-identifiers are explained because low level meta-identifiers such as alpha numeric, integer, and letter are straightforward to understand the meaning of the meta-identifiers. The full EBNF is provided in Appendix.
Example: Create a parametric database whose name is EmpDB, id is 200504, and the size of the database is 500 gigabytes, respectively.

CREATE DATABASE EmpDB
DATABASE ID 200504
DATABASE SIZE 500G

3.3.2 create table
The syntax of create table meta-identifier is shown in Figure 3.3.

figure 3.3: The syntax of “create table” in ParaSQL

create table = "CREATE TABLE", table name, column list;

column list = "(" single column description,
               {"," single column description},
               ",", primary key list,
               ")";

single column description = column name, data type,
                           type list;

primary key list = "PRIMARY KEY",
                  "(" column name,
                  { ",", column name},
                  ")" |
                  "EMPTY KEY";

type list = "temporal" | "spatial" | "snapshot"
           | "spatiotemporal" | "belief";

create table meta-identifier starts with keyword CREATE TABLE. table name gives a name of a relational table and column list describes information on columns (attributes) such as a name, a type, and a parametric element of the column. The parametric element can be temporal, spatial, spatiotemporal, belief element. For the sake of simplicity, let’s assume that EmpDB is a temporal database and that we need to define two relations-EMP and MANAGEMENT-so as to record the information about employees and departments. EMP records information such as the name of an employee, the salary, and the departments which he or she worked or is working on. MANAGEMENT relation has to record information about the history of a department and its manager(s). The schemas on two relations are as follows:

EMP (Name: string, Salary: real, Dept: string)
MANAGEMENT (Dept: string, Manager: string)

We only consider spatiotemporal databases for implementation, but the grammar of ParaSQL is extendible to the other dimensional databases.
The underlined attribute is a key of each relation. Based on these schemas, now we can create two tables by using ParaSQL. The following example shows how to create EMP and MANAGEMENT relations.

**Example:** Create EMP and MANAGEMENT relations based on the schemas.

```sql
CREATE TABLE EMP
(
    Name string temporal,
    Salary real temporal,
    Dept string temporal,
    PRIMARY KEY (Name)
)

CREATE TABLE EMP
(
    Dept string temporal,
    Manager string temporal
    PRIMARY KEY (Manager)
)
```

We have to note that unlike classical databases, ParaSQL allows *empty keys* for parametric relations. The empty key is for the case that a relation consists of only one object, that is, there is no requirement to distinguish objects in the relation. This situation might happen in spatial databases, or even in spatiotemporal databases.

### 3.4 Data Manipulation Language

In this section, we will discuss the DML in ParaSQL. The DML is composed of two expressions—modify expression and query expression. Modify expression is composed of insert, delete, and update expressions. We will discuss these expressions in the following subsections. These expressions are to insert, delete, and update tuples in a parametric database. Query expression is the most important expression in ParaSQL because it contains all concepts developed in the parametric data model such as relational expression, domain expression, and boolean expression discussed in Chapter 2. The syntax of the DML is shown in Figure 3.4.

```
modify expression = insert expression
| delete expression
| update expression;

query expression = relational expression
| domain expression
| boolean expression;
```

Figure 3.4: DML in ParaSQL
3.4.1 insert expression

| insert expression = |
| "INSERT INTO", object name, |
| "(" |
| parametric assignment, |
| {"","", parametric assignment}, |
| ")"; |
| parametric assignment = |
| "<", parametric element, value, |
| {"","", parametric element, value, |
| }, |
| ">"; |

Figure 3.5: The syntax of “insert expression” in ParaSQL

The syntax of insert expression is shown in Figure 3.5. insert expression starts with keyword INSERT INTO followed by object name meta-identifier. object name is the name of a relation, and it should exist in a parametric database. A tuple to be inserted consists of parametric assignments corresponding to attributes. parametric assignment identifier consists of two meta-identifiers–parametric element and value\(^7\). The assignment should be parenthesized with "<" and ">" at the beginning of and at the end of the assignment. Therefore the semantics of insert expression will be of the form:

\[
\text{INSERT INTO } r \quad (<\xi_1>, <\xi_2>, \ldots, <\xi_n>)
\]

where \(r\), \(R\), and \(\xi_i\) are a parametric relation, the scheme for \(r\), and a temporal assignment for attribute \(A_i\) such that \(A_i \in R\) for \(1 \leq i \leq n\). If \(\tau\) is the newly inserted tuple by INSERT statement, and \(r'\) is the parametric relation after \(\tau\) is inserted into relation \(r\), then

\[
r' = r \cup \{\tau\}
\]

One thing to note here is parametric assignment \(\xi_i\). Since an attribute has multiple values as an attribute value, \(\xi_i\) can be defined as follows:

\[
\xi_i = \begin{cases} 
\xi_i & \text{one parametric element} \\
\{\xi_{i1}, \ldots, \xi_{ij}\} & \text{otherwise}
\end{cases}
\]

where \(1 < j \leq k\) and \(j, k\) are integers.

In order to understand the DML in ParaSQL more intuitively, let’s assume that a base relation created by section 3.3.2 is given and that some records already exist in the database as shown in Figure 3.6.

The following example shows how to insert tuples by using ParaSQL.

\(^7\)Values in the syntax of the ParaSQL are not parametric values consisting of parametric elements and their corresponding values of the domain of attributes. Values are just values of domains of attributes. Therefore, it should be distinguished from parametric values (or values).
<table>
<thead>
<tr>
<th>EMP</th>
<th>Name</th>
<th>Salary</th>
<th>Dept</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>[50,54] 20K</td>
<td>[45,60] Shoes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[55,60] 25K</td>
<td></td>
</tr>
<tr>
<td>[31, NOW] Leu</td>
<td>[31,NOW] 15K</td>
<td>[31,NOW] Toys</td>
<td></td>
</tr>
<tr>
<td>⋃ [0,44] Mary</td>
<td>[0,40] 25K</td>
<td>[0,40] Credit</td>
<td></td>
</tr>
<tr>
<td>⋃ [50,NOW]</td>
<td>⋃[50,NOW]</td>
<td>⋃[50,NOW]</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.6: Initial EMP table in EmpDB database

**Example:** Insert a tuple for employee Inga and Tom into EMP relation. Inga has been working from 71 to NOW in Clothing department with salary 25K. Tom worked during [0,20] and [41,51] in Hardware and Clothing departments, with salaries 20K and 30K, respectively.

```sql
INSERT INTO EMP
(  <[71,NOW] Inga>,
    <[71,NOW] 25K>,
    <[71,NOW] Clothing>
)

INSERT INTO EMP
(  <[0, 20]+[41,51] Tom>,
    <[0, 20] 20K, [41,51] 30K>,
    <[0,20] Hardware, [41,51] Clothing>
)
```

We have to note here that the tuples inserted into EMP table keep the homogeneity property in the parametric model. The result of the two INSERT statements is shown in Figure 3.7.

### 3.4.2 delete expression

The syntax of delete expression is shown in Figure 3.8. delete expression meta-identifier starts with keyword DELETE followed by object name. An object name is the name of a relation and the relation should exist in a parametric database. WHERE clause in DELETE statement has boolean key expression meta-identifier indicating a condition such that a key attribute has to have a specific value. Since a key in a relation may consist of multiple attributes, attribute $A_i$ should appear in the boolean key expression, if $A_i \in K$ and $K$ is a key of the relation.

The semantics of delete expression will be of the form:

```
DELETE  r
WHERE    K_1 = k_1 \land K_2 = k_2 \land \cdots K_m = k_m
```

where $r$, $K_i$, and $K$ are a parametric relation, a key attribute, and a key for relation $r$, respectively. $K_i$ is a subset of $K$ for $1 \leq i \leq m$. If $R$ is a scheme for $r$, then $K \subseteq R$. $k_i$ is a value for key $K_i$. 
<table>
<thead>
<tr>
<th>Name</th>
<th>Salary</th>
<th>Dept</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>15K</td>
<td>Toys</td>
</tr>
<tr>
<td></td>
<td>20K</td>
<td>Shoes</td>
</tr>
<tr>
<td>Leu</td>
<td>15K</td>
<td>Toys</td>
</tr>
<tr>
<td>Mary</td>
<td>25K</td>
<td>Credit</td>
</tr>
<tr>
<td>Tom</td>
<td>20K</td>
<td>Hardware</td>
</tr>
<tr>
<td></td>
<td>30K</td>
<td>Clothing</td>
</tr>
<tr>
<td>Inga</td>
<td>25K</td>
<td>Clothing</td>
</tr>
</tbody>
</table>

Figure 3.7: After inserting two tuples into EMP table

```sql
delete expression = "DELETE", object name, "WHERE", boolean key expression;

boolean key expression = key condition, 
{"AND", key condition};

key condition = key attribute name, 
"="", parametric value;
```

Figure 3.8: The syntax of “delete expression” in ParaSQL

Let $\tau$ be a tuple identified by values $k_1, k_2, \cdots, k_m$ for $K_1, K_2, \cdots, K_m \in K$ and $r'$ be a relation after deleting $\tau$ from $r$. Therefore, $r'$ can be defined as follows:

$$ r' = r - \{ \tau \} $$

Following example provides a concrete example of delete expression in ParaSQL.

**Example:** Remove information about employee Tom from EMP relation.

```
DELETE EMP
WHERE EMP.Name = 'Tom'
```

The example removes the information such that the name of an employee is “Tom” from EMP relation. Figure 3.9 shows the result of DELETE statement.

### 3.4.3 update expression

The syntax of update expression is shown in Figure 3.10. update expression starts with keyword UPDATE followed by an object name. Since there might be only some of attributes to be updated, the attributes should be updated without changing the other attributes. In order to do
EMP

<table>
<thead>
<tr>
<th>Name</th>
<th>Salary</th>
<th>Dept</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[50,54] 20K</td>
<td>[45,60] Shoes</td>
</tr>
<tr>
<td>[31, NOW] Leu</td>
<td>[31,NOW] 15K</td>
<td>[31,NOW] Toys</td>
</tr>
<tr>
<td>∪ [50,NOW] Mary</td>
<td>[0,40] 25K</td>
<td>[0,40] Credit</td>
</tr>
<tr>
<td>[71, NOW] Inga</td>
<td>[71,NOW] 25K</td>
<td>[71,NOW] Clothing</td>
</tr>
</tbody>
</table>

Figure 3.9: After deleting a tuple for Tom from EMP table

this, ParaSQL provides keyword SET followed by parametric assignments consisting of pairs of an attribute name and a parametric assignment. If there are more than one attribute to be updated, assignments are separated by commas. update expression has WHERE clause to retrieve tuples satisfying conditions indicated by boolean key expression. Like delete expression, each attribute \( K_i \in K \) should appear in a condition made of a conjunctive normal form of key attribute \( K_i \), for \( 1 \leq i \leq m \).

\[
\text{update expression} = \\
\text{"UPDATE"}, \ \text{object name}, \\
\text{"SET"}, \ \text{attribute term}, \\
\text{"="}, \ \text{parametric assignment}, \\
\text{","}, \ \text{attribute term}, \\
\text{"="}, \ \text{parametric assignment} \\
\text{""}, \\
\text{"WHERE"}, \ \text{boolean key expression};
\]

Figure 3.10: The syntax of “update expression” in ParaSQL

The semantics of update expression can be formed as follows:

\[
\text{UPDATE} \ r \\
\text{SET} \ \ A_1 = \xi_1, A_2 = \xi_2, \ldots A_n = \xi_n, \\
\text{WHERE} \ \ K_1 = k_1 \land K_2 = k_2 \land \ldots K_m = k_m
\]

where \( r, A_i, K_j, \) and \( \xi_k \) are a parametric relation, an attribute, a key attribute, and a parametric assignment, respectively. Parametric relation \( r \) is an instance of scheme \( R \) and \( K \) is a key of the relation satisfying \( K_j \in K \) and \( K \subseteq R \). Since attribute \( A_i \) might have multiple values, parametric assignment \( \xi_i \) for \( A_i \) will be either an atomic assignment or multiple assignments to the attribute. In WHERE clause, all key attribute \( K_i \), for \( 1 \leq i \leq m \) should appear to indicate a tuple.

Following example shows how to update a tuple in EMP relation.
Example: Change the salary of John during [55,60] to 30K.

```
UPDATE EMP
WHERE EMP.Name = 'John'
```

The result of the example is shown in Figure 3.11. As we can see, the old value of salary of John has been updated to 30K.

![EMP](image)

Figure 3.11: After updating a tuple for John in EMP table

3.4.4 relational expression

relational expression meta-identifier is the concrete description of relational expression in the algebra of the parametric data model. The syntax of the expression is depicted in Figure 3.12.

In the expression, the symbols +, −, and * represent “union,” “difference,” and “intersection,” respectively. As shown in Figure 3.12, select statement may contain domain expression and boolean expression in RESTRICTED TO and WHERE clause, respectively. attribute list is either * or a set of attribute term consisting of object term and attribute name. The following example shows a query including a relational expression without domain and boolean expressions.

Example: Show department and salary history of employees.

```
SELECT E.Name, E.Dept, E.Salary
FROM EMP E
```

Result: The result of the ParaSQL query will be all tuples as follows:

<table>
<thead>
<tr>
<th>Name</th>
<th>Salary</th>
<th>Dept</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0,44] Mary</td>
<td>[0,40] 25K, [0,40] Credit</td>
<td></td>
</tr>
</tbody>
</table>
relational expression = select statement,
  { ("+", select statement)
   | ("-", select statement)
   | ("*", select statement)
  };

select statement = "SELECT" attribute list,
  ["RESTRICTED TO", domain expression],
  "FROM", object list,
  ["WHERE", boolean expression];

attribute list = "*" | attribute term,
  {"",", attribute term};

object list = object name, object nickname,
  {""," object name, object nickname};

attribute term = object name, ".", attribute name;

3.4.5 domain expression

domain expression meta-identifier is the concrete description of domain expression in the algebra of the parametric data model. The syntax of domain expression is shown in Figure 3.13. domain expression might have domain expression term, or the union/the intersection of two domain expression terms. In order to preserve the precedence among union (+), difference (−), intersection (∗), and complementation (∼), domain expression term and domain expression factor are introduced as low level meta-identifiers. domain expression factor is either atomic domain expression or the complementation of domain expression with/without parentheses. In atomic domain expression, relational expression can be included in the inside of the domain expression [[:]]. Therefore, relational expression and domain expression are mutually recursive because a relational expression may have domain expressions in its RESTRICTED TO clause. The following ParaSQL example shows a query consisting of a relational expression and a domain expression.

Example: Give details about the employees while they worked either in ‘Toys’ or in ‘Shoes’ and
Figure 3.13: domain expression in ParaSQL

did not earn a salary greater than 24K.

```
SELECT *
RESTRICTED TO ([E.Dept = 'Toys'] + [E.Dept = 'Shoes'])
* ~[E.Salary > 24K]
FROM EMP E
```

**Result:** The ParaSQL query retrieves tuples from EMP relation and restricts the tuples to domain expression made up of the intersection of [E.Dept='Toy'] ∪ [E.Dept='Shoes'] and ~[E.Salary>24K]. The domain expression in RESTRICTED TO clause is intersected with each domain of retrieved tuple. If the result is empty, the tuple will not be selected. Therefore, in our example, there exists only one tuple satisfying the restriction and the result will be as follows:

<table>
<thead>
<tr>
<th>Name</th>
<th>Salary</th>
<th>Dept</th>
</tr>
</thead>
<tbody>
<tr>
<td>[11,54]</td>
<td>John</td>
<td>Toys</td>
</tr>
<tr>
<td>[50,54]</td>
<td>John</td>
<td>Shoes</td>
</tr>
</tbody>
</table>
### 3.4.6 boolean expression

boolean expression is a concrete implementation of boolean expression in the algebra of the parametric data model. The syntax of boolean expression is shown in Figure 3.14.

```plaintext
boolean expression = (boolean expression factor
| "NOT", boolean expression factor),
{ ("OR", boolean expression factor)
| ( "AND", boolean expression factor) }
};

boolean expression factor =
atomic boolean expression
| 
| "(" boolean expression ")"
| "NOT", "(" boolean expression ")";

atomic boolean expression =
attribute, operation, value
| attribute, operation, attribute
| domain expression, set operation, domain expression;

set operation = equal | not equal | subsets
| superset | proper subsets
| proper superset | not subsets
| not superset | not proper subsets
| not proper superset;
```

Figure 3.14: The syntax of “boolean expression” in ParaSQL

boolean expression meta-identifier returns either TRUE or FALSE after the evaluation of conditions. It can be either an atomic boolean expression or a boolean expression with/without a negation (NOT). atomic boolean expression can be represented as algebraic comparisons between an attributes value and a constant, or algebraic comparisons between two attributes, or set comparisons between two domain expressions. Algebraic comparisons includes “<,” “≤,” “>,” “≥,” “=,” “!=" and so on. Set comparisons includes “eq,” “!eq,” “⊂,” “⊄,” “⊆,” “⊈,” “⊂,” “⊄,” and so on.

In order to illustrate a query including relational, domain, and boolean expressions, suppose that MANAGEMENT relation defined in section 3.3.2 is given as shown in Figure 3.15. EMP relation shown in Figure 3.11 is added to Figure 3.15.

The following example shows a query consisting of relational, domain, and boolean expressions.

**Example:** Give detail information of employees whose manager is John.

```sql
SELECT *
RESTRICTED TO [[E.Dept = M.Dept]]
FROM EMP E, MANAGEMENT M
WHERE M.Name = 'John'
```
EMP

<table>
<thead>
<tr>
<th>Name</th>
<th>Salary</th>
<th>Dept</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>15K</td>
<td>Toys</td>
</tr>
<tr>
<td>Leu</td>
<td>15K</td>
<td>Toys</td>
</tr>
<tr>
<td>Mary</td>
<td>25K</td>
<td>Credit</td>
</tr>
<tr>
<td>Inga</td>
<td>25K</td>
<td>Clothing</td>
</tr>
</tbody>
</table>

MANAGEMENT

<table>
<thead>
<tr>
<th>Dept</th>
<th>Manager</th>
</tr>
</thead>
<tbody>
<tr>
<td>Toys</td>
<td>John</td>
</tr>
<tr>
<td>Shoes</td>
<td>John</td>
</tr>
<tr>
<td>Toys</td>
<td>Leu</td>
</tr>
<tr>
<td>Clothing</td>
<td>Tom</td>
</tr>
<tr>
<td>Clothing</td>
<td>Inga</td>
</tr>
</tbody>
</table>

Figure 3.15: EMP and MANAGEMENT relations

Result: The ParaSQL query first retrieves tuples from MANAGEMENT relation such that the name of an employee is “John.” It is important to note that the formal definition of literal cross products in the parametric data model is not the same as the classical relational data model. In the parametric data model, it is defined as a set of iterators for the relations in the cross product. In this example, from FROM and WHERE clauses, only one tuple of MANAGEMENT relation will be selected such that the name of manager is “John.” Then it restricts the tuples of EMP relation to the domain expression such that the department of the employee is the same as the department of a manager. The result of the query will be as follows:

<table>
<thead>
<tr>
<th>Name</th>
<th>Salary</th>
<th>Dept</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>15K</td>
<td>Toys</td>
</tr>
<tr>
<td>Leu</td>
<td>15K</td>
<td>Toys</td>
</tr>
<tr>
<td>Mary</td>
<td>25K</td>
<td>Credit</td>
</tr>
<tr>
<td>Inga</td>
<td>25K</td>
<td>Clothing</td>
</tr>
</tbody>
</table>
3.5 Summary

In this chapter, we have discussed the ParaSQL for the parametric data model. ParaSQL is described by EBNF developed by IOS/IEC and consists of the DDL and the DML. The DDL is create expression including create database and create table. DML consists of modification expression and query expression. modification expression mainly inserts, deletes, and updates a parametric database and query expression retrieves tuples from relations. The most important meta-identifier is query expression consisting of relational expression, domain expression, and boolean expression. Each expression is the concrete description of an algebra of parametric model. We have also seen that the formal definition on cross products is not the same as the classical relational data model, instead it is defined as a set of iterators for relations. ParaSQL provides easy ways to express natural language queries including “and,” “or,” and “not” without introducing any additional variables or self-join processes because of the nature of the model such as the closure property.
Chapter 4

Implementation of Parametric Database

In this chapter, we will discuss the implementation issues and the entire system architecture at an abstract level. The abstract system architecture is made up of three layers based on the scope of implementation categories. We will also discuss the current status of the implementation in this project.

4.1 The Scope of Implementation

The scope of the implementation of the parametric database (ParaDB) consists of seven categories—ParaSQL Parser, Logical Query Planner, Physical Query Planner, Storage, Buffering, Iterators, Catalog.

The ParaSQL Parser is the concrete implementation of ParaSQL grammar discussed in Chapter 3. This parser is a syntax analyzer in a compiler viewpoint, that is, it cannot detect any semantic errors from a given ParaSQL query. The second category is Logical Query Planner. This category includes a semantic checker checking the semantics of queries. The Logical Query Planner represents logical query plans with annotations such as $\sigma$, $\phi$, $\times$, and so on. The internal representation of logical query plans will be XML documents. Since XML is very powerful to represent tree structures with self explanations, it is adapted for the internal information exchanges. The third category is Physical Query Planner. The purpose of this is to create a sequence of calls of functions based on a logical query plan. The other categories—Storage, Buffering, and Iterators are related to HDF5 discussed in Chapter 8. As we have discussed in Chapter 1, HDF5 is adapted as the storage model for ParaDB. Therefore, the storage model for ParaDB is basically the same as HDF5. For Buffering and Iterators, they should be studied and implemented by using HDF5 library in this project. For Catalog, it will be constructed as an HDF5 file.

In this chapter, we will discuss three layers of ParaDB. Currently ParaSQL parser and Logical Query Plan are partially implemented and the other modules are not implemented. Therefore we will discuss Layer1 based on two partially implemented modules. For the other layers, we will discuss the brief plans for implementing the modules.

4.2 Abstract System Architecture

ParaDB consists of three main Layers and Reporters shown in Figure 4.1. Layer1 mainly focuses on query processing such as parsing a given ParaSQL query, generating logical query plans and
generating physical query plans. Layer2 executes the query based on the physical query plans, that is, it calls a sequence of functions implemented by using HDF5 library. This layer has Buffer Manager to help reduce the number of disk I/Os. Whenever queries are read to execute, Query Executor checks whether or not certain pages exist in the buffer. If it does, the executor retrieves the pages from the buffer, otherwise, it does from HDF5 file repository. Layer3 is the file repository constructed by HDF5 file storage. The repository contains a system catalog and database files. Whenever a user inserts, deletes, or updates databases, the system catalog will keep the information. Reporters are to generate errors encountered while query processing or executing queries, and to provide query results in which the query has been successfully executed.

![Figure 4.1: Abstract level system architecture](image)

### 4.3 Layer1-Query Processing Layer

The purpose of Query Processing Layer (QPL) is to generate physical query plan from a ParaSQL query posed by a user. This layer consists of three main subsystems—ParaSQL Parser, Logical Query Planner, and Physical Query Planner as shown in Figure 4.2. Semantic Checker is adapted to Logical Query Planner. Optimizer is not included in this proposal, but it should be interacted with Logical Query Planner if it needs to be implemented. In the following subsections, we will discuss the subsystems in detail.

![Figure 4.2: Query Processing Layer](image)
4.3.1 ParaSQL Parser

In order to generate the query plan, it first retrieves a ParaSQL query and analyzes the syntax for the query based on the ParaSQL grammar. Figure 4.3 shows ParaSQL Tracer which has been implemented to parse ParaSQL queries and generate logical query plans.

![ParaSQL Tracer](image)

**Figure 4.3: The ParaSQL Tracer**

In order to illustrate how to parse queries and construct logical query plans, let's assume that a user issues the following ParaSQL query which asks the information of Lue's salary while John was a manager.

```sql
SELECT E1.Salary
RESTRICTED TO [[SELECT M.Manager
               FROM MANAGEMENT M
               WHERE M.Manager = 'John']]
FROM EMP E
WHERE E.Name = 'Lue';
```

Figure 4.4 shows a parse tree constructed by ParaSQL Tracer for the ParaSQL query. The internal representation of the parse tree is an XML document as discussed in section 4.1. As of now, the implementation is only for temporal queries and it needs to be extended.

38
4.3.2 Semantic Checker

The ParaSQL Parser only checks the syntax of the query, that is, it checks only if the query follows the grammar or not. Therefore, QPL needs to introduce a semantic checker for checking if the query is meaningful. For example, suppose that a user poses the following ParaSQL query.

```
SELECT E.Salary
FROM EMPLOYEE EMP
```

The ParaSQL query is correct in the view of syntax, but wrong in the view of semantics because the object name of EMPLOYEE relation in SELECT clause is different from that of FROM clause. This difference cannot be detected by checking syntax because it follows the ParaSQL grammar—attribute term = object name, ".", attribute name;. Therefore, before generating a logical query plan from the query, it is required to check the semantics of the query. The Semantic Checker will check the consistency of object names.
4.3.3 Logical Query Planner

There might be many different ways to generate logical query plans for a ParaSQL query. The best logical query plan might be the plan to get results fast. In order to achieve this, optimization and indexing mechanisms should be introduced. However, they are full fledged topics by themselves in implementing a database system. Since our purpose of this project is to build the ParaDB for spatiotemporal data and it is the initial step to do this, two topics are excluded in the project. But they should be included after successfully implementing the database.

In order to generate a logical query plan (without considering optimization), we need to formalize the steps to construct logical query plans. As we have discussed in the previous chapter, the skeleton of ParaSQL is as follows:

\[
\text{SELECT } x \\
\text{RESTRICTED TO } \mu \\
\text{FROM } r_1, r_2, \cdots, r_n \\
\text{WHERE } f
\]

The Logical Query Planner (LQP) goes through the following four steps suggested in [77] as follows:

1. First a cross product\(^1\) \(r_1 \times r_2 \times \cdots \times r_n\).
2. For each tuple in the cross product, the condition \(f\) is verified.
3. If the tuple does not satisfy \(f\), it is rejected.
4. If the tuple satisfies \(f\), its attributes are restricted to the select list \(x\) and its parametric domain is restricted to \(\mu\).

From the skeleton of ParaSQL, we can construct a parse tree at the abstract level as shown in Figure 4.5.

To construct an abstract logical query plan from a parse tree, we use top down approach to construct it. From the abstract parse tree above, we make a logical execution sequence as follows: \(1 \rightarrow 2 \rightarrow 4 \rightarrow 3\). Note that subparse tree 3 should be the last in the sequence, therefore, it is executed first.

\(^1\)There is no formal definition on the cross product in the parametric data model, but we use notation \(\times\) for indicating that relations are involved in the query processing.
Figure 4.6: The relationship between a parse tree and a logical query plan at the abstract level.

Figure 4.6 shows the relationship between the execution sequence and its abstract level query plan. Every ParaSQL query can be translated into an abstract parse tree and the abstract parse tree can be transformed into an abstract logical query plan. Abstract logical query plans might be divided into four different categories based on whether there exist relational expressions in a ParaSQL query. Let REL be the universal set of relational expressions defined as follows:

$$REL = \{ e | e \text{ is a terminal relational expression.} \}$$

The four categories can be expressed as follows:

1. $$\forall \mu \forall f (e_i \notin \mu \land e_i \notin f)$$
   This case is the simplest query plan because domain and boolean expressions do not include relational expressions. Therefore, the logical execution sequence will be
   $$1 \rightarrow 2 \rightarrow 4 \rightarrow 3$$

   Figure 4.7 shows the abstract logical query plan for this case.

Figure 4.7: Abstract logical query plan for case 1.
2. $\exists_{\mu_j \in \mu}(e_i \in \mu_j) \land \forall_f(e_i \notin f)$

This case is that some domain expressions include relational expressions and there are no relational expressions in boolean expressions. Therefore, $\mu_j$ can be either an atomic domain expression or relational expression and the logical execution sequence will be

$$1 \rightarrow (1 \rightarrow 2 \rightarrow 4 \rightarrow 3) \rightarrow 4 \rightarrow 3$$

Figure 4.8 shows the abstract logical query plan for this case.

![Abstract logical query plan for case 2](image)

3. $\forall_{\mu}(e_i \notin \mu) \land \exists_{f_j \in f}(e_i \in f_j)$

This case is that there are no relational expressions in domain expressions but in boolean expressions. In this case, $f_j$ can be an atomic boolean expression, an atomic domain expression, or a relational expression. Therefore, the logical execution sequence will be

$$1 \rightarrow 2 \rightarrow 4 \rightarrow (1 \rightarrow 2 \rightarrow 4 \rightarrow 3)$$

Figure 4.9 shows the abstract logical query plan for this case.

4. $\exists_{\mu_j \in \mu}(e_k \in \mu_j) \land \exists_{f_j \in f}(e_i \in f_j)$

This case is the extreme case that both domain and boolean expressions contain relational expressions. Figure 4.10 shows the abstract logical query plan for this case, where $\mu_j$ can be either an atomic domain expression or a relational expression. $f_j$ can be an atomic boolean expression, an atomic domain expression, or a relational expression. The logical execution sequence will be

$$1 \rightarrow 2 \rightarrow (1 \rightarrow 2 \rightarrow 4 \rightarrow 3) \rightarrow (1 \rightarrow 2 \rightarrow 4 \rightarrow 3)$$

In this categories, we omit the case that relational expressions include domain and boolean expressions in either SELECT clause or FROM clause because abstract logical query plans for these cases can be easily extended by plugging in the abstract logical query plans introduced in this section.

Figure 4.11 depicts a logical query plan generated by Logical Query Planner in ParaSQL Tracer for the parse tree shown in Figure 4.4.
4.3.4 Physical Query Planner

The last subsystem in LQP is Physical Query Planner (PQP). PQP analyzes logical query plans generated by LQP. It basically extracts information from the logical query plan and generates a sequence of iterators with parameters such as relations and projections of attributes, and so on.

As of now, PQP is not implemented and it will be done after implementing iterators because physical query plans are a sequence of calls for iterators.

4.4 Layer2-Query Execution Layer

The second layer of the ParaDB is Query Execution Layer. This layer consists of five main subsystems-Query Executor, Data Manipulator, Data Definer, Buffer Manager, and HDF5 Library as shown in Figure 4.12. Query Executor manages the other subsystems based on a physical query plan. Buffer Manager handles memory blocks to reduce disk I/Os in the case that certain pages already exist in the buffer. Since iterators retrieve tuples from a disk storage, it can save disk I/Os
if the pages are in the buffer by just retrieving the pages from the buffer. It also helps to modify tuples without accessing disk storage when the pages to be modified are in the buffer. Suppose that there is no buffer manager and a user updates a certain tuple (fitted in a page) three times consequently. In this circumstance, it is required to read and write the page three times, respectively. Therefore, there are six disk I/Os. Then now suppose that there exists a buffer manager and assume that there are no pages in the buffer. In this circumstance, it is required to read and write the page only once. Therefore, there are two disk I/Os. Providing Buffer Manager promises the improved performance of the database.

The Query Executor calls either Data Manipulator or Data Definer based on a physical query plan. Data Manipulator calls the iterators implemented by using HDF5 library. Data Definer creates databases and tables in parametric databases. Both Data Manipulator and Data Definer communicate with the HDF5 repository in Layer 3. Buffer Manager, Data Manipulator, Data Definer and Iterators are implemented by C and they use HDF5 library except Buffer Manager. One thing to note here is that Layer 1 will be implemented by Java. Therefore it is required that Query Executor should be implemented to communicate between two different programming languages without causing gaps. Current proposal will use Java Native Interface (JNI)\(^2\) to do this.

\(^2\)http://java.sun.com/j2se/1.3/docs/guide/jni/
4.5 Layer3-Database Repository

The last layer in the ParaDB is HDF5 database repository. The repository mainly consists of a system catalog and files for the database as shown in Figure 4.13. Since the ParaDB uses HDF5 as its file repository, the system catalog and database files follow the storage mechanism of HDF5.

Figure 4.13: Database Repository

Figure 4.14 shows a detail explanation of constructing the system catalog in ParaDB. The catalog is a hierarchical structure and each database information is attached to the root node in the catalog file. Each database has its own catalog node as a child node of the root and a catalog node for a database consists of relation nodes which the database contains. Each relation has a dataset for the information on the relation. The structure of the actual database files are not decided yet, but they also follow the hierarchical data format.

4.6 Reporters

ParaDB has two reporters shown in Figure 4.15. Error Reporter generates error messages occurred while processing and executing queries. ParaSQL Parser, Logical Query Planner, and Physical Query Planner directly call Error Reporter whenever they have encountered errors and Semantic Checker indirectly calls it via Logical Query Planner in Layer 1. In Layer 2, Error Reporter is
directly called by Query Executor and the other subsystems in the layer call it via Query Executor when errors occurred.

The other reporter–Execution Result Reporter–is called by Query Executor after successfully executing a given query. These two reporters are only connected to Layer 1 and Layer 2, not Layer 3. Since Layer 3 is the lowest level in the ParaDB and they are controlled by HDF5 library, the internal errors are taken care of by HDF5 itself.

Figure 4.15: Error Reporter and Execution Result Reporter in ParaDB

### 4.7 Summary

In this chapter, we have discussed the implementation issues in the ParaDB. The abstract system architecture consists of three layers and the implementations are divided into seven categories–ParaSQL Parser, Logical Query Planner, Physical Query Planner, Storage, Buffering, Iterators and Catalog. ParaSQL Parser parses a ParaSQL query based on the ParaSQL grammar and Logical Query Planner generates a logical query plan for the query. Logical Query Planner uses the Semantic Checker to check the consistency of object names. Physical Query Planner generates a sequence of calls based on the logical query plan. These three systems are in Layer 1.

Buffering and Iterators are in Layer 2 and they are handled by Query Executor to be implemented by JNI. Storage and Catalog are in Layer 3 and they follow the hierarchical data format for constructing the catalog and database files. Figure 4.16 shows the entire system architecture of ParaDB.
Figure 4.16: System architecture of ParaDB
Chapter 5

Temporal Query Languages

5.1 Introduction

Conventional database systems are capable of storing only the current perception of reality and the current relationship among objects. Such databases enforce currency of data by excluding old data values when newer ones become available. After such currency updates, the old values are lost from the logical level and only the current state remains available. On the other hand, temporal database systems are capable of storing multiple versions of data, thereby allowing users the facility of examining complete object histories [3].

Chomicki defined a temporal database as a repository of temporal information and a temporal query language as any query language for temporal databases in [21]. Jensen in [46] argued that most applications of database technology are temporal in nature and listed applicable examples such as financial applications, record-keeping applications, scheduling applications, and scientific applications. He also pointed out that temporal database management is a vibrant field of research and an active community of several hundred researchers had produced some 2000 papers\(^1\).

Temporal databases can be classified into two groups based on schemes of timestamping. The first one is a tuple level timestamping and the second one is an attribute level timestamping. Temporal databases also can be classified into three categories based on the feature of timestamps—point-based, interval-based, and temporal element-based. Therefore, there are six different combinations possible. The parametric database is the attribute timestamping and temporal elements are used as timestamp.

In this chapter, we will discuss temporal data models and their temporal query languages. There exist many temporal query languages and examples include TQuel, TSQL2, SQL\(^T\), IXSQL, SQL/TP, TOLAP, TOSQL, HISQL, ChronoSQL, SQL/Temporal, and so on. Since ParaSQL is an extended SQL and we need to compare it with other temporal query languages, we only select four temporal query languages—SQL/TP, IXSQL, SQL\(^T\), and TSQL2. All queries used in this chapter are from literature describing each query language.

5.2 SQL/TP

5.2.1 SQL/TP Data Model

In SQL/TP, point-based references to time is used as the basis of the query language. The domain of time is viewed as a discrete countably infinite linearly ordered set without endpoints (e.g., the\(^1\)

\(^1\)His thesis was published in April 2000.
integers). The individual element of the set represents the actual time instants while the linear order represents the progression of time [78].

In the comparison with ParaSQL, we use an abstract temporal database with relation INDEPENDENCE defined as follows:

INDEPENDENCE (Name, Year)

The INDEPENDENCE relation captures the independence of countries in central Europe. Figure 5.1 shows the abstract data for INDEPENDENCE relation.

<table>
<thead>
<tr>
<th>INDEPENDENCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
</tr>
<tr>
<td>Poland</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Poland</td>
</tr>
<tr>
<td>Poland</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Poland</td>
</tr>
<tr>
<td>Poland</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Czech Kingdom</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Czechoslovakia</td>
</tr>
<tr>
<td>Czechoslovakia</td>
</tr>
<tr>
<td>Czechoslovakia</td>
</tr>
<tr>
<td>Czechoslovakia</td>
</tr>
<tr>
<td>Czech Republic</td>
</tr>
<tr>
<td>Slovakia</td>
</tr>
<tr>
<td>Slovakia</td>
</tr>
<tr>
<td>Slovakia</td>
</tr>
</tbody>
</table>

Figure 5.1: Independence countries in central Europe [78]

5.2.2 SQL/TP Syntax

SQL/TP consists of two basic syntactic constructs-Select block and Set operations. Select block is the main block of SQL/TP and it is similar to the standard SQL as shown below:

SELECT <list of attribute identifiers>
FROM <sequence of relations>
[WHERE <conditions>]
[GROUP BY <list of attribute identifiers>]

Set operations provide a way to combine the individual select blocks using set operations such as union, except, intersect, and so on.

5.2.3 Examples of Queries

Before the comparison, we need to find a parametric database for Figure 5.1. Figure 5.2 shows the correspondence database for INDEPENDENCE relation in the parametric model.

1. List all countries that were independent while Czech Kingdom was independent².

²We use INDEP instead of INDEPENDENCE.
As we can see, ParaSQL uses `INDEPENDENCE` relation only once, rather than twice. The reason that SQL/TP uses two variables for `INDEPENDENCE` is that the model of SQL/TP stores an object in different tuples. The ParaSQL query retrieves tuples from `INDEPENDENCE` relation and restricts the tuples to the temporal element such that the name of a tuple is ‘Czech Kingdom.’ If the intersection is not empty, then the country in the tuple was independent while Czech Kingdom was independent. Therefore, the intersected temporal elements says the time when two countries were independent.

2. List all years when no country was independent.

ParaSQL:

```
~[[SELECT I.Name FROM INDEP I]]
```

Domain expression gives the temporal element—the finite union of time stamps of each country. Therefore, just complimenting the result from the domain expression gives times when
countries were dependent.

3. List all countries that became independent before Slovakia.

**SQL/TP:**

```sql
SELECT Name
FROM INDEP, (SELECT min(Year) AS y0
FROM INDEP
WHERE Name='Slovakia')
WHERE Year < y0
```

**ParaSQL:**

```sql
SELECT I.Name
FROM INDEP I
WHERE before([r.Name], [r.Name = 'Slovakia'])
```

In this query, we use function `before` which has a prototype as follows:

```
before(te1, te2)
```

It returns TRUE if the first time stamp happens earlier than the second one at any specific time event. Therefore it can be formalized as below if the return value is TRUE:

```
\exists t_1 \in te_1 \forall t_2 \in te_2 (t_1 < t_2)
```

Comparing with SQL/TP, ParaSQL does not need to use a nested query. The ParaSQL query retrieves each tuple from INDEPENDENCE relation, and checks if the temporal element of the tuple happened before Slovakia was independent in a specific time event.

### 5.3 IXSQL

#### 5.3.1 IXSQL Data Model

IXSQL differs from all the other temporal query languages in that it does not provide support for a special, built-in notion of time. Rather IXSQL adds the ability to define columns of a parameterized interval abstract data type, and it provides special query facilities for manipulating tables with rows that have such interval values. There exists at least two different versions of IXSQL and we consider the latter version in [4, 49].

IXSQL stands for Internal Extension to SQL and it is syntactically and semantically upwards consistent with SQL2. *Time intervals* are used to mark the duration of events. For the comparison, we use the following relations. and Figure 5.3 shows a database for the relations [49].

- `TRANSPLANTATION (Name, Date)`
- `DRUG (Name, Drug, Level, Time)`
- `INFECTION (Name, Cause, Time)`
A date has format ‘yyyy-mm-dd,’ but, for simplicity reason, it is represented by an integers, preceded by ‘d.’ Thus, ‘d30’ could represent 1993-05-01. Similarly, [d30, d40) could represent the interval [1993-05-01, 1993-05-11).

5.3.2 Syntax of IXSQL

```
SELECT [sel qualifier] <select list> (1)
FROM <table ref list> (2)
[WHERE <search condition>] (3)
[GROUP BY <grouping column ref list>] (4)
[HAVING <search condition>] (5)
[REFORMAT AS <reformat item>] (6)
[NORMALIZE ON <reformat column list>] (7)
[ORDER BY <sort spec list>] (8)

<reformat item> ::= <reformat column list>
[<reformat item>] | UNFOLD [ALL] <reformat column list>
[<reformat item>]

<reformat column list> ::= <column name>
| <table name>.<column name>
| <correlation name>.<column name>
| <unsigned integer>
```

Line (1)–(5) are executed as in SQL2 and lines (6)–(8), if they exist, are executed in this order. Line 6 (reformats a table with respect to a sequence of columns): REFORMAT AS introduces a sequence of unfold or fold operations which have to be performed on the table retrieved by the
execution of the code in lines (1)–(5). Since REFORMAT AS is applied to the table obtained by the execution of line (1)–(5), the <reformat column list>, which follows in UNFOLD or FOLD, must be a sublist of the columns in <select list>. In particular, if \( R \) is the table obtained by the execution of lines (1)–(5) then FOLD \( A_1, A_2, \ldots, A_n \) and UNFOLD \( A_1, A_2, \ldots, A_n \) are, respectively equivalent to the algebraic operations \( \text{fold}[A_1, A_2, \ldots, A_n](R) \) and \( \text{unfold}[A_1, A_2, \ldots, A_n](R) \) [49].

Basically, unfold operation changes the underlying structure of an interval-based relation to a point-based relation, and fold operation is the reverse of unfold operation.

### 5.3.3 Examples of Queries

Figure 5.4 shows the Parametric database for the relation shown in Figure 5.3.

<table>
<thead>
<tr>
<th>TRANSPLANTATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
</tr>
<tr>
<td>[d30, d30] John</td>
</tr>
<tr>
<td>[d40, d40] Peter</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DRUG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>INFECTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
</tr>
<tr>
<td>( \cup ) [d40, d45]</td>
</tr>
<tr>
<td>( \cup ) [d50, d59]</td>
</tr>
</tbody>
</table>

Figure 5.4: Correspondence relations to Figure 5.3 in the parametric data model

1. Give the level of Cyclosporine administered to John, on each of the dates in \([d32, d38)\).

IXSQL:

```sql
SELECT intersect(Time, [d32, d38)), Level
FROM DRUG
WHERE Name = ‘John’ AND Drug=‘Cyclosporine’
   AND Time cp [d32, d38]
REFORMAT AS UNFOLD 1
```
ParaSQL:

```sql
SELECT D.Level
RESTRICTED TO [d32, d37]
FROM DRUG D
WHERE D.Drug = 'Cyclosporine'
AND D.Name = 'John'
```

In IXSQL, it uses `intervsect` function to compute the intersection of two intervals. For example,

```
intervsect([p5, p10], [p7, p12]) = [p7, p10]
```

The ParaSQL query extracts tuples such that the name is John and drug is Cyclosporine from DRUG relation. There is only one tuple satisfying the condition. It restricts the timestamps with [d32, d37]. Therefore, it provides ⟨[d30, d34], 121⟩ and ⟨[d35, d37], 58⟩. In IXSQL, it uses UNFOLD operation. As we can expect, it would require additional execution time to change the structure of an interval-based relation to a point-based relation.

2. List the days within [d31, d35) on which John was suffering from some disease.

**IXSQL:**

```sql
SELECT intervsect(Time, [d31, d35))
FROM INFECTION
WHERE Name = 'John' AND Time cp[d31, d35)
REFORMAT AS UNFORDL ALL 1
```

**ParaSQL:**

```
count([[ SELECT I.Name
RESTRICTED TO [d31, d34]
FROM INFECTION I ]])
```

Since the domain expression in RESTRICTED TO clause restricts the timestamps in INFECTION relation with [d31, d34], ParaSQL does not need to check whether or not there exist some disease in specific duration. If there is disease in [d31, d34], tuples will be returned with a timestamp such that \( te \cap [d31, d34] \), where \( te \) is a temporal element (or time stamp for the tuple). In the query, we use `count` function. It returns the number of time granule (in this case, day).

3. Give the patients who were administered with Cyclosporine for all the days in [d30, d41).

**IXSQL:**

```sql
SELECT DISTINCT Name
FROM DRUG D1
WHERE [d30, d41] subinterv ANY
(SELECT Time
FROM DRUG D2
```

54
WHERE D1.Name = D2.Name
AND D2.Drug = 'Cyclosporine'
NORMALIZE ON Time)

ParaSQL:

SELECT D.Name
FROM DRUG D
WHERE [d30, d40] SUBSET
[[D.Drug = 'Cyclosporine']]}

The above query examines if the given interval [d30, d40] is a subset of a temporal element such that D.Drug is ‘Cyclosporine’ for each tuple. If it is, then the tuple is selected. As we can see, ParaSQL does not introduce any nested relational expression and additional functions.

5.4 SQL$^T$

5.4.1 SQL$^T$ Data Model

The basic approach of SQL$^T$ is based on a point-based temporal data model and on explicit time queries. Data model in SQL$^T$ assumes that the use of some granularity for representing valid time—for instance days, every temporal relation contains an additional column, say the last column, called VTime, storing single time–granules, and the relation contains one row for each (time) point at which the database fact is valid. Thus, the valid time has become the last column in SQL$^T$ data model. Let’s consider the following relation provided in [14].

PRESCRIPTION (Name, Physician, Drug, Dosage, Frequency, VTime)

Since there are not concrete databases for the relation in [15, 17, 18], let’s assume that there exists an imaginary database for the relation. Things that we have to note here are that the data model used in SQL$^T$ is a point-based, a time column is explicit for queries, and timestamps are labeled in the tuple level. Therefore, we can consider that the data model of SQL$^T$ is similar to that of SQL/TP. In PRESCRIPTION relation, the granularity of VTime is day.

5.4.2 Syntax of SQL$^T$

The syntax of SQL$^T$ is very similar to SQL/TP because both use a point-based model for representing time granules. The following syntax shows the subset of SQL$^T$ syntax.\(^3\)

\[^3\]I could not find a concrete syntax for the SQL$^T$ in [15, 17, 18], but we can construct the essential subset of the language because it is a minimal extension of SQL.
5.4.3 Examples of Queries

1. Find the patients who have been prescribed some drug for more than 240 days.

SQLT:

```
SELECT Name
FROM PRESCRIPT
GROUP BY Name, Drug
HAVING LENGTH (Vtime) > 240
```

ParaSQL:

```
SELECT P.Name
FROM PRESCRIPT P
WHERE length([P.Drug]) > 250
```

In this ParaSQL, we use \texttt{length} function. It returns the number of days for the time stamps for Drug attribute. One thing to note here is that ParaSQL does not use GROUP BY in the query because the parametric data model for the relation is not a point-based—it does not need to coalesce the tuples based on VTime. Since SQL\textsuperscript{T} is built up on a point-based model, it should coalesce the tuples by Name and Drug because the objects might be stored in separate tuples.

2. Find the patients who have been prescribed, throughout 1996.

SQLT:

```
SELECT P.Name
FROM PRESCRIPT AS P
WHERE ((SELECT P1.Name
        FROM PRESCRIPT AS P1
        WHERE P1.Name = P.Name
        AND P1.Drug = P.Drug
        AND P1.Drug = 'Proventil')
        CONTAINS (SELECT C.Vtime
                   FROM Calendar AS C
                   WHERE C.Year = 1996)
        GROUP BY P.Name
```

ParaSQL:

```
SELECT P.Name
FROM PRESCRIPT P
WHERE interval(year, 1996)
    SUBSET [[P.Drug = 'Proventil']]
```

The above example shows that the query can be easily expressed by ParaSQL. In the query, it uses \texttt{interval} function returning the time interval for year 1996. When retrieving each tuple, it checks whether or not the timestamp for P.Drug = ‘Proventil’ contains the interval. If it is,
then it extracts the name of a patient. That is, the patient has been prescribed throughout 1996. As we can see, ParaSQL query does not introduce additional variables for PRESCRIPT and nested SELECT statements. Since the data model used by SQLT maintains an object in different tuples, it should introduce additional variables to check two tuples are for the same object.

3. Find the patients who have been prescribed some drugs for more than 240 consecutive days.

**SQLT:**

```sql
CREATE VIEW PartitionedP (Name, Drug, PerNo, VTime) AS
SELECT P1.Name, P1.Drug, COUNT(P2.VTime), P1.VTime
FROM PRESCRIPT AS P1, P2
  AND P1.VTime >= P2.VTime
  AND NOT EXIST (SELECT P3.*
                 FROM PRESCRIPT AS P3
                 WHERE P3.VTime = P2.VTime - 1
                   AND P3.Name = P2.Name
                   AND P3.Drug = P2.Drug)
GROUP BY P1.Name, P1.Drug, P1.VTime

SELECT Name
FROM Partitioned P
GROUP BY Name, Drug, PerNo
HAVING LENGTH (Vtime) > 240
```

**ParaSQL:**

```sql
SELECT P.Name
FROM PRESCRIPT P
WHERE interval(day, 240)
  SUBSET [[P.Drug]]
```

In this example, the ParaSQL query uses `interval` function. The first argument, day, is to indicate the interval is for day and the result will return the interval for 240 consecutive days. In SQLT, it creates a view to query the example, but the ParaSQL query just uses a function. It checks if the interval for 240 days is a subset of a domain of Drug attribute. If it is, then there exists an interval for 240 consecutive days in the temporal element, that is, there exists a patient who has been prescribed some drugs for more than 240 consecutive days. As we can see, the ParaSQL query can be expressed in easy and natural ways without introducing another view and a complicated nested SELECT statements.

5.5 **TSQL2**

**TSQL2 Data Model**

The TSQL2 [4, 21, 75, 76] specification was developed at the University of Arizona By Richard Snodgrass. TSQL2 data model is based on tuple timestamping and 1NF. Time in TSQL2 is multi-dimensional-(valid time and transaction time, or bitemporal). Valid time concerns the time when
a fact is true in reality. Transaction time concerns the time the fact was present in the database as stored data. TSQL2 data model uses implicit timestamps. The transaction time of facts are supported by the system itself. In contrast, the valid times of facts are usually supported by the user. Let’s consider the following relations from [75].

**EMPLOYEE**(ID, Name, Salary, Gender, D-brith, DeptName)

**SKILL**(EmpID, Name)

**DEPARTMENT**(DeptName, Budget, MgrID)

Figure 5.5 shows table instances for EMPLOYEE, SKILLS, and DEPARTMENT relations.

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Salary</th>
<th>Gender</th>
<th>D-birth</th>
<th>DeptName</th>
<th>VTime</th>
</tr>
</thead>
<tbody>
<tr>
<td>ED</td>
<td>Ed</td>
<td>20</td>
<td>M</td>
<td>7/1/55</td>
<td>Toy</td>
<td>[2,4]</td>
</tr>
<tr>
<td>ED</td>
<td>Ed</td>
<td>30</td>
<td>M</td>
<td>7/1/55</td>
<td>Toy</td>
<td>[5,10]</td>
</tr>
<tr>
<td>ED</td>
<td>Ed</td>
<td>40</td>
<td>M</td>
<td>7/1/55</td>
<td>Toy</td>
<td>[11,16]</td>
</tr>
<tr>
<td>ED</td>
<td>Ed</td>
<td>40</td>
<td>M</td>
<td>7/1/55</td>
<td>Book</td>
<td>[17,NOW]</td>
</tr>
<tr>
<td>DI</td>
<td>Di</td>
<td>30</td>
<td>F</td>
<td>10/1/60</td>
<td>Toy</td>
<td>[0,7]</td>
</tr>
<tr>
<td>DI</td>
<td>Di</td>
<td>40</td>
<td>F</td>
<td>10/1/60</td>
<td>Toy</td>
<td>[8,12]</td>
</tr>
<tr>
<td>DI</td>
<td>Di</td>
<td>50</td>
<td>F</td>
<td>10/1/60</td>
<td>Toy</td>
<td>[13,NOW]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>EmpID</th>
<th>Skill</th>
<th>VTime</th>
</tr>
</thead>
<tbody>
<tr>
<td>ED</td>
<td>Typing</td>
<td>[3,NOW]</td>
</tr>
<tr>
<td>ED</td>
<td>Filing</td>
<td>[9,NOW]</td>
</tr>
<tr>
<td>ED</td>
<td>Driving</td>
<td>[0,4]</td>
</tr>
<tr>
<td>DI</td>
<td>Directing</td>
<td>[6,19]</td>
</tr>
<tr>
<td>DI</td>
<td>Directing</td>
<td>[0,NOW]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Budget</th>
<th>MgrID</th>
<th>VTime</th>
</tr>
</thead>
<tbody>
<tr>
<td>Toy</td>
<td>150</td>
<td>DI</td>
<td>[0,7]</td>
</tr>
<tr>
<td>Toy</td>
<td>200</td>
<td>DI</td>
<td>[8,14]</td>
</tr>
<tr>
<td>Toy</td>
<td>100</td>
<td>DI</td>
<td>[15,NOW]</td>
</tr>
<tr>
<td>Book</td>
<td>50</td>
<td>ED</td>
<td>[18,NOW]</td>
</tr>
</tbody>
</table>

Figure 5.5: EMPLOYEE, SKILLS, and DEPARTMENT relations in TSQL2 data model

### 5.5.1 Syntax of TSQL2

The syntax of TSQL2 can be referred in [76]. The following syntax shows a subset of the syntax of TSQL2.

```
SELECT [SNAPSHOT] <select list>
FROM  <table source> [AS <correlation>]
[WHERE <search condition>]
[<group  by clause>]
[<having clause>]
```

4In [75], the timestamps are formatted as [mm/dd/yyyy-mm/dd/yyyy]. In order to simplify the relations, we ordered the timestamps and allocated unique numbers to the timestamps. The queries are not affected by the changes.
SNAPSHOT specifies that the resulting table will be a snapshot table. The SNAPSHOT keyword tells the query to return a table without a valid-time column, i.e., a non-temporal table [59].

### 5.5.2 Examples of Queries

Figure 5.6 shows relation tables in a parametric database corresponding to relation tables shown in Figure 5.5.

1. Find ED’s salaries when he worked in the same department as DI.

   **TSQL2:**
   ```sql
   SELECT SNAPSHOT E3.Salary
   FROM EMP(ID, DeptName) AS E1 E2,
   E1(Salary) AS E3
   WHERE E1.ID = 'ED' AND E2.ID = 'DI'
   AND E1.DeptName = E2.DeptName
   ```

   **ParaSQL:**
   ```sql
   SELECT E1.Salary
   RESTRICTED TO [[E1.DeptName=E2.DeptName]]
   FROM EMP E1, EMP E2
   WHERE E1.ID = 'ED' AND E2.ID = 'DI'
   ```

   The ParaSQL query retrieves tuples such that the IDs of employees are ED and DI from two EMPLOYEE relations, respectively. Therefore, only one tuple is retrieved from each relation. It is important to understand that relations in FROM clause will not be cross-producted, instead they are separately retrieved based on WHERE conditions. We will discuss it later. After retrieving tuples, it restricts the tuples to a temporal element satisfying that the names of departments are same each other, that is, it retrieves the time when two employees worked together in the same department. If the intersection is not empty with the domain in RESTRICTED TO clause, employee ED worked with employee DI during the finite union of intervals. Therefore, each interval has its correspondent salary and the salaries will be shown as a result with the time stamps. As we can see in this example, the ParaSQL query uses two variables for EMPLOYEE relation, but TSQL2 uses three.

2. Find the names of departments that always had a budget greater than $90K during the times when managed by someone named Di.

   **TSQL2:**
   ```sql
   SELECT SNAPSHOT D2.Name
   FROM DEPT(ID, Name, MgrID) AS D1
   DEPT(ID, Budget) (PERIOD) AS D2
   EMP(ID, Name) AS M
   WHERE D1.MgrID = M.ID AND M.Name = 'Di'
   AND D2.Budget > 90 AND D1.ID = D2.ID
   AND VALID(D2) CONTAINS VALID(D1)
   ```

---

5In the following examples, we use EMP and DEPT as names of EMPLOYEE and DEPARTMENT, respectively
Figure 5.6: Relations corresponding to Figure 5.5 in the parametric data model
ParaSQL:

```sql
SELECT D.DeptName
FROM DEPT D
WHERE [[D.MgrID='DI']] SUBSET [[D.Budget > 90]]
```

The above ParaSQL query retrieves tuples satisfying that the domain of manager DI is a subset of the domain of the budget greater than $90K. If there exists a tuple, the department always had a budget greater than $90K while DI was a manager. The reason that TSQL2 uses a variable for EMPLOYEE relation is that an object is stored in different tuples. But, in the parametric data model, we know that the ID of an employee should be unique. Therefore, the ParaSQL query in this example does not need to use several relations and variables.

3. When did ED work in Toy department while the department was managed by DI?

TSQL2:

```sql
SELECT SNAPSHOT
    INTERSECT(VALID(E1), VALID(E2))
FROM EMP(ID, Name, Salary, DeptName) AS E1, E2
    DEPT(MgrID, Name) AS D1
WHERE E1.Name = 'Di' AND E2.Name = 'Ed'
    AND E1.ID = D1.MgrID
    AND D1.Name = 'Toy'
    AND E2.DeptName = 'Toy'
    AND VALID(D1) OVERLAPS VALID(E2)
```

ParaSQL:

```sql
SELECT E.Name
RESTRICTED TO [[D.MgrID='DI']]*[[D.DeptName='Toy']]
FROM EMP E, DEPT D
WHERE E.ID = 'ED'
```

The query statement in this example can be written in a ParaSQL in very simple way. The above ParaSQL query retrieves tuples such that the ID of an employee is ‘ED.’ Note that the query does not need to find the id of employee ‘Ed’ because ID attribute is the primary key of EMPLOYEE relation in the parametric data model. But in TSQL2 data model, it needs a condition finding the ID of employee ‘Ed’ because the object for the employee is stored in several tuples. After retrieving the tuples, it restricts the tuples to the intersection of two temporal dimensions satisfying that the ID of a manager is ‘DI’ and the name of the department is ‘Toy.’ The RESTRICTED TO clause guarantees that employee ED worked during the intersection if the intersection is not empty.

As we have seen in the examples provided in [76], the queries can be expressed by using ParaSQL queries in simple and natural ways.

5.6 Summary

In this chapter, we discussed four different temporal query languages and the query examples provided in the papers describing each language. As we have seen, ParaSQL can express the queries in easy and natural ways.
Temporal data model introduced in this chapter can be classified into two groups—implicit time data model and explicit time data model based on approaches to access time granules. Therefore, TSQL2 is in the implicit time data model and the others in explicit time data model. Because of the implicit time, TSQL2 time columns cannot be explicitly referred in the SELECT and WHERE clauses of an SQL query [17].

The temporal query languages except ParaSQL also can be classified into two categories—point-based models and interval-based models. SQL\textsuperscript{T} and SQL/TP use point-based model and TSQL2 use interval-based model. IXSQL can change the underlying interval-based structure to point-based structure by calling `unfold` function.

Because of the drawbacks of interval-based and time implicit model, there is no implementation of TSQL2 even though TSQL2 is the standard for temporal query languages [59].

No matter what models—either a point-based model or an interval-based model—the temporal query languages are built on, they cannot satisfy the closure properties on union, intersection, and complementation. That is the main reason that the queries expressed by SQL/TP, IXSQL, SQL\textsuperscript{T}, and TSQL2 are so complicated compared to those by ParaSQL.

\footnote{In [15], TSQL2 is classified as an interval-based model, but the authors in [4] argue that TSQL2 is classified as a point-based model.}
Chapter 6

Spatial Query Languages

6.1 Introduction

The purpose of spatial databases is to correlate data in space and they provide answers to questions such as how far has waste product extended from the spill location? How many miles away is the closest hospital of his house? Most spatial databases do not stand on their own, but instead are just an extension to relational databases. They use a dialect of SQL called Spatial Feature Structured Query Language—which simply adds spatial functions to SQL such as distance, touches, centroid, inside, area, and extent [63]. Table 6.1 shows the usability of Spatial Database Management Systems (SDBMS).

<table>
<thead>
<tr>
<th>User</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mobile phone user</td>
<td>Where is the nearest gas station?</td>
</tr>
<tr>
<td>Army field commander</td>
<td>Has there been any significant enemy troop movement since last night?</td>
</tr>
<tr>
<td>Medical doctor</td>
<td>Based on this patient’s MRI, have we treated somebody with similar condition?</td>
</tr>
<tr>
<td>Farmer</td>
<td>How can I minimize the use of pesticide on my farm?</td>
</tr>
<tr>
<td>Emergency service</td>
<td>Where is the person calling for help located?</td>
</tr>
<tr>
<td>Transport specialist</td>
<td>How should the road network be expanded to minimize traffic congestion?</td>
</tr>
</tbody>
</table>

At the time of writing, there are many query languages for spatial database management systems found in literature. They can be classified into two categories—extended SQL style query languages and visual spatial query languages. Extended SQL style query languages are SQL/OGIS [61, 72, 73], PSQL [69], QL/G [12], Spatial SQL [27], GeoSQL [80], CSQL [54], GEOQL [60], SQL/SDA [48] and so on. Visual spatial query languages are Spatial-Query-by-Sketch [25, 26], Query-by-Visual-Example [43], Cigales [11], and so on.

Authors in [55] argue that the extended SQL approach is the more natural interface language to query a spatial database since the SQL query language is now widely accepted to query a relational
database. Therefore, in this chapter, we will discuss four different extended SQL style spatial query languages—SQL/OGIS, QL/G, SQL/SDA, and PSQL, and compare them with ParaSQL.

6.2 SQL/OGIS

6.2.1 SQL/OGIS Data Model

The OGIS\(^1\) consortium was formed by major software vendors to formulate an industry wide standard related to GIS interoperability. The OGIS spatial data model can be embedded in SQL3. The OGIS Consortium [61] has standardized spatial feature geometry and spatial operations. The OGIS specification defines a standard for SQL which supports the storage and query of spatial data. The spatial data is based on the OGIS Geometry Object Model in [61]. The non-instantiable class Geometry serves as the base class with subclasses for Point, Curve(line) and Surface(Polygon).

Conceptually, spatial entities are stored as tables with geometry valued columns. Instances of the entities are stored as rows in a table. Datatypes of spatial attributes are drawn from the Geometry Model while those of non-spatial attributes are from SQL3. Implementation of a spatially-enabled table called the feature table, are described for two target environments: SQL3 and SQL3 with Geometry Types. In the SQL3 environment, a geometry-valued column is implemented as a Foreign Key reference into a geometry table. A geometric value is stored using one or more rows in the geometry table. The geometry table may be implemented using either standard SQL numeric types or SQL binary types. In SQL3 with Geometry Types, a geometry-valued column is implemented as a column whose SQL type is drawn from the set of Geometry Types.

The SQL functions(methods) specified by the OGIS specification fall into three categories: 1) basic functions on the Geometry datatypes, 2) operators for testing topological relationships, and 3) functions that support spatial analysis.\(^2\)

We define COUNTRY, CITY, and RIVER relations in SQL/OGIS data model as follows:

\[
\begin{align*}
\text{COUNTRY}(\text{Name, Cont, Pop, GDP, Life-Exp, Shape: Ploygon}) \\
\text{CITY}(\text{Name, Country, Pop, Captital, Shape: Point}) \\
\text{RIVER}(\text{Name, Origin, Length, Shape: LineString})
\end{align*}
\]

The primary keys for each relation scheme are Name attribute in each relation and only Shape attributes are indicated with OGIS data types. Figure 6.1 shows the tables of the World database.

6.2.2 Syntax of SQL/OGIS

The syntax of SQL/OGIS follows the SQL. OGIS provides Geometry Types and functions used in SQL3. Geometry types includes Point, Curve, Surface and Geometry Collection. Each geometric object is associated with a Spatial Reference System, which describes the coordinate space in which the geometric object is defined. Table 6.2 shows the functions used in SQL/OGIS.

Examples of Queries

For the comparison with ParaSQL, we transform relations in Figure 6.1 into a parametric database. Figure 6.2 shows the tables of World database in the parametric data model. We change Polygonid, Pointid and LineStringid to reg, preg, lreg in the parametric data model, respectively.

\(^1\)OGIS stands for Open Geodata Interchange Standard.

\(^2\)For more detail information, refer to [61, 73].
1. Find the names of all countries which are neighbors of the United States in the COUNTRY table.

**SQL/OGIS:**

```sql
SELECT C1.Name AS 'Neighbors of USA'
FROM COUNTRY C1, COUNTRY C2
WHERE Touch(C1.Shape, C2.Shape) = 1
AND C2.Name = 'USA'
```

**ParaSQL:**

```sql
SELECT C.Name
RESTRICTED TO [[C.Name = 'USA']]"
FROM COUNTRY C
WHERE C.Name != 'USA'

The above ParaSQL query retrieves tuples from COUNTRY relation, and checks whether or not the name of a country is 'USA.' If it is, then it rejects the tuple because the query asks to find all neighbors of the USA. It is different from SQL/OGIS-see the second boolean expression in SQL/OGIS. If the name is not 'USA,' then the spatial element of the tuple is restricted to the spatial element such that Name attribute is 'USA.' If there exists intersected area between two spatial elements, the country is a neighbor of the USA.

Here, we have to note that the ParaSQL query introduces only one variable for COUNTRY relation, but SQL/OGIS does not. Therefore, the query structure of ParaSQL can be executed much simpler than that of SQL/OGIS because it can avoid join processes.

2. The St. Lawrence River can supply water to cities that are within 300 km. List the cities that can use water from the St. Lawrence.

<table>
<thead>
<tr>
<th>Function Type</th>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic Functions</td>
<td>SpatialReference()</td>
<td>Returns the Reference System of the geometry</td>
</tr>
<tr>
<td></td>
<td>Envelope()</td>
<td>The minimum bounding rectangle of the geometry</td>
</tr>
<tr>
<td></td>
<td>Export()</td>
<td>Convert the geometry into a different representation.</td>
</tr>
<tr>
<td></td>
<td>IsEmpty()</td>
<td>Tests if the geometry is a empty set or not.</td>
</tr>
<tr>
<td></td>
<td>IsSimple()</td>
<td>Returns TRUE if the geometry is simple.</td>
</tr>
<tr>
<td></td>
<td>Boundary()</td>
<td>Returns the boundary of the geometry.</td>
</tr>
<tr>
<td>Topological/</td>
<td>Equal</td>
<td>Tests if the geometries are spatially equal.</td>
</tr>
<tr>
<td>Set Operators</td>
<td>Disjoint</td>
<td>Tests if neither interiors nor boundaries intersect.</td>
</tr>
<tr>
<td></td>
<td>Intersect</td>
<td>Tests if the geometries intersect.</td>
</tr>
<tr>
<td></td>
<td>Touch</td>
<td>Tests if the boundaries intersects and interior does not.</td>
</tr>
<tr>
<td></td>
<td>Cross</td>
<td>Tests if the geometries cross each other.</td>
</tr>
<tr>
<td></td>
<td>Within</td>
<td>Tests if the given geometry is within another given geometry.</td>
</tr>
<tr>
<td></td>
<td>Contains</td>
<td>Tests if the given geometry contains another given geometry.</td>
</tr>
<tr>
<td></td>
<td>Overlap</td>
<td>Tests if the interiors intersect.</td>
</tr>
<tr>
<td>Spatial Analysis</td>
<td>Distance</td>
<td>Returns the shortest distance between two geometries.</td>
</tr>
<tr>
<td></td>
<td>Buffer</td>
<td>Returns a geometry that represents all points whose distance from the given is less than or equal to the specified distance.</td>
</tr>
<tr>
<td></td>
<td>ConvexHull</td>
<td>Returns the convex hull of the geometry.</td>
</tr>
<tr>
<td></td>
<td>Intersection</td>
<td>Returns the intersection of two geometries.</td>
</tr>
<tr>
<td></td>
<td>Union</td>
<td>Returns the union of two geometries.</td>
</tr>
<tr>
<td></td>
<td>Difference</td>
<td>Returns the difference of two geometries.</td>
</tr>
<tr>
<td></td>
<td>SymDiff</td>
<td>Returns the symmetric difference of two geometries.</td>
</tr>
</tbody>
</table>
### COUNTRY

<table>
<thead>
<tr>
<th>Name</th>
<th>Cont</th>
<th>Pop(millions)</th>
<th>GDP(billions)</th>
<th>Life-Exp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Canada</td>
<td>NAM</td>
<td>30.1</td>
<td>658.0</td>
<td>77.08</td>
</tr>
<tr>
<td>Mexico</td>
<td>NAM</td>
<td>107.5</td>
<td>694.3</td>
<td>69.36</td>
</tr>
<tr>
<td>Brazil</td>
<td>SAM</td>
<td>183.3</td>
<td>1004.0</td>
<td>65.60</td>
</tr>
<tr>
<td>Cuba</td>
<td>NAM</td>
<td>11.7</td>
<td>16.9</td>
<td>75.95</td>
</tr>
<tr>
<td>USA</td>
<td>NAM</td>
<td>270.0</td>
<td>8003.0</td>
<td>75.75</td>
</tr>
<tr>
<td>Argentina</td>
<td>SAM</td>
<td>36.3</td>
<td>348.2</td>
<td>70.75</td>
</tr>
</tbody>
</table>

### CITY

<table>
<thead>
<tr>
<th>Name</th>
<th>Country</th>
<th>Pop(millions)</th>
<th>Capital</th>
</tr>
</thead>
<tbody>
<tr>
<td>Havana</td>
<td>Cuba</td>
<td>2.1</td>
<td>Y</td>
</tr>
<tr>
<td>Washington, D.C</td>
<td>USA</td>
<td>3.2</td>
<td>Y</td>
</tr>
<tr>
<td>Monterrey</td>
<td>Mexico</td>
<td>2.0</td>
<td>N</td>
</tr>
<tr>
<td>Toronto</td>
<td>Canada</td>
<td>3.4</td>
<td>N</td>
</tr>
<tr>
<td>Brasilia</td>
<td>Brazil</td>
<td>1.5</td>
<td>Y</td>
</tr>
<tr>
<td>Rosario</td>
<td>Argentina</td>
<td>1.1</td>
<td>N</td>
</tr>
<tr>
<td>Ottawa</td>
<td>Canada</td>
<td>0.8</td>
<td>Y</td>
</tr>
<tr>
<td>Mexico City</td>
<td>Mexico</td>
<td>14.1</td>
<td>Y</td>
</tr>
<tr>
<td>Buenos Aires</td>
<td>Argentina</td>
<td>10.75</td>
<td>Y</td>
</tr>
</tbody>
</table>

### RIVER

<table>
<thead>
<tr>
<th>Name</th>
<th>Origin</th>
<th>Length(kilometers)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rio Parana</td>
<td>Brazil</td>
<td>2600</td>
</tr>
<tr>
<td>St. Lawrence</td>
<td>USA</td>
<td>1200</td>
</tr>
<tr>
<td>Rio Grande</td>
<td>USA</td>
<td>3000</td>
</tr>
<tr>
<td>Mississippi</td>
<td>USA</td>
<td>6000</td>
</tr>
</tbody>
</table>

Figure 6.2: The World database in the parametric data model

**SQL/OGIS:**

```sql
SELECT C.Name
FROM City C, River R
WHERE Overlap(C.Shape, Buffer(R.Shape, 300) = 1
  AND R.Name='St. Lawrence'
```

**ParaSQL:**

```sql
SELECT C.Name
RESTRICTED TO extend([[R.Name = 'St. Lawrence']], 300)
FROM City C, River R
```
In ParaSQL, it uses `extend` function. The prototype of the function is defined as follows:

\[
\text{extend}(\text{region}, x)
\]

It extends the region to \( x \) km from its boarder. ParaSQL first gets an extended region such that the name of a river is ‘St. Lawrence.’ Then it restricts the spatial elements to the region. If there exists an intersected region, then the city can use water from the river.

3. List the length of the rivers in each of the countries they pass through.

**SQL/OGIS:**

```sql
SELECT R.Name, C.Name,
     Length(Intersection(R.Shape, C.Shape) AS Length
FROM RIVER R, COUNTRY C
WHERE Cross(R.Shape, C.Shape) = 1
```

**ParaSQL:**

```sql
SELECT R.Name, C.Name, Length([R.Name]*[C.Name]) AS Length
FROM RIVER R, COUNTRY C
WHERE [R.Name]*[C.Name] != empty
```

In this example, we have to note that ParaSQL uses a set operation-intersection to check whether or not a river passes through a country. SQL/OGIS uses two functions-`Intersection` and `Cross`, but two functions are doing the same work in this query.

### 6.3 QL/G

#### 6.3.1 QL/G Data Model

QL/G stands for Query Language for Geometric databases. It has been developed at the University of Waterloo and is intended to be a general-purpose spatial query language for manipulating both alphanumeric as well as geometric data. QL/G is a modular, strongly-typed functional language with an SQL flavor. QL/G is an extension of SQL and the data model of QL/G is a nested relational model extended with geometric data and operators. However, the geometric data and operators are not dependent on the nested relational model. The geometric component is independent of any existing set-oriented data models, and it takes a modular and functional approach to the design of the query language to achieve the objective. The basic constructs or operators in QL/G can be considered as functions. A function takes in arguments and yields some output. A query is a function whose arguments may be other queries/functions. This feature allows queries be combined in any manner to produce more complicated queries [12].

In QL/G, there are six disjoint sets as shown in Table 6.3. The set \( R \cup S \cup \{\text{TRUE, FALSE}\} \) is said to be the set of *atomic values*. The tokens `REAL`, `STR` and `BOOLEAN` denote the *atomic type*. The tokens `POINT`, `LINE`, `REGION`, `LINE*`, `REGION`, `REGION*` are the *geometric data types*. `REGION*` is of type either `REGION`, `LINE` or `POINT`. Similarly, `LINE*` denotes either type of `LINE` or `POINT`. 

68
Table 6.3: Six disjoint set in QL/G

<table>
<thead>
<tr>
<th>Set</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( R )</td>
<td>the set of real numbers</td>
</tr>
<tr>
<td>BOOLEAN</td>
<td>the set of {TRUE, FALSE}</td>
</tr>
<tr>
<td>( B )</td>
<td>{REAL, STR, BOOLEAN, POINT, S_LINE, POINT, S_LINE, S_REGION, LINE, LINE*, REGION, REGION*}</td>
</tr>
<tr>
<td>( A )</td>
<td>a countably infinite set of symbols which are called attributes.</td>
</tr>
<tr>
<td>( C )</td>
<td>a countably infinite set of symbols which are called relation names.</td>
</tr>
</tbody>
</table>

6.3.2 Syntax of QL/G

QL/G has two main categories—structure transformation operators and geometric operators. Structure transformation operators are an extension of constructs in SQL to a nested relational model. Geometric operators are defined independently from the nested relational operators, and any other suitable set of operators that produce set values could replace those in structure transformation operators. The language includes algebraic operators like union and minus as well as all SQL built-in functions such as distinct, min, max, count, sum and average. However, the operators union, minus and distinct are extended to accept geometric values.

```
SELECT tuple(newname_1: result_1(x_1, \ldots, x_n),
            \ldots,
            newnames_k: result_k(x_1, \ldots, x_n))
FROM  x_1 IN expr_1,
      \vdots
      x_n IN expr_n
WHERE  qualifications;
```

Every SELECT-FROM-WHERE block returns a set of tuples defined on \(k\) attributes. There are several major differences from the popular SQL. Table 6.4 shows the description on arguments. For geometric operator, many of the proposed geometric operators are overloaded, meaning that the same name is used for different computations. They includes inside, overlap, adjacent, length, mindist, maxdist, and so on.

6.3.3 Examples of Queries

For the comparison, let’s define following relations whose italic attributes form the key of the corresponding relation.

```
CITIES (Name:STR, Surface:S_REGION)
HIGHWAYS (Name:STR, Route:S_LINE)
ROADS (City:STR, Streets(Name:STR, Route:S_LINE))
RIVERS (Name:STR, Route:LINE)
OIL (Oid:REAL, Potential: REAL, Surface: REGION)
COAL (Cid:REAL, Potential: REAL, Surface: REGION)
```
Table 6.4: Description on arguments in QL/G syntax

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>new name</td>
<td>New attribute names.</td>
</tr>
<tr>
<td>qualification</td>
<td>The <em>qualification</em> allows nesting of SELECT-FROM-WHERE and is augmented with operators and geometric operators.</td>
</tr>
<tr>
<td>$expr_i$</td>
<td>Each $expr_i$ in the FROM clause is evaluated and returns a set of values.</td>
</tr>
<tr>
<td>$x_j$</td>
<td>Each variable $x_j$ is ranging over a value in the corresponding set. The <em>qualification</em> is then evaluated. If it returns TRUE, expressions in the SELECT clause are evaluated as the values of the corresponding columns.</td>
</tr>
<tr>
<td>$result_i(x_1, \cdots, x_n)$</td>
<td>It is a function returning a value on the current assignment of variables $x_1, \cdots, x_n$.</td>
</tr>
</tbody>
</table>

1. Find all neighboring cities of Toronto.

QL/G:

```
SELECT tuple(neighbor: Other.name)
FROM Toronto IN CITIES, Other IN CITIES
WHERE Toronto.Name='Toronto' AND Toronto.Surface ADJACENT Other.Surface;
```

ParaSQL:

```
SELECT C.Name
RESTRICTED TO [[C.Name = 'Toronto']]
FROM CITIES C
WHERE C.Name != 'Toronto'
```

As we have seen in the previous SQL/OGIS section, ParaSQL query uses just one relation, but QL/G uses two relations of CITIES. The query retrieves tuples from CITIES relation and checks if the name of city is ‘Toronto.’ If it is, then it rejects the tuple; otherwise, it restricts the domain to the spatial element such that the name of city is ‘Toronto.’

2. Consider a geological exploration application. Based on two preliminary independent geological surveys on oil and coal, a province is partitioned into a large number of regions according to their potentials. The information is recorded in the relations OIL and COAL, respectively. The higher the number of potential, the greater the chance we find oil or coal in a region. Notice that, however, we need to do the actual drilling to verify the existence of either kind of reserves.

Find the regions that the high in potential (>80) for both oil and coal.

QL/G:

```
SELECT C.Name
RESTRICTED TO [[C.Name = ‘Toronto’]]
FROM CITIES C
WHERE C.Name != ‘Toronto’
```
DISTINCT (UNNEST(
    SELECT tuple(high_pot: regions(intersection(c.Surface, o.Surface)))
    FROM c IN COAL, o IN OIL
    WHERE (c.Potential > 80) AND (o.Potential > 80)
    AND (c.Surface OVERLAP o.Surface))
ON high\_pot);

ParaSQL:

[[ SELECT C.Name
    RESTRICTED TO [[C.Potentail > 80]]*[[O.Potentail > 80]]
    FROM COAL C, OIL O
]]

The above ParaSQL query retrieves the tuples from relations-COAL and OIL. It restricts spatial elements to the domain expression of COAL relation such that both potentials from COAL and OIL are greater than 80. If there exists a region, then the potential of COAL and OIL is over 80. By taking the domain expression to the relational expression, above query returns the region satisfying the restriction.

3. What are the area of the city Hamilton?³

QL/G:

SELECT tuple(size: area(H.Surface))
FROM H IN CITIES
WHERE H.Name = 'Hamilton';

ParaSQL:

[[ SELECT C.Name
    FROM CITIES C
    WHERE C.Name = 'Hamilton'
]]

The above ParaSQL query uses a domain expression including a relational expression in its inside. The relational expression returns Name attribute of CITIES relation such that the name of a city is ‘Hamilton.’ Due to the domain expression outside of the relational expression, the final return value will be the region of the city because the domain expression returns a spatial element of an attribute.

6.4 SQL/SDA

6.4.1 SQL/SDA Data Model

SQL/SDA (Spatial Data Analysis) has been designed to satisfy the requirement that GIS development is to provide easy and effective access to spatial analysis functionalities for supporting

³The original query was to find the area as well as the length of the boundary of the city. To illustrate the difference between two query languages, the query was modified to find only the area of the city.
decision making based on geo-referenced data within the framework of SQL standard for spatial
extensions. Within such a framework, the objective of SQL/SDA is to support the expression of
complicated spatial queries dealing with various spatial analysis problems. SQL/SDA adopts SQL
with geometry types and a geometry-valued column is implemented as a column whose data type
is drawn from the set of geometry types which are defined by OGIS [48].

In SQL/SDA data model, we can define LANDUSE, SOIL, BUILDING, SEWER, STREAM,
and SHOP relations by using geometry datatypes. Creating tables are exactly following the stan-
dard SQL except it uses geometry datatypes. In the tables, attribute Location has one of geometry
types defined in SQL/SDA.

<table>
<thead>
<tr>
<th>Relation</th>
<th>Columns</th>
</tr>
</thead>
<tbody>
<tr>
<td>LANDUSE</td>
<td>ID, Type, Location</td>
</tr>
<tr>
<td>SOIL</td>
<td>ID, Type, Location</td>
</tr>
<tr>
<td>BUILDING</td>
<td>ID, Name, Owner, Usage, Location</td>
</tr>
<tr>
<td>SEWER</td>
<td>ID, Type, Capacity, Location</td>
</tr>
<tr>
<td>STREAM</td>
<td>ID, Name, Location</td>
</tr>
<tr>
<td>SHOP</td>
<td>ID, Name, Location</td>
</tr>
</tbody>
</table>

In the above feature table, the LANDUSE, SOIL and BUILDING features are of polygon type,
the SEWER and STREAM of linestring type, and the SHOP of point type. Figure 6.3 shows maps
and their corresponding relational tables.

![Figure 6.3: LANDUSE and SOIL relations in the SQL/SDA data model [48]](image)

6.4.2 Syntax of SQL/SDA

The syntax of SQL/SDA is based on the conventional SQL and only the FROM clause is different
from the standard SQL. The basic syntax of SQL/SDA and the BNF form for FROM clause are
defined as follows:

```
SELECT <select-clause>
FROM <from-clause>
WHERE <where-clause>
```
The spatially derived attributes are treated in the same way as those in the source relations and, thus, can be applied as constraints in the main WHERE clause, and/or referenced for further analysis, aggregation, or graphical display in the main SELECT clause. Since SQL/SDA needs to comply with both the general spatial analysis procedure and the SQL concepts, the subquery in the FROM clause is employed [48].

6.4.3 Examples of Queries

For LANDUSE and SOIL relations in the parametric data model, we use notations \textit{lreg} and \textit{sreg} for regions in LANDUSE and SOIL relations, respectively. Figure 6.4 shows the tables in the parametric data model.

<table>
<thead>
<tr>
<th>LANDUSE</th>
<th></th>
<th>SOIL</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>Type</td>
<td>ID</td>
<td>Type</td>
</tr>
<tr>
<td>[lreg]1</td>
<td>Brushland</td>
<td>[sreg]1</td>
<td>A</td>
</tr>
<tr>
<td>[lreg]2</td>
<td>Water</td>
<td>[sreg]2</td>
<td>B</td>
</tr>
<tr>
<td>[lreg]3</td>
<td>Forest</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 6.4: LANDUSE and SOIL relations in the parametric data model

1. Find the land parcels and their corresponding area on the condition that the landuse type of each parcel is brushland and soil type is ‘A’ and area is between 700 hectares to 900 hectares.

SQL/SDA:

```sql
SELECT lu.ID, sl.ID, ILocation, areaval
FROM (SELECT *, OVERLAP(lu.Location, sl.Location) AS overlapval,
       INTERSECTION(lu.Location, sl.Location) AS lLocation,
       AREA(lLocation) AS areaval
FROM LANDUSE AS lu, SOIL AS sl)
```
WHERE lu.type='Brushland' AND sl.type='A'
  AND overlapval=TRUE AND areaval > 700
  AND areaval < 900

ParaSQL:
[[ SELECT *
    RESTRICTED TO [[L.Type='Brushland']]*[[S.Type='A']]]
FROM LANDUSE L, SOIL S
WHERE areaval([[L.Type='Brushland']]
  *[[S.Type='A']]) > 70
  AND areaval([[L.Type='Brushland']]
    *[[S.Type='A']]) < 90
]]

The above ParaSQL query seeks tuples satisfying that the size of the intersection of two areas-'Brushland' in LANDUSE relation and type 'A' in SOIL relation is between 700 and 900, exclusively. If there exists a tuple satisfying the condition, then it restricts the spatial element to the intersection of two areas such that the land type is 'Brushland' and the soil type is 'A.' Therefore, the tuples after executing the relational expression satisfies the condition. By taking domain expression outside of the relational expression, the ParaSQL query returns the region such that the land type is 'Brushland' and the type of soil is 'A.' We can rewrite the above query by using SETFUNC clause, it will be discussed later.

2. This query is to select a lab site. The selection criteria are:
   (a) Preferred landuse is brushland.
   (b) Soil type should be 'A.'
   (c) Site must be within 300 meters of existing sewer lines.
   (d) Site must be beyond 20 meters of existing streams.
   (e) Site must contain an area at least 2,000 square meters.

SQL/SDA:
SELECT labLocation
FROM (SELECT *, INTERSECT(lu.Location, s1.Location)
    BUFFER(sw.Location, 300)
    AS buf1Location,
    INTERSECTION(lsLocation, bufLocation)
    AS lsbLocation,
    BUFFER(sm.Location, 20)
    AS buf2Location,
    DIFFERENCE(lsbLocation, buf2Location)
    AS labLocation,
    AREA(labLocation) AS areaval
FROM LANDUSE AS lu, SOIL AS s1,
    sewer AS sw, stream AS sm)
WHERE lu.Type = 'Brushland' AND s1.Type = 'A'
  AND areaval > 2000
ParaSQL:

[[ SELECT *
  RESTRICTED TO [[L.Type='Brushland']]*[[S.Type='A']]
FROM  LANDUSE L, SOIL S, SEWER SE, STREAM ST
WHERE  distance([[L.Type='Brushland']]*[[S.Type='A']],
  [[SE.ID]]) < 300
  AND  extend([[L.Type='Brushland']]*[[S.Type='A']],
          20)*[[ST.ID]] != empty)
  AND  areaval([[L.Type='Brushland']]*[[S.Type='A']])
  > 2000]]

In order to prevent the query from using same expression repeatedly, let’s define SETFUNC clause in ParaSQL as follows:

```
SETFUNC < function name > AS <function definition>
```

If we use the definition of SETFUNC, then we can define a function- interDom as follows:

```
SETFUNC = interDom AS
[[[LANDUSE.Type = 'Brushland']] * [[[SOIL.Type='A']]]
```

Therefore, the above ParaSQL query can be rewritten more neatly as follows:

```
SETFUNC = interDom AS
[[[LANDUSE.Type='Brushland']]*[[SOIL.Type='A']]]
[[ SELECT *
  RESTRICTED TO interDom
FROM  LANDUSE L, SOIL S, SEWER SE, STREAM ST
WHERE  distance(interDom, [[SE.ID]]) < 300
  AND  extend(interDom, 20) * [[ST.ID]] != empty)
  AND  areaval(interDom) > 2000]]
```

3. The last query concerns site selection. This one is related to land suitability evaluation for building an institute, in which all the possible sites need to be classified into different suitability levels. Assuming that there are two maps: LANDUSE and SOIL, and suitability levels are “high (III),” “medium (II),” and “low (I).” The evaluation includes the following steps:

(a) Overlay the landuse map and soil map.

(b) Classify the overlay map in terms of the evaluation criteria: i) If the landuse type is “Brushland” and soil type is ‘A,’ then the suitability level is ‘III.’ ii) If the landuse type is “Water” and soil type is ‘A,’ then the suitability level is ‘I.’ iii) Otherwise the suitability level is ‘II.’

(c) Merge the parcels (area > 100 hectares) with the same suitability level and display them.

This query is formulated in SQL/SDA as follows:
SQL/SDA:

```sql
SELECT FUSION(ILocation)
FROM (SELECT *
      INTERSECTION (lu.Location,
                    s1.Location) AS ILocation,
      AREA(ILocation) AS areaval,
      classfyval =
      (CASE lu.Type || s1.Type
          WHEN 'Brushland' AND 'A' THEN 'III'
          WHEN 'Brushland' AND 'B' THEN 'II'
          WHEN 'Water' AND 'A' THEN 'I'
          WHEN 'Water' AND 'B' THEN 'II'
          WHEN 'Forest' AND 'A' THEN 'II'
          WHEN 'Forest' AND 'B' THEN 'II'
          END)
      FROM LANDUSE AS lu, SOIL AS s1)
WHERE ILocation <> NULL AND areaval > 100
GROUP BY classfyval
```

In this query, the INTERSECTION operation is employed to obtain the basic parcels that have the attributes of both landuse and soil type. After the classification of each overlaid parcel, the parcels with the same “classfyval” are merged. This is performed by the FUSION function in the SELECT clause coupled with the GROUP BY clause [48].

ParaSQL:

```sql
SELECT *,
      level =
      (CASE
          WHEN L.Type='Brushland' AND S.Type='A' THEN 'III'
          WHEN L.Type='Water' AND S.Type='A' THEN 'I'
          ELSE 'II'
          END)
      RESTRICTED TO [[L.ID]]*[[S.ID]]
FROM LANDUSE L, SOIL S
WHERE areaval([[L.ID]]*[[S.ID]]) > 100
```

In the above ParaSQL query, it does not use a nested relational expression. It first retrieves tuples such that the size of the intersection of two areas from LANDUSE and SOIL relation, is greater than 100 hectares. Then it restricts the spatial element to the intersection, that is, the spatial stamp will be only the intersection of two regions. After restricting the tuples, it classifies the regions based on the evaluation criteria. Figure 6.3 shows the result of this query using the LANDUSE and SOIL maps. The central small polygon is not labeled with a suitability level because its area is less than 100 hectares.

### 6.5 PSQL

#### 6.5.1 PSQL Data Model

PSQL (Pictorial SQL) extends the definition of relations over spatial and other types of domains. Every domain in PSQL is an abstract data type. PSQL supports three basic pictorial domains:
points, line segments, and regions. In addition to these three basic pictorial domains, PSQL also supports the standard alphanumeric domains, integers, reals, and strings.

Relations are defined over alphanumeric and/or pictorial domains. They model inter-domain relationships. Every tuple models a relationship among those alphanumeric and pictorial elements of the domains [69]. The following relation is defined over a set of pictorial domains (point, segment, region).

COUNTY (CName, Crop, CRegion)

The pictorial domain of CRegion is of type “region.” Figure 6.6 shows COUNTY relation in PSQL data model.

<table>
<thead>
<tr>
<th>COUNTY</th>
</tr>
</thead>
<tbody>
<tr>
<td>CName</td>
</tr>
<tr>
<td>story</td>
</tr>
<tr>
<td>story</td>
</tr>
<tr>
<td>orange</td>
</tr>
<tr>
<td>orange</td>
</tr>
<tr>
<td>orange</td>
</tr>
<tr>
<td>polk</td>
</tr>
</tbody>
</table>

Figure 6.6: COUNTY relation in PSQL data model

6.5.2 Syntax of PSQL

PSQL is based on the SQL and the skeleton of the PSQL is as follows:

```
SELECT <attribute-target-list>
FROM   <relation-list>
[ON    <picture-list>]
[WHERE <qualification>]
```
The `<attribute-target-list>` is the set of tuples resulting from the query. The `<relation-list>` defines the source relations that will be queried. The optional ON `<picture-list>` in the mapping is a name list that specifies the picture the query is on. This is used when tuples of the same relation corresponding to different pictures. The ON clause can be omitted provided that the domains of the relations are associated with a unique picture. The search area on the picture is specified in the `<qualification>` clause. It can either be a bound variable or a location given in absolute constant coordinates or in variable coordinates. The location variable may just be the name of a location predefined outside the retrieve mapping. Furthermore, a search area in `<qualification>` may be followed by a pictorial operator “cover,” “overlap,” etc., followed by another area specification [69].

### 6.5.3 Examples of Queries

In order to compare with ParaSQL, let’s define COUNTY relation in a parametric database as COUNTY(CName, Crop) and CName → Crop. Therefore, CName is a primary key of COUNTY relation. Figure 6.7 shows the COUNTY table in the parametric data model.

<table>
<thead>
<tr>
<th>COUNTY</th>
</tr>
</thead>
<tbody>
<tr>
<td>CName</td>
</tr>
<tr>
<td>[creg₁] ∪ [creg₂]</td>
</tr>
<tr>
<td>[creg₁] ∪ [creg₂]</td>
</tr>
<tr>
<td>[creg₃] ∪ [creg₄]</td>
</tr>
<tr>
<td>[creg₃] ∪ [creg₄]</td>
</tr>
<tr>
<td>[creg₅] ∪ [creg₆]</td>
</tr>
<tr>
<td>[creg₆]</td>
</tr>
<tr>
<td>[creg₆]</td>
</tr>
</tbody>
</table>

Figure 6.7: COUNTY relation in the parametric data model [33]

1. Retrieve complete information about counties which grow wheat or corn.

PSQL:

```sql
SELECT C1.*
FROM COUNTY C1 C2
WHERE (C1.Crop = 'wheat' OR C1.Crop = 'corn')
AND C1.CName = C2.CName
```

ParaSQL:

```sql
SELECT *
FROM COUNTY C
WHERE [[C.Crop = 'wheat']] != EMPTY OR
      [[C.Crop = 'corn']] != EMPTY
```

To express this query in PSQL we need two variables C1 and C2. The variable C1 is needed to make sure that the crop is wheat, and C2 is needed to make sure that the county is same as that in C1. Note that the FROM clause “FROM COUNTY C1 C2” in the following expression is meant to create two aliases of the COUNTY relation [33].
2. Retrieve complete information about counties which grow wheat and corn.

PSQL:

```
SELECT C1.*
FROM COUNTY C1 C2 C3
WHERE (C1.Crop = 'wheat' AND C1.Crop = 'corn')
    AND C1.CName = C2.CName AND C1.CName = C3.CName
```

ParaSQL:

```
SELECT *
FROM COUNTY C
WHERE [[C.Crop = 'wheat']] != EMPTY AND
    [[C.Crop = 'corn']] != EMPTY
```

In the query, we just change or to and in the English query of the previous query. This is expressed in ParaSQL simply by replacing OR in WHERE clause in query (1) with AND, but the corresponding transformation does not work in PSQL. To express the new query in PSQL, we need three independent variables as shown in the query statement. Note that for every occurrence of and in an English query of the form given above, we need an additional variable in the PSQL query. Thus if there are \( n \) properties to be checked for a given county, we need \( n + 1 \) variables in PSQL leading to an \((n+1)\)-way join; in ParaSQL simply one variable suffices [33].

3. Retrieve information about counties that grow wheat.

PSQL:

```
SELECT C1.*
FROM COUNTY C1 C2
WHERE C1.Crop = 'wheat'
    AND C1.CName = C2.CName
```

ParaSQL:

```
SELECT *
FROM COUNTY C
WHERE [[C.Crop = 'wheat']] != EMPTY
```

4. Retrieve information about counties that do not grow wheat.

PSQL:

```
( SELECT *
    FROM COUNTY
)
DIFFERENCE
( SELECT C1.*
    FROM COUNTY C1 C2
    WHERE C1.Crop = 'wheat'
        AND C1.CName = C2.CName
)
```
ParaSQL:

```
SELECT *
FROM COUNTY C
WHERE [[C.Crop = 'wheat']] = EMPTY
```

In this query, we just add a not to the previous English query. In ParaSQL, it is simply expressed by adding a negation, that is,

$$\sim ([\text{C.Crop} = \text{'wheat'}]) \neq \text{EMPTY} \iff [\text{C.Crop} = \text{'wheat'}] = \text{EMPTY}$$

But, it is more complex in PSQL.

### 6.6 Summary

In this chapter, we discussed spatial query languages and compared with ParaSQL. Spatial query languages can be categorized into two classes—SQL style spatial query languages and visual spatial query languages.\(^4\) It has been argued whether the relational database query language SQL can be successfully extended for spatial applications. However, since SQL is still a popular database language and its functionalities have been enhanced considerably, it is considered as the most preferred option for this study.

For the comparison with ParaSQL, we have looked at SQL style query languages such as SQL/OGIS, a spatial query language for supporting OGIS standard, QL/G, SQL/SDA, and PSQL. As we have seen in the examples excerpted from literature describing the query languages, we could conclude that ParaSQL provides more easy and natural ways to express the queries. In some queries, we have seen that it made quite complicated to transform English queries into the other query languages when changing “and” to “or,” or adding “not” to the English queries. But ParaSQL did not introduce any additional complexity, but it could express the queries in a natural way.

---

\(^4\)In [2], the authors classify the query languages into three kinds—textual approaches, non-textual approach, and hypermedia approaches.
Chapter 7

Spatiotemporal Query Languages

7.1 Introduction

Many data objects in the real world have attributes about location and time. For example, a database about sea turtles records the location and time data for turtles that carry radio transmitters. Other examples include tracking vehicles with global positioning systems (GPS), mobile phone users within mobile phone networks, and environmental changes over time. Traditional relational database technology is not suitable for managing spatiotemporal data, which are multi-dimensional with complex structures and behaviors [81].

Spatiotemporal databases have been the focus of considerable research activity over a significant periods. However, there still exist very few prototypes of complete systems, and far less products that provide effective support for applications tracking changes to spatial and aspatial data\textsuperscript{1} over time. This is because the design and implementation of a complete spatiotemporal database is a challenging undertaking, involving extensions to all aspects of a non-spatiotemporal architecture-data model, query language, query optimizer, query evaluator, programming environment, storage manager, indexes, etc [41].

In the past, research in spatial and temporal data models and database systems has mostly been done independently. Spatial database research has focused on supporting the modeling and querying of geometries associated with objects in a database. Temporal databases have focused on extending the knowledge kept in a database about the current state of the real world to include the past in the two senses of “the past of the real world” (valid time) and “the past state of the database” (transaction time). Nevertheless, many people have felt that the two areas are closely related, since both deal with “dimensions” or “spaces” of some kind, and that an integration field of “spatiotemporal databases” should be studied and would have important applications [29].

Authors in [67] suggest two directions to accommodate spatial and temporal databases: 1) the embedding of a temporal awareness in spatial systems, and 2) the accommodation of space into temporal data mining systems. And they points out that the former approach has been the more popular because of the relative maturity of geographic information systems and the availability of time-stamped snapshots of geographic/spatial test data.

Erwig in [28] defines spatiotemporal data as a particular example of temporal data in general. For example, we can deal with temporally changing numbers through a type like \(\text{time} \rightarrow \text{num}\). Such a temporal number could give the size of the oil spill depending on the time. In [40], the authors define the spatiotemporal databases as set of moving \(n\)-dimensional figures described by means of an set of tuples \((x_1, x_2, \cdots, x_n; t)\) in \(R^n \times R\), where \(R\) is the set of real numbers, \((x_1, x_2, \cdots, x_n)\)

\textsuperscript{1}Time and spatial independent data, that is, ordinary data over classical databases.
represent the spatial coordinates of a point in the $n$-dimensional real space $\mathbb{R}^n$ and $t$ is the time coordinate in $\mathbb{R}$. In [81], the authors define spatiotemporal databases as a database that embodies spatial, temporal, and spatiotemporal database concepts, and captures simultaneously spatial and temporal aspects of data.

Spatiotemporal applications fall into the category of data intensive applications, often referred to as “non-standard”, including, among others, multimedia, VLSI design, and artificial intelligence based systems [64]. Spatiotemporal data analysis plays an important role in many scientific applications like environmental epidemiology and public health. Data analysis of multidimensional data like spatial, temporal and statistical data occurs in many scientific applications and has to be supported by modern database technology. An adequate data model, comfortable querying and special implementation techniques have to be considered [51].

In spite of the fact that there are many different definitions and approaches to spatiotemporal models and query languages in the spatiotemporal literature, the definitions have the common features in that spatiotemporal databases should deal with time and space dimensions and/or the combined dimension. Therefore, query languages for spatiotemporal databases should provide powerful expressive mechanism to express queries asking spatial, temporal, or spatiotemporal object in effective and in natural.

In the following sections, we will discuss four different spatiotemporal query languages–$\text{SQL}^{ST}$, E.S. STQL, K.R.P STQL and STSQL. We also compare them with ParaSQL. The all queries have been excerpted from the literature describing each spatiotemporal query language.

### 7.2 SQL$^{ST}$

#### 7.2.1 SQL$^{ST}$ Data Model

The objective of SQL$^{ST}$ is to minimize the extensions required in SQL to support spatiotemporal queries. SQL$^{ST}$ is based on a directed-triangulation model to represent spatial data, and a point-based model to represent time at the conceptual level. According to [16], the authors defined SQL$^T$ and SQL$^S$ components based on Worboy’s suggestion [83]. Therefore, SQL$^{ST}$ is the combined query language with SQL$^T$ and SQL$^S$. To model time at the conceptual level, a point-based time model is used, where information is repeated for each time granule where it is valid. In the spatial model, SQL$^{ST}$ use triangles to represent polygons; a similar approach was proposed in [39, 47].

The authors explain two reasons for the polygon-oriented representation. The first is that coalescing is needed much less frequently than in temporal queries. The second is that two dimensional shapes offer a more natural representation for many application domains. Therefore, SQL$^{ST}$ views reality as a sequence of snapshots of objects that are moving and/or changing in shape [14, 16].

Figure 7.1 shows an example of spatial objects changing with time. At time $t = 0$, there are two spatial objects in the graph—a square $O_1$ and a triangle $O_2$. At time $t = 10$, $O_1$ changes its shape and $O_2$ moves to a new position. At time $t = 20$, $O_1$ has some more changes in shape while $O_2$ stays unchanged [16].

An internal representation of Figure 7.1 could be shown as follows:
Here, the regions are represented by a circular list of vertexes, and the time elements are stored as intervals. Figure 7.2 shows how the changes are recorded in the database at the conceptual level.

![Figure 7.1: Graphs representing spatiotemporal data [16]](image)

\[
\begin{align*}
(O_1 & \quad [(2,6),(2,2),(6,2),(6,6)], [0,10)) \\
(O_2 & \quad [(6,6),(6,2),(10,4)], [0,10)) \\
(O_1 & \quad [(2,6),(2,2),(4,4),(6,6)], [10,20)) \\
(O_2 & \quad [(4,4),(4,0),(8,2)], [10,20)) \\
(O_1 & \quad [(2,6),(2,2),(6,6)], [20,30)) \\
(O_2 & \quad [(4,4),(4,0),(8,2)], [20,30))
\end{align*}
\]

7.2.2 Syntax of SQL\textsuperscript{ST}

There are no concrete syntax for SQL\textsuperscript{ST} in [14, 16] at the time of writing. But, we can derive the syntax as shown below because SQL\textsuperscript{ST} is the minimal extension from the standard SQL. It provides interval operators suggested by Allen [1] such as overlap, precede, contain, equal,
meet and intersect. It also support commonly used spatial predicates such as equal, disjoint, overlap, meet, contain, adjacent, and common border, etc.

\[
\text{SELECT <attribute list>}
\text{FROM <relation list>}
\text{[WHERE <where condition>]} \\
\text{[<group by clause>]} \\
\text{[<having clause>]} 
\]

### 7.2.3 Examples of Queries

In this section, we will look at SQL\textsuperscript{ST} queries transforming some examples introduced in [42]. For the comparison with ParaSQL, we consider three relations–FOREST, FOREST\_FIRE, and FIRE\_FIGHTER as shown below:

- FOREST (Forestname CHAR(30), Territory REGION, VTime DAY)
- FOREST\_FIRE (Firename CHAR(30), Extent REGION, VTime DAY)
- FIRE\_FIGHTER (Fightername CHAR(30), Location POINT, VTime DAY)

FOREST relation has records of the location and the development of forests changing over time. FOREST\_FIRE relation has records of the evolution of forest fires. FIRE\_FIGHTER relation has records of the motion of fighters.

When translating a database consisting of those relations into a parametric database, we have to note that each attribute in a relation consists of spatiotemporal dimension and its corresponding value. The primary keys for the relations will be Forestname, Firename, and Fightername, respectively. The columns Territory and Extent have a spatial data type as REGION and Location has a type as POINT; temporal data column VTime has a granularity of DAY in SQL\textsuperscript{ST} data model.

1. When and where did the fire called “The Big Fire” reach what largest extent?

**SQL\textsuperscript{ST}**:

\[
\text{SELECT F1.VTime, F2.extent, AREA(F1.extent)} \\
\text{FROM FOREST\_FIRE as F1 F2} \\
\text{WHERE F1.Firename = 'The Big Fire'} \\
\text{AND F2.Firename = 'The Big Fire'} \\
\text{AND F1.VTime = F2.VTime} \\
\text{GROUP BY F1.VTime} \\
\text{HAVING AREA(F1.Extent) = (SELECT MAX(AREA(extent))} \\
\text{FROM FOREST\_FIRE} \\
\text{WHERE Firename = 'The Big Fire')} 
\]

**ParaSQL**:

\[
\text{SELECT *} \\
\text{RESTRICTED TO [[PROJ SPACE} \\
\text{[[F.Name='The Big Fire']]} \\
\text{WHERE maxarea(SPACE)]]} \\
\text{FROM FOREST\_FIRE F} \\
\text{WHERE F.Firename = 'The Big Fire'} 
\]

84
This query is very interesting and makes ParaSQL develop a new sublanguage for internal navigation. The concept of the parametric data model is to store an object in a single tuple, not separately. Therefore, it can retrieve tuples without introducing another variables for combining tuples into one object compared to other spatiotemporal query languages. But it has a limitation to compare parametric elements each other in a same tuple. Comparing parametric elements which exist in a same tuple is one type of restriction of domain of the tuple.

Therefore, we can decide the position of the sublanguage as the inside of RESTRICTED TO clause. Since we handles only spatiotemporal elements, there exist two domains-spatial and temporal domains. Based on this observation, we can define *dimension projection operator* $\Psi$ as follows:

$$\Psi^t([[]]) = \{ x | x \text{ is a dimension and } c(x) \text{ is true} \}$$

where, $t$ is a target dimension and $c$ is a condition. To apply the definition, let’s consider FOREST\_FIRE relation. Figure 7.3 shows extents of ‘The Big Fire’ associated with temporal element $t_i$ and Figure 7.4 shows the relation in the parametric data model. Note that $t_i$ is a temporal element, not a time instance.

![Figure 7.3: Fire extents of ‘The Big Fire’](image)

![Figure 7.4: FOREST\_FIRE relation in the parametric data model for Figure 7.3](image)

Suppose that a user poses a query asking the largest extent of ‘The Big Fire.’ Then we can retrieve the largest extent of the fire by using the dimension projection as follows:

$$\Psi_{\text{maxarea}}([[[\text{Firename} = \text{‘The Big Fire’}]]) = \{ s_4 \}$$
The execution of the dimension projection consists of three steps: 1) extract a parametric
domain, 2) project specified domain, and 3) apply conditions.

By the first step of the dimension projection, a spatiotemporal element is extracted as follows:
\[
[[\text{Firename = 'The Big Fire']]} = s_1 \times t_1 \cup s_2 \times t_2 \cup s_3 \times t_3 \cup s_4 \times t_4
\]

By the second step, only spatial elements are projected from the spatiotemporal element as follows:
\[
\Psi^S([[\text{Firename = 'The Big Fire']}}) = \{s_1, s_2, s_3, s_4\}
\]

By the last step, only qualified spatial elements are returned as follows:
\[
\Psi_{\text{maxarea}}^S([[\text{Firename = 'The Big Fire']}}) = \{s_4\}
\]

The ParaSQL query uses the sublanguage defined for navigating the internal parametric
elements of tuples. The definition and the grammar of the language are not formally defined
yet and they should be studied. But the definition of the dimension projection that we have
discussed is sufficient to solve the problem of this example. Let’s define the format of the
sublanguage tentatively as follows:

\[
\begin{align*}
\text{PROJ} & \quad \{\text{space, time}\} \\
\text{FROM} & \quad <\text{domain expression}> \\
\text{WHERE} & \quad <\text{condition}>
\end{align*}
\]

The ParaSQL query first retrieves tuples such that the name of a fire is ‘The Big Fire’ and
find the largest extent from the domain of the fire. In this case, the largest region will be
returned. Since the return value is wrapped up with domain expression in RESTRICTED
TO clause, the underlying system of parametric database aligns the dimension with \([s_4]\) × \(T\),
where \(T\) is the universal set of time. By the restriction, the domain of tuple will be intersected
with the spatiotemporal element of the tuple. Therefore, we can get the time and the largest
extent of the fire.

2. When and where was the spread of fires larger than 500\(km^2\)?

**SQLST:**

```
SELECT F1.VTime, F2.Extent
FROM FOREST_FIRE as F1 F2
WHERE F1.VTime = F2.VTime
AND F1.Firename = F2.Firename
GROUP BY F1.VTime, F2.Extent, F1.Firename
HAVING AREA(F1.Extent) > 500
```

**ParaSQL:**

```
SELECT *
FROM FOREST_FIRE F
WHERE area([[F.Extent]]) > 500
```
The above ParaSQL query retrieves tuples from FOREST\_FIRE relation, and checks if the extent area of a fire is greater than 500\(\text{km}^2\). If it is, then it returns the tuples; otherwise, it rejects them. As we can see, the ParaSQL query uses only one variable for FOREST\_FIRE relation, but SQL\textsuperscript{ST} uses two variables for the relation. It is because the object is stored in separate tuples in SQL\textsuperscript{ST} data model. Since the ParaSQL query uses only one variable, we can expect that the ParaSQL can provide more efficient way to execute the query far faster than that of SQL\textsuperscript{ST} because it can avoid join processing. One thing that we have to note here is that SQL\textsuperscript{ST} uses GROUP BY clause to group the tuples based on VTime, Extent, and Firename. It is because of the nature of the base model of SQL\textsuperscript{ST}. Since it uses a point-based model for time, it should group the tuples for an object stored in different tuples to query on the object.

3. Determine the times and locations when “The Big Fire” started.

SQL\textsuperscript{ST}:

```
SELECT VTime, Extent
FROM FOREST\_FIRE
WHERE Firename = 'The Big Fire'
AND VTime = (SELECT MIN(VTime)
 FROM FOREST\_FIRE
 WHERE Firename = 'The Big Fire')
```

ParaSQL:

```
SELECT F.Firename
FROM FOREST\_FIRE F
WHERE F.Firename = 'The Big Fire'
```

The query expressed by SQL\textsuperscript{ST} can be written in a ParaSQL query as above. As we can see, the ParaSQL query does not introduce any function that calculates the time event that the fire occurred. It is because of the nature of the parametric data model. Since the parametric data model maintains parametric elements with attributes, there are no needs to introduce the function to find the starting time of the event. User can easily determine the time by browsing the results from the query. If we use function \texttt{starttime} to do that, the above query will be rewritten as follows:

ParaSQL:

```
SELECT starttime([F.Firename])
FROM FOREST\_FIRE F
WHERE F.Firename = 'The Big Fire'
```
7.3 E.S STQL

7.3.1 E.S STQL Data Model

E.S STQL stands for Erwig and Shneider’s Spatio-Temporal Query Language. E.S STQL models spatiotemporal data as abstract data types which can be employed as attribute types in a relation. The objective of E.S STQL data model is to be compatible with smoothly changing spatiotemporal objects. The temporal version of a value of type $\alpha$ that changes over time is modeled as a temporal function of type $\tau(\alpha) = \text{time} \rightarrow \alpha$, where $\alpha$ is a spatial data type such as point and region. Temporal functions are basis of an algebraic data model for spatiotemporal data types. Therefore, $\tau(\text{point})$ represents a point changing its location over time. Similarly, an element of type $\tau(\text{region})$ is a region that can move and/or grow/shrink. In addition, E.S STQL data model also has changing numbers and booleans when defining operations on temporal objects. For example, if we want to compute the time dependent distance of an airplane and a storm. This could be achieved by an operator:

$$\text{Distance} : \tau(\text{point}) \times \tau(\text{region}) \rightarrow \tau(\text{real})$$

The data model of E.S STQL has a special operator lift to make non-temporal operation work on temporal objects and return a temporal object as a result. For non-temporal function $f : \alpha_1 \times \cdots \times \alpha_n \rightarrow \beta$, its corresponding lifted versions defined as follows:

$$\uparrow f : \tau(\alpha_1) \times \cdots \times \tau(\alpha_n) \rightarrow \tau(\beta)$$

with

$$\uparrow f(S_1, \cdots, S_n) := \{(t, f(S_1(t), \cdots, S_n(t))) | t \in \text{time}\}$$

For example, consider the spatial predicate inside : point $\times$ region $\rightarrow$ bool. The lifted version of this predicate has the type

$$\uparrow \text{inside} : \text{Point} \times \text{Region} \rightarrow \text{Bool}$$

with the meaning that it yields true for each time at which the point is inside the region, undefined whenever the point or the region is undefined, and false in all other cases [30].

7.3.2 Syntax of E.S STQL

E.S STQL extends the widespread database query language SQL. There is no formal syntax definition on E.S STQL in literature. The following syntax is derived based on the examples in [30] and the data model of E.S STQL. Since it adapts the standard SQL, the main skeleton of the syntax will be SELECT-FROM-WHERE clause and SELECT clause may have attributes forming temporal functions of type.

```sql
SELECT <attribute list> | <temporal function of type>
FROM   <relation list>
[WHERE <where condition>]
[group by clause]
[having clause]
```

---

2 Martin Erwig and Markus Shneider developed a spatiotemporal query language, and they named it STQL. During the literature survey, I found that there existed another spatiotemporal query language with the same name. In this paper, E.S STQL is the spatiotemporal query language developed by Erwig and Shneider.

3 According to [30], the authors denote non-temporal types, entities, functions, and predicates by lower case letters while their temporal counterparts start with capital letters to make notations more comprehensible.
7.3.3 Examples of Queries

For comparisons, let’s define FLIGHT and WEATHER relations as follows:

\[
\text{FLIGHTS(ID: string, Route: Point)}
\]
\[
\text{WEATHER(Kind: string, Extent: Region)}
\]

The attribute ID identifies a flight, and Route records the route of a flight over time. The attribute Kind classifies different weather events like hurricanes, high pressure areas, or snowfall, and Extent yields the evolving extent of each weather event.

1. Where was United Airlines flight 207 at time 8:00?

E.S STQL:

\[
\text{SELECT Route(8:00)}
\]
\[
\text{FROM FLIGHTS}
\]
\[
\text{WHERE ID = ‘UA207’}
\]

ParaSQL:

\[
\text{SELECT F.ID}
\]
\[
\text{RESTRICTED TO [8, 8]}
\]
\[
\text{FROM FLIGHTS F}
\]
\[
\text{WHERE F.ID = ‘USA 207’}
\]

In E.S STQL, it retrieves Route attribute with a time argument. Based on the data model used in E.S STQL, the attribute value is the function of time as defined below:

\[
\text{Route : time} \rightarrow \text{Point}
\]

Therefore Route(8:00) returns the point of airline ‘UA 207’ at 8:00. In ParaSQL, it uses RESTRICTED TO clause to find the position when time was 8:00. In the RESTRICTED TO clause, the parametric domain is a spatiotemporal domain. Therefore the underlying system will take care of to align the temporal domain to a spatiotemporal domain.

2. When was a plane over the Eiffel Tower?

E.S STQL:

\[
\text{SELECT dom(Intersection(Route, ^EiffelTower))}
\]
\[
\text{FROM FLIGHTS}
\]

ParaSQL:

\[
\text{SELECT F.ID}
\]
\[
\text{RESTRICTED TO [EiffelTower]}
\]
\[
\text{FROM FLIGHTS F}
\]

In E.S STQL, the lifting operator is denoted by ^\text{. The Intersection operator is lifted and computes the time-dependent intersection of two moving points. The result is a moving point comprising all those (time, point)-pairs where the two original moving points met.}
In the above E.S STQL and ParaSQL, they use EiffelTower to describe a point containing the coordinates of the Eiffel Tower. Since there is no relation on Eiffel Tower, we have to assume that a user knows the region (here, point) of the tower. The ParaSQL query retrieves the parametric elements (the region of Eiffel Tower). If the result is non-empty after intersecting two domains, the flight was over the tower at a specific time. Here, we have to note the domain alignment. Two query languages have different approach to align domains. In E.S STQL, users have to handle the domain alignment by using the lift operator, but in ParaSQL, the work is left to a underlying system. Therefore, in E.S STQL users are required to determine which attributes are time independent before asking queries.

3. Determine the time when flight UA207 flew into a hurricane

E.S STQL:
```
SELECT MIN(dom(Intersection(Route, Extent)))
FROM FLIGHTS, WEATHER
WHERE ID = 'UA207' AND Kind = 'hurricane'
```

ParaSQL:
```
SELECT F.ID
RESTRICTED TO [[SELECT W.Kind
    FROM WEATHER W
    WHERE W.Kind = 'hurricane']]
FROM FLIGHTS F
WHERE F.ID = 'UA 207'
```

The above ParaSQL query retrieves tuples such that the name of a flight is ‘UA 207.’ After retrieving tuples, it restricts them to the domain such that the kind of a weather is ‘hurricane.’ Up to this point, the retrieved and restricted tuples satisfy two conditions such that the identification of a flight is ‘UA 207’ and it flew into a hurricane. Even though ParaSQL query uses a nested relational expression in RESTRICTED TO clause, it does not introduce WEATHER relation in FROM clause. Therefore, ParaSQL query can avoid the join processing.

7.4 K.R.P STQL

7.4.1 K.R.P STQL Data Model

The data model of K.R.P STQL supports a bitemporal concept for a spatial object such as point, line, and polygon objects. In this model, an object is represented using a hierarchical three-dimensional architecture that consists of two-dimensional space domain and linear valid time domain on the basis of another time domain, entitled as linear transaction time domain. Figure 7.5 shows the logical spatiotemporal database and describes a table that consists of primitive, spatial, valid time, and transaction time attributes. The spatial attribute has one or more values; the point type has a pair of spatial coordinates, i.e., \((x, y)\). The line type has two pairs of spatial coordinates

---

K.R.P STQL is the spatiotemporal query language developed by the authors in [52, 53]. They named it STQL, but there already exists another STQL introduced in the previous section, in this paper we call it K.R.P STQL.
that stand for the first and final points, respectively. Also, the polygon type has a sequence of points. K.R.P STQL data model uses the spatiotemporal first normal form (ST-1NF) sustaining a spatial attribute to have one or more values. [52, 53].

<table>
<thead>
<tr>
<th>Primary Key</th>
<th>Primitive attribute (char, int, bool)</th>
<th>Spatial attribute (point, line, polygon)</th>
<th>Temporal attribute (valid, transaction time)</th>
</tr>
</thead>
</table>

Figure 7.5: Table structure of STDB [53]

Let’s consider the following relations–BUILDING and CABLE. This BUILDING relation has Name, Owner, Price, Location, and Shape attributes. Especially, Location and Shape attributes are point and polygon types, respectively. CABLE relation has Name, Manhole, and Section attributes. Section attribute is line type. Since K.R.P STQL has been built upon a bitemporal model, there are additional columns such as VF, VT, TS, and TE. The synonym VF stands for valid from time. Also the VT, TS, and TE mean valid to, transaction start, and transaction end, respectively [53].

BUILDING (Name, Owner, Price, Location, Shape, VF, VT, TS, TE)
CABLE (Name, Manhole, Section, VF, VT, TS, TE)

Figure 7.6 and Figure 7.7 shows snapshot tables for BUILDING and CABLE relations.5

7.4.2 Syntax of K.R.P STQL

K.R.P STQL is designed on the basis of SQL3 and TSQL2 and the syntax is defined as follows:

```
SELECT attribute_name [, attribute_name]*
FROM table_name [AS alias_name][, table_name [AS alias_name]]*
[VALID temporal_expression]
[WHERE general_predicate | spatial_expression]
[WHEN temporal_expression]
```

The SELECT statement is a major one in DML (Data Manipulation Language) and it retrieves the spatiotemporal information for objects from the past to current using specified temporal and spatial predicates. The target list of the select statement is filled with attributes to be displayed or stored into another table. An alias name for table can be used in the FROM clause. The VALID clause is used to specify the time to be displayed for retrieved tuples. The WHERE clause includes the relational and spatial predicates. It works on primitive and spatial attributes, respectively. The spatiotemporal predicates are described in WHERE and WHEN clauses. In queries, the spatial operation AREA and temporal operation VALID can be used as a function type [52, 53].

7.4.3 Examples of Queries

For the comparison, we need to find a parametric database for Figure 7.6 and Figure 7.7. Figure 7.8 shows the parametric database for BUILDING table. In the table, we use \textit{reg}_i for regions indicated as polygon type in Figure 7.6. Since there five different shapes in the relation, each polygon shape maps to its corresponding region. Figure 7.9 shows a parametric database for Figure 7.7. Since there are three different regions, we denote them as \textit{creg}_i.

\footnote{In order to simplify the date, we listed the date shown in the table in [53], and gave unique numbers to every date.}
<table>
<thead>
<tr>
<th>Name</th>
<th>Owner</th>
<th>Price</th>
<th>Address</th>
<th>Location</th>
<th>Shape</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amazon Book Store</td>
<td>Jane</td>
<td>100,000</td>
<td>Gaeshin 25</td>
<td>(100,100)</td>
<td>((90,90),(90,110), (110,110),(110,90))</td>
</tr>
<tr>
<td>KFC</td>
<td>Tom</td>
<td>35,000</td>
<td>Sajik 77</td>
<td>(50,50)</td>
<td>((40,40),(40,60), (60,60),(60,40))</td>
</tr>
<tr>
<td>KFC</td>
<td>Tom</td>
<td>35,000</td>
<td>Sajik 77</td>
<td>(50,50)</td>
<td>((40,40),(40,60), (60,60),(60,40))</td>
</tr>
<tr>
<td>MacDonald</td>
<td>Nick</td>
<td>70,000</td>
<td>Sajik 77</td>
<td>(50,50)</td>
<td>((40,40),(40,60), (60,60),(60,40))</td>
</tr>
<tr>
<td>Lotteria</td>
<td>Jane</td>
<td>90,000</td>
<td>Gakyung 45</td>
<td>(200,200)</td>
<td>((190,190),(190,210), (210,210),(210,190))</td>
</tr>
<tr>
<td>Lotteria</td>
<td>Jane</td>
<td>90,000</td>
<td>Gakyung 115</td>
<td>(300,300)</td>
<td>((290,290),(290,310), (310,310),(310,290))</td>
</tr>
<tr>
<td>Lotteria</td>
<td>Jane</td>
<td>90,000</td>
<td>Gakyung 115</td>
<td>(300,300)</td>
<td>((290,290),(290,310), (310,310),(310,290))</td>
</tr>
<tr>
<td>Lotteria</td>
<td>Jane</td>
<td>90,000</td>
<td>Gakyung 115</td>
<td>(300,300)</td>
<td>((190,190),(190,410), (410,410),(410,190))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VF</th>
<th>VT</th>
<th>TS</th>
<th>TE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>NOW</td>
<td>1</td>
<td>UC</td>
</tr>
<tr>
<td>1</td>
<td>NOW</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>UC</td>
</tr>
<tr>
<td>2</td>
<td>NOW</td>
<td>3</td>
<td>UC</td>
</tr>
<tr>
<td>6</td>
<td>NOW</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>NOW</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td>9</td>
<td>10</td>
<td>UC</td>
</tr>
<tr>
<td>9</td>
<td>NOW</td>
<td>10</td>
<td>UC</td>
</tr>
</tbody>
</table>

Figure 7.6: BUILDING table snapshot [53]
<table>
<thead>
<tr>
<th>Name</th>
<th>Manhole</th>
<th>Section</th>
<th>VF</th>
<th>VT</th>
<th>TS</th>
<th>TE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heungdok-1</td>
<td>(120, 80)</td>
<td>((10,10),(20,50), (30,40),(150,190))</td>
<td>0</td>
<td>NOW</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>Heungdok-1</td>
<td>(120, 80)</td>
<td>((10,10),(20,50), (30,40),(150,190))</td>
<td>1</td>
<td>4</td>
<td>5</td>
<td>UC</td>
</tr>
<tr>
<td>Heungdok-1</td>
<td>(120, 80)</td>
<td>((10,10),(20,50), (30,40),(110,105), (140,135), (150,190))</td>
<td>5</td>
<td>NOW</td>
<td>5</td>
<td>UC</td>
</tr>
<tr>
<td>Heungdok-2</td>
<td>(40, 275)</td>
<td>((10,310),(20,250), (30,280),(70,390))</td>
<td>6</td>
<td>NOW</td>
<td>6</td>
<td>UC</td>
</tr>
</tbody>
</table>

Figure 7.7: CABLE table snapshot [53]

1. Retrieve the building name from BUILDING table that had been built before 1 March 1985\(^6\) and is over 50\(m^2\).

K.R.P STQL:
```
SELECT Name
FROM BUILDING AS B
WHEN BEGIN(VALID(B)) PRECEDE TIMESTAMP 1985-03-01
WHERE AREA(B.Shape) > 50
```

ParaSQL:
```
SELECT B.Name
RESTRICTED TO ~[2, NOW]
FROM BUILDING B
WHERE area([[B.Name]]) > 50
```

The above ParaSQL query retrieves tuples such that the size of a region is greater than 50\(m^2\). In order to do this, it uses function \texttt{area} returning the size of spatial element. After retrieving tuples satisfying the condition from BUILDING relation, it restricts parametric elements to the complement of interval [2, NOW]. Therefore, the RESTRICTED TO clause extracts tuples such that the parametric elements are valid before 85/03/01.

2. Retrieve the manhole that was built within 1 year from now and at 1 \(km\) distance from the line across the building. The building should have been built before 1 March 1985 and is over 50\(m^2\).

K.R.P STQL:
```
SELECT C.Manhole
FROM BUILDING AS B, CABLE AS C
WHEN BEGIN(VALID(B)) PRECEDE TIMESTAMP 1985-03-01
```

\(^6\)1985/03/01 has been denoted as 2 in the relational tables.
<table>
<thead>
<tr>
<th>Name</th>
<th>Owner</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1, NOW] Amazon</td>
<td>[1, NOW] Jane</td>
<td>[1, NOW] 100,000</td>
</tr>
<tr>
<td>Book Store</td>
<td>[reg1]</td>
<td>[reg1]</td>
</tr>
<tr>
<td>[1, 2] KFC</td>
<td>[1, 2] Tom</td>
<td>[1, 2] 35,000</td>
</tr>
<tr>
<td>[reg2]</td>
<td>[reg2]</td>
<td>[reg2]</td>
</tr>
<tr>
<td>[1, NOW] MacDonald</td>
<td>[1, NOW] Nick</td>
<td>[1, NOW] 70,000</td>
</tr>
<tr>
<td></td>
<td>[reg2]</td>
<td>[reg2]</td>
</tr>
<tr>
<td>[6, NOW] Lotteria</td>
<td>[6, NOW] Jane</td>
<td>[6, NOW] 90,000</td>
</tr>
<tr>
<td></td>
<td>[reg3]</td>
<td>[reg3]</td>
</tr>
<tr>
<td></td>
<td>[reg4]</td>
<td>[reg4]</td>
</tr>
<tr>
<td></td>
<td>[reg5]</td>
<td>[reg5]</td>
</tr>
<tr>
<td>[6, 9]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 7.8: An instance of BUILDING table in the parametric data model

<table>
<thead>
<tr>
<th>Name</th>
<th>Manhole</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0, 4] Heungdok-1</td>
<td>[0, 4] (120, 80)</td>
</tr>
<tr>
<td>[creg1] ∪ [5, NOW] [creg2]</td>
<td>[creg1] ∪ [5, NOW] [creg2]</td>
</tr>
<tr>
<td>[6, NOW] Heungdok-2</td>
<td>[6, NOW] (40, 275)</td>
</tr>
<tr>
<td>[creg3]</td>
<td>[creg3]</td>
</tr>
</tbody>
</table>

Figure 7.9: An instance of CABLE table in the parametric data model
ParaSQL:

```
SELECT C.Manhole
RESTRICTED TO [NOW-1, NOW]
FROM CABLE C
WHERE distance(C.Manhole,
    (SELECT B.Location
     RESTRICTED TO ~[2, NOW]
     FROM BUILDING B
     WHERE area(B.Shape) > 50
    ) <= 1
```

In ParaSQL, it uses a nested SELECT statement. It first retrieves a tuple from CABLE relation and checks the distance between the location of a manhole and a location of a building returned from the nested relational expression. The nested relational expression is exactly same as the query discussed in the previous example. It returns the locations of buildings such that the building was built before 85/03/01 and the size of area is 50 m$^2$. If the distance between two locations is less than 1 km then the tuple from CABLE relation satisfies the condition. Then the tuple is restricted to a temporal element [NOW-1, NOW] restricting the time to be valid only within 1 year.

As we can see in this example, ParaSQL uses a nested SELECT statement, but K.R.P STQL uses two variables in FROM clause causing a join operation between two relations-BUILDING and CABLE. Since the join operation is very expensive operation, it should be avoided if possible.

## 7.5 STSQL

### 7.5.1 STSQL Data Model

Authors in [10] extended SQL to spatiotemporal SQL (STSQL) and it is based on a temporally extended SQL, termed ATSQL [9]. STSQL supports the two temporal aspects, valid time and transaction time. STSQL has been designed at TimeCenter\(^7\) and the purpose of the query language is to support spatiotemporal query over spatiotemporal databases. In order to naturally generalize the snapshot relational model to a dimensional relational model, they adopt the view that a dimensional table simply is a collection of snapshot tables, with each snapshot table having an associated multi-dimensional point and containing all the snapshot tuples that have an associated multi-dimensional region that contains the point.

STSQL introduces new datatypes that capture time and space values. For time values STSQL uses anchored time periods. Spatial values are unions of regions. Regions are either defined over 1-, 2-, or 3-dimensional spatial domains. The corresponding datatypes are PERIOD, 1D_REGION, 2D_REGION, and 3D_REGION, respectively.

The below defines three relations-STANDS, ESTATES, and PLANDS.

\(^7\)http://www.cs.auc.dk/general/DBS/tdb/TimeCenter
In these relations, VT and TT represent a valid time and a transaction time of a tuple, respectively. In [10], the authors alter base tables for STANDS, ESTATES, and PLANS to add columns representing temporal and spatial information. But it is basically same work to create tables based on the relation schemas.

<table>
<thead>
<tr>
<th>STID</th>
<th>Index</th>
<th>Specie</th>
<th>Planted</th>
<th>Survey</th>
<th>STVT</th>
<th>STTT</th>
<th>StArea</th>
</tr>
</thead>
<tbody>
<tr>
<td>st100</td>
<td>high</td>
<td>pine</td>
<td>1935</td>
<td>1984-1986</td>
<td>1989-NOW</td>
<td>1996-NOW</td>
<td>regst100</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ESID</th>
<th>Owner</th>
<th>ESArea</th>
<th>ESVT</th>
<th>ESTT</th>
</tr>
</thead>
<tbody>
<tr>
<td>es34</td>
<td>Paul</td>
<td>reges34</td>
<td>1995-NOW</td>
<td>1994-NOW</td>
</tr>
<tr>
<td>es63</td>
<td>Mary</td>
<td>reges63</td>
<td>1996-NOW</td>
<td>1996-NOW</td>
</tr>
<tr>
<td>es80</td>
<td>Peter</td>
<td>reges80</td>
<td>1996-NOW</td>
<td>1995-1996</td>
</tr>
<tr>
<td>es401</td>
<td>Mary</td>
<td>reges401</td>
<td>1996-NOW</td>
<td>1995-1996</td>
</tr>
<tr>
<td>es80</td>
<td>Peter</td>
<td>reges80</td>
<td>1996-1999</td>
<td>1997-NOW</td>
</tr>
<tr>
<td>es401</td>
<td>Mary</td>
<td>reges401</td>
<td>1996-1999</td>
<td>1997-NOW</td>
</tr>
<tr>
<td>es100</td>
<td>Tom</td>
<td>reges100</td>
<td>2000-NOW</td>
<td>1997-NOW</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PLID</th>
<th>STID</th>
<th>Volume</th>
<th>Ripe</th>
<th>PLVT</th>
<th>Harvest1</th>
<th>Harvest2</th>
</tr>
</thead>
</table>

Figure 7.10: The spatiotemporal example database [10]

Figure 7.10 shows the tables for the relations, and the descriptions are as follows [10]:

1. The STANDS table models the (surveyed and analyzed) status of stands. For each stand, the specie of the stand’s dominant tree population, the soil fertility of the stand, the stand’s location, and a period of validity are recorded.

---

8The base relation does not have VT and TT columns. They are ordinary relational tables.
2. The ESTATES table records for each estate its owner, the validity period of the ownership, and the area that it covers.

3. The PLANS table records how stands are cultivated. For each stand, the volume to be harvested and the ripe year are recorded. Each plan has two harvest periods, calculated according to different scheduling methods that emphasize some growth conditions differently, e.g., according to soil fertility, climate, etc.

7.5.2 Syntax of STSQL

STSQL follows the syntax of SQL. In addition, it has a flag before SQL statements are described. Flag is the central construct and is used to indicate the desired evaluation mode(s) such as whether the statements have to be evaluated sequentially and/or non-sequentially. The following EBNF defines the syntax of flag. The <cursor specification> is the standard’s production for the SELECT statement.

<cursor specification> ::= flags <query expressions> [ <order by clause>]
flags ::= [flag { “AND” flag } | range_spec { “AND” range_spec}]
flag ::= modifier dimensions [ domain constant ] [range_spec]
range_spec ::= “SET” <identifier> dim_datatype range_expression
modifier ::= “SEQUENCED” | “NONSEQUENCED”
dimensions ::= “(” column_reference { column_reference } “)” [ “AS” <identifier>]
dim_datatype ::= PERIOD | 1D_REGION | 2D_REGION | 3D_REGION

SEQUENCED string indicates how to handle the dimension attributes in the queries and also restricts the qualifying tuples based on the values of their dimension attributes. NONSEQUENCED indicates dimension attributes that should be treated as regular attributes in a query [10].

7.5.3 Examples of Queries

For the comparison with ParaSQL, we need to transform the database shown in Figure 7.10 into a parametric database. Figure 7.11 and Figure 7.12 show the corresponding relations in the parametric database.

1. For each stand that is ripe in 2000, determine its harvest periods.

STSQL:

```
> SEQUENCED (STVT, PLVT) AS VT AND
NONSEQUENCED (Harvest1, Harvest2)
SELECT Harvest1, Harvest2
FROM STANDS ST, PLANS PL
WHERE PL.STID = ST.STID
AND ST.Ripe = 2000;
```

ParaSQL:

```
SELECT P.Harvest1, P.Harvest2
FROM PLANDS P
WHERE P.Ripe = 2000
```
Figure 7.11: The parametric database for STANDS relation

The above STSQL query needs to join STANDS and PLANS relations as well as the join over the valid times to associate stands with relevant plans. Then it retrieves the harvest periods like regular attributes.

In the ParaSQL query, it uses one variable for PLANS relation. Because the primary key of PLANS relation is STID, the ParaSQL query does not need to join STANDS and PLANS relations. Therefore, the above query will returns Harvest1 and Harvest2 columns in tuples of qualifying that Ripe is 2000.

2. Determine all stands that did not change status for more than 5 years, together with the corresponding estate(s).

STSQL:

\[
> \text{SEQUENCED (STArea, ESArea) AS STESArea AND}
\]
\[
\text{SEQUENCED (STVT, ESVT) AS STESVT}
\]
\[
\text{SELECT STID, ESID}
\]
\[
\text{FROM STANDS, ESTATES}
\]
ESTATES

<table>
<thead>
<tr>
<th>ESID</th>
<th>Owner</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1995, NOW] × [reg_es34] es34</td>
<td>[1995, NOW] × [reg_es34] Paul</td>
</tr>
<tr>
<td>[1996, NOW] × [reg_es80] es80</td>
<td>[1996, NOW] × [reg_es80] Peter</td>
</tr>
<tr>
<td>[1996, NOW] × [reg_es100] es100</td>
<td>[1996, NOW] × [reg_es100] Tom</td>
</tr>
</tbody>
</table>

PLANS

<table>
<thead>
<tr>
<th>STID</th>
<th>PLID</th>
<th>Volume</th>
<th>Ripe</th>
<th>…</th>
</tr>
</thead>
</table>

Figure 7.12: The parametric database for ESTATES and PLANS relations

WHERE DURATION(STVT, YEAR) > 5;

ParaSQL:

SELECT S.STID, E.ESID
RESTRICTED TO [[S.STID]]*[[E.ESID]]
FROM STANDS S, ESTATES E
WHERE duration([[[S.STID]], YEAR] > 5

The above STSQL query requires to sequentially join the locations of STANDS and ESTATES and the valid time of both relations. It restricts the valid time with function DURATION predefined in STSQL, and projects STID and ESID.
In ParaSQL query, it retrieves joined tuples from STANDS and ESTATES relations such that the duration of valid time of a stand is greater than 5 years. The query adopts function `duration` defined in STSQL to avoid introducing a new function in ParaSQL. After retrieving the qualified tuples, it restricts dimensions of the tuple to the intersection of the dimension of STID from STANDS relation and the dimension of ESID from ESTATES relation, respectively. If the result is not empty, the stand has not changed status for more than 5 years and there exists its corresponding estate(s).

3. For all stands, determine when the two harvest periods are scheduled contemporary.

**STSQL:**

```sql
> SEQUENNCED (PL1.Harvest1, PL2, Harvest2) AS AgreedHarvest
AND NONSEQUENCED (PL1.Harvest2, PL2.Harvest1)
SELECT PL1.STID
FROM PLANS PL1, PLANS PL2
WHERE PL1.STID = PL2.STID;
```

**ParaSQL:**

```sql
SELECT P.STID
FROM PLANS P
WHERE overlap(P.Harvest1, P.Harvest2)
```

In the above STSQL query, it needs a self-join two PLANS relations and sequentially joining Harvest1 and Harvest2. Since we are not interested in Harvest2 in PL1 and Harvest1 in PL2, a nonsequential semantics has to be specified for those harvest periods.

In the above ParaSQL query, it retrieves tuples such that attribute value of Harvest1 is overlapped with that of Harvest2. Here, the ParaSQL query uses function `overlap`. This function is defined a little bit differently because it takes two parameters which are not dimensions of the attribute, but simple values. In ParaSQL query, there are no needs to use two variables for PLANS relation because STID is the primary key of PLANS relation.

### 7.6 Summary

There has been considerable research on general temporal databases and on spatial databases [83]. However, temporal databases and spatial databases have long been separate and important areas of database research, and researchers in both areas have felt that there are important connections in the problems addressed by each area, and the techniques and tools utilized for their solution [23].

In this chapter, we have discussed four different spatiotemporal data model and extended SQL style spatiotemporal query languages-SQL$^{ST}$, E.S. STQL, K.R.P STQL, and STSQL as well as compared them with ParaSQL. All these query languages have their specific features handling problems that the authors in the literature had encountered.

SQL$^{ST}$ is the minimal extension from the standard SQL and it can provide users much similar standard SQL style spatiotemporal query language. It, however, cannot grasp the specific features of spatiotemporal data with natural and convenient ways because of the restriction of the standard SQL for spatiotemporal query.
E.S STQL defined attribute values as temporal functions of types. Therefore, the query language is very suitable for applications to query moving objects over time. One disadvantage of E.S STQL is that users have to handle the dimension alignment between two different domains by using lift operation.

K.R.P STQL has been built based on TSQL2 data model incorporated with spatial attributes. As mentioned in [84], TSQL2 proposal was met with little success. Since the model in K.R.P STQL stores an object in different tuples, it requires to do a self-join to process queries on the object.

STSQL was proposed in reference [9] and developed at TimeCenter. Because space and time are captured by separate attributes, STSQL is intended for applications that do not involve storing the movement of continuously moving objects [81]. Since STSQL data model is also based on TSQL2 data model, it has the same drawbacks that K.R.P STQL has.
Chapter 8
Hierarchical Data Format

8.1 Introduction
Hierarchical Data Format (HDF) is a scientific data management project providing basic tools for managing scientific data, and support for scientists who use HDF software. HDF products include two distinct data formats, HDF4 and HDF5, supporting libraries for reading and writing HDF data, and utilities for working with HDF. Available since 1988, HDF has become a standard for scientific data archiving and data sharing. HDF5 is a completely new version of HDF that provides an improved format and more general data model, supports efficient storage of very large datasets, and is supported by an I/O library that runs on both serial and parallel platforms [31]. According to [57], the motivation of the development of HDF5 is to overcome the limitations of HDF4 which are summarized as follows:

1. A single file cannot store more than 20,000 complex objects, and a single file cannot be larger than 2 gigabytes.

2. The data models are less consistent than they should be, there are more object types than necessary, and datatypes are too restricted.

3. The library source is old and overly complex, does not support parallel I/O effectively, and is difficult to use in threaded applications.

HDF5 can store large number of large data objects, such as Multi-dimensional arrays, tables, and computational meshes, and these can be mixed together in any way that suits a particular application. HDF5 supports cross platform portability of the interface and corresponding file format, as well as ease of access for scientists and software developers. HDF5 is used by many scientific and engineering applications, including the NASA Earth Observing System (NASA-EOS)\(^1\) and DOE Advanced Simulation and Computing (DOE-ASCI)\(^2\) projects [68].

The HDF implements a model for managing and storing data. The model includes an abstract data model and an abstract storage model (the data format), and libraries to implement the abstract model and to map the storage model to different storage mechanisms. The HDF5 library provides a programming interface to a concrete implementation of the abstract models. The library also implements a model of data transfer, i.e., efficient movement of data from one stored representation to another stored representation. Figure 8.1 illustrates the relationships between the models and implementations [56].

\(^1\)http://eospso.gsfc.nasa.gov/
\(^2\)http://www.dp.doe.gov/asc/home.htm
As we can see in Figure 8.1, HDF5 consists of three models and two implementations. In the following sections, we will discuss Abstract Data Model and Programming Model only. Since the Library\(^3\) implements the programming model, and is Storage Model is a method to represent objects from the Abstract Data Model, introducing the Abstract Model and Programming Model is sufficient for our purpose. For the Stored Data\(^4\), it is beyond the scope of the project.

### 8.2 Abstract Data Model

The Abstract Data Model (ADM) defines concepts for defining and describing complex data stored in files. The HDF5 ADM is a very general model which is designed to conceptually cover many specific models of data. Many different kinds of data can be mapped to objects of the HDF5 ADM, and therefore stored and retrieved using HDF5. The ADM is not, however, a model of any particular problem or application domain. Users need to map their data to the concepts of the ADM \[56\]. The key concepts include File, Group, Dataset, Datatype, Dataspace, and Property List.

#### 8.2.1 File

Abstractly, an HDF5 File is a container for an organized collection of objects. The objects are Groups and Datasets and other objects. The objects are organized as a rooted, directed graph. Every HDF5 file has at least one object, the root Group. All objects are members of the root Group or descendents of the root Group. HDF5 objects have a unique identity within a single HDF5 file, and can be accessed only by its names within the hierarchy of the file. HDF5 objects in different files do not necessarily have unique identities, and it is not possible to access a permanent HDF5 object except through a file \[56\].

---

\(^3\)The HDF5 Library implements the HDF5 Data Model and Storage Model as described above. In order to be as portable as possible, the library is implemented in portable C, which is not an object-oriented language.

\(^4\)The Store Data is the concrete implementation of the Storage Model. The Storage Model is mapped to several storage mechanisms, including single disk files, multiple files (family of files), and memory representations \[56\].
8.2.2 Group

As suggested by the name Hierarchical Data Format, an HDF5 file is hierarchically structured. The HDF5 group and link objects implement this hierarchy. In the simple and most common case, the file structure is a tree structure; in the general case, the file structure may be a directed graph with a designated entry point. The tree structure is very similar to the file system structures employed on UNIX systems, directories and files, and on Apple Macintosh and Microsoft Windows systems, folders and files. HDF5 groups are analogous to the directories and folders; HDF5 datasets are analogous to the files. The one very important difference between the HDF5 file structure and the above-mentioned file system analogs is that HDF5 groups are linked as a directed graph, allowing circular references; the file systems are strictly hierarchical, allowing no circular references [56].

8.2.3 Dataset

An HDF5 dataset is an object composed of a collection of data elements, or raw data, and metadata that stores a description of the data elements, data layout, and all other information necessary to write, read, and interpret the stored data. From the viewpoint of the application the raw data is stored as a one-dimensional or multi-dimensional array of elements (the raw data), those elements can be any of several numerical or character types, small arrays, or even compound types similar to C structs. A Dataset objects is stored in a file in two parts: a header and a data array. The header contains information that is needed to interpret the array portion of the dataset, as well as metadata (or pointers to metadata) that describes or annotates the dataset. Header information includes the name of the object, its dimensionality, its number-type, information about how the data itself is stored on disk (the storage layout), and other information used by the library to speed up access to the dataset or maintain the file's integrity [56, 57].

The Figure 8.2 illustrates the range of possibilities—the group structure is strictly hierarchical, and the structure takes advantage of the directed graph’s allowance of circular references, In Figure 8.2-(b), GroupA is not only a member of the root group, /, but a member of GroupC. Since GroupC is a member of GroupB and GroupB is a member of GroupA, Dataset1 can be accessed by means of the circular reference /GroupA/GroupB/GroupC/GroupA/Dataset1. Figure 8.2-(c) illustrates an extreme case in which GroupB is a member of itself, enabling a reference to a member dataset such as /GroupA/GroupB/GroupB/GroupB/Dataset2 [56].

8.2.4 Dataspace

The HDF5 Dataspace describes the layout of the elements of a multidimensional array. Conceptually, the array is a hyper-rectangle with one to 32 dimensions. HDF5 Dataspaces can be extendable. Therefore, each dimension has a current and maximum size, and the maximum may be unlimited. The Dataspace describes this hyper-rectangle: it is a list of dimensions, with the current and maximum (or unlimited) size. Dataspace objects are also used to describe hyperslab selections from a dataset. Any subset of the elements of a Dataset can be selected for read or write by specifying a set of hyperslabs. A non-rectangular region can be selected by the union of several (rectangular) Dataspaces [56].

8.2.5 Datatype

The HDF5 Datatype object describes the layout of a single data element. A data element is a single element of the array; it may be a single number, a character, an array of numbers or carriers, or other data. The Datatype object describes the storage layout of this data. Data types are
categorized into 11 classes of Datatype–Time, Bitfield, String, Reference, Opaque, Integer, Float, Array, Enumeration, Variable Length, and Compound. Each class is interpreted according to a set of rules and has a specific set of properties to describe its storage [56].

Basically, the Datatype Class consists of two main data types—Atomic Datatype and Composite Datatype. Atomic Datatypes are indivisible, each may be a single object. The first seven classes are in Atomic Datatype, and the others in Composite Datatype composed of multiple elements of Atomic Datatypes.
8.2.6 Attribute

Any HDF5 Named Data Object–Group, Dataset, or Named Datatype–may have zero or more user defined Attributes. Attributes are used to document the object. The Attributes of an object are stored with the object. An HDF5 Attribute has a name and data. The data is described analogously to the Dataset: the Dataspace defines the layout of an array of Data Elements, and the Datatype defines the storage layout and interpretation of the elements. In fact, an Attribute is very similar to a Dataset with the following limitations as follows [56]:

1. An attribute can only be accessed via the object, attribute names are significant only within the object. Attributes cannot be shared.

2. For practical reasons, an Attribute should be a small object, no more than 1000 bytes.

3. The data of an Attribute must be read or written in a single access, selection is not allowed.

4. Attributes do not have Attributes.

8.2.7 Property List

HDF5 has a generic Property List object, which is a collection of pairs consisting of names and values. Property List has a specific set of Properties. Each Property has an implicit name, an HDF5 Datatype, and a value. A Property List object is created and used similar to the other objects of the HDF5 library. Property Lists are attached to the object in the library, they can be used by any part of the library. Some properties are permanent (e.g., the chunking strategy for a dataset), others are transient (e.g., buffer sizes for data transfer). A common use of a Property List is to pass parameters from the calling program to a Virtual File Layer (VFL) driver or a module of the pipeline. Property Lists are conceptually similar to Attributes. Property Lists are information relevant to the behavior of the library, while Attributes are relevant to the user’s data and application. Properties are used to control optional behavior for file creation, file access, dataset creation, dataset transfer (read, write), and file mounting [56].

8.3 The Programming Model

8.3.1 Create an HDF5 File

This programming model shows how to create a file and also how to close the file. Figure 8.3 shows a code fragment to illustrate these steps. If there is a possibility that the file already exists, the user must add the flag H5ACC_TRUNC to the access mode to overwrite the previous file’s information [56, 57].

8.3.2 Create and Initialize a Dataset

The Datatype and dimensionality (dataspace) are independent objects, which are created separately from any dataset that they might be attached to. Because of this the creation of a dataset requires, at a minimum, separate definitions of datatype, dimensionality, and dataset. Hence, to create a dataset the following steps need to be taken:

1. Create and initialize a dataspace for the dataset to be written.

2. Define the datatype for the dataset to be written.
3. Create and initialize the dataset itself.

The code in Figure 8.4 illustrates the creation of these three components of a dataset object.

```c
hid_t   dataset, datatype, dataspace; /* declare identifiers */

/*
 * Create dataspace: Describe the size of the array and
 * create the data space for fixed size dataset.
 */
dimsf[0] = NX;
dimsf[1] = NY;
dataspace = H5Screate_simple(RANK, dimsf, NULL);

/*
 * Define datatype for the data in the file.
 * We will store little endian integer numbers.
 */
datatype = H5Tcopy(H5T_NATIVE_INT);
status = H5Tset_order(datatype, H5T_ORDER_LE);

/*
 * Create a new dataset within the file using defined
 * dataspace and datatype and default dataset creation
 * properties.
 * NOTE: H5T_NATIVE_INT can be used as datatype if conversion
 * to little endian is not needed.
 */
dataset = H5Dcreate(file, DATASETNAME, datatype, dataspace, H5P_DEFAULT);
```

Figure 8.4: A creation of a dataset with essential components
8.3.3 Write a Dataset to a New File

Having defined the datatype, dataset, and dataspace parameters, we can write out the data with a call to function H5Dwrite. Figure 8.5 shows an example of how to write to a dataset.

```c
/*
 * Write the data to the dataset using default transfer
 * properties.
 */
status = H5Dwrite(dataset, H5T_NATIVE_INT, H5S_ALL, H5S_ALL,
                  H5P_DEFAULT, data);
```

Figure 8.5: Write a dataset to a new file

The third and fourth parameters of H5Dwrite in the example describe the dataspaces in memory and in the file, respectively. They are set to the value H5S_ALL to indicate that an entire dataset is to be written. By changing the options, we can access a portion of a dataset.

Reading is analogous to writing. If we wish to read an entire dataset, we would use the same basic calls with the same parameters with replacing H5Dwrite with H5Dread [56, 57].

8.3.4 Getting Information about a Dataset

Although reading is analogous to writing, it is often necessary to query a file to obtain information about a dataset. For instance, we often need to know about the datatype associated with a dataset, as well dataspace information (e.g. rank and dimensions). There are several “get” routines for obtaining this information. The code segment in Figure 8.6 illustrates how to retrieve this kind of information [56, 57].

8.3.5 Creating a Group

To create a group, H5Gcreate is used. For example, the following code shown in Figure 8.7 creates a group called “Data” in the root group.

A group may be created in another group by providing the absolute name of the group to the H5Gcreate function or by specifying its location. For example, to create the group “Data_new” in the “Data” group, one can use the following sequence of calls as shown in Figure 8.8.

Note that the group identifier grp is used as the first parameter in the H5Gcreate function when the relative name is provided. Third parameter in H5Gcreate optionally specifies how much file space to reserve to store the names that will appear in this group. If a non-positive value is supplied, then a default size is chosen. H5Gclose closes the group and releases the group identifier [56, 57].

8.4 Summary

The main building blocks of HDF5 are the “dataset” and the “group.” An HDF5 dataset is a multidimensional array of elements of a specified datatype. Datatypes can be atomic (integers, floats, and others) or compound (like C structs). HDF5 groups are similar to directory structures in that they provide a way to explicitly organize the datasets in an HDF5 file [68].
/*
 * Get datatype and dataspace identifiers and then query
 * dataset class, order, size, rank and dimensions.
 */

datatype = H5Dget_type(dataset);  /* datatype identifier */
class = H5Tget_class(datatype);
if (class == H5T_INTEGER) printf("Data set has INTEGER type \n");
order = H5Tget_order(datatype);
if (order == H5T_ORDER_LE) printf("Little endian order \n");

size = H5Tget_size(datatype);
printf(" Data size is %d \n", size);

dataspace = H5Dget_space(dataset);  /* dataspace identifier */
rank = H5Sget_simple_extent_ndims(dataspace);
status_n = H5Sget_simple_extent_dims(dataspace, dims_out);
printf("rank %d, dimensions %d x %d \n", rank, dims_out[0], dims_out[1]);

Figure 8.6: Retrieve information about a dataset

;/*
 * Create a group in the file.
 */
grp = H5Gcreate(file, "/Data", 0);

Figure 8.7: Create a group

;/*
 * Create group "Data_new" in the group "Data" by specifying
 * absolute name of the group.
 */
grp_new = H5Gcreate(file, "/Data/Data_new", 0);

or

;/*
 * Create group "Data_new" in the "Data" group.
 */
grp_new = H5Gcreate(grp, "Data_new", 0);

Figure 8.8: Create a group in another group
When reading and writing an HDF5 dataset, an application describes two datasets: a source dataset and a destination dataset. These can have different sizes and shapes and, in some instances, can involve different datatypes. When an application writes a subset from the source to the destination, it specifies the subset of the source data that is to be written and the subset of the destination that is to receive the data. The only restriction on this operation is that the two subsets contain the same amount of data; like the datasets they can have entirely different shapes [68].

An HDF5 file is organized as a rooted, directed graph. The Named Data Objects are the nodes of the graph, and the links are the directed arcs. Each arc of the graph has a name, the root group has the name “/”. Objects are created and then inserted into the graph with the link operation, which creates a named link from a Group to the object.
Appendix A
EBNF of Parametric Structured Query Language

(*
Following grammar of ParaSQL is described by using EBNF developed by
ISO/IEC. Current version of ParaSQL is oriented to create and manipulate
temporal databases in the parametric data model. It is evolving and
there will be modifications for spatial, spatiotemporal, and multilevel
security databases. Up to this point, we don’t consider the features of
multi-dimensional databases except temporal databases. This version of
ParaSQL is limited to a minimal coverage of the standard SQL.
The features will be added if necessary.
*)

parametric sql =
  expression list, ";";
expression list =
  query expression | modification expression | create expression;

(* Data Definition Language *)
create expression =
  create database | create table;

(* Crate Database *)
create database =
  "CREATE DATABASE", database name, database parameter list;
database name =
  alpha numeric;
database parameter list =
  database parameter, {",", database parameter};
database parameter =
  "DATABASE ID", database id | "DATABASE SIZE", database size;
database id =
  integer;
database size =
  integer, ("K" | "M" | "G");
number of pages =
  integer;

(* Create Table *)
create table =
  "CREATE TABLE", table name, column list, [temporal ceiling];

table name =
  alpha numeric;
column list =
  "(", single column description,
{""," single column description},
""," primary key list,
"}"
}
single column description =
column name, data type, type list;

primary key list =
"PRIMARY KEY",
"(",
column name, { ",", column name},
")"
| "EMPTY KEY"

column name =
alpha numeric;
type list =
"temporal" | "static" | "snapshot" | "spatial" | "belief"
| "spatiotemporal"
temporal ceiling =
temporal element | "NOW"
data type =
"integer" | "string" | "real"
parametric element =
temporal element | spatial element | belief element | empty;
temporal element =
"[", natural number, ",", natural number, "]"

(* Data Manipulation Language *)
modify expression =
insert expression | delete expression
| update expression;

(* Insert Statement *)
insert expression =
"INSERT INTO", object name,
"("
parametric assignment, { ",", parametric assignment},
")"

parametric assignment =
"<", parametric element, parametric value,
{"", parametric element, parametric value, },
">

(* Delete Statement *)
delete expression =
"DELETE", object name, "WHERE", boolean key expression;

(* Update Statement *)
update expression =
    "UPDATE", object name,
    "SET", attribute term, "=" parametric assignment,
    {",", attribute term, "=" parametric assignment},
    "WHERE" boolean key expression;

boolean key expression =
    key condition, {"and", key condition};
key condition =
    key attribute name, "=" parametric value;
key attribute name =
    attribute term;
parametric value = value;

(* Expression for retrieving tuples *)
query expression =
    relational expression | domain expression | boolean expression;

relational expression =
    select statement,
    { ("+", select statement)
        | ("-", select statement)
        | ("*", select statement)
    };

(* Select Statement *)
select statement =
    "SELECT" attribute list,
    ["RESTRICTED TO", domain expression],
    "FROM", object list,
    ["WHERE", boolean expression];

attribute list =
    "*" attribute term, {",", attribute term};
object list =
    object name, object nickname, {"," object name, object nickname};
attribute term =
    object name, ".", attribute name;
object name =
    alpha numeric;
object nickname =
    alpha numeric;
attribute name =
    alpha numeric;

(*
    Symbols, ~,+,*,-, represent "complement", "union", "intersection",
    "difference", respectively.

113
domain expression =
  domain expression term,
  {  ("+", domain expression term) |
      ("-", domain expression term) |
    };

domain expression term =
  domain expression factor, {"*", domain expression factor};

domain expression factor =
  atomic domain expression | "~", atomic domain expression |
  "(" domain expression, ")" | "-", "(" domain expression, ")";

atomic domain expression =
  "[[", attribute, "]"]" |
  "[[", attribute, operation, attribute, "]"]" |
  "[[", attribute, operation, value, "]"]" |
  "[[", value, operation, attribute, "]"]" |
  "[[", value, operation, value, "]"]" |
  "[[", relational expression, "]"]" |
  parametric element;

boolean expression =
  (boolean expression factor | "NOT", boolean expression factor),
  {  ("OR", boolean expression factor) |
      ("AND", boolean expression factor) |
    };

boolean expression factor =
  atomic boolean expression |
  "(" boolean expression, ")" |
  "NOT", "(" boolean expression, ")";

(*
 parametric expression is to cover the following examples,
 [[A.p]] = [1,NOW]
 [[A.p]] = empty
 where [1,NOW] can be replaced with [region1] for spatial element.
*)

atomic boolean expression =
  attribute, operation, value |
  attribute, operation, attribute |
  domain expression, set operation, domain expression;

parametric element =
  temporal element | spatial element | belief element | empty;
temporal element =
    interval, {"+", interval};
interval = "[", natural number, ",", natural number, "]";
operation = equal | not equal | less than |
    less than or equal | greater than | greater than or equal;

set operation =
    equal | not equal | subsets | supersets | proper subsets |
    proper supersets | not subsets | not supersets |
    not proper subsets | not proper supersets;

number = natural number | float number;

(*
   Natural number should be either 0 or a number not starting with '0'
 *)
natural number = "0" | (digit - "0"), {digit};
float number = [(digit - "0"), {digit}], ",", {digit | "0"};
alpha numeric = letter, {letter | digit};
string = {letter | digit}, {" ", (letter | digit)};
value = ([sign], number) | """, string, "";

(*
   From here, we describe terminal notations.
 *)
subsets = "subsets";
supersets = "supersets";
proper subsets = "psubsets";
proper supersets = "psupersets";
not subsets = "nsubsets" | "not subsets";
not supersets = "nsupersets" | "not supersets";
not proper subsets = "npsubsets" | "not psubsets";
not proper supersets = "npsupersets" | "not psupsets";
empty = "empty";
sign = "-" | "+";
equal = "=";
not equal = "!=";
less than = "<";
less than or equal = "<=";
greater than = ">";
greater than or equal = ">=";
letter = "a"|"b"|"c"|"d"|"e"|"f"|"g"|"h"|"i"|"j"|"k"|"l"|"m"|"n"|
     "o"|"p"|"q"|"r"|"s"|"t"|"u"|"v"|"w"|"x"|"y"|"z";
digit = "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9";
Bibliography


