Object-relational spatio-temporal databases

Tsz-Shing Cheng

Iowa State University

Follow this and additional works at: https://lib.dr.iastate.edu/rtd

Recommended Citation
https://lib.dr.iastate.edu/rtd/10772
INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

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

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

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
Object-relational spatio-temporal databases

by

Tsz-Shing Cheng

A Dissertation Submitted to the
Graduate Faculty in Partial Fulfillment of the
Requirements for the Degree of
DOCTOR OF PHILOSOPHY

Department: Computer Science
Major: Computer Science

Approved:

Signature was redacted for privacy.

In Charge of Major Work

Signature was redacted for privacy.

For the Major Department

Signature was redacted for privacy.

For the Graduate College

Iowa State University
Ames, Iowa

1995
# TABLE OF CONTENTS

1. **INTRODUCTION**
   - 1.1. The existing relational model
   - 1.2. The type hierarchy and oids
   - 1.3. Algebraic identities and optimization
   - 1.4. A pattern matching language
   - 1.5. Related works
     - 1.5.1. A survey of temporal databases
     - 1.5.2. A survey of spatial database models
     - 1.5.3. Placement of computed types in existing object data models
   - 1.6. Thesis organization

2. **RelDimDB: GADIA'S RELATIONAL MODEL**
   - 2.1. AgriDB: a running example
   - 2.2. Dimensional elements
   - 2.3. Attribute values
   - 2.4. Tuples and relations
   - 2.5. Weak equality and key restructuring
   - 2.6. The query language RelDimSQL
     - 2.6.1. A sublanguage for associative navigation
     - 2.6.2. Relational expressions
     - 2.6.3. The select statement
     - 2.6.4. Dimension alignment

3. **ORDimDB: AN OBJECT-RELATIONAL DIMENSIONAL DATABASES**
   - 3.1. ORPersonnelDB: an object-relational database for personnel database
   - 3.2. Object identity
   - 3.3. Dimensional elements
   - 3.4. Attributes and object fragments
   - 3.5. Type hierarchy
     - 3.5.1. Attributes of a type
3.6. Path expressions 31
3.7. Tuple objects and keyed set objects 31
3.8. Equality of set objects 33
3.9. The query language ORDimSQL 34
  3.9.1. Dimensional expressions and boolean expressions 35
  3.9.2. Set expressions 35
  3.9.3. The projection operator 36
  3.9.4. The select statement 39
3.10. ORAgriDB: an object-relational database for AgriDB application 39
3.11. Dimension alignment 40
3.12. Upward compatibility for the classical user 43
3.13. Comparison of ORDimDB with other works 44
  3.13.1. Comparison of ORDimDB with relational model RelDimDB 45
  3.13.2. Comparison with the temporal extension of OODAPLEX 46

4. IDENTITIES AND ALGEBRAIC OPTIMIZATION 49
4.1. Algebraic identities 50
4.2. A dimensional framework for size estimation 55
4.3. Algorithm for algebraic optimization of spatio-temporal queries 58
  4.3.1. Principles for algebraic manipulation 58
  4.3.2. An algorithm for optimizing set expressions 60

5. THE PATTERN MATCHING LANGUAGE 65
5.1. The pattern matching process 66
  5.1.1. Temporal pattern matching 67
  5.1.2. Spatial pattern matching 69
  5.1.3. Spatial-temporal pattern matching 70
5.2. Construction of patterns 71
  5.2.1. Spatio-Temporal pattern 71
  5.2.2. Spatial pattern 75
  5.2.3. Temporal pattern 77
5.3. Storing patterns in functions 77
5.4. The construct \([ \ldots ]\) in ORDimSQL 78
5.5. IF-THEN-ELSE as dimensional expression 79
5.6. Examples 79
5.7. Modeling continuous data 83
  5.7.1. Incorporating continuous data 84

6. CONCLUSIONS AND FUTURE WORKS 88

BIBLIOGRAPHY 90

ACKNOWLEDGEMENTS 94
LIST OF FIGURES

Figure 1  AgriDB: A case study. 12
Figure 2  RelAgriDB: the relational design for AgriDB of Figure 1. 15
Figure 3  Crop relation with TILLAGE as the key. 16
Figure 4  Entity relationship diagram for ORPersonnelDB. 26
Figure 5  An object-relational design for ORPersonnelDB. 27
Figure 6  Instances in ORPersonnelDB. 28
Figure 7  An interesting type hierarchy. 30
Figure 8  Different forms of equality for set objects. 34
Figure 9  A type hierarchy illustrating the projection operation of Example 8. 38
Figure 10 ORAgriDB: the object-oriented design for AgriDB of Figure 1. 41
Figure 11 Instances in ORAgriDB. 42
Figure 12 A relational design for personnel database. 45
Figure 13 Example alignment factors and output factors for various selections. 57
Figure 14 Various patterns described in Section 5.1. 68
Figure 15 Creation of view in Example 28. 86
Figure 16 Creation of view in Example 29. 87
LIST OF TABLES

Table 1  Comparisons of savings with various parameters.  64
1

1. INTRODUCTION

In this thesis we term any assortment of ordinary, temporal, spatial and spatio-temporal data as a *dimensional database*. Different forms of dimensional data share considerable similarities. Conventional databases only maintain ordinary data, i.e. data which is space and time independent. Furthermore, research in dimensional databases has been concentrated on one particular form of dimensional data and there are no models to handle different forms of dimensional data uniformly. Therefore, the application software developed in these models may need to be re-coded when system migration from one form of dimensional data to another takes place. The development of application software is one of the most expensive components of computer systems. Therefore, the query languages for dimensional data should be user friendly and scalable.

In our ongoing research we model various forms of dimensional data uniformly. We have given a relational model RelDimDB and an algebraic query language RelDimSQL for dimensional data in [GCT93].

In this thesis we add type hierarchy and oids (object identifiers) to the relational model to obtain an object-relational model ORDimDB and an algebraic query language ORDimSQL. Complex objects are beyond the scope of this thesis, and they are left as future work. Thus, we only consider tuple objects and set objects, where a tuple cannot have a set object as a value of an attribute. This is the reason we term our model object-relational rather than object-oriented. We study algebraic optimization for our algebra. Finally, we give a pattern matching language to extend expressibility of our SQL-like query language ORDimSQL.

The organization of the remainder of this section is as follows. A brief introduction to the existing relational model is given below in Section 1.1 (a more detailed account of the model is presented in Chapter 2). An introduction to the topics covered this thesis is given below in Sections 1.2 to 1.4. In Section 1.5 we discuss related works. An organization for the remaining chapters of the thesis is given in Section 1.6.
1.1. The existing relational model

A detailed account of our existing relational model RelDimDB and SQL-like query language for dimensional data is given in Chapter 2. Here we discuss some interesting features of the model. It is helpful to keep in mind that our central goal has been to obtain query languages which are as natural as possible.

**Dimensional elements and value dimensioning**

To model real world objects in a database as naturally as possible we use *dimensional elements* that are subsets of the underlying dimensional space. The exact nature of dimensional elements is left open to implementation; the only required property is that they must be closed under union, intersection, complementation and difference. The closure properties achieve two objectives. First a real world object can be represented as a single database object, without having to fragment it into potentially unbounded number of database objects. Second this correspondence remains intact under algebraic operations, laying a foundation for query languages which handle "or," "and" and "not" of natural languages symmetrically. For temporal data a dimensional element, called temporal element, may be defined as a finite union of time intervals. Temporal elements are widely used in temporal databases as a timestamp of choice. Note that the dimensional elements are the most primitive types of dimensional domains in our model.

The only primitive type of value in our models is a function from a dimensional element into ordinary values, e.g. the salary history of an employee: \((11,45) \cup (71,75) \rightarrow 30K, (51,60) \rightarrow 40K\). Note that even a constant such as 50K is treated as a function of time with 50K as its value. The addition of dimension in this way is termed *value dimensioning*. It allows the complexities of dimensional data to be incorporated at the lowest level in the model. Another extreme in the database literature is *tuple dimensioning*, which essentially amounts to appending an attribute to an ordinary tuple to establish the domain of validity of the tuple. Tuple dimensioning makes query languages complex as they lead to query languages which fail to achieve symmetry between "or," "and" and "not" of natural languages [GY91,GN93].
Homogeneity and weak equality

We assume that the dimensional domain of all attributes in a tuple is the same. This is called the homogeneity assumption. Note that tuples using tuple dimensioning are a priori homogeneous. Under this assumption, we are guaranteed that information in a relation at a single point in the dimensional space is isomorphic to a classical snapshot relation without nulls. In this thesis we do not consider natural join operator.

Dimensional data is parametrization of ordinary data. Therefore, many concepts for classical relations, including the operators in classical relational algebra, carryover to the dimensional case. This is done by concentrating only on snapshots, which are guaranteed to be (isomorphic to) classical snapshot relations (without nulls). The notions which carry over to dimensional data in this way are termed weak. Clearly, the weak notions are based on snapshots of relations and ignore their structure. Perhaps one of the most fundamental definitions in dimensional database is that of weak equality which holds between two relations if they have the same snapshots. Weak equality is weaker than equality. The notions which are not weak are said to be strong. We view dimensional data largely as a study of the strong notions, because this is where they really depart from ordinary data.

Object fragments and restructuring

Formally, a fragment of an object o is information about that object during a dimensional domain \( \mu \), and it is denoted as \( \mu_0 \). Unlike ordinary objects, dimensional objects are interwoven. For example, consider the following fragment of an object X: “X managed a department Y during the time \( \mu \)” Note that X may have managed different departments at other times, and also that Y may have been managed by different managers at other times. In a natural language the given information about manager X can also be interpreted as information about department Y: “department Y was managed by the manager X during the time \( \mu \)” in the relational model one can switch from one view to the other, e.g. from department objects to manager objects, by using a key restructuring operator. The restructuring operators brings our query languages closer to natural languages. It also eliminates the need for storing an object fragment in multiple places, eliminating potential data redundancy at the logical level.
**Dimension alignment**

The purpose of dimension alignment is to make different dimension sorts to coexist as seamlessly as possible. As an example we allow a user to write $\mu \cap \text{reg}$, where $\mu$ is a temporal element and reg is a spatial element. The system interprets it as $(\mu \times \text{Space}) \cap (\text{Time} \times \text{reg})$, by padding each argument with the dimensional universes Space and Time in the missing dimensions. This allows a mix of temporal and spatial queries to be used in the spatio-temporal context without any changes. Also note that an ordinary constant such as 50K automatically becomes $(\text{Time} \ 50\text{K})$ in the temporal context.

**Algebraic SQL-like query language**

Our SQL-like language RelDimDB for dimensional data is algebraic. A literal upward dimensional compatibility with classical SQL is achieved at the level of a classical user. A classical user is a user in any dimensional database who only deals with the classical relations and classical SQL.

Now we discuss the contribution made in this thesis in extending our existing framework for relational dimensional databases.

**1.2. The type hierarchy and oids**

Although our relational and object-relational query languages RelDimSQL and ORDimSQL share some common features, syntactically similar queries in their two languages have different semantics because of the difference in the underlying database paradigms. This is because associative navigation in relational model is value based, but in the object-relational model inheritance through oids plays a more powerful role. More specifically, an attribute in the object-relational model is inherited by all subtypes, and use of such an attribute in a object-relational query would correspond to a cascade of natural joins in a relational query. Now we discuss the major features of the type hierarchy.
Object interleaving, object fragments and restructuring

In the relational case the attributes are purely value-valued, so the object fragments may be termed value fragments. In the object-relational case attributes can be oid-valued, and necessitate recognition of oid fragments to reduce data redundancy at logical level. Like value fragments, oid fragments not only eliminate data redundancy arising from object interleaving but also simplify queries because associative navigation can be incorporated in form of oid-valued attributes reducing the need for join operations in queries. It makes unrestricted use of natural language phrases such as “the manager of manager of the department of X” to be expressed naturally through path expressions. In the relational case such navigation would need a complex cascade of joins.

Placement of computed types and weak typing

The operators in the relational model are generalized in the object-relational model and placed naturally in a type hierarchy to maximally avail existing methods. Our type structure is deliberately designed to be weak. This eliminates potential seams arising from type hierarchy which can hinder equality among different database representations of the same real world information. For example, instead of associating a key with a type we associate it with an instance.

Dimension alignment

Different data dimensions are treated uniformly by the concept of dimension alignment, which simplifies queries and provides upward dimensional compatibility. With our approach upgrading from relational model to object-relational model, and, upgrading from ordinary data to dimensional data, become orthogonal. In other words, starting from ordinary data, in our framework the industry and users can smoothly migrate to object-relational paradigm, or more complex forms of dimensional data in any sequence without the need to recode the existing applications.
1.3. Algebraic identities and optimization

Algebraic identities and optimization for relational temporal data have been given in [NG92]. In the relational case the algebraic identities from that work carry over immediately to general forms of dimensional data. With a careful design of the type hierarchy and special attention to projection and cross product operators, the identities also carry over to the object-relational model. But the rules of heuristics for algebraic optimization do not carry over to general form of dimensional data even in the relational case. For example, rather than decreasing the size of an object, a selection can increase it, because representation of a small but complex region may require more disk space than a large and simple region. We identify such problems for optimizing dimensional data, and give a direction for algebraic optimization. However, we do not address all issues, e.g. taking path expressions into account in optimization is left as future work.

1.4. A pattern matching language

The existing query languages in temporal and spatial databases confine to simple associative navigation (A0B) and seem to lack the expressive power needed for application development. For application development one often has to resort to embedding a query language in a lower level language (e.g., C++). Although languages such as C++ are very powerful, they are general purpose languages and do not provide an appropriate level of abstraction for application development in spatio-temporal data. Motivated by this need, we design a pattern matching language for complex querying of spatio-temporal data.

Linguistics of space and time have similar as well as different features. On one hand, space and time both have a boolean algebraic structure of sets; on the other hand we experience and perceive space and time differently. Time has an evolutionary nature, where the concept of change is important.¹ in contrast, spatial regions are there, all at once, located in

---

¹ Three different types of logical times have been suggested in the temporal database literature: valid time, transaction time and decision time. However, we do not distinguish among them and our time dimension is generic.
some directions relative to each other, having certain shapes and sizes.

The pattern matching language is essentially a generalization of [[A0B]], our construct for associative navigation in dimensional SQLs. We make the philosophical observation that similarities can often be considered a special case of differences. This is reflected in our model: the associative navigation A0B in our SQL-like languages captures similarities, but its pattern matching generalization captures the difference in the linguistics of space and time. Even though the pattern matching languages seamlessly extends DimSQLs, there is a distinct shift of paradigm. The DimSQLs are declarative and algebraic, whereas the pattern matching is more imperative. We have not studied the optimization of the pattern matching language.

Syntactically a temporal pattern or a spatial pattern is a special case of spatio-temporal pattern. An advantage is that one can migrate from temporal databases or spatial databases to spatio-temporal data seamlessly, without having to rewrite the existing queries involving patterns. Also, the user needs to learn only one pattern matching language for a dimension irrespective of whether this dimension appears as the only dimension or coexists with other dimensions. The pattern matching language is very expressive, yet the patterns are easy to construct.

Our pattern matching language is intended to be independent of the choice of a model. Use of patterns is not confined to query languages. We feel that our pattern matching language would find applications in spatio-temporal active databases [Da88,SC91,DG93], where patterns can be used as triggers.

1.5. Related works

Now we survey some of the related works in temporal databases, spatial databases and works involving type hierarchy in object-oriented databases.

1.5.1. A survey of temporal databases

Ben-Zvi [BZ82] proposed a model for temporal databases, called the Time Relational model, in which timestamping is done at the tuple level. (An exposition of Ben-Zvi’s work
A relation is defined as a collection of sets of tuples sharing the same key values. Thus, Ben-Zvi's relations are not in 1NF (first normal form). Snodgrass [Sn87] gives a 1nf model with tuple dimensioning, and a query language TQuel, by adding a WHEN clause to specify temporal properties of what is to be retrieved and a VALID clause to attach a timestamp to what is to be retrieved. As stated above, a major problem with 1NF approaches is that they lead to complicated query languages.

Gadia and Vaishnav [GV85, Gad88] give a model for temporal databases with value-dimensioning and based on temporal elements rather than intervals. In [Gad92] Gadia introduces TempSQL, a SQL-like query language for temporal data. Clifford and Croker [CC87] use temporal element, called lifespan in their terminology. A relation can have both temporal attributes and non-temporal attributes. Tansel [Tan93] allows attributes to be atomic values and sets of atomic values, possibly with timestamps which are temporal elements. Whereas TempSQL collapses tuples with respect to keys, the query language SQLT in [Tan93] leaves this as an option and introduce collapse operation for combining tuples.

Wuu and Dayal [WD92] model temporal objects within an existing object-oriented data model and query language, OODAPLEX for ordinary data by treating time points as abstract data type. They support a somewhat limited form of oid fragments. Our oid fragments generalize the oid fragments in OODAPLEX and value fragments in [GCT93]. A drawback of the approach in [WD92] is that OODAPLEX was primarily designed for ordinary data, temporal queries in OODAPLEX are complex (see Section 3.13.) [DW92] extends the query processing for ordinary data in OODAPLEX to temporal queries. Such optimization may not be very efficient because of the 1NF nature of their model.

Segev and Shoshani [SS87] define the concept of a time sequence, such as salary history of an individual or the measurements taken by a particular sensor in an experiment. Rose and Segev [RS91] present an object-oriented model based on the concept of time sequence. In that model all properties of an object type (attributes, methods, and constraints) and values of the structures of the types can be treated as time-sequence objects.

---

1. A tuple is in first normal form if all its attribute values are atomic, having no structure.
Su and Chen [SC91] propose an object-oriented knowledge model to handle temporal data using interval timestamps. Rules express temporal and other semantic constraints of an application domain. There are three types of rules: a state rule verifies the state of a knowledge base, an operation rule performs an operation under various temporal conditions (or states), and a deductive rule deduces objects' data values which are not explicitly stored in the knowledge base.

Pissinou and Makki [PM92] give a model for object versioning. A version is of the form \{[t,t'], \{ver\}\}, where \([t,t']\) is a time interval and \(\{ver\}\) is a version or a sequence of versions. Binary relationships among objects are modeled by "(domain-object \([t_1,t_2], v\), mapping-object \([t_3,t_4], v\), range-object \([t_5,t_6], v\)"") triples. For example, the phrase “the telephone number of James is 743 during 1980-1990” is represented as (James ([50,\(\infty\]),1), HasNo([80,90],1), 743([80,90],1)).

1.5.2. A survey of spatial database models

Scholl and Voisard [SV89] define elementary regions as lines, polygons, etc. A region is defined either as an elementary region or a set of elementary regions. Guting [Gut88] uses strong typing and classifies several different types of spatial domains, e.g., point type, line type (curves), polygon type, etc. Roussopoulos et al. [RFS88] give an SQL-like query language called PSQL. Engenhofer [Egn94] proposed an SQL-like query language and a graphical presentation language for spatial databases. Several types of graphic specification are identified. All these languages use tuple dimensioning.

Gadia and Chopra [GC93b] postulate the notion of spatial elements as spatial domains of interest to a user as any subsets of the universe of space, such that spatial elements are closed under union, intersection, subtraction and complementation. In contrast to existing languages for spatial data, [GC93b] favors weakly typed regions with no hierarchy among them and value dimensioning, i.e. adding dimension literally to the most primitive values in a model. As stated above, a serious drawback of languages with tuple dimensioning is that they do not handle and, or, and not of natural languages in a symmetric and a natural way [GN93].
1.5.3. Placement of computed types in existing object data models

In this thesis, placement of computed types is an important issue that we pay close attention to. Therefore, we briefly survey some of the placement methods for computed types found in object-oriented database models.

Kim [Kim89, Kim94] proposed an object-oriented model, UniSQL/X, by extending the relational model. We note that in UniSQL, the projection operator is always placed under the system root, OBJECT. Scholl and Schek [SS90] gave an object-oriented model built upon the nested relational model. In their model computed type for every operator has to be either a subtype or a supertype of the input types. A projection is placed as a supertype of the input by removing the unprojected attributes. A projected attribute cannot be a path expression.

We do not postulate a root for our type hierarchy. Like our other operators, projection is placed in the existing type hierarchy to maximally avail existing methods. Projected attributes can be path expressions. Thus our projection is perhaps more interesting than those in UniSQL/X and [SS90]. However, it must be kept in mind that whereas the scope of these two models extends to complex objects, the current scope of our model does not.

Rundensteiner [Run94] proposed a classification mechanism for supporting object-oriented views to solve the type inheritance mismatch and the is-a incompatibility problem. To place a new type into an existing hierarchy, new virtual types may be created and existing type hierarchy may be modified.

1.6. Thesis organization

The rest of the thesis is organized as follows. In Chapter 2 we present Gadia’s relational model. In Chapter 3 we present our object-relational model, and we compare our object-relational model with the relational model and OODAPLEX. In Chapter 4 we classify two kinds of identities and propose an algebraic framework for optimizing dimensional data. In Chapter 5 we present our pattern matching language for spatio-temporal databases, and illustrate how continuous data can be incorporated. In Chapter 6 we give our conclusion and future works.
2. RelDimDB: GADIA'S RELATIONAL MODEL

In this chapter we introduce our relational model RelDimDB and its query language RelDimSQL for dimensional data. As a running example in this thesis, a case study in agriculture is described in the next Section.

2.1. AgriDB: a running example

In this section we introduce an application, called AgriDB, in agriculture environmental management. The application is a mix of spatial information, spatio-temporal information, and ordinary data. The purpose of the application is to determine and make decisions about the environmental consequences of using various chemicals in agriculture. The following is a detailed description of the application. A summary and spatial maps where this application experiment is conducted are shown in Figure 1.

We are given a fixed spatial region, which we can assume to be 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 applied to the region. Some of these chemicals seep through the soil and contaminate ground water. The seepage depends on the chemical being applied, the crop type, the tillage method, and the soil texture. We are given some U.S. Environmental Protection Agency (EPA) data about chemicals. This data specifies the range of acceptable contaminant levels of the chemicals in ground water. We are given some wells, where readings are taken from time to time to monitor contaminants in the ground water.

As is customary in agriculture, for this application we pair up the wells such that each well belongs to one and only one pair. A pair of wells is usually treated as a single entity (located at a single point) and the two wells in the pair are classified as up-gradient (u/g) and down-gradient (d/g) depending upon 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
- **Information about soil texture.** This data is only spatial and it is time independent. A map of soil texture is displayed above.

- **Information about crops and tillage methods.** In this example application, crops are not rotated through the year; thus crop information is only spatial, and time independent. A map of this data is displayed above.

- **EPA data.** This data specifies the range of acceptable contaminant levels of the chemicals in ground water. This data is ordinary; it is space independent, as well as time independent.

- **Chemical readings.** This data consists of readings taken from various wells at different times. It is space and time dependent. A map showing the location of wells is displayed above.

Figure 1. AgriDB: A case study.
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. Time is assumed to be acyclic.

2.2. Dimensional elements

We assume an underlying universal dimensional domain $P$. The user views it as a set of points in the dimensional space. A point in the dimensional space is sometimes referred to as a dimensional point. We postulate that certain subsets of $P$, called dimensional elements, are of interest to users, and they are closed under union ($\cup$), intersection ($\cap$), subtraction ($-$) and complementation ($\neg$). For example if the dimensional domain is the set of time instants $Time = [0,NOW] = \{0,1,\cdots,NOW\}$, then the dimensional elements are temporal elements [Ga88, GY88]. If the dimensional domain is the spatial domain $Space$, then the dimensional elements are spatial elements [GC92T]. If the dimensional domain is $Time \times Space$ we get spatio-temporal elements. The closure properties of dimensional elements make it possible to store information about a stored or computed object in a single tuple. This results in a simpler query language, as discussed in more detail in Section 2.6. Note that we do not make specific assumptions about the constitution of $P$. For example, we do not assume that $P$ is discrete or continuous.

2.3. Attribute values

To capture the changing value of an attribute through different points in a dimensional space, we introduce the notion of a dimensional assignment: a dimensional assignment (or simply assignment) $\xi$ to an attribute $A$ is a function from some dimensional element $\mu$ into domain of $A$, where domain of $A$ is of primitive type (e.g., real, integer, $\cdots$). An example of a spatial assignment to the attribute CROP is $(\text{reg}_1 \text{ wheat, reg}_2 \text{ corn})$, where $\text{reg}_1$ and $\text{reg}_2$ are spatial elements. An example of a temporal assignment to the attribute SALARY is $(\[60,65\] 20K, \[66,80\] 35K)$. The domain of an assignment is called its dimensional domain. The operator $[\ ]$ denotes the dimensional domain of a dimensional assignment. Thus, for the spatial
assignment above, \[\{(\text{reg}_1 \text{ wheat}, \text{reg}_2 \text{ corn})\}\] = \text{reg}_1 \cup \text{reg}_2.

2.4. Tuples and relations

A tuple is simply a concatenation of dimensional assignments whose dimensional domains are the same. The dimensional domain of a tuple \(\tau\), denoted \([\tau]\), is simply the domain of any of its assignments. The assumption that all assignments in a tuple have the same domain is called the homogeneity assumption [Ga88]. A dimensional relation \(r\) over \(R\), with \(K \subseteq R\) as its key, is a finite set of non-empty tuples, such that no key attribute value of a tuple changes from one dimensional point to another, and no two tuples agree on all their key attributes.

Suppose \(r\) is a relation and \(p\) is a dimensional point. The pointal of a relation \(r\) at a dimensional point \(p\), denoted by \(r(p)\), is the relation obtained by restricting each tuple of \(r\) to \(p\). Homogeneous relations are a parameterization of classical pointal relations without nulls. If a base relation is not homogeneous, its pointal would have null values. A study of null values is beyond the scope of this thesis.

We are now ready to model the information described in our AgriDB application from Section 2.1. Figure 2 shows how it is modeled as a spatio-temporal database, called RelA-griDB.

2.5. Weak equality and key restructuring

The purpose of a key is to provide a persistent identity to objects. Sometimes in forming a query, it is necessary to view the relation as having a different key. A pointal relation can have different (sets of attributes as its) keys, but the choice of key has no effect on the pointal relation. Two relations \(r\) and \(s\) over \(R\) are said to be weakly equal if at every dimensional point \(p\), the pointals of \(r\) and \(s\) are equal. If the key of a dimensional relation is changed, the resulting relation is unique and weakly equal to the one we start with, but its structure is different [Gad88]. We illustrate the idea of key restructuring through an example.
(a) The soil relation

<table>
<thead>
<tr>
<th>TEXTURE</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>sreg₁οsreg₄</td>
<td>sandy loam</td>
</tr>
<tr>
<td>sreg₂οsreg₃</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>

(b) The crop relation

<table>
<thead>
<tr>
<th>CNAME</th>
<th>TILLAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>creg₁οcreg₅</td>
<td>corn</td>
</tr>
<tr>
<td>creg₁</td>
<td>no till</td>
</tr>
<tr>
<td>creg₃</td>
<td>min till</td>
</tr>
<tr>
<td>creg₂οcreg₃</td>
<td>wheat</td>
</tr>
<tr>
<td>creg₂</td>
<td>conven till</td>
</tr>
<tr>
<td>creg₃</td>
<td>no till</td>
</tr>
<tr>
<td>creg₄</td>
<td>soybean</td>
</tr>
<tr>
<td>creg₄</td>
<td>no till</td>
</tr>
</tbody>
</table>

(c) The epa relation with concentration in parts per billion (ppb)

<table>
<thead>
<tr>
<th>CHEM-NAME</th>
<th>MAX</th>
<th>MIN</th>
</tr>
</thead>
<tbody>
<tr>
<td>atrazine</td>
<td>3.00</td>
<td>0</td>
</tr>
<tr>
<td>simazine</td>
<td>35.00</td>
<td>0</td>
</tr>
</tbody>
</table>

(d) The chems-in-wells relation with concentration in parts per billion (ppb)

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

- **The soil relation.** This relation is spatial, but it is time independent. The key is TEXTURE.
- **The crop relation.** This relation is also spatial, but time independent. The key of the relation is CNAME.
- **The epa relation.** This relation is space and time independent, and CHEM-NAME is its key.
- **The chems-in-wells relation.** For each chemical, it shows readings taken in various wells at different times. It is spatio-temporal. The key of this relation is CHEM-NAME.

Figure 2. RelAgriDB: the relational design for AgriDB of Figure 1.
Example 1. Suppose in the crop relation, we want the key to be TILLAGE instead of CNAME. This is done by restructuring the crop relation to obtain the crop relation as shown in Figure 3. (The use of key restructuring is illustrated in Example 5.)

<table>
<thead>
<tr>
<th>CNAME</th>
<th>TILLAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>creg₁</td>
<td>corn creg₄ \cup creg₃ \cup creg₄ no till</td>
</tr>
<tr>
<td>creg₃</td>
<td>wheat</td>
</tr>
<tr>
<td>creg₄</td>
<td>soybean</td>
</tr>
<tr>
<td>creg₅</td>
<td>corn creg₂</td>
</tr>
<tr>
<td>creg₂</td>
<td>wheat creg₅ min till</td>
</tr>
<tr>
<td>creg₂</td>
<td>conven. till</td>
</tr>
</tbody>
</table>

Figure 3. Crop relation with TILLAGE as the key.

2.6. The query language RelDimSQL

Now we introduce an SQL-like query language RelDimSQL. We assume that a database consisting of dimensional relations is given. The set of all RelDimSQL expressions can be divided into three mutually exclusive groups: dimensional expressions, boolean expressions, and relational expressions. The most interesting operator in RelDimSQL is its SQL-like select statement. The select statement draws its simplicity and power through associative navigation made possible by dimensional expressions and boolean expressions. Now we present our sublanguage for associative navigation. We then describe the relational operators such as union and difference of dimensional relations, followed by the select statement.

2.6.1. A sublanguage for associative navigation

The associative navigation in our model is done through dimensional expressions and boolean expressions. Note that boolean expressions are defined in terms of dimensional expressions. These expressions are defined as follows.

- A constant dimensional element is a dimensional expression. An example of such an expression is \( \text{reg} \times [0,5] \) which is spatio-temporal.
• If A is an attribute, then \([A]\) is a dimensional expression that evaluates to the domain of the attribute A. For example, for the first tuple in the crop relation of Figure 4.1, \([CNAME]\) evaluates to \(creg_1 \cup creg_5\).

• If A and B are attributes and \(\theta\) is an operator, then \([A\theta B]\) is a dimensional expression. This extracts points in the dimensional space where A and B are in \(\theta\)-relationship. For example, if A is the dimensional assignment \(\langle [0,5] a_1, [6,10] a_2 \rangle\), and B is \(\langle [3,8] a_1, [11,20] a_2 \rangle\), then \([A=B]\) evaluates to \([3,5]\). If A is an attribute, and b is a constant, then \([A\theta b]\) is a dimensional expression. For example, if A is the above dimensional assignment, then \([A=a_1]\) evaluates to \([0,5]\). This evaluation is done after identifying the constant \(a_1\) with \(\langle [0, NOW] a_1 \rangle\). Thus the dimensionless constant \(a_1\) has been aligned in the temporal dimension. A more general discussion of dimension alignment is given below in Section 2.6.4.

• If e is a relational expression, then \([e]\) is a dimensional expression, whose value is the union of domains of tuples in the relation computed by e. For example, \([\text{crop}]\) is \(creg_1 \cup creg_2 \cup creg_3 \cup creg_4 \cup creg_5\). This construct is a source of powerful nesting among RelDimSQL expressions. It can also be used by itself as a query.

• If \(\mu\) and \(\nu\) are dimensional expressions, then so are \(\mu \cup \nu, \mu \cap \nu, \mu - \nu\) and \(\neg \mu\).

• If \(\mu\) and \(\nu\) are dimensional expressions, then \(\mu \subseteq \nu\) is a boolean expression.

• We define \(A\theta B\) to be an abbreviation of the boolean expression \([A\theta B] \neq \emptyset\).

• If \(f\) and \(g\) are boolean expressions then so are \(f \text{ or } g, f \text{ and } g\) and \(\text{not } f\).

2.6.2. Relational expressions

Relational expressions are the syntactic counterparts of relations. There are seven operators for relational expressions: union, difference, intersection, selection, cross product, restructuring, and projection. In this thesis, however, we do not consider natural join and full form of cross product.
• Suppose \( r \) and \( s \) are dimensional relations with the same scheme and key. Then \( r \cup s \), \( r \setminus s \), and \( r \cap s \) are also relational expressions with same scheme and keys. The semantics of these operators are given below.

To arrive at \( r \cup s \) we first compute the union of \( r \) and \( s \) treating them as sets, and then collapse each pair of tuples of \( r \) and \( s \) which agree on all key attributes, into a single tuple. The resulting relation is \( r \cup s \).

The relation \( r \setminus s \) is computed as follows. We start with \( r \). Then for each tuple \( \tau \) of \( r \) we check to see if there is a tuple in \( s \) which agrees with \( \tau \) on the key attributes. If there is no such tuple in \( s \), then \( \tau \) does not change. If \( s \) has such a tuple \( \tau' \), then those dimensional points where \( \tau \) and \( \tau' \) agree on all attributes are removed from the domain of \( \tau \). This basically amounts to removing the overlap between \( \tau \) and \( \tau' \) from \( \tau \). If the resulting domain of \( \tau \) is not empty it is kept, otherwise it is discarded. The resulting relation is \( r \setminus s \).

The relation \( r \cap s \) is computed as follows. We start with \( r \). Then for each tuple \( \tau \) of \( r \) we check to see if there is a tuple in \( s \) which agrees with \( \tau \) on the key attributes. If there is no such tuple in \( s \), then \( \tau \) is discarded. If \( s \) has such a tuple \( \tau' \), then those dimensional points where \( \tau \) and \( \tau' \) disagree on some attributes are removed from the domain of \( \tau \). If the surviving portion of \( \tau \) is not empty it is kept, otherwise it is discarded. The resulting relation is \( r \cap s \).

• Suppose \( r \) is a relation, \( f \) is a boolean expression and \( \mu \) is a dimensional expression. Then the selection \( \sigma(r;f;\mu) \) evaluates to \{ \( \tau \) where \( \mu(\tau) \) is not empty \}. If \( f \) evaluates to \( \text{true} \) for a tuple, \( \sigma \) allows us to select only a relevant part of it. This relevant part is specified by a user through the temporal expression \( \mu \).

• Suppose \( r \) is a relation over \( R \) and \( s \) is a relation over \( S \), where \( R \cap S = \emptyset \). A tuple in the \( r \times s \) is obtained by concatenating a tuple in \( r \) and a tuple in \( s \), and only preserving the dimensional points where both tuples are defined. This assures the homogeneity of \( r \times s \). The key of \( r \times s \) is the union of the keys of \( r \) and \( s \). Let \( \tau_r, \tau_s \) be two tuples. Define hom(\( \tau_r \circ \tau_s \)) to be the homogeneous part of the concatenation of \( \tau_r \) and \( \tau_s \). We then define \( r \times s = \{ \text{hom}(\tau_r \circ \tau_s) : \tau_r \in r \land \tau_s \in s \land \text{hom}(\tau_r \circ \tau_s) \text{ is not empty} \} \).
• Suppose $r$ is a relation over $R$ with $K$ as its key. Then if $K' \subseteq R$ such that $K' \rightarrow R$ in all pointals of $r$, then the restructuring operation $I_{K'}(r)$ is the unique relation weakly equal to $r$ but with key $K'$.

• Suppose $r$ is relation over $R$ with key $K$. Then the projection $\Pi_L(r)$ is a relation over $L$ with key $K$, if $K \subseteq L$; otherwise, its key is $L$. The projection operator is the same as the classical projection operator, except that the resulting key is important.

2.6.3. The select statement

In RelDimSQL, the select statement is the most interesting one. The select statement is of the form given below.

```
select attributelist : K
restricted_to dimensionalExpression
from relationList
where booleanExpression
```

The semantics of the above select statement is as follows. A tuple $\tau$ is formed by selecting tuples from each relation in the relationList. Each item in relationList is of the form $r$ or $r:K$ where $r:K$ is the relation $r$ restructured so that $K$ is the key. For this tuple $\tau$, booleanExpression is evaluated. If $\tau$ does not satisfy booleanExpression, it is rejected. If $\tau$ satisfies booleanExpression, then dimensionalExpression is evaluated for this tuple. This gives us the portion of domain of $\tau$, which is of interest to us. The tuple $\tau$ is now restricted to this domain. If its domain becomes empty, it is rejected; otherwise its attribute values specified by attributelist are retrieved.

The key of the resulting relation can be explicitly provided by the construct : Key in the select clause. If the key is not explicitly provided, a default key is determined. We say that a relation $r$ is represented in the attributelist if the attributelist contains an attribute of $r$. If all the key attributes of such a relation $r$ are in the attributelist, these key attributes become part of the default key; otherwise all the attributes from $r$ appearing in the attributelist become part of the default key. A complete default key is constructed in this manner from all the relations represented in the attributelist.
The select statement of RelDimSQL is very powerful. It is clear that the booleanExpression and dimensionalExpression are both quite versatile. Both of them can involve relational expressions. This will be illustrated through several examples as below. All these examples refer to the RelAgriDB shown in Figure 2.

Example 2. The query *find all crops which are being grown in an area larger than 2000 square miles* can be expressed in RelDimSQL as follows.

```
select c.CNAME
from crop c
where area([c.CNAME]) > 2000
```

Example 3. The query *find the region where corn is being grown and soil texture is sandy loam* can be expressed in following two ways.

```
[[select c.CNAME
   restricted_to [[c.CNAME=corn]] \[s.TEXTURE=sandy loam]]
from crop c, soil s]

[[select c.CNAME restricted_to [[c.CNAME=corn]] from crop c]]
\[select * restricted_to [[s.TEXTURE=sandy loam]] from soil s]]
```

Example 4. The query *give information about those crops which are being grown somewhere using the no till method, limiting the retrieval to the area where min till method is used* is expressed below.

```
select *
   restricted_to [[c.TILLAGE = min till]]
from crop c
where [[c.TILLAGE = no till]] \neq \emptyset
```

Example 5. The query *give complete information about tillage methods which are used for corn* requires restructuring of the crop relation, and is expressed below.

```
select *
   from crop:TILLAGE c
where [[c.CNAME = corn]] \neq \emptyset
```
2.6.4. Dimension alignment

It is sometimes necessary to align dimensions while expressions are being evaluated. For example, a user can write \( \text{reg} \cap \mu \), where \( \text{reg} \) is purely a spatial region, 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 \( \text{reg} \cap \mu \) will be treated as \( (\text{reg} \times \text{Time}) \cap (\text{Space} \times \mu) \), where \( \text{Time} \) is the universe of time, and \( \text{Space} \) is the universe of space. An interesting corollary of this phenomenon is that \( \lfloor \rfloor \) operator can be applied to ordinary data. Thus, in the context of spatial alignment, if \( a \) is an ordinary constant, then \( \lfloor a \rfloor \) evaluates to \( \text{Space} \), and in the spatio-temporal context it evaluates to \( \text{Space} \times \text{Time} \). Our concept of alignment, presented here yields substantial dividends in terms of ease of querying.

Another example of dimension alignment in the context of select statement of RelDimSQL is given in Example 6. We note that removing a dimension can be problematic. The reason is that if we remove time dimension from an attribute value such as \( \langle \text{reg}_1 \times [0,5] \text{ red} \rangle \) to obtain \( \langle \text{reg}_1 \text{ red} \rangle \), in subsequent alignments it will be treated as \( \langle \text{reg}_1 \times \text{Time} \text{ red} \rangle \). This amounts to manufacturing new information not present in the database, which should be avoided. Similar problems occur if we remove space dimensions.

While evaluating the relational expressions, to be introduced next, we must keep in mind the idea of dimensional alignment. For instance, the \text{SALARY} \ history of an employee is space independent but time dependent. Hence, \( \text{emp}(\text{NAME SALARY DEPT}) \) may be a purely temporal relation. Hence, a legitimate assignment to the \text{SALARY} \ attribute is the temporal assignment \( \langle [0,10] \text{ 20K}, [11,20] \text{ 30K} \rangle \). However, this temporal assignment is equivalent to spatio-temporal assignment \( \langle \text{Space} \times [0,10] \text{ 20K}, \text{Space} \times [11,20] \text{ 30K} \rangle \) where \( \text{Space} \) is the entire spatial domain. Thus, in the following definitions of the relational operators we assume that the dimensions in the assignments of operand relations are the same (for if they are not they can be aligned by the system as described above).

Example 6. The query \textit{find the information about the wells, which are located in the region where soybean is grown and soil texture is of type clay loam, and for which the d/g concentration of atrazine exceeds the maximum allowable concentration} can be expressed as displayed
Note that this query involves all four relations in RelAgriDB. The restricted_to clause has three intersection operators with four operands. The first two are spatial, and the system will align them to spatio-temporal to match them with the third and fourth operands.

```sql
select w.*
restricted_to [[select c.CNAME restricted_to [[c.CNAME=soybean]] from crop c]]
\land [[select s.* restricted_to [[s.TEXTURE=clay loam]] from soil s]]
\land [[w.D/G-CONC > (select e.MAX from epa e where e.CHEM-NAME=atrazine)]]
\land [[w.CHEM-NAME=atrazine]]
from chems-in-wells w
```

The following is perhaps a simpler way of expressing the above query. The from clause consists of a mix of inputs: crop and soil are spatial, epa is ordinary, and chems-in-wells is spatio-temporal. They will all be aligned to be spatio-temporal. We also note that without optimization, this expression would be more expensive to execute than the one above, since it involves a cross product of four operands.

```sql
select w.*
restricted_to [[c.TEXTURE=clay loam]] \land [[c.CNAME=soybean]]
\land [[w.D/G-CONC > e.MAX]]
from crop c, soil s, epa e, chems-in-wells w
where w.CHEM-NAME = e.CHEM-NAME
    and w.CHEM-NAME = atrazine
```
3. ORDimDB: AN OBJECT-RELATIONAL DIMENSIONAL DATABASES

In this chapter we present our object-relational model ORDimDB for dimensional databases. We desire two forms of seamlessness in our model: first dimensional seamlessness and second value-oid seamlessness. Dimensional seamlessness requires that the queries in one dimensional database should be usable in a dimensional database with additional dimensions. Value-oid seamlessness means that queries in our value-oriented relational model should be usable in the oid-oriented object-relational model. We achieve the two forms of seamless without sacrificing the natural evolution of our model.

The dimensional seamlessness is achieved quite literally in our object-relational model at every level: we have seamlessness among dimensional domains, dimensional values, type hierarchy and path expressions. We resort to weak typing which disregards keys and dimension types of objects.

The value-oid seamlessness is built at the most primitive level, i.e. the attribute value level, in the model: an attribute value is a function from a dimensional domain; its values are either ordinary values or they are oids. Note that complex objects are beyond the scope of this thesis. In other words in our model an attribute value cannot be a complex object. This is the reason we have termed our model as object-relational, rather than object-oriented. An oid-valued attribute inherently carries more information than a value-valued attribute. An oid-valued attributes captures a hierarchy of many object fragments. In the relational model the counterpart of an oid-value would syntactically need several natural join operators. Therefore, the object-relational paradigm is necessarily more natural and richer than its relational counterpart. (A comparison of the two languages is given in Section 3.13.) This leads us to redefine the value-oid seamlessness at the level of a user. Imagine that we have a relational user of a relational database application. When the database is upgraded to the object-relational model, the given database can be mapped to object-relational model by creating a stand alone type for every relation. There is no hierarchy in the resulting object-relational database application. The seamlessness arises from the fact that an existing relational query will be a
an object-relational query without any modifications, and it will yield a result which is same as what the relational user would expect, except for oids. Clearly by a simple filtering of oids, the relational user can be given the illusion that it is interacting with a relational database application. Often when one migrates from relational to object-relational model a redesign of applications becomes desirable. This is not covered by value-oid seamlessness.

The relational algebraic operators are generalized taking type hierarchy into account. A computed object is naturally placed in the type hierarchy to maximally avail existing methods. As stated before, key is not part of the type hierarchy, but it is associated with an instance. Factoring key out of the type hierarchy has two advantages. One is that it provides compatible with the ordinary database, where changing a key of a relation (set object) would not change the relation (set object) and scheme (type). Another advantage is that an identity such as $O = I_K(I_K'(O))$, where $I_K(O)$ changes the key of set Object $O$ to $K$, would hold immediately in our model.

Our approach is conservative in the sense we do not want to loose the pragmatic benefits, mainly algebraic optimization, offered by the relational model. The backbone of algebraic optimization is a good set of identities. Example of an important identity is the associative law for cross product: $(e_1 \times e_2) \times e_3 = e_1 \times (e_2 \times e_3)$. This is one of many identities making algebraic optimization possible in a relational databases. This identity allows a syntactic form "from $e_1, e_2, e_3$" to be used in an SQL-like select statement which (conceptually) forms a cross product $e_1 \times e_2 \times e_3$ in a declarative manner. This identity does not necessarily hold in all object-oriented systems. We introduce several forms of equality in our model, with one of the equalities, called strong equality, as our most favorite one. The associative law of cross products hold for this form of equality, but not for stronger forms of equalities. We show that we can preserve all the relational identities [NG92] in the object-relational model. This sets up a good stepping stone for algebraic optimization in object-relational model.

For ease of reading, we will first present a personnel database and then introduce our terminology.
3.1. ORPersonnelDB: an object-relational database for personnel database

In this section we consider a personnel database. An entity-relationship diagram for the database is shown in Figure 4. The design of ORPersonnelDB is shown in Figure 5 and the instances of ORPersonnelDB is shown in Figure 6. As shown in Figure 5, there are two *is-a* relationships in ORPersonnelDB, namely *an employee is a person* and *a manager is an employee*. Furthermore, there are two oid-valued attributes: the attribute WORKS-IN in employee type is of dept type; and the attribute MANAGED-BY in dept is of manager type. Oids and type hierarchy are important features in object-oriented databases, as they allow considerable associative navigation without joins. In this chapter, we will see how oids and type hierarchy are modeled in dimensional databases.

3.2. Object identity

An object is identified by its *object identity (oid)*, which is independent of its internal values and invisible to the user. Theoretically an oid is generated only once and thus unique even across different database systems.

3.3. Dimensional elements

As in the relational model, we assume an underlying universal dimensional domain $P$. The user views it as a set of points in the dimensional space. We postulate that certain subsets of $P$, called *dimensional elements*, are of interest to users, and they are closed under union ($\cup$), intersection ($\cap$), subtraction ($-$) and complementation ($\neg$). Again, we do not make any assumptions about the constitution of $P$. For example, we treat the time dimension to be generic. We note that dimensional elements are the most primitive data type for dimensional domains in our model.
Figure 4. Entity relationship diagram for ORPersonnelDB.
type person
  NAME : string
  SSNO : string
  GENDER : string
  DOB : string

type employee is person
  SALARY : integer
  WORKS-IN : dept

type manager is employee

type dept
  DNAME : string
  MANAGED-BY : manager

(a) The type hierarchy of ORPersonnelDB.

(a) The pictorial representation of (a).

Figure 5. An object-relational design for ORPersonnelDB.
<table>
<thead>
<tr>
<th>OID</th>
<th>NAME</th>
<th>SSNO</th>
<th>GENDER</th>
<th>DOB</th>
</tr>
</thead>
<tbody>
<tr>
<td>p₁</td>
<td>John</td>
<td>121-20-2098</td>
<td>M</td>
<td>12/12/58</td>
</tr>
<tr>
<td>p₂</td>
<td>Tom</td>
<td>232-09-4382</td>
<td>M</td>
<td>02/12/45</td>
</tr>
<tr>
<td>p₃</td>
<td>Inga</td>
<td>232-12-3931</td>
<td>F</td>
<td>11/11/60</td>
</tr>
<tr>
<td>p₄</td>
<td>Leu</td>
<td>948-32-0973</td>
<td>F</td>
<td>03/09/61</td>
</tr>
<tr>
<td>p₅</td>
<td>Mary</td>
<td>293-09-9129</td>
<td>F</td>
<td>04/23/59</td>
</tr>
<tr>
<td>p₆</td>
<td>Peter</td>
<td>278-04-9739</td>
<td>M</td>
<td>01/20/66</td>
</tr>
</tbody>
</table>

(a) person: set object of person type

<table>
<thead>
<tr>
<th>OID</th>
<th>OID:person</th>
<th>SALARY</th>
<th>WORKS-IN</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>[50,54] 20K</td>
<td>[45,60] d₃</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[35,60] 25K</td>
<td></td>
</tr>
<tr>
<td>e₂</td>
<td>[18,21] ∪ p₂</td>
<td>[18,20] 20K</td>
<td>[18,21] d₁</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[41,51] 30K</td>
<td>[41,51] d₂</td>
</tr>
<tr>
<td>e₃</td>
<td>[71,NOW] p₃</td>
<td>[71,NOW] 25K</td>
<td>[71,NOW] d₂</td>
</tr>
<tr>
<td>e₄</td>
<td>[31,NOW] p₄</td>
<td>[31,NOW] 15K</td>
<td>[31,NOW] d₁</td>
</tr>
<tr>
<td>e₅</td>
<td>[41,44] ∪ p₅</td>
<td>[41,44] ∪ 25K</td>
<td>[41,44] ∪ d₂</td>
</tr>
</tbody>
</table>

(b) emp: set object of employee type

<table>
<thead>
<tr>
<th>OID</th>
<th>OID:employee</th>
</tr>
</thead>
<tbody>
<tr>
<td>m₁</td>
<td>[11,60] e₁</td>
</tr>
<tr>
<td>m₂</td>
<td>[41,47] e₂</td>
</tr>
<tr>
<td>m₃</td>
<td>[71,NOW] e₃</td>
</tr>
<tr>
<td>m₄</td>
<td>[45,49] e₄</td>
</tr>
</tbody>
</table>

(c) mgr: set object of manager type

<table>
<thead>
<tr>
<th>OID</th>
<th>DNAME</th>
<th>MANAGED-BY</th>
</tr>
</thead>
<tbody>
<tr>
<td>d₁</td>
<td>[11,49] Toys</td>
<td>11,44 m₁</td>
</tr>
<tr>
<td></td>
<td></td>
<td>45,49 m₄</td>
</tr>
<tr>
<td>d₂</td>
<td>[41,47] ∪ [71,NOW] Clothing</td>
<td>41,47 ∪ m₂</td>
</tr>
<tr>
<td></td>
<td></td>
<td>71,NOW m₃</td>
</tr>
<tr>
<td>d₃</td>
<td>[45,60] Shoes</td>
<td>45,60 m₁</td>
</tr>
</tbody>
</table>

(d) management: set object of dept type

Figure 6. Instances in ORPersonnelDB.
3.4. Attributes and object fragments

We define a value-function to be a function from a dimensional element into a fixed value domain (e.g. domain of SALARY), and an oid-function to be a function from a dimensional element into a oid-domain of fixed type. For example, in the employee type of Figure 5, the attribute WORKS-IN is of oid-type dept. Unlike traditional object-oriented databases, objects in dimensional databases are interwoven. An example of this interleaving can be seen in the crop relation in Figure 2(b) and crop:TILLAGE relation in Figure 3. In the relational dimensional model this problem is solved by allowing attribute values to be value-functions. This is not adequate in the object-relational dimensional model because of the type hierarchy. Therefore, in the object-relational model ORDimDB, we define an attribute value to be a value-function or an oid-function. Note that the value-functions and oid-functions are the most primitive type of values in our model.

Suppose o is a tuple object and μ a dimensional domain, then \( o \downarrow \mu \), denoted \( \mu_o \), is called an object fragment. The domain of object fragment \( \mu_o \), denoted \( \llbracket \mu_o \rrbracket \) is \( \mu \cap \llbracket o \rrbracket \). In an object fragment \( \mu_o \), nothing is known about object \( o \) from outside of \( \mu \). An oid-function assembles fragments of any number of objects.

Example 7. In Figure 6(d), the oid function \( \langle [11,44] m_1, [45,49] m_4 \rangle \) is a value of the MANAGER attribute, where \( m_1 \) and \( m_4 \) are tuple objects of manager type. Here \( [11,44] m_1 \) and \( [45,49] m_4 \) are examples of oid fragments.

Defining attribute values as functions allows us to capture a real world object as an unfragmented logical object. In addition the interleaving of objects is captured through object fragments, reducing the data redundancy at the logical level. The incorporation of object fragments also simplifies queries considerably. This is because associative navigation can be done through oids without explicit join operations. To support such navigation, we use path expressions, which will be introduced in Section 3.6.
3.5. Type hierarchy

One of our objectives is to obtain a reasonable set of algebraic identities which make our model amenable to algebraic optimization. We introduce a weak type hierarchy for set objects. Our types neither capture keys nor the dimensionality of data which are pushed to the instance levels.

A type $T$ is a pair $(\text{sup}, \text{local})$ where $\text{sup}$ is a set of types, called immediate supertypes of $T$ and $\text{local}$ is a set of attributes called local attributes of $T$. For example, the employee type of Figure 5 is $\langle \{\text{person}\}, \{\text{SALARY, WORKS-IN}\} \rangle$. A more interesting example of a type hierarchy is given in Figure 7.

![Type Hierarchy Diagram]

Figure 7. An interesting type hierarchy.

It is possible for a type to have only one supertype and no local attributes. Although such a type may seem redundant, but this is not always so, because it may have some methods in addition to its supertype. It is customary in object-oriented models to postulate a common root, usually called $\text{OBJECT}$, for the type hierarchy. In our terminology, this common root would be the type $\langle \varnothing, \varnothing \rangle$. In ORDimDB, a common root would not contribute anything to the modeling power; therefore it is not postulated.
3.5.1. Attributes of a type

The set of attributes of a type consists of both the local attributes and the attributes inherited from its supertypes. All the attribute names must be distinct. Formally, the set of all attributes of type $T$, denoted by $\text{Attrs}(T)$, is $\text{local}(T) \cup (\cup_{T' \in \text{sup}(T)} \text{Attrs}(T'))$. For example, the attributes of manager type of Figure 5, $\text{Attrs}($employee$)$, is $\{\text{NAME}, \text{SSNO}, \text{GENDER}, \text{DOB}, \text{SALARY}, \text{WORKS-IN}\}$.

3.6. Path expressions

With the incorporation of oids, queries become more powerful because associative navigation can be done through a sequence of oid fragments without explicit join operations. Such navigation can be specified as a path expression. Let the type of attribute $A_i$ be $T_i$ for $1 \leq i \leq N-1$. A path expression $e$ to an attribute $A_N$ originating from object variable $o$ of type $T_0$ is of the form $o.A_1.A_2...A_{N-1}.A_N$, where $A_i$ is in $\text{Attrs}(T_{i-1})$ for $1 \leq i \leq N$. To arrive at the value of $A_N$, for each $p \in [o.A_1]$, $o.A_1(p)$ is substituted by $A_1(p).A_2(p)...A_{N-1}(p).A_N(p)$, where $A_i(p)$ is an oid for $1 \leq i \leq N-1$. Note that an attribute in a relational query is a degenerate path expression with $N = 1$. We also say that $A_N$ is the attribute of $e$, and its target type is $T_{N-1}$.

For example, consider instances of ORPersonnelDB of Figure 6. The phrase the name of manager of $e$ can be expressed by the path expression $e.$WORKS-IN.DNAME. The attribute of the path expression is DNAME and its target type is dept type. If $e$ is the object $e_1$ in Figure 6, then the value of the path expression $e.$WORKS-IN.DNAME is $([11,44]$ Toys, $[45,60]$ Shoes$)$.

3.7. Tuple objects and keyed set objects

A tuple object of a type $T$ is a tuple over the set of attributes $\text{Attrs}(T)$ together with an oid. If $o_1$ and $o_2$ are tuple objects over the same set of attributes $R$ (not unnecessarily having the same type hierarchy), then we say that $o_1$ and $o_2$ are coincident at $p$, if $o_1.A(p) = o_2.A(p)$ for every attribute $A$ in $R$. Note that the notion of being coincident captures equality between object segments without having to force the object fragments to be of the same type. This def-
inition forms the basis for definition of several forms of equalities for set objects in our object-relational model.

A set object \( O \) of type \( T \), with key \( K \subseteq \text{Attrs}(T) \), is a finite set of tuple objects of type \( T \), such that no key attribute value of an object changes from one dimensional point to another, and no two tuple objects agree on all their key attributes. Suppose \( O \) is a set object of type \( T \). We denote the key of \( O \) by \( \text{key}(O) \). The notions \( \text{Attr}(O) \), \( \text{local}(O) \) and \( \text{sup}(O) \) are induced by the type \( T \). We note that a key of a set object does not necessarily consist of local attributes. For example, the set object \( \text{emp} \) in Figure 6(b) has the key NAME, which is not a local attribute of its type employee.

Since key is factored out of the type hierarchy, changing the key of a set object would not change its type. This yields compatibility with a classical database for ordinary data, where changing a key of a set object would not change the set object itself. In other words, the type of a set object \( O \) with key \( K \) is the same as the type of \( I_{K'}(O) \), where \( I_{K'} \) is the key restructuring operator, which will be introduced in Section 3.9.2. As a result, the identities such as \( O = I_K(I_{K'}(O)) \), where \( K \) is the key of \( O \), would readily hold in our model.

Homogeneity is assumed in our model: \( \llbracket o.A \rrbracket \) is the same for every attribute \( A \) of an object \( o \). The dimensional domain of a tuple object \( o \), denoted \( \llbracket o \rrbracket \), is \( \llbracket o.A \rrbracket \), for any attribute \( A \) of \( o \). The dimensional domain of a set object \( O \), denoted \( \llbracket O \rrbracket \), is the union of \( \llbracket o \rrbracket \) of tuple objects \( o \) in \( O \).

In our model, we have made the assumption that for any tuple objects \( o_1 \) and \( o_2 \), \( \llbracket o_1.A = o_2 \rrbracket \subseteq \llbracket o_2 \rrbracket \), where \( A \) is an attribute of \( o_1 \). In other words, if object \( o_1 \) refers to another object \( o_2 \) at point \( p \), then the attributes of object \( o_2 \) must be defined at \( p \). For example, in ORPersonnelDB of Figure 6, \( \llbracket d_1.\text{MANAGED-BY} = m_1 \rrbracket = [11,44] \subseteq [11,60] = \llbracket m_1 \rrbracket \), where \( d_1 \) is a dept object and \( o_2 \) is a manager object. The assumption is to ensure that the value of a path expression is always well defined.
3.8. Equality of set objects

Equality plays an important role in databases. Khoshafian and Copeland [KC86] define three levels of equality for objects: identical, shallow equal and deep equal. Two objects are identical if they are the same values or oids; shallow equal if they contain the same values and oids; deep equal if they have the same values when oids are recursively followed. These notions of equality are too stratified for our use. In our model, in addition to key, we want our notions of equality to take type hierarchy into account. Now we introduce our notions of equality for set objects.

Suppose O₁ and O₂ are two set objects over a dimensional space P, such that Attrs(O₁) = Attrs(O₂). Then

- O₁ and O₂ are weakly equal, written O₁ ≅₁ O₂, if for all p in P, O₁(p) and O₂(p) are coincident as sets of tuple objects.
- O₁ and O₂ are strongly equal, written O₁ =ₚ O₂, if they are weakly equal and have the same key.
- O₁ and O₂ are type equal, written O₁ =ₜ O₂, if they are weakly equal and their types are equal.
- O₁ and O₂ are strongly type equal, written O₁ =ₜ O₂, if they are strongly equal and type equal.

In Figure 8 we show the four notions of equality. The weakest form of equality is the weak equality, and it implies the other three forms of equalities. Type equality requires the objects to be strictly of the same type. Under this notion of equality, even the cross product fails to be associative. In ordinary as well as parametric databases this becomes a hindrance to algebraic optimization, and without any visible gain. However, we do consider type equality to contrast a database system with a general purpose programming system from our point of view. Our strongest form of equality combines the type and strong equalities. This suffers from the shortcomings of the type equality.
Our favorite equality is the strong equality, which we simply term as equality. This form of equality recognizes object fragments as a legitimate basis for determining if two objects have the same information. We believe that this form of equality would lead to database languages which are closer to natural languages. In addition it leads to a set of natural identities, forming a strong foundation for algebraic optimization.

For ordinary data, attribute values are degenerate forms of value-functions and oid-functions. Consequently, for such data strong equality degenerates to weak equality, and strong-type equality degenerates to type equality.

3.9. The query language ORDimSQL

ORDimSQL consists of *dimensional expressions, boolean expressions* and *set expressions*. These expressions are mutually recursive, and evaluate to dimensional elements, bool-
ean values TRUE and FALSE, and set objects, respectively.

3.9.1. Dimensional expressions and boolean expressions

Dimensional expressions are $\mu, [A_1], [A_1 \theta A_2], [o]$ and $[e]$, where $\mu$, $A_i$, $o$ and $e$ are constant dimensional element, attribute, a tuple object and set expression, respectively. If $\mu$ and $\nu$ are dimensional expressions, then so are $\mu \cup \nu$, $\mu \cap \nu$, $\mu - \nu$ and $-\mu$.

If $\mu$ and $\nu$ are dimensional expressions, then $\mu \subseteq \nu$ is a boolean expression. If $f$ and $g$ are boolean expressions then so are $f \vee g$, $f \wedge g$ and $\neg f$.

3.9.2. Set expressions

Set expressions are the syntactic counterparts of set objects. We have seven operators for set expressions: union, difference, intersection, selection, cross product, restructuring and projection. In this thesis, however, we do not consider natural join and full form of cross product. The semantics of every operator except projection is the same as the relational counterpart in Section 2.6, except that an oid created for each new tuple objects. Thus, we only need to define the new type (hierarchy) for the resulting set objects generated by these operations. (For ease of reading, the resulting key is restated from Section 2.6 here.) Projection will be covered in the next section.

- Suppose $O_1$ and $O_2$ are set expressions of type $T_1$ and $T_2$ with the same key $K$, and same attributes $\text{Attrs}(O_1) = \text{Attrs}(O_2)$. Then $O$ union $O'$, $O$ difference $O'$, $O$ intersection $O'$ are set expressions of type $T_1$ if $T_1 = T_2$, and $(\emptyset, \text{Attrs}(T_1))$ otherwise. The key of these expressions is $K$.

- Suppose $O$ is a set expression of type $T$ with key $K$, $f$ is a boolean expression and $\mu$ is a dimensional expression. Then the selection operation $\sigma(O; f; \mu)$ is also a set expression of type $T$ with key $K$.

- Suppose $O_1$ and $O_2$ are set expressions of type $T_1$ and $T_2$ where $\text{Attrs}(T_1) \neq \text{Attrs}(T_2)$ and with key $K_1$ and $K_2$, respectively. Then $O_1 \times O_2$ is a set expression of type $(\{T_1, T_2\}, \emptyset)$
with key $K_1 \cup K_2$.

- Suppose $O$ is a set expression of type $T$ with key $K$, and $K' \subseteq \text{Attrs}(T)$. Then the restructuring operation $\Pi_{K'}(O)$ is also a set expression of type $T$ but with key $K'$.

### 3.9.3. The projection operator

In an object-oriented database, methods are defined for a type. Like attributes, methods of a type are both locally defined as well as inherited from its supertypes. When a new type is computed, we need to place the new type into the existing type hierarchy. In order to reap the maximum possible benefit of existing methods for a type hierarchy, our goal is to preserve as much of the existing type hierarchy as possible. Yet the placement of a new type should be comprehensible to the user, and at the same time we hope that it would provide backward compatibility to the relational model. The projection operator is extended from the one in relational model to incorporate path expressions and type hierarchy. Because of path expressions, the attributes to be projected may not be a subset of the attributes of the input type.

We could always define the resulting type $T'$ as a new type in the existing hierarchy. However, in some cases this is not desirable, because an existing type in the hierarchy can do the job. A simple example of this is when all attributes of a set object of a stand alone type $T$ are projected, the resulting type should be $T$.

Suppose $O$ is a set expression of type $T$ with key $K$, and $L$ is a set of path expressions with attributes originating from $O$ and the projection $\Pi_L(O)$ is a set expression of type $T'$. To define $T'$, we introduce $\text{sup}(T')$ and $\text{local}(T')$ in two stages. In the first stage we compute a new type. When necessary, in the second stage we collapse the new type onto an existing one.

Let $\text{Attrs}(L)$ denotes the attributes appearing in $L$ and $\text{targetTypes}(L)$ denotes the target types appearing in $L$. It is assumed that types in $\text{targetTypes}(L)$ are mutually independent in the sense that the attributes among them are distinct. It is clear that a type we wish to salvage is in or a supertype of a type in $\text{targetTypes}(L)$. Therefore, we say that a type $T''$ is preserving in $L$ if $\text{Attrs}(T'') \subseteq \text{Attrs}(L)$, and $T''$ is in or a supertype of a type in $\text{targetType}(L)$.
In the first stage, we define $\text{sup}(T') = \{T'' : T'' \text{ is preservable in } L \text{ and } T'' \text{ does not have any subtype } T''' \text{ which is also preservable in } L\}$, and $\text{local}(T') = \text{Attrs}(L) - \bigcup_{T'' \in \text{sup}(T')} \text{Attrs}(T'')$.

In the second stage we examine if the new type $T'$ can be collapsed onto an existing one. This situation arises when $\text{local}(T')$ is empty and $\text{sup}(T')$ is a singleton, say $\{T''\}$. We substitute $T''$ for $T'$ (abandoning the existing $T'$).

Finally, the key of $O \downarrow\text{L}$ is defined to be $K$, the key of $O$, if $K$ is a subset of $L$. Otherwise the key is defined to be $\text{Attrs}(L)$.

**Example 8.** Consider the type hierarchy in Figure 7. T's and A's represent types and attributes. All attributes are of primitive types (i.e., real, string, ...) except that $A_{14}$ is of type $T_{11}$ and $A_{13}$ is of type $T_{15}$. Suppose we are given two projections $Q_1$ and $Q_2$ on a set object $O$ of type $T_{14}$ with key $A_{15}$ as below. The new types, $T_{16}$ and $T_{17}$, formed by $Q_1$ and $Q_2$ are shown in Figure 9(a) and Figure 9(b), respectively.

- $Q_1$: $\Pi_{A_1, A_2, A_{11}, A_{14}}(O)$
  
  To help understand the formation of the new type, $T_{16} = \langle \text{sup}, \text{local} \rangle$, in Figure 9(a), the parameters in the definition of projection are shown below.
  
  $\text{targetTypes} = \{T_{14}\}$
  
  $\text{preservable types} = \{T_1, T_2, T_{10}\}$
  
  $\text{sup} = \{T_{10}\}$
  
  $\text{local} = \{A_{14}\}$
  
- $Q_2$: $\Pi_{A_1, A_2, A_{11}, A_{15}, A_{14}, A_5, A_{14}, A_6, A_{14}, A_{12}, A_{14}, A_{13}, A_9, A_{14}, A_{13}, A_{10}, A_{14}, A_{13}, A_{16}}(O)$,
  
  or equivalently, $\Pi_{A_1, A_2, A_{11}, A_{15}, A_{14}, \{A_5, A_6, A_{12}\}, A_{14}, A_{13}, \{A_9, A_{10}, A_{16}\}}(O)$
  
  To help understand the formation of the new type of $Q_2$, $T_{17} = \langle \text{sup}, \text{local} \rangle$, in Figure 9(b), the parameters in the definition of projection are shown below.
  
  $\text{targetTypes} = \{T_{14}, T_{11}, T_{15}\}$
  
  $\text{preservable types} = \{T_1, T_2, T_4, T_5, T_8, T_9, T_{10}, T_{13}\}$
  
  $\text{sup} = \{T_4, T_5, T_{10}, T_{13}\}$
  
  $\text{local} = \{A_{12}, A_{15}, A_{16}\}$
Figure 9. A type hierarchy illustrating the projection operation of Example 8.
3.9.4. The select statement

In ORDimSQL the select statement has the following form.

```sql
select SelectList :Key
   restricted_to DimExp
from FromList
where BoolExp
```

Here SelectList is a list of path expressions; DimExp is a dimensional expression; FromList is a list of set objects, where each item is of the form O or O:K where O:K is the set object O restructured so that K is the key; and BoolExp is a boolean expression. The semantics of the select is as follows. First a virtual literal cross product of set objects in FromList is formed. For each tuple object o in the cross product, the condition BoolExp is verified. If object does not satisfy BoolExp, it is rejected. If the object satisfies BoolExp, the attributes in SelectList are retrieved and its dimensional domain is restricted to DimExp. If the resulting object is empty, it is rejected; otherwise it is retrieved. If necessary, the dimensional domain of the retrieved object is further contracted so that a retrieved object is homogeneous. Note that the value of DimExp depends upon the substituted object.

The key of the resulting set object can be explicitly provided by the construct : Key in the select clause. If the key is not explicitly provided, a default key is determined. We say that a set object O is represented in the SelectList if the SelectList contains an attribute of O. If all the key attributes of such a set object O are in the SelectList, these key attributes become part of the default key; otherwise all the attributes originated from O appearing in the SelectList become part of the default key. A complete default key is constructed in this manner from all the set objects represented in the SelectList. Furthermore, the type of the resulting set object is determined as follows. Suppose O₁, O₂, ..., Oₙ are in the FromList. Then the type of the resulting set object is defined to be the type of \(\prod_{\text{SelectList}} (O₁ \times O₂ \times \cdots \times Oₙ)\).

3.10. ORAgriDB: an object-relational database for AgriDB application

In this Section, we show an object-relational database, ORAgriDB, for AgriDB application of Figure 1. The design of ORAgriDB is shown in Figure 10, and the instances in ORA-
griDB is shown in Figure 11. As shown in Figure 10, ORAgriDB includes five types and there is one set object for each type. The five set objects include two ordinary (space/time independent) set objects, two spatial set objects and one spatio-temporal set. The attributes forming the key for each set object are underlined (keys are introduced later). At first glance ORAgriDB may seem more complex than their relational counterpart RelAgriDB: ORAgriDB contains five set objects, whereas RelAgriDB contains only four relations. We note that the chemical entities appear in the set objects epas and well-chems. Following the object-oriented philosophy, it is more natural to let both instances emanate from a common source by using a type hierarchy. Therefore, we have created a chemical type which acts as a supertype of epa and chems-in-wells types. The data items appearing in ORAgriDB are the same as those in RelAgriDB, except that two more chemicals have been incorporated in the instance of chemical type to illustrate the role of chemical type.

3.11. Dimension alignment

Dimension alignment is an interesting feature in our query language ORDimSQL. Such alignment becomes necessary when evaluating expressions using a mix of dimensional types. In our model, the alignment is automatically done by the system by padding the whole spaces in the missing dimensions of operands. For example, let reg and $\mu$ be a spatial element and a temporal element respectively. The expression $\text{reg} \cap \mu$ will be treated by the system as $(\text{reg} \times \text{Time}) \cap (\text{Space} \times \mu)$, where $\text{Time}$ is the universe of time, and $\text{Space}$ is the universe of space. Dimension alignment leads to true seamless in users view of data. It also allows query reuse. For example existing temporal and spatial queries can be mixed and embedded into spatio-temporal queries without any syntactic changes.

**Example 9.** The query *find the information about the wells, which are located in the region where soybean is grown and soil texture is of type clay loam, and for which the d/g concentration of atrazine exceeds the maximum allowable concentration* can be expressed as follows.

```sql
select c
restricted_to [[select p.CNAME restricted_to [[p.CNAME = soybean]] from crops p]]```
(a) The type hierarchy of ORAgriDB

```
type chemical
    CHEM-NAME : string

type epa is chemical
    MAX : real
    MIN : real

type chems-in-wells is chemical
    U/G-CONC : real
    D/G-CONC : real

type soil
    TEXTURE : string

type crop
    CNAME : string
```

(b) The pictorial representation of (a)

```
chemical          soil          crop
| CHEM-NAME | TEXTURE | CNAME |

epa              | chemicals-in-wells
| MAX           | U/G-CONC | D/G-CONC |
```

Figure 10. ORAgriDB: the object-oriented design for AgriDB of Figure 1.
### (a) Chemicals: set object of chemical type

<table>
<thead>
<tr>
<th>OID</th>
<th>CHEM-NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>c₁</td>
<td>Atrazine</td>
</tr>
<tr>
<td>c₂</td>
<td>Simazine</td>
</tr>
<tr>
<td>c₃</td>
<td>Iodine</td>
</tr>
<tr>
<td>c₄</td>
<td>Lead</td>
</tr>
</tbody>
</table>

### (b) EPAs: set object of epa type

<table>
<thead>
<tr>
<th>OID</th>
<th>OIDchem</th>
<th>MAX</th>
<th>MIN</th>
</tr>
</thead>
<tbody>
<tr>
<td>d₁</td>
<td>c₁</td>
<td>3</td>
<td>0.05</td>
</tr>
<tr>
<td>d₂</td>
<td>c₂</td>
<td>35</td>
<td>0.05</td>
</tr>
<tr>
<td>d₃</td>
<td>c₃</td>
<td>20</td>
<td>0.05</td>
</tr>
</tbody>
</table>

### (c) Well-Chems: set object of chems-in-wells type

<table>
<thead>
<tr>
<th>OID</th>
<th>OIDchem</th>
<th>U/G-CONC</th>
<th>D/G-CONC</th>
</tr>
</thead>
<tbody>
<tr>
<td>e₁</td>
<td>p₁×[0,NOW] ∪ p₂×[0,NOW]</td>
<td>c₁</td>
<td></td>
</tr>
<tr>
<td></td>
<td>p₂×[0,5]</td>
<td>1.5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>p₂×[6,NOW]</td>
<td>3.5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>p₁×[0,NOW]</td>
<td>0.9</td>
<td></td>
</tr>
<tr>
<td></td>
<td>p₂×[10]</td>
<td>1.4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>p₂×[11,NOW]</td>
<td>2.9</td>
<td></td>
</tr>
<tr>
<td>e₂</td>
<td>p₁×[0,NOW]</td>
<td>c₂</td>
<td></td>
</tr>
<tr>
<td></td>
<td>p₁×[0,9]</td>
<td>10.0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>p₁×[10,NOW]</td>
<td>12.2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>p₁×[0,NOW]</td>
<td>9.2</td>
<td></td>
</tr>
</tbody>
</table>

### (d) Soils: set object of soil type

<table>
<thead>
<tr>
<th>OID</th>
<th>TEXTURE</th>
</tr>
</thead>
<tbody>
<tr>
<td>s₁</td>
<td>sreg₁ ∪ sreg₅</td>
</tr>
<tr>
<td>s₂</td>
<td>sreg₂ ∪ sreg₃</td>
</tr>
<tr>
<td>s₃</td>
<td>sreg₄</td>
</tr>
<tr>
<td>s₄</td>
<td>sreg₆</td>
</tr>
</tbody>
</table>

### (e) Crops: set object of crop type

<table>
<thead>
<tr>
<th>OID</th>
<th>CNAME</th>
<th>TILLAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>r₁</td>
<td>creg₁ ∪ creg₅</td>
<td>corn</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r₂</td>
<td>creg₂ ∪ creg₃</td>
<td>wheat</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r₃</td>
<td>creg₄</td>
<td>soybean</td>
</tr>
</tbody>
</table>

Figure 11. Instances in ORAgriDB.
\[ \cap \{ \text{select } s \text{ restricted_to } \{ s.\text{TEXTURE} = \text{clay loam} \} \text{ from soils } s \} \cap \{ \text{c.D/G-CONC} > (\text{select } e.\text{MAX} \text{ from epas } e \text{ where } e.\text{CHEM-NAME} = \text{atrazine}) \} \cap \{ \text{c.CHEM-NAME} = \text{atrazine} \} \text{ from well-chems } c \]

In the above query, the restricted_to clause has three intersection operators with four operands. The first two operands are spatial, and the remaining two are spatio-temporal. When evaluating the system will align the first two operands to spatio-temporal to match them with the third and the fourth operands.

Furthermore, the concept of dimension alignment is also applied in path expressions. It treats ordinary, temporal, spatial and spatial-temporal attributes uniformly without any distinction in the database. When following a tuple object o of a dimensional type (e.g. temporal) from an attribute of a more complex dimensional type (e.g. spatio-temporal), the attributes of o will be padded with the whole spaces of the missing dimensions (e.g. spatial). An illustration is given in Example 10.

**Example 10.** Suppose we add an attribute \text{PRODUCT} to the manager type in ORPersonnelDB of Figure 5, where \text{PRODUCT} is of crop type in ORAgriDB of Figure 10. From Figure 11(e), we have \text{r1.CNAME} = \langle \text{creg1} \cup \text{creg2 corn} \rangle, which is an spatial attribute. Let \text{m1.PRODUCT} to be the spatio-temporal attribute \langle [11,60] \times \text{creg1 r1} \rangle. When evaluating the path expression \text{m1.PRODUCT.CNAME}, the attribute \text{r1.CNAME} is padded with the whole time domain \text{Time} and becomes \langle \text{Time} \times \text{creg1} \cup \text{creg2 corn} \rangle. Thus the path expression is evaluated to be \langle [11,60] \times \text{creg1 corn} \rangle.

### 3.12. Upward compatibility for the classical user

Under our approach, the upward compatibility for the classical user is along two orthogonal paths: from ordinary data to dimensional data, and from relational model to object-relational model. This is useful for industry as well as for its user community. The industry and users may choose to take one upgrading path now and the other path at a later time. Users do not have to abandon their investment in application programs when they upgrade their database developed under our framework.
Our object-relational framework can be directly used by a relational user. A relational user is one who interacts with a relational database and asks only relational queries. A relational database is embedded into an object-relational database in a straightforward manner by converting each relation to a set object of a stand-alone type. A relational user is not aware of path expressions. When a relational user stores computed queries, the system imposes a type hierarchy. However, this type hierarchy is devoid of any path expressions. Therefore, the system is unable to put additional semantics on such data over and above the relational model. As a result, a relational user does not notice in any way that he or she is interacting with a relational system rather than an object-relational system.

In [GN93], it has been shown that the transition from SQL to RelDimSQL is seamless. Keeping this goal in mind, the transition from SQL to ORDimSQL is also seamless by having ORDimSQL a generalization of RelDimSQL. In other words, a RelDimSQL query is also ORDimSQL query, with no transformation is needed. When a user submits a SQL query to our object-relational system, the SQL query is transformed to a ORDimSQL query using the rules in [GN93]. The query is then executed and the result is returned to the user.

3.13. Comparison of ORDimDB with other works

In this Section we compare ORDimDB with other models closely related to ours. First, we compare our object-relational model ORDimDB with our own relational model RelDimDB presented in Chapter 2. (RelDimDB is derived from [GC93b].) This comparison shows some pronounced differences between our two models, particularly due to type inheritance and incorporation of oids in ORDimDB. Then we compare our model ORDimDB with the temporal extension of OODAPLEX of [WD92, DW92] and argue that our query language ORDimSQL is higher level and more user friendly than the query language in OODAPLEX.

To make the above comparisons, it is enough to consider a temporal database. Our remarks are about the inherent philosophy behind these models. Temporal data is much simpler than spatial data, and our remarks only become amplified if one migrates to spatial or spatio-temporal data. We will use ORPersonnelDB as our benchmark example.
3.13.1. Comparison of ORDimDB with relational model RelDimDB

Now we compare our object-relational model ORDimDB with our relational model RelDimDB given in Chapter 2. To allow us to make our comparison as clear as possible, we give a relational database scheme which is syntactically closest to the design in Figure 6. We include four relational schemes as shown in Figure 12.

```
person(NAME, SSNO, GENDER, DOB)
emp(NAME, SALARY, WORKS-IN)
mgr(NAME)
management(DEPT, MANAGER)
```

Figure 12. A relational design for personnel database.

In Figure 5 and Figure 12, we have two designs for a personnel database: first for the object-relational model ORDimDB of this thesis, and second for the relational model [GC93b]. Although, these designs capture the same information, because of type inheritance in ORDimDB, the first also captures the complex semantics, an employee is a person and a manager is an employee. Now we can compare the two models at the query level. To show the subtle difference between the object-relational and relational nature of the two models, we give the following example.

**Example 11.** Consider the query *give full details of managers of John.* In our object-relational model ORDimDB, the above query is expressed as follows. The attribute m.MANAGED-BY, in the select clause, represents one of possibly many managers of John and it is of manager type. Therefore, the query retrieves the NAME, SSNO, GENDER, DOB, SALARY and WORKS-IN values of John's manager(s). Note that due to the restricted_to clause, each man-
The following is a relational expression which is syntactically similar to the one above. However, it retrieves a result which is quite different from the one above. The only information the following query gives is the **MANAGER** attribute value from the management relation.

```
select m.MANAGER
restricted_to [[e.WORKS-IN = m.DEPT]]
from emp e, management m
where e.NAME = "John"
```

The relation scheme retrieved by RelDimSQL query

To obtain the same result as in the object-relational case, for the relational model, the query has to be expressed as follows.

```
select p.*, e2.SALARY, e2.WORKS-IN
restricted_to [[e1.WORKS-IN = m.DEPT]] \(\cap\) [[m.MANAGER = e2.NAME]]
\(\cap\) [[e2.NAME = p.NAME]]
from emp e1 e2, management m, person p
where e1.NAME = "John"
```

### 3.13.2. Comparison with the temporal extension of OODAPLEX

We compare our work to [WD92, DW92]. In [WD92, DW92] generic time type is point type, while in our model it is temporal element type. Also, they use primitives = and ≤ among instants, while we use the primitives = and ⊆ among temporal elements. Our primitives are higher order leading to simpler queries. Moreover, they readily extend from temporal databases to dimensional databases. Now we give two examples to illustrate our point.

**Example 12.** Consider the following query taken from [WD92, DW92]: *list all employees who have worked for department which has ever been headed by a manager named John.*

The query is expressed as follows in the two languages.
ORDimSQL: select e.NAME
from emp e, management m
where [e.WORKS-IN = m] ≠ ∅
and [m.MANAGED-BY.NAME = "John"] ≠ ∅

OODAPLEX: for each e in extent(employee) where
for some d in extent(dept)
for some t₁ in extent(time)
WORKS-IN(e)(t₁) = d and
MANAGED-BY(d)(t₁) = "John"
print(NAME(e))
end

In [DW93], OODAPLEX is cast into a SQL-like syntax, even though a formal transformation is not given. To illustrate this change, we cast the above OODAPLEX query into the following SQL-like syntax. The complexity of the query, however, seems to remain the same.

OODAPLEX: select NAME(e)
from extent(employee) e
where exists ( select d
from extent(dept) d, extent(time) t₁
where WORKS-IN(e)(t₁) = d and
MANAGED-BY(d)(t₁) = "John"
)

Example 13. The query give information about managers who were managers at least during [11,50] is expressed as follows in the two languages. Note that our query makes use of the restructuring function. The construct :MANAGER changes the key of management to MANAGER and r is an object pointer for their result.

ORDimSQL: select r.MANAGED-BY.*
from management: MANAGED-BY r
where [r.MANAGED-BY] ⊆ [11,50]

OODAPLEX: for each e in extent(employee) where
for all t in [11,50]
for some m₁ in extent(dept)
NAME(MANAGED-BY(m₁)(t)) = NAME(e)
for each t in lifespan(e)
print(time:t, name:NAME(e), dept:DNAME(WORKS-IN(e)))
end
end

Even though the papers [WD92, DW92] only treat temporal data, the notion of dimensionality is potentially available in OODAPLEX. In OODAPLEX the user thinks in terms of instants, where as in ORDimDB the user thinks in terms of dimensional elements. At the syntactic level, dimensional elements give rise to dimensional expressions. This makes ORDimDB much higher level than OODAPLEX. Dimensional expressions are natural, making user queries simple.
4. IDENTITIES AND ALGEBRAIC OPTIMIZATION

Unlike ordinary relational and object-oriented databases, optimization in dimensional databases is a relatively new problem. The problem is non-trivial due to the complex nature of various operators and dimensional domains. In this thesis our goal is to identify the distinct issues of optimization in dimensional databases, and propose a framework as a first step for further research. We will define the equality of objects, and identify certain identities in our object-relational model.

In temporal and spatial databases, much research on query processing has concentrated on physical implementation aspects such as access methods [SOL94, KKEW94, GS91, BK90, Gut84, Gun89, Sa84, Sa90, SK90, SR87, LL91]. Algebraic optimization, on the other hand, has received relatively little attention [DW92, oid fragments, NG92].

Nair and Gadia [NG92] proposed certain algebraic identities and an algebraic optimizer for relational temporal databases. The optimizer in [NG92] does not take various dimensional domains and dimension alignment into account. This may cause a problem since unlike in a database for ordinary data, an algebraic operation such as “selection” does not necessarily reduce the size of an operand in dimensional databases. This is due to the complex nature of dimensional domains. For example, representation of a spatial element which covers a small area of irregular shape may take considerably more disk space than a spatial element which covers the whole space. In other words, a smaller region does not necessary require less space to represent. Thus the optimizer needs more information in the algebraic manipulation process. Towards a solution of the problem, we introduce a dimensional framework, called size estimation, to estimate the output sizes from algebraic operations. This framework is only a first step in the optimization problem for dimensional databases and various issues within the framework still need to be further investigated.

Four forms of equalities among algebraic expressions introduced in Section 3.8 give rise to four different notions of identities. However, we only focus on strong identities and strong-type identities, as they both take key into account. In our model, the identities in [NG92] are
strong identities. Among these strong identities, only a subset of them, namely the strong-type identities, will hold if the type hierarchy is taken into account. In short, a strong-type identity is also a strong identity, but not the other way around. In this chapter, we will first identify which strong identities are strong-type identities. Then we will list certain principles and give an optimizer based on the framework.

4.1. Algebraic identities

As stated above, in this section we consider two kinds of algebraic identities: strong identities (=s) and strong-type identities (=st). A strong identity is when the two sides of the identity is strongly equal. As stated in previous chapter, strong equality is our preferred form of equality, we simply term it equality and denote it as "=" without the subscript s. A strong-type identity is when the two sides of the identity is strong-type equal. Basically, the difference between the two is that for strong identities the type hierarchy is not considered but for strong-type identities the type hierarchy is considered. Thus a strong-type identity is also a strong identity, but not the other way around.

Nair [NG92] identified certain algebraic identities for relational temporal databases. These identities are also valid for relational spatio-temporal databases, as we assume that only one value or one oid can exist at any given dimensional (spatio-temporal in our case) point for any tuple objects. In our object-relational model, the algebraic operators are generalizations of their relational counterparts, the identities in [NG92] will remain valid as strong identities in our model. Clearly, not every strong identity is a strong-type identity.

In this section, we will identify the strong identities which are also strong-type identities. Since the only difference between strong identities and strong-type identities is the equivalence of types, the proofs will only be on the type equivalence. Furthermore, when showing \( T(e_1) = T(e_2) \), where \( T(e_1) \) and \( T(e_2) \) are the types of the two sides of an identity, it suffices to show that \( \text{sup}(T(e_1)) = \text{sup}(T(e_2)) \). This is because \( \text{sup}(T(e_1)) = \text{sup}(T(e_2)) \) implies \( \text{local}(T(e_1)) = \text{local}(T(e_2)) \), when \( \text{Attrs}(T(e_1)) = \text{Attrs}(T(e_2)) \), which is always the case in these identities.
We will use the following notation in this section: $O_1, O_2, \cdots$ represent set objects of type $T_1, T_2, \cdots$ respectively; $O_i, O_i', O_i''$, \cdots represent set objects of type $T_i$ with the same key; $T(e)$ represents the type of the set expression $e$.

I 1 Commutativity of cross product

$O_1 \times O_2 \equiv_{st} O_2 \times O_1$

Proof: This is a strong-type identity since \( \sup(T(O_1 \times O_2)) = \{T_1, T_2\} = \sup(T(O_2 \times O_1)) \).

I 2 Associativity of cross-product

$O_1 \times (O_2 \times O_3) = (O_1 \times O_2) \times O_3$

Proof: The strong identity is not a strong-type identity since $T_1$ is in \( \sup(T(O_1 \times (O_2 \times O_3))) \) but not in \( \sup(T((O_1 \times O_2) \times O_3)) \).

I 3 Cascade of projections

$\Pi_{L_1}(\Pi_{L_2}(O_1)) =_{st} \Pi_{L_1'}(O_1)$, where $L_1' \subseteq L_2$ and $\text{Attrs}(L_1) = \text{Attrs}(L_1')$

Proof: The identity obviously holds if $L_1' = L_2$. Therefore, we assume $L_1' \subset L_2$, or $\text{Attrs}(L_1) \subset \text{Attrs}(L_2)$. Denote $P(L)$ to be the set of all preservable types in $L$.

From the definition of projection, it is clear that the resulting type of a projection $\Pi_L(O)$ is completely determined by $P(L)$. Therefore, to show the identity, it suffices to show that $P(L_1) = P(L_1')$.

Since $L_1' \subset L_2$, $P(L_1') \subseteq P(L_2)$. It is clear that if a type $T$ is in $P(L_2)$ and $\text{Attrs}(T) \subseteq \text{Attrs}(L_1')$ then $T$ must be in $P(L_1')$. It is also clear that the set of all the supertypes of $T(\Pi_{L_2}(O_1))$ is $P(L_2)$, unless collapsing occurs in $\Pi_{L_2}(O_1)$. If collapsing occurs, then the set of all the supertypes of $T(\Pi_{L_2}(O_1))$ is $P(L_2) - \{T(\Pi_{L_2}(O_1))\}$, which is still a superset of $P(L_1')$. This is because $\text{Attrs}(T(\Pi_{L_2}(O_1)))$ is not a subset of $\text{Attrs}(L_1')$, implying that $T(\Pi_{L_2}(O_1))$ is not in $P(L_1')$. (I)

$P(L_1) = \{T \mid \text{Attrs}(T) \subseteq \text{Attrs}(L_1) \text{ and } T \text{ is in or a supertype of a type in targetType}(L_1)\}$

$= \{T \mid \text{Attrs}(T) \subseteq \text{Attrs}(L_1) \text{ and } T \text{ is in or a supertype of a type in } \{T(\Pi_{L_2}(O_1))\}\}$

(since $\text{Attrs}(L_1) \subseteq \text{Attrs}(T(\Pi_{L_2}(O_1)))$)
\[ \{ T \mid \text{Attrs}(T) \subseteq \text{Attrs}(L_1) \text{ and } T \text{ is or a supertype of } T(\Pi_{L_2}(O_1)) \} \]

\[ = \{ T \mid \text{Attrs}(T) \subseteq \text{Attrs}(L_1') \text{ and } T \text{ is a supertype of } T(\Pi_{L_2}(O_1)) \} \]

(since \( \text{Attrs}(L_1') \subseteq \text{Attrs}(L_2) \))

\[ = \{ T \mid \text{Attrs}(T) \subseteq \text{Attrs}(L_1') \text{ and } T \text{ is in } P(L_2) \} \text{ or } \]

\[ \{ T \mid \text{Attrs}(T) \subseteq \text{Attrs}(L_1') \text{ and } T \text{ is in } P(L_2) - \{ T(\Pi_{L_2}(O_1)) \} \} \text{ (from (I))} \]

In either cases, we obtain \( P(L_1') \).

**I 4 Cascade of selections**

\[ 14.1 \quad \sigma(\sigma(O_1; f_1; ); f_2; ; ) =_{st} \sigma(O_1; f_1 \land f_2; ; ) \]

\[ 14.2 \quad \sigma(\sigma(O_1; f_1; ); f_2; ; ) =_{st} \sigma(\sigma(O_1; f_2; ; ); f_1; ; ) \]

\[ 14.3 \quad \sigma(\sigma(O_1; ; \mu_1); ; \mu_2) =_{st} \sigma(O_1; ; \mu_1 \land \mu_2) \]

\[ 14.4 \quad \sigma(\sigma(O_1; ; \mu_1); ; \mu_2) =_{st} \sigma(\sigma(O_1; ; \mu_2); ; \mu_1) \]

\[ 14.5 \quad \sigma(\sigma(O_1; f; ; ); ; \mu) =_{st} \sigma(O_1; f; ; \mu) \]

**Proof:** Here we only prove 14.1 and the proofs for others are similar. Since the type of the resulting set object from selection is the same as that of the input set, \( T(\sigma(\sigma(O_1; f_1; ); f_2; ; )) = T(O_1; f_1; f_2; ; ) = T(O_1; f_1 \land f_2; ; ) \).

**I 5 Commutativity of selection and projection**

\[ 5.1 \quad \text{If key of } O_1 \subseteq \text{Attrs}(L) \text{ and } f \text{ involves only attributes in } \text{Attrs}(L), \text{ then} \]

\[ \sigma(\Pi_L(O_1); f; ; ) =_{st} \Pi_L(\sigma(O_1; f; ; )) \]

\[ 5.2 \quad \text{If } \mu \text{ involves only attributes in } \text{Attrs}(L), \text{ then} \]

\[ \sigma(\Pi_L(O_1); ; \mu) =_{st} \Pi_L(\sigma(O_1; ; \mu)) \]

\[ 5.3 \quad \text{If key of } O_1 \subseteq \text{Attrs}(L), \text{ and } f \text{ and } \mu \text{ involve only attributes in } \text{Attrs}(L), \text{ then} \]

\[ \sigma(\Pi_L(O_1); f; ; \mu) =_{st} \Pi_L(\sigma(O_1; f; ; \mu)) \]

**Proof:** Here we only prove 5.1 and the proofs for others are similar. Since the type of the resulting set object from selection is the same as that of the input set, \( T(\sigma(\Pi_L(O_1); f; ; )) = \)
\[ T(\Pi_L(O_1)) = T(\Pi_L(\sigma(O_1; f; ))) \]

I 6 Commutativity of selection and cross-product

I 6.1 If all the attributes of \( \mu \) are involved in \( \text{Attrs}(O_1) \) then

\[ \sigma(O_1 \times O_2; \mu) =_{st} \sigma(O_1; \mu) \times O_2 \]

I 6.2 If all the attributes of \( \mu \) are involved in \( \text{Attrs}(O_2) \) then

\[ \sigma(O_1 \times O_2; \mu) =_{st} O_1 \times \sigma(O_2; \mu) \]

I 6.3 If all the attributes of \( \mu \) are involved in \( \text{Attrs}(O_1) \) then

\[ \sigma(O_1 \times O_2; f; \mu) =_{st} \sigma(\sigma(O_1; \mu \neq \emptyset; ) \times O_2; f; \mu) \]

Proof: Here we only prove I 6.1 and the proofs for others are similar. Since the type of the resulting set object from selection is the same as that of the input set, \( T(\sigma(O_1 \times O_2; \mu)) = T(O_1 \times O_2) = T(\sigma(O_1; \mu) \times O_2) \).

I 7 Commutativity of selection and union

\[ \sigma(O_1 \cup O_2; \mu) =_{st} \sigma(O_1; \mu) \cup \sigma(O_2; \mu) \]

Proof: Since the type of the resulting set object from selection is the same as that of the input set, \( T(\sigma(O_1 \cup O_2; \mu)) = T(O_1 \cup O_2) = T(\sigma(O_1; \mu) \cup \sigma(O_2; \mu)) \).

I 8 Commutativity of selection and difference

\[ \sigma(O_1 - O_2; \mu) =_{st} \sigma(O_1; \mu) - \sigma(O_2; \mu) \]

Proof: Since the type of the resulting set object from selection is the same as that of the input set, \( T(\sigma(O_1 - O_2; \mu)) = T(O_1 - O_2) = T(\sigma(O_1; \mu) - \sigma(O_2; \mu)) \).

I 9 Commutativity of projection and cross-product

If Key of \( O_1 \subseteq \text{Attrs}(L_2) \) and Key of \( O_2 \subseteq \text{Attrs}(L_3) \), or both are not, and \( \text{Attrs}(L_1) \) is a list of attributes of which \( \text{Attrs}(L_2) \) are originated in \( \text{Attrs}(O_1) \) and \( \text{Attrs}(L_3) \) are originated in \( \text{Attrs}(O_2) \) then

\[ \Pi_L(O_1 \times O_2) = \Pi_L(O_1) \times \Pi_L(O_2) \]
Proof: The strong identity is not a strong-type identity since the cardinality of \( \text{sup}(T(\Pi_{L_2}(O_1) \times \Pi_{L_3}(O_2))) \) is always two while it is not true for that of \( \text{sup}(T(\Pi_{L_1}(O_1 \times O_2))) \).

I 10 Commutativity of projection and union.

I 10.1 \( \Pi_L(O_1 \cup O_1') =_st \Pi_L(O_1) \cup \Pi_L(O_1') \)

Proof: Recall that if the types of the two operands of union are the same, the resulting type will be the same as that of the operands. Thus we have \( T(\Pi_L(O_1 \cup O_1')) = T(\Pi_L(O_1)) = T(\Pi_L(O_1')) \).

I 10.2 If \( \text{Attrs}(T_1) = \text{Attrs}(T_2) \) where \( T_1 \neq T_2 \) and \( O_1 \) and \( O_2 \) have the same key then

\[ \Pi_L(O_1 \cup O_2) = \Pi_L(O_1) \cup \Pi_L(O_2) \]

Proof: The strong identity is not a strong-type identity. For example, let \( T_1 = \langle \{T_3, T_4\}, \emptyset \rangle \), \( T_2 = \langle \{T_3\}, \emptyset \rangle \), \( T_3 = \langle \{T_3\}, \emptyset \rangle \), and the projection list be \( \text{Attrs}(T_3) \).

Then \( T(O_1 \cup O_2) = (\emptyset, \text{Attrs}(T_3)) \) which implies \( T(\Pi_{\text{Attrs}(T_3)}(O_1 \cup O_2)) = (\emptyset, \text{Attrs}(T_3)) \). On the other hand, \( T(\Pi_{\text{Attrs}(T_3)}(O_1)) = T_3 = T(\Pi_{\text{Attrs}(T_3)}(O_2)) \), which implies \( T(\Pi_{\text{Attrs}(T_3)}(O_1) \cup \Pi_{\text{Attrs}(T_3)}(O_2)) = T_3 = \langle \{T_3\}, \emptyset \rangle \).

Thus \( T(\Pi_{\text{Attrs}(T_3)}(O_1 \cup O_2)) \neq T(\Pi_{\text{Attrs}(T_3)}(O_1) \cup \Pi_{\text{Attrs}(T_3)}(O_2)) \).

I 11 Commutativity of restructuring and union

If \( K \rightarrow \text{Attrs}(O_1) \) and \( K \rightarrow \text{Attrs}(O_2) \) then

\[ I_K(O_1 \cup O_2) =_st I_K(O_1) \cup I_K(O_2) \]

Proof: Since the type of the resulting set object from restructuring operation is the same as that of the input set object, \( T(I_K(O_1 \cup O_2)) = T(O_1 \cup O_2) = T(I_K(O_1) \cup I_K(O_2)) \).

I 12 Commutativity of restructuring and difference

If \( K \rightarrow \text{Attrs}(O_1) \) and \( K \rightarrow \text{Attrs}(O_2) \) then

\[ I_K(O_1 - O_2) =_st I_K(O_1) - I_K(O_2) \]

Proof: Since the type of the resulting set object from restructuring operation is the same as that of the input set object, \( T(I_K(O_1 - O_2)) = T(O_1 - O_2) = T(I_K(O_1) - I_K(O_2)) \).
13 Commutativity of restructuring and cross-product

If $K_1 \rightarrow \text{Attrs}(O_1)$ and $K_2 \rightarrow \text{Attrs}(O_2)$ and $K = K_1 \cup K_2$ where $K_1$ is in Attrs($O_1$) and $K_2$ is in Attrs($O_2$) then

$I_K(O_1 \times O_2) = \mathcal{I}_{K_1}(O_1) \times \mathcal{I}_{K_2}(O_2)$

Proof: Since the type of the resulting set object from restructuring operation is the same as that of the input set object, $T(I_K(O_1 \times O_2)) = T(O_1 \times O_2) = T(I_{K_1}(O_1) \times I_{K_2}(O_2))$.

14 Commutativity of restructuring and projection

If $K \subseteq \text{Attrs}(L)$ then

$I_K(\Pi_L(O_1)) = \mathcal{I}_{\Pi_L(I_K(O_1))}$

Proof: Since the type of the resulting set object from restructuring operation is the same as that of the input set object, $T(I_K(\Pi_L(O_1))) = T(\Pi_L(O_1)) = T(\Pi_L(I_K(O_1)))$.

15 Commutativity of restructuring and selection

If $K \rightarrow \text{Attrs}(O_1)$ then

$I_K(\sigma(O_1; \mu)) = \mathcal{I}_{\sigma(I_K(O_1); \mu)}$

Proof: Since the type of the resulting set object from selection or restructuring operation is the same as that of the input set object, $T(I_K(\sigma(O_1; \mu))) = T(I_K(O_1)) = T(\sigma(I_K(O_1); \mu))$.

16 Cascade of restructuring operations

If $K_1 \rightarrow \text{Attrs}(O_1)$ and $K_2 \rightarrow \text{Attrs}(O_1)$ then

$I_{K_1}(I_{K_2}(O_1)) = \mathcal{I}_{K_1}(O_1)$

Proof: Since the type of the resulting set object from restructuring operation is the same as that of the input set object, $T(I_{K_1}(I_{K_2}(O_1))) = T(I_{K_1}(O_1))$.

4.2. A dimensional framework for size estimation

In this section we present a framework for size estimation for algebraic operations. In our model traditional heuristics for optimization alone may not always work due to the nature of our operations and various domains.
For example, consider the "selection" operation. For ordinary data, the classical selection always reduces the size of the input. Therefore, the classical heuristic "perform selections as early as possible" usually works well during optimization. For dimensional data, our selection may not always reduce the size of a set. For instance, let \( O \) be a temporal set object and \( \mu \) be a complex spatio-temporal expression. Then \( \sigma(O; \mu) \) may result in an output which is larger than the input. This is because the retrieved tuple objects may have a complex spatial dimension, which requires considerable space to represent. As a result, the classical heuristic "perform selections as early as possible" may not work all the time.

Therefore, when applying the heuristics, we need to know the estimated size of the output in order to determine the applications of algebraic identities. Size estimation for our operators is inherently a complex problem, since it depends on the nature of each operation and the complexity of its operands, such as the complexity of spatio-temporal expressions. As a first step, we give a framework for size estimation and identify certain issues. Each of the issues within the framework will need to be studied thoroughly in the future.

We define two kinds of parameters, *alignment factors* and *output factors*, for dimension alignments and algebraic operations respectively. The alignment factors are the estimated sizes for alignment from one dimensional type to a more complex dimensional type (e.g., from temporal to spatio-temporal). The output factors are the estimated sizes resulting from algebraic operations. Here we assume that alignment factors are always used in conjunction with output factors, and the added domain (e.g., spatial domain is added if aligning from temporal to spatio-temporal) after the operation may not be the same for each object. The reason is that if a set object is aligned, for example, only for printing or storage purposes, the additional size is negligible since the added domain can be factored out from the set object itself. On the other hand, if it is determined that the added domain can be factored out after the operation, the set object will not need any alignment and the added domain will be carried over to the next operation.

To illustrate the usage of the two factors, consider the operation \( \sigma(O; \mu) \). To estimate the output size, we need to first determine whether dimension alignment is needed for the oper-
and. If so, say a dimension alignment is needed from temporal to spatio-temporal. Then we can use the alignment factor for temporal/spatio-temporal to calculate the size of the aligned operand. Finally we can compute the output size by using the output factor for \( \sigma(O; \mu) \) and the aligned size. Note that idea here does not limit to spatio-temporal databases but any dimensional databases can apply the same methodology.

Both parameters alignment factors and output factors may be determined at compile time. The alignment factors would depend on specific representations and the output factors would depend on the operation and its operands. As an example, the tables for the alignment factors and the output factors for selections are shown in Figure 13.

![Alignment factors for various dimension alignments.](image)

<table>
<thead>
<tr>
<th>Input size = n</th>
<th>ordinary</th>
<th>temporal</th>
<th>spatial</th>
<th>spatial-temporal</th>
</tr>
</thead>
<tbody>
<tr>
<td>ordinary</td>
<td>( n )</td>
<td>( c_1n )</td>
<td>( c_2n )</td>
<td>( c_3n )</td>
</tr>
<tr>
<td>temporal</td>
<td>-</td>
<td>( n )</td>
<td>-</td>
<td>( c_5n )</td>
</tr>
<tr>
<td>spatial</td>
<td>-</td>
<td>-</td>
<td>( n )</td>
<td>( c_6n )</td>
</tr>
</tbody>
</table>

(a) Alignment factors for various dimension alignments.

![Output factors for various selections.](image)

<table>
<thead>
<tr>
<th>Input size = n</th>
<th>output factors</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \sigma(O; \mu) )</td>
<td>( d_\mu n )</td>
</tr>
<tr>
<td>( \sigma(O; f; \mu) )</td>
<td>( d_f n )</td>
</tr>
<tr>
<td>( \sigma(O; f; \mu) )</td>
<td>( d_{\mu f} n )</td>
</tr>
</tbody>
</table>

(b) Output factors for various selections.

Figure 13. Example alignment factors and output factors for various selections.

In Figure 13(a), we see that for alignment factor the input size is multiplied by a coefficient \( c_i \). These coefficients estimate the output size from the alignments and are always greater than one. In Figure 13(b), these coefficients estimate the output size from the various selection operations and they depend on the actual operands such as \( \mu \). For example, the coefficient \( d_f \) should always be less than or equal to 1, since the operation \( \sigma(O; f; \mu) \) will never increase the size of the input set; \( d_\mu \) and \( d_{\mu f} \) could be any real numbers, depending on the actual \( \mu \) and \( f \). (One remark on \( d_{\mu f} \) is that in general, \( d_{\mu f} \) may not be \( d_\mu \ast d_f \), as \( \mu \) and \( f \) may be dependent.) Note that, however, not every output factor is a function of \( n \). Some output
factors may be constants. For example, a selection may always result in one tuple regardless of the input size. Similarly, the output factors for other operations can also be defined.

**Example 14.** Suppose that we want to estimate the size of \( \sigma(O_1; \mu) \) and compare it with that of \( O_1 \). In an ordinary database, a selection will always decrease the size of the input set object and thus no comparison is needed. But now let \( O_1 \) be a temporal set object of size 1000 and \( \mu \) be a spatio-temporal expression. Also let the alignment factor for temporal/spatio-temporal be \( 2.5*n \) and the output factor for \( \sigma(O_1; \mu) \) be \( 0.7*n \). (The assumption is for illustration only and may not be very realistic.) In this case, a dimension alignment on \( O_1 \) is needed from temporal to spatio-temporal. To estimate the size of \( \sigma(O_1; \mu) \), we first calculate the estimated size of the aligned \( O_1 \) by multiplying the size of \( O_1 \) by the alignment factor for temporal/spatio-temporal, which is \( 1000*2.5 = 2500 \). Then we obtain the final estimated size by multiplying the result with the output factor for \( \sigma(O_1; \mu) \), which is \( 2500*0.7 = 1750 \). In this example, we see that the output size of this selection operation is larger than the size of the input set object \( O_1 \). If the calculation was for the algebraic rule, \( \sigma(O_1 \times O_2; \mu) = \sigma(O_1; \mu) \times O_2 \), then the decision should be that the selection should not be pushed into the cross product.

**4.3. Algorithm for algebraic optimization of spatio-temporal queries**

In this section we will first give certain optimization heuristics and then sketch an algorithm to convert a given spatio-temporal query to a more optimal query. The algorithm uses the strong identities and the size estimation from previous sections. If only the strong-type identities were used, the algorithm will not be as effective since they are only a subset of strong identities. We now list the principles for algebraic manipulation and then an optimizer will be sketched.

**4.3.1. Principles for algebraic manipulation**

- Delay dimension alignment as much as possible.
  
  Dimension alignment increases the size of a set. Delaying alignment tends to make the intermediate results of multistep evaluations small. In a spatio-temporal model, algebraic
expressions can be manipulated by taking the dimensional types of the operands into account. Generally, the sizes of data of various dimensional types are according to the following heuristic:

\[ \text{ordinary} \ll \text{temporal} \ll \text{spatial} \ll \text{spatio-temporal} \]

- Delay computation with complex representation as much as possible.
  Complex representation may require more disk space to represent. This principle is similar to the first one. Delaying computation with complex representation tends to make the intermediate results of multistep evaluations small.

- Perform selections as early as possible.
  A selection reduces the size of a set. It tends to make the intermediate results of multistep evaluations small.

- Perform projections as early as possible.
  In relational model, a projection usually reduces the size of a relation smaller by reducing the number of attributes. In object-relational model, a projection sometimes does the otherwise because of path expressions. However, optimizing path expressions is beyond the scope of the thesis.

- Reduce size of operands in a cross-product.
  As it is well known, the cross-product operator is the most expensive operator. Reducing the size of the operands would drastically reduce the cost of the operation. The two heuristics above are used to reduce the size of the operands. Note that, however, associativity of cross-product holds only as a strong identity, but not as a strong-type identity.

- Reduce size of operands in the restructuring operation.
  The restructuring operator is also a fairly expensive operator and if the size of the operand is reduced the cost of the operation can be reduced significantly. This can be done by using the first two principles above. The size of the operand can also be reduce by pushing the restructuring operator into a cross product to yield two restructuring operations but on much smaller operands. Note that on the average the cost of the cross-product stays the same.
• Combine cascades of unary operations.

It is sometimes possible to combine a cascade of unary operations. For a cascade of selections, it may be possible to combine them into a single selection. For a cascade of projections, all but the last projection can be removed. The same is for restructuring.

• Remove redundant operations.

If T is the type of a set expression O, then \( \Pi_{\text{Attr}(T)}(O) = O \) and the redundant projection can be removed. Similarly, if K is the key of an expression O, the \( I_K(O) = O \) and the redundant restructuring operation can be removed.

• Combine operations that can be executed simultaneously.

Sometimes operations can be executed simultaneously in a single step. For example, in the expression \( D_1 \cup C \cup O \cup x \); f; \( \mu \) the selection and projection can be executed simultaneously while performing the cross-product.

4.3.2. An algorithm for optimizing set expressions

Perform each of the following steps, in order. Use alignment factors and output factors at each step, if applicable.

1. Use rules 4.1-4.5 to separate each selection into a cascade of selections. We do this because it may be easier to push a smaller selection further down the expression tree rather than a larger selection. Also note that the order of the cascade is important and various orders must be still tried in a practical implementation. For instance, it may be better to convert \( \sigma(O; ; \mu_1 \cap \mu_2) \) to \( \sigma(\sigma(O; ; \mu_2); ; \mu_1) \) rather than \( \sigma(\sigma(O; ; \mu_1); ; \mu_2) \) since it may be possible to push the selection on \( \mu_2 \) (but not the selection on \( \mu_1 \)) inside the expression O.

2. Use rules (I 4), (I 5), (I 6), (I 7), (I 8) and (I 15) to move the selections as far down the tree as possible, subject to the following conditions.

As stated in the previous section, one of our heuristics is to delay dimension alignment as much as possible. Therefore, when we push a selection into a projection, cross product, union or difference, the alignment factors and output factors should be taken into account. For example, \( \sigma(O_1 \times O_2; ; \mu) \) should not be converted to \( \sigma(O_1; ; \mu) \times O_2 \) if the estimated size of
\( \sigma(O_1; \mu) \) is larger than that of \( O_1 \). Otherwise, it would make the operand for the cross product larger. Note that this alternation may seem to violate the heuristic "do selection as soon as possible," but it should be clear that this is more efficient.

3. Use rules (I 3), (I 5), (I 9) and (I 14) to move the projections as far down the tree as possible, when the projection list is smaller than the attributes of the input type. Also, eliminate redundant projections where possible.

   Since our projection operator may not reduce the number of attributes because of path expression, sometimes we may not want to push the projection down the tree. For example, \( \Pi_L(\sigma(O_1; f; )) \) should not be converted to \( \sigma(\Pi_L(O_1); f; ) \) when \( |L| > |\text{Attrs}(O_1)|. \) This is because the conversion will increase the number of attributes rather than decreasing it. Again, we have not considered optimizing path expression in this thesis.

4. Use rules (I 3), (I 4) and (I 5) to combine a cascade of selections and projections into a single selection, a single projection or a selection followed by a projection.

5. Use rules (I 13) and (I 16) to move restructuring operations down the tree. Rule (I 13) moves restructuring operations into the cross product. Rule (I 16) eliminates all but the last restructuring operation in a cascade of restructuring operations. Also, eliminate redundant restructuring operations, i.e., eliminate a restructuring operation if it restructures an expression on its own key.

6. Identify sequences of operations that can be executed simultaneously. For instance, in \( \Pi_L(\sigma(I_K(O); f; \mu) \) the selection and projection can be performed while doing the restructuring.

**Example 15.** Suppose \( e_1(A, B, C) \) is a spatio-temporal set expression of estimated size 10000 blocks, and \( e_2(D, E, F, G) \) is a temporal set expression \( e_2 \) of estimated size 1000 blocks. Consider the query \( \sigma(e_1 \times e_2 ; \mu_1(B) \cap \mu_2(E, F) \cap \mu_3(A, D)) \). For illustration purpose, suppose \( \mu_1, \mu_2 \) and \( \mu_3 \) are such that \( d_{\mu_1} \) and \( d_{\mu_2} \) are both 0.7*n and \( d_{\mu_3} \) is 10. Also let the alignment factor c for temporal/spatio-temporal be 2.5*n. A possible scenario for this example is that both spatio-temporal expressions \( \mu_1 \) and \( \mu_2 \) cover 70% of the entire domain, \( \mu_3 \) (e.g. \( [A = D] \)) will result in one object with estimated size of 10 blocks regardless of the input size, and the representation of a spatio-temporal description for a tuple requires additional 1.5 times of
the original tuple size.

• Let us calculate the disk accesses without algebraic optimization.
  Read: 1000 + 1000*10000 = 1001000
  Write: 10
  Total disk accesses = 1001000 (read) + 10 (write) = 10001010

• We first optimize the above query without considering dimension alignment.
  Applying rule I 4.3 twice, we get
  \( \sigma(\sigma(e_1 \times e_2 ; ; \mu_1(B)) ; ; \mu_2(E,F)) ; ; \mu_3(A,D)) \)
  Applying rule I 6.1, we get
  \( \sigma(\sigma(e_1 ; ; \mu_1(B) \times e_2 ; ; \mu_2(E,F)) ; ; \mu_3(A,D)) \)
  Applying rule I 6.1, we get
  \( \sigma(\sigma(e_1 ; ; \mu_1(B)) \times \sigma(e_2 ; ; \mu_2(E,F)) ; ; \mu_3(A,D)) \)
  This query can be evaluated in the following steps:

  **Step 1.** Compute \( s_1 = \sigma(e_1 ; ; \mu_1(B)) \) in a single step.
  Read: 10000
  Write: 10000*0.7 = 7000
  Total disk accesses = 10000 (read) + 7000 (write) = 17000.

  **Step 2.** Compute \( s_2 = \sigma(e_2 ; ; \mu_2(E,F)) \) in a single step.
  Read: 1000
  Write: Since \( e_2 \) is a temporal set, an alignment is needed for \( e_2 \) and the estimated aligned size for \( e_2 \) is 1000*2.5 = 2500. The estimated output size for selection is thus 2500 * 0.7 = 1750
  Total disk accesses = 1000 (read) + 1750 (write) = 2750.

  **Step 3.** Compute \( \sigma(s_1 \times s_2 ; ; \mu_3(A,D)) \)
  Read: 1750 + 175*7000 = 12251750
  Write: 10
  Total disk accesses = 12251750 (read) + 10 (write) = 12251760

Therefore the total disk accesses = 17000 + 2750 + 12251760 = 12271510
Now we optimize the same query, taking dimension alignment into account and using size estimation.

Applying rule I 4.3 twice, we get

\[ \sigma(\sigma(\sigma(e_1 \times e_2 ; ; \mu_1(B)) ; \mu_2(E,F)) ; \mu_3(A,D)) \]

Applying rule I 6.1, we get

\[ \sigma(\sigma(\sigma(e_1 ; \mu_1(B)) \times e_2 ; \mu_2(E,F)) ; \mu_3(A,D)) \]

We do not apply I 6.1 again to push \( \mu_2 \) into the second operand because doing so will increase the operand size from 1000 to 100*2.5*0.7 = 1750.

Applying rule I 4.3, we get

\[ \sigma(\sigma(e_1 ; \mu_1(B)) \times e_2 ; \mu_2(E,F) \cap \mu_3(A,D)) \]

This query can be evaluated in the following steps:

Step 1. Compute \( s_1 = \sigma(e_1 ; ; \mu_1(B)) \) in a single step.

Read: 10000
Write: 10000*0.7 = 7000
Total disk accesses = 10000 (read) + 7000 (write) = 17000

Step 2. Compute \( \sigma(s_1 \times e_2 ; ; \mu_2(E,F) \cap \mu_3(A,D)) \) in a single step.

Read: 1000 + 1000*7000 = 7001000
Write: Since \( \mu_3(A,D) \) will result 10 blocks of output, the estimated size for the query is then 1.

Total disk accesses = 7001000 (read) + 10 (write) = 7001010
Therefore the total disk accesses = 17000 + 7001010 = 7018010

Comparing the two approaches, the savings of the second approach over the first one is (12271510 - 7018010) / 7018010 = 75%. The savings is mainly due to that in the second case, the estimated operand sizes for the cross product are smaller than the ones in the first case.

Table 1 shows various savings over different optimization plans with different values for \( d_{\mu_1} \) and \( d_{\mu_2} \).

In Table 1 we see that for the middle rows (shaded) each parameter value is less than 1. In such cases our plan works same as Nair's. In other cases our plan shows improvement over
Nair's plan. This is because in Nair's plan, all the selections are performed before the cross product, but they do not reduce but rather enlarge the sizes of the operands. In our plan and the optimized plan, on the other hand, the cross product is performed before the selections. Thus the cross product can be executed with smaller operands, which makes the whole execution more efficient. In this illustration, our optimized plan performs well in all cases.

Table 1. Comparisons of savings with various parameters.

<table>
<thead>
<tr>
<th>Parameter values</th>
<th>Savings (%) using one plan vs. another</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>d_{u1}</td>
</tr>
<tr>
<td>0.7</td>
<td>1.75</td>
</tr>
<tr>
<td>0.7</td>
<td>1.5</td>
</tr>
<tr>
<td>0.7</td>
<td>1</td>
</tr>
<tr>
<td>0.7</td>
<td>0.1</td>
</tr>
<tr>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>0.05</td>
<td>0.001</td>
</tr>
<tr>
<td>1</td>
<td>1.75</td>
</tr>
<tr>
<td>5</td>
<td>1.75</td>
</tr>
<tr>
<td>10</td>
<td>1.75</td>
</tr>
<tr>
<td>100</td>
<td>1.75</td>
</tr>
<tr>
<td>1000</td>
<td>1.75</td>
</tr>
</tbody>
</table>
5. THE PATTERN MATCHING LANGUAGE

Our model and SQL-like languages treat space and time uniformly. This uniformity is made possible by using dimensional elements and the concept of a key. Dimensional elements are a primitive data type, on which the concepts of attribute values and tuple objects are built. The concept of key is used to form meaningful set objects. The algebra captures the similarity in linguistics of space and time. This similarity is based upon the boolean valued structure of the dimensional elements.

The algebraic language SQL is weaker in expressive power than the lower level languages such as C or Pascal. Thus to gain additional expressive power in application programming, SQL is sometimes embedded in a lower level language. As the syntax and semantics of SQL and the lower level programming language are very different, sometimes the term *impedance mismatch* is used to describe it. Eliminating the impedance mismatch continues to be a fundamental issue in databases. However, elimination of impedance mismatch is a rather illusive goal as the feasibility of optimization seems to diminish with the expressive power.

A tuple object in spatio-temporal databases can be very rich but complex in information content. Therefore we need a lower level language to express complex navigation within a tuple object. This need in the context of temporal database was first expressed in [Gad86]. Note that such a language should recognize linguistics peculiarities of space as well as time. For example, in human perception time evolves, whereas the space is present all at once. We present a pattern matching language for this purpose.

In this chapter we present a pattern matching language for spatio-temporal databases [CG94]. The concept of pattern matching presented in this thesis is meant to be independent of the choice of a model for spatio-temporal databases. However, for illustrations we use ORDimSQL to show where the pattern matching fits in the big picture. In this thesis we only demonstrate the use of patterns in querying spatio-temporal databases; patterns can also be used as triggers in active databases.
Our pattern matching language smoothly integrates into our SQL-like language. In the context of ORDimSQL, the pattern matching language extends the construct \[\text{[A0B]}\] of ORDimSQL. (Note that \[\text{[A0B]}\] is counterpart of “A0B” of ordinary relational databases allowing queries such as “select C from r where A<B”. ) As a result, the integration provides backward compatibility to the previous version of ORDimSQL. Note that we are not embedding our SQL like language into the pattern matching language, rather quite the opposite is true. We do not claim that the addition of pattern matching language will eliminate need for embedding, but it should reduce it substantially. This helps reduce the impedance mismatch. The pattern matching language is not meant to be as powerful as C or Pascal, however it is custom designed for spatio-temporal data. It includes mechanism for assignment of values to variables, control structures and subroutines.

In the spatio-temporal model, a pattern can be a temporal pattern, a spatial pattern or a mix of the two. The proposed language supports all these varieties. This yields considerable expressiveness, yet the patterns are easy to compose. Furthermore, since temporal patterns and spatial patterns are special cases of spatio-temporal patterns, migration to spatio-temporal model from either temporal or spatial model will be seamless. In other words, no temporal or spatial queries need to be rewritten when migrating to a spatio-temporal generalization. Also, the user needs to know only one pattern matching language for a data dimension. For example, the temporal pattern constructs for the temporal model is the same as the one for the spatio-temporal model and thus user-friendliness is enhanced.

Pattern matching consists of two elements: the construction of patterns (the syntax) and the matching process (the semantics). Before we introduce the language, it is necessary to understand the matching process in order to know how to construct patterns. We begin with pattern matching process in next section.

5.1. The pattern matching process

There are two basic pattern matching processes in spatio-temporal model: temporal pattern matching and spatial pattern matching. Temporal pattern matching is based on the con-
cept of time cursor [GG73], while spatial pattern matching is based on the relative positions of regions. Spatio-temporal pattern matching is then done by interleaving the two processes.

5.1.1. Temporal pattern matching

Temporal pattern matching is basically an examination of time instant/property values of tuple objects from the beginning to the end of the time line in which an attempt is made to match a construction specified by the pattern. To achieve this, we need the concept of time cursor. A time cursor (or simply cursor) is a pointer to a time instant on the 1-dimensional time line. We call the time instant that the cursor points to the cursor time. The default movement of the cursor is from left to right (increasing value of time), one time unit at a time. The nature of the cursor movement and its resolution can be altered by setting cursor options. As patterns match certain values encountered can be stored in program variables. In each step in pattern matching, the object and the values of the user variables may be examined. If the matching succeeds, then the cursor will automatically advance as specified by the cursor options. In the default case the cursor moves to the instant, following the domain of the pattern just matched. To illustrate the mechanism, let us consider a simple example.

Example 16. The pattern salary was raised from 40K to 50K on 1/1/91 can be expressed as follows. (Also see Figure 14.).

\[
\text{cursorResolution(DAY)}
\]
\[
[ \text{AT(12/31/90)} \ o.SALARY =40K =50K ]
\]

As we will see from the syntactic rules to be introduced shortly, the above pattern consists of an option and a temporal pattern. The option "cursorResolution(DAY)" sets the cursor granularity to be one day, for the following temporal pattern in brace-brackets. The temporal pattern consists of two subpatterns "AT(12/31/90)" and "o.SALARY =40K =50K".

- \text{AT(12/31/90)} is an attribute independent pattern (termed autonomous pattern). It simply says "the cursor is at 12/31/90." When this pattern is being examined, the match will take place if and only if the cursor time is 12/31/90. Note that the AT pattern does not move the cursor, it only checks for position.
$SALARY = \frac{40K}{50K}$

*Time line*

12/31/90  \hspace{1cm} 1/1/91  \hspace{1cm} 1/2/91

(a) The temporal pattern described in Example 16.

(b) The spatial pattern described in Example 17.

(c) The spatio-temporal pattern described in Example 18.

*Figure 14. Various patterns described in Section 5.1.*
• The second pattern \texttt{o.SALARY = 40K = 50K} is an attribute dependent pattern. An
attribute dependent pattern consists of an attribute followed by a sequence of compari-
sions. In this case the attribute is \texttt{SALARY}. This is followed by “\texttt{40K = 50K}”, called the
attribute follow-up pattern. The pattern “\texttt{o.SALARY = 40K = 50K}” says that attribute
values should be 40K at the cursor time and 50K immediately after (the next cursor time).

To begin the cursor granularity is set to one day. When matching the entire pattern, the
system first checks if the cursor time is 12/31/90. No cursor movement takes place, as speci-
fied by the semantics of “\texttt{AT}.” Now the \texttt{SALARY} of the object \texttt{o} is checked to determine if it is
40K. If this is the case, then the cursor moves to 1/1/91; otherwise, the matching is failed.
Similarly if the salary is 50K at the current cursor time (1/1/91), the pattern is matched and the
cursor advances to 1/2/94.

5.1.2. Spatial pattern matching

Spatial pattern matching is an examination of the properties and the relative positions of
one or more spatial domains. Since spatial patterns can be nested, inner patterns are evaluated
first and then the outer ones. Let us consider a simple example.

Example 17. The pattern \textit{area within 50 miles of earthquake zone is adjacent to the area
where the population density higher than 500} can be expressed as follows. (Also see Figure
14.)

\texttt{CIRCLE(o.EARTH-QUAKE = high, 50) ADJACENT (o.POPUL_DENS >= 500)}

There are three subpatterns embedded in the spatial pattern: \texttt{o.EARTH-QUAKE=high, CIR-
CLE(o.EARTH-QUAKE=high, 50)} and \texttt{o.POPUL_DENS >= 500}. Many times a spatial pattern
may be formed by one spatial pattern followed by a binary operator and then another spatial
pattern. In this case \texttt{ADJACENT} is such an operator. A spatial pattern may also be formed by
using one of the built-in function. In this case \texttt{CIRCLE} is such a built-in function. It takes a
spatial region specified by \texttt{o.EARTH-QUAKE=high} as its input and returns the spatial region
within 50 miles of the input region.
When matching, the system first computes the spatial region, say $\text{reg}_1$, where $\text{odb.EARTH-QUAKE} = \text{high}$. It then computes the region, say $\text{reg}_2$, where it is area within 50 miles of the region $\text{reg}_1$. On the other hand, the system computes the region, say $\text{reg}_3$, where $\text{odb.POPUL_DENS} \geq 500$. Finally $\text{reg}_2$ is checked if it is adjacent to $\text{reg}_3$, using certain tolerance parameters.

5.1.3. Spatial-temporal pattern matching

In the spatial-temporal context, we view that there is a spatial plane at each time instant. Spatio-temporal pattern matching is like the temporal pattern matching, except that at each time instant, a spatial pattern is examined instead of an attribute value comparison. Now let us consider a simple example.

Example 18. Assuming that the cursor granularity is set to one year. The pattern earth quake area is larger than it was 10 years ago can be expressed as follows. (Also see Figure 14.)

```plaintext
cursorResolution(YEAR)
[ ((odb.EARTH-QUAKE = high) =: _quake-area)
 ARB(9)
 ((odb.EARTH-QUAKE = high) PROPER_CONTAIN _quake-area) ]
```

The above pattern consists of an option and a spatial-temporal pattern. The option "cursorResolution(YEAR)" sets the cursor granularity to be one year, for the following spatial-temporal pattern in brackets.

The spatial-temporal pattern can be viewed as one spatial pattern followed by one temporal pattern and then another spatial pattern. ARB(9) is the temporal pattern and the two patterns before and after are spatial patterns. In the first spatial pattern, the operator "=" is an assignment operator. In this case, it assigns the spatial pattern (odb.EARTH-QUAKE=high) to the program variable _quake-area. (A variable always starts with an underscore.) Once a spatial variable is defined/assigned, the variable can then be treated as another spatial pattern. In this case the variable _quake-area is used as a spatial pattern in the second spatial pattern.

When matching, the system first computes the spatial region where odb.EARTH-QUAKE = high, and stores it in the variable _quake-area. Assuming that the default resolution of the cursor is one year, the cursor is then advanced by 1 year. Then the operator ARB(9) advances
the cursor by 9 more years. Now the system computes the spatial region, say reg₁, where
\( EARTH-QUAKE = \text{high} \) at the cursor time and checks if it properly contains the stored region
\( _\text{quake-area} \).

### 5.2. Construction of patterns

In this section we present the language for constructing spatio-temporal patterns, of
which, temporal patterns, spatial patterns are special cases. Normally, a (sub)pattern will be
matched at the current cursor time and the domain matched by the pattern will be returned. If
the returned domain is empty, the matching process fails and an empty domain is returned,
unless the failure is otherwise allowed by the (loop) operator. After matching of a pattern the
cursor will be at the last time instant matched plus one.

#### 5.2.1. Spatio-Temporal pattern

\[
\text{<SpatioTemporal pattern>} ::= \begin{array}{l}
(\{\text{<option>}\}) \ [\text{<SpatioTemporal pattern>} ] \ |
\text{<spatial pattern>} \ |
\text{<autonomous pattern>} \ |
\text{<attribute>} \ \{\text{<attribute followup pattern>}\} \ |
\text{<SpatioTemporal loop pattern>} \ |
\text{<SpatioTemporal logical pattern>} \ |
\text{<SpatioTemporal aggregate pattern>} \ |
\text{<SpatioTemporal pattern>} \ \{\text{<SpatioTemporal pattern>}\}
\end{array}
\]

A spatio-temporal pattern is basically a sequence of patterns. These patterns can be
of various forms. For example, a loop pattern is to define a repeated pattern to be
matched. Note that if we take away the syntax of spatial pattern, \(<\text{spatial pattern}>\), the
grammar then becomes temporal pattern. For \(<\text{spatial pattern}>\) alone, we have spatial
pattern. Therefore the language can migrate from either domain to spatio-temporal
domain seamlessly.

\[
\text{<option>} ::= \begin{array}{l}
\text{suppressOperand} \ |
\text{cursorJump} \ |
\end{array}
\]
Several options can be specified for a pattern. If no options are specified, the options from the outer scope will be used. The scope of these options only applies to the following pattern within the brackets [ ]. Using brackets is only for ease of reading in this thesis. One could always use parentheses for specifying the scope of the options. Now we introduce these options. suppressOperand suppresses the domain matched by the pattern. cursorJump attempts to find the first match of the pattern by advancing the cursor. cursorReset resets the cursor to its starting position after the pattern is matched. cursorReverse causes the cursor advancing from right to left, instead from left to right. cursorResolution specifies the resolution of the cursor such as DAY, YEAR or user defined.

<autonomous pattern> ::= 
  AT( <attr indep time instant> ) | 
  ARB [( <period> )] | 
  REV_ARB [( <period> )]

There are several autonomous (attribute independent) patterns. AT, with attribute independent time instant, specifies where the cursor should be. This operator will not cause the cursor to advance and it will not return the time domain. ARB specifies how much the cursor should advance. (If no parameter is specified, it will cause the cursor to advance until the following pattern can have a match.) REV_ARB is the same as ARB except in the opposite direction.

<attribute followup pattern> ::= 
  {({<option>}) [ <followup pattern> ]} | 
  {<followup pattern>}

There are patterns that must follow attributes. Each of the follow up patterns may or may not have options. If no options are specified, the options from the outer scope will be used.
<followup pattern> ::= 
  <autonomous pattern> | 
  AT( <attr dep time instant> ) |
 <temporal comparison pattern>

AT, with attribute dependent time instant, is such a pattern. AT checks if the cursor is at the input time point. If so, the cursor stays at the same time point and the matching continues; otherwise, the matching fails. The other is the temporal comparison pattern.

<period> ::= 
  <integer constant> | 
  <timeGranule> ( <integer constant> )

<timeGranule> ::= 
  <userTimeType> | SECOND | MINUTE | HOUR | DAY | MONTH | YEAR | . . .

The time period is either an integer or a system or user-defined function. For example, HOUR(10) will return the length of period equal to the length of 10 hours.

<temporal comparison pattern> ::= 
  <comparison or assignment> | 
  = SPAN (<set of constants>) | 
  = EVOL (θ [{<computation> [=: <program variable>] } [,<period>]}) 
[::<boolean expression>]

SPAN matches the longest possible domain consisting of consecutive attribute values given in its argument. EVOL matches the θ relationships with the attributes values at the two ends of the <period>. Some built-in computation variables are provided for the difference of the two values, such as _DIFF and _RATIO. Potentially, EVOL can also be used to match continuous pattern. The built-in variable _FUNCTION is provided for the function during the <period>. User defined computation can be defined. Furthermore, computation variables can also be assigned, using the assignment operator =:, to some program variables. An optional <boolean expression> can also be specified.

<comparison or assignment> ::= 
  θ <constant> | 
  θ <attribute> | 
  θ <program variable> | 
  = <program variable> [: <boolean expression>]
<computation> ::= 
  _DIFF | 
  _RATIO | 
  _PERCENT | 
  _FUNCTION | 
  USER_DEFINED

<program variable> ::= 
  _<identifier>

<SpatioTemporal loop pattern> ::= 
  <scan type> [ALL] [count] (<SpatioTemporal pattern>) [:(<boolean expression>)]

<scan type> ::= 
  scanNormal | 
  scanRepeat

Patterns can be matched repeatedly by using the loop construct. <scan type> specifies whether to reset the cursor after each match. scanNormal reset the cursor to the previous starting time instant plus one before the next matching begins. scanRepeat does not reset the cursor and starts the next pass at the current cursor time. ALL specifies that the matching process would not stop even if a pattern fails to match, and the domain of all matched patterns will be returned. Without ALL, the matching process fails if once a pattern fails to match. <count> specifies the number of occurrences of the pattern, and is default to be infinite. <boolean expression> will stop the looping and return if it evaluates to false.

<SpatioTemporal logical pattern> ::= 
  <SpatioTemporal pattern> AND <SpatioTemporal pattern> | 
  <SpatioTemporal pattern> OR <SpatioTemporal pattern> | 
  NOT <SpatioTemporal pattern>

AND specifies that both of its operands must be matched and the intersection of their domains will be returned. For both operands of AND, the cursor starts at the same point. After matching, the cursor will be at the last time instant of the returned domain plus one. OR specifies that either of its operands must be matched and the union of their domains will be returned. If an operand pattern fails to match, it returns empty domain for that operand. For both operands of OR, the cursor starts at the same point.
After matching, the cursor will be at the last time instant of the resulting domain plus one. NOT specifies that the operand must not be matched and the cursor is set to the starting time plus one after matching. The domain containing the starting time is returned.

\[<\text{attr indep time instant}> ::= \]
\[\langle\text{time instant constant}\rangle \mid \text{BEGIN} \mid \text{END} \mid \text{NOW} \mid \ldots\]

BEGIN, END and NOW refer to the beginning, the end and the current time respectively.

\[<\text{attr dep time instant}> ::= \]
\[<\text{attr indep time instant}> \mid \text{FIRST} \mid \text{LAST} \mid \text{INTERVAL\_FIRST} <\text{integer constant}> \mid \text{INTERVAL\_LAST} <\text{integer constant}> \mid \ldots\]

FIRST and LAST refer to the first and the last time instants where the attribute is defined. INTERVAL\_FIRST and INTERVAL\_LAST refer to the first and the last time instants in the specified time interval where the attribute is defined.

\[<\text{SpatioTemporal aggregate pattern}> ::= \]
\[\text{MIN\_PATTERN} (<\text{SpatioTemporal loop pattern}>, <\text{numeric expression}>) \mid \text{MAX\_PATTERN} (<\text{SpatioTemporal loop pattern}>, <\text{numeric expression}>) \mid \text{OCCURS} (<\text{SpatioTemporal loop pattern}>), <\text{comparison or assignment}> \mid \text{DURATION} (<\text{SpatioTemporal pattern}>), <\text{comparison or assignment}>\]

The aggregate operators, MIN\_PATTERN and MAX\_PATTERN, given a loop pattern, returns the domain of the pattern selected by the min or max of a given numeric expression. OCCURS returns the number of occurrences matched in the given loop pattern, while DURATION returns the duration (the number of time units with respect to the cursor resolution) of the domain returned by the given loop pattern. These numbers are then compared or assigned.

### 5.2.2. Spatial pattern

\[<\text{spatial pattern}> ::= \]
\[\text{(suppressOperand)} \mid <\text{spatial pattern}> \mid <\text{program variable}> \mid <\text{spatial pattern}> \text{IN\_CONTEXT} <\text{spatial pattern}> \mid <\text{program variable}>\]
A spatial pattern defines a spatial region. suppressOperand suppresses the spatial domain returned by the operand. A spatial pattern can be assigned to a variable using the assignment operator =:. IN_CONTEXT limits the context to a given spatial region instead of the whole spatial domain.

```
<spatial attribute pattern> ::=  
  <attribute> <constant>  
  <attribute> <attribute>

<spatial function pattern> ::=  
  <spatial region function> (<spatial pattern>, [<parameters>])

<spatial region function> ::=  
  CIRCLE | SQUARE | BOUNDARY | INTERIOR | . . .

  Certain spatial functions, such as CIRCLE and SQUARE, are for computing a new region from its input region.

<spatial measurement pattern> ::=  
  <1-spatial measurement function> (<spatial pattern>) <comparison or assignment>  
  <2-spatial measurement function> (<spatial pattern>, <spatial pattern>) <comparison or assignment>

<1-spatial measurement function> ::=  
  AREA | PERIMETER | LENGTH | . . .

<2-spatial measurement function> ::=  
  SHORTEST_DIST | LONGEST_DIST | CENTER_DIST | . . .

  Spatial patterns can also be specified by the properties of the regions. The property of a region could be AREA or PERIMETER. The property between two regions could be SHORTEST_DIST or LONGEST_DIST.
```

```
<comparison or assignment> ::=  
  <constant>  
  <attribute>  
  <program variable>  
  =: <program variable>[: <boolean expression>]
```
<spatial comparison pattern> ::=<spatial pattern> <2-spatial op> [<tolerance parameter>] <spatial pattern> |<3-spatial op> (<spatial pattern>, <spatial pattern>, <spatial pattern> [, <tolerance parameter>])
<2-spatial op> ::= ADJACENT | TOUCH | LEFT | RIGHT | ABOVE | BELOW | NEAREST | FARTHEST | NORTH | SOUTH | WEST | EAST | EQUAL | CONTAINED_IN | PROPER_CONTAINED_IN |...
<3-spatial op> ::= IN_BETWEEN |...
Spatial patterns can also be specified by the relative positions of the regions. For example, ADJACENT returns the union of the set of spatial points where they are adjacent to each other in the two operands. If the resulting set is empty, the pattern is not matched. An optional tolerance parameter can also be specified.

<spatial logical pattern> ::=<spatial pattern> AND <spatial pattern> |<spatial pattern> OR <spatial pattern> | NOT <spatial pattern>
In spatial context, AND, OR and NOT correspond to intersection, union and complementation respectively. NOT is computed with respect to a context region or the whole region.

5.2.3. Temporal pattern
The grammar for temporal patterns would be exactly the same as <spatioTemporal pattern> except that <spatial pattern> is not included.

5.3. Storing patterns in functions
Patterns are sometimes complex and frequently used. Therefore it is desirable to store patterns as functions in the library. Below is the syntax of a function in our language and example of its use is shown in Example 19.

<function> ::= FUNCTION <identifier> ([<parameter>: <type>]) : <return type>
RETURN(<return expression>)
<type> ::= DOMAIN | ATTRIBUTE | CURSOR | PATTERN | BOOLEAN | ...
<return type> ::= DOMAIN | PATTERN | BOOLEAN
<return expression> ::= [SpatioTemporal pattern] | <SpatioTemporal pattern> | <boolean expression>

A function with zero or more input parameters returns a domain, a pattern or a boolean value. When a function returning a pattern is called, the function will be substituted by the returned value in the calling pattern.

**Example 19.** The query *list all the managers during their salaries being between 50K and 70K who had been promoted during 1990-1993* can be expressed as below. (To illustrate the use of functions, the query uses two defined functions `time_having_salary` and `once_promoted` in its `restricted_to` clause and where clause respectively.)

FUNCTION `time_having_salary`(salary: ATTRIBUTE, sal1, sal2: INTEGER): DOMAIN
return([[salary > sal1] ∩ [salary < sal2]])

FUNCTION `once_promoted`(rank: ATTRIBUTE, start_time, end_time: CURSOR): BOOLEAN
return([[AT(start_time) cursorJump[rank = EVOL(<):(BEFORE(end_time))] ]] ≠ ∅)

select m.MANAGER
restricted_to time_having_salary(m.SALARY, 50K, 70K)
from management m
where once_promoted(m.RANK, 1990, 1993)

#### 5.4. The construct [[ . ]] in ORDimSQL

The original ORDimSQL contains the constructs `[[AθB]]` which is interpreted as the set of all dimensional points \( p \) where \( A(p) \) is in \( θ \) relationship with \( B(p) \). Now with the addition of patterns, \( AθB \) in `[[AθB]]` syntactically qualifies as a pattern. The pattern \( AθB \) retrieves only \( \{ p = (t,s) \in Time \times Space \mid t \) is the current cursor time and \( A(p) \) is in \( θ \) relationship with \( B(p) \} \). This problem is solved by treating `[[ . ]]` in `[[AθB]]` as a loop. Therefore an alternative definition of `[[AθB]]` is "scanNormal ALL AθB". Thus the pattern matching language in this thesis
consistently extends the original ORDimSQL. In general, if \( p \) is a pattern, \( [p] \) is defined as “scanNormal ALL \( p \)”. We note that ORDimSQL also has the constructs \( [A] \) and \( [r] \); but syntactically, \( A \) in \( [A] \) and \( r \) in \( [r] \) do not qualify as patterns, and they continue to work in the original way, with no potential for ambiguity.

5.5. IF-THEN-ELSE as dimensional expression

In addition to the pattern language we have introduced, we have also added the if-then-else construct as dimensional expression. In our language, IF-THEN-ELSE has the following syntax.

\[
<\text{dimensional expression}> ::= \\
\text{IF} <\text{boolean expression}> \text{THEN} <\text{dimensional expression}> [\text{ELSE} <\text{dimensional expression}>] \\
\]

If the boolean expression is true then the dimensional expression in the THEN clause is evaluated; otherwise the dimensional expression in ELSE clause is evaluated. If the ELSE clause is missing, then it is the same as having the corresponding dimensional expression being empty.

5.6. Examples

In the previous section we presented our language for pattern matching. As stated before, the concept of pattern matching in this thesis is independent of the choice of an underlying model for spatio-temporal databases. In particular, the pattern matching language is seamlessly integrated into ORDimSQL. In this section we give a few interesting examples.

Example 20. The query list all employees who have worked in computer department for at least 10 years continuously can be expressed as follows.

\[
\text{select} \quad \text{e.NAME} \\
\text{from} \quad \text{emp e} \quad /* \text{temporal set object */} \\
\text{where} \quad [\text{DURATION}(\text{e.DEPT} = \text{SPAN} ("computer")) \geq \text{YEAR}(10)] \neq \emptyset
\]
In the where clause in the above expression, the temporal pattern specifies that the duration of `DEPT` of object `e` being "computer" continuously must be greater than or equal to 10 years. The domain extractor `[ ]` then matches the pattern at each `t ∈ Time`. If there is at least one successful match, the resulting domain will be nonempty and thus the object will be retrieved.

Example 21. The query **list John's salary when the first time John relocated from CA to IA?** can be expressed as follows.

```sql
select e.SALARY
restricted_to [e.LOCATION AT(FIRST) cursorJump[suppressOperand [= "CA"] = "IA"]]
from emp e /* temporal set object */
where [e.NAME = "John"] ≠ Ø
```

There are two temporal patterns in the above query: one is in the restricted_to clause and the other is in the where clause. In the where clause, it requires that the `NAME` of the object must be "John" at some time `t`. In the restricted_to clause, the pattern specifies that the first time instant where the `LOCATION` changes from "CA" to "IA." `AT(FIRST)` specifies that the cursor time must be at the first time instant where `LOCATION(e)` is defined. `cursorJump[suppressOperand [= "CA"] = "IA"]` specifies that `e.LOCATION` must be "CA" at time `t` and "IA" at time `t+1`. The `suppressOperand` operator indicates that the time domain for `[= "CA"]` is not included in the resulting domain. The `cursorJump` operator indicates that it may need to advance the cursor some number of steps before there is a match. Since the matching process is from left to right, it finds the first occurrence of the relocation. Note that the operator `AT(FIRST)` is necessary in correctly specifying the pattern. This is because the domain extractor `[ ]` will match the pattern at each `t ∈ Time`. `AT(FIRST)` ensures that the starting cursor time is before the first occurrence of such pattern, if it exists. Without `AT(FIRST)` the clause will then return the time domain of all of such relocation.

Example 22. The query **list John's department when the first time he got a salary raise and a location change at the same time** can be expressed as follows.
select e.DEPT
    restricted_to [AT(BEGIN) cursorJump[e.SALARY = EVOL(<) AND e.LOCATION = EVOL(≠)]]
from emp e /* temporal set object */
where [[e.NAME = "John"] ≠ Ø]

The where clause in the above query is the same as in Example 21. In the restricted_to clause AT (BEGIN) specifies that the starting cursor time must be the beginning of the time line. (Note that an equivalent approach, instead of having AT (BEGIN) at the beginning, is to put AT (FIRST) after e.SALARY or e.LOCATION in the pattern.) The pattern e.SALARY = EVOL(<) specifies that the attribute value at cursor time must be greater than the one at the previous cursor time. The same is true for e.LOCATION = EVOL(≠) except that the condition is “not equal” instead of “greater than.” The AND operator in between them specifies that both pattern must be matched at the same starting cursor time. Finally cursorJump operator attempts to find the first occurrence of the pattern.

Example 23. The query list the air pressure when the first time the temperature had dropped linearly in a 24hrs period in Des Moines, Iowa, since 12:00am 5/1/94 can be expressed as follows.

    select w.PRESSURE
        restricted_to [AT(12:00am 5/1/94)
            cursorJump[w.TEMPERATURE = EVOL(>_FUNCTION,HOURS(24))
                (_FUNCTION=LINEAR)]
        from weather_in_Iowa w /* spatio-temporal set object */
        where [[ w.CITY = "Des Moines"] ≠ Ø]

AT(12:00am 5/1/94) specifies that the cursor time must be (12:00am 5/1/94) when the matching begins. The pattern w.TEMPERATURE = EVOL (>_FUNCTION,HOURS(24)) (>_FUNCTION=LINEAR) specifies that the starting temperature must be greater than the ending temperature in a 24hrs period and the function in the period must be linear. Then cursorJump attempts to find the first occurrence of the pattern.

Example 24. The query list the air pressure when the first time the temperature had dropped linearly by at least 25°F in a 24hrs period in Des Moines, Iowa, since 12:00am 5/1/94 can be expressed as follows.
select w.PRESSURE
restricted_to [\[ AT(12:00am 5/1/94)
cursorJump [w.TEMPERATURE = EVOL (> FUNCTION _DIFF, HOURS (24))
:(FUNCTION = LINEAR and _DIFF >= 25) \]]
from weather_in_Iowa w /* spatio-temporal set object */
where [[ w.CITY = "Des Moines"]\] \| \| \ ن

The above expression is the same as the previous one except it requires that the tempera-
ture difference at the two end time instants must be at least 25°F.

Example 25. The query list all possible sites for biomass energy production in Iowa, subject
to the following criteria can be expressed as follows.

• not in a flood zone
• amount and location of resources in terms of per capita income
• extent of manure and other wastes from livestock operations

select s
restricted_to [[(s.FLOOD = very low NEAR suppressOperand [s.PER_CAPITA > $400])
NEAR suppressOperand [s.MANURE > 100]]
from states s /* spatial set object */
where [[s.STATE = "Iowa"]\] \| \| \ ن

In the above query the where clause specifies that the tuple for Iowa should be selected.
In the restricted_to clause, the spatial pattern specifies that the region should not be a flood
zone, be near to the household resources and manure resources from livestock operations.
Furthermore, the second and third subpatterns have the operator suppressOperand in front,
indicating that the regions for those two patterns are not included in the resulting domain.

Example 26. The query list potential sites for a nuclear plant in Nevada, having minimal
possibility of earth quake/flooding and very low population density within 50 miles can be
expressed as follows.

select s
restricted_to [[CIRCLE((s.EARTH_QUAKE = very low AND s.FLOOD = very low), 50)
CONTAINED_IN suppressOperand [s.POPU_DENS = very low]]
from states s /* spatial set */
where [[s.STATE = "Nevada"]\] \| \| \ ن
In the above query the where clause specifies that the tuple for Nevada should be selected. In the restricted_to clause, the region within 50 miles of where s.EARTH_QUAKE=very low AND s.FLOOD=very low is checked if it is contained in the region where s.POPU_DENS=very low. If so, the former region is returned.

**Example 27.** The query *find the population density during the last ten years for each state, within 50 miles of the area where there had been a strong earthquake ten years ago* can be expressed as follows.

```sql
select s.POPU_DENS
restricted_to [[ cursorResolution(YEAR) [ AT(NOW-10)
((s.EARTH_QUAKE=strong) =: _quake_zone)
scanNormal ALL 9 (CIRCLE(_quake_zone,50))] ]]
from states s /* spatio-temporal set object */
```

In the restricted_to clause, the cursorResolution(YEAR) operator sets the cursor resolution to year. AT (NOW-10) ensures the starting cursor time to be 10 years ago. Then in the spatial pattern ((s.EARTH_QUAKE = strong) =: _quake_zone), the region where s.EARTH_QUAKE = strong is retrieved and assigned to the variable _quake_zone. The loop pattern, scanNormal ALL 9 (CIRCLE(_quake_zone,50)), then computes the region with 50 miles of _quake_zone each year, for the next 9 years.

**5.7. Modeling continuous data**

In applications of spatio-temporal data one often comes across models which regard data as continuous, rather than discrete. Even discrete data can consist of multiple granularities. Sometimes we have discrete observations and continuous functions are fitted. These continuous functions are then supposed to represent the reality and they may be queried.

Note that observations are usually not queried directly. For example the temperature readings may be taken every three hours, but may be queried for any time of the day. Rainfall may be measured at some points in a country, but one can query for rainfall everywhere. Observations are almost certainly not relations in our sense. They may not incorporate the concept of a tuple object and a key. All different sets of observations may not be a good data-
In this section we introduce a language interface which can convert observations into views. A view is just like a set object, however it is virtual in the sense that it may not be stored and only its definition may be stored. From the point of view of querying a view is treated just as a stored relation.

We need to keep in mind the boundaries of databases when dealing with continuous data. To illustrate this, let us suppose that in a certain scientific application we are given two temporal attribute values $g_1$ and $g_2$, both continuously differentiable functions of time. Suppose in that application one needs to determine points where the two functions agree, e.g., $g_1(t) = g_2(t)$. It is well known that this problem is hard to solve in general; even in case $g_1$ and $g_2$ are polynomials in $t$, one has to resort to iterative approximation (e.g. see Newton's Method of Approximation in any calculus textbook). The point of this example is that such application specific problems have to be solved in the application disciplines and not databases. On the other hand a temporal database should provide a storage model for values such as $g_1(t)$ and $g_2(t)$, and allow desirable primitives to be integrated into a (query) language.

5.7.1. Incorporating continuous data

To incorporate these measurements and methods, we provide a construct `createView`.

Once the measurements are incorporated into a view, the view can be queried upon in the same way as other relations. Since the resolution of continuous data has a profound impact on the run time, accuracy and storage requirement of the result, the user can specify the resolution of the continuous data or function when creating the view. The general syntax and semantics of `createView` are given below.

```
createView view_name
select attr1:res1, attr2:res2, ..., attrn:resn [Key]
with attr1 = userMethod1(paraList1)
  ...
  attrn = userMethodn(paraListn)
restricted_to DimExp
from FromList
where BoolExp
```
The syntax of createView is similar to that of the select statement in ORDimSQL. There are several differences however.

- A new clause `createView` has been added. This clause provides the name of the view to be created. The attributes and the key of this view are listed in the select clause. The resolution for each attribute is user definable.

- A new clause `with` has been added which says how each attribute value in the select clause is to be computed: \( \text{attr} = \text{userMethod}(\text{paraList}) \): the new attribute \( \text{attr} \) is computed by the user supplied method \( \text{userMethod} \) with parameters in \( \text{paraList} \).

- \( \text{DimExp} \) and \( \text{BoolExp} \) are dimensional and boolean expressions respectively.

- FromList is list of set objects. The purpose of the from clause is to set up a cursor, a tuple variable for each set object. Note that the concept of such a cursor is formal in a model. It is implicitly based upon structure of a set object. The FromList cannot include observations. This is because a set of observations can have a very complex structure. Only the user methods can interpret and make use of these measurements properly. Thus no cursor construct is provided within the createView.

The semantics of createView is also very similar to that of the select statement. First a virtual literal cross product of set objects in FromList is formed. For each tuple object \( o \) in the cross product, the condition \( \text{BoolExp} \) is verified. If object does not satisfy \( \text{BoolExp} \), it is rejected. If the object satisfies \( \text{BoolExp} \), the new attributes \( \text{attr}_1: \text{res}_1, \text{attr}_2: \text{res}_2, \ldots, \text{attr}_n: \text{res}_n \) are computed with their corresponding user methods. Then all the attributes in the select clause are retrieved and its dimensional domain is restricted to \( \text{DimExp} \). If the resulting object is empty, it is rejected; otherwise it is retrieved. If necessary, the dimensional domain of the retrieved object is further contracted so that a retrieved object is homogeneous. Note that the new attributes are stored according to the resolutions specified. The key and the type of the view are determined in the same way as that of the select statement of Section 3.9.4.

**Example 28.** Assuming that \( \text{city} (\text{CNAME}) \) is a spatial set object and \( \text{pData} \) is a set of observations. Create a view for precipitation in each city where its area is greater than 500.
createView precipitation
select c.CNAME, PRECIP:10
with PRECIP = mPrecip(pData, \{c.CNAME\})
from city c
where AREA(\{c.CNAME\}) \geq 500

In this example, the precipitation for each city is extracted from the set of observation pData and stored with the resolution of 10. For each object c in the set object city, the function AREA(\{c.CNAME\}) calculates its area. If the result is greater than 500, the method mPrecip computes the precipitation level for the attribute PRECIP; otherwise, the object is rejected. The computation takes pData and the domain of c.CNAME as parameters, as illustrated in Figure 15.

![Figure 15. Creation of view in Example 28.](image)

**Example 29.** Assuming that city(CNAME) is a spatial set object and tData is a set of observations. Create a continuous temporal view for temperature for years [1990,1994] in cities in Iowa with a resolution of 1.0 day.

createView temperature
select c.CNAME, CTEMP: day(1.0)
with CTEMP = mTemp(tData, \{c\})
In this example, the temperature for each city is extracted from the set of observation tData and stored with the resolution of 10. For each object c in the set object city, mTemp computes the temperature for the attribute CTEMP. Unlike in Example 28, the view is restricted to the time domain year([1990,1994]). An example query on this view is given in Example 30.

**Example 30.** The query *list the temperature for each city, when the derivative of the temperature is zero* can be expressed as follows.

```sql
select t.*
restricted_to llDERW(t.DEGREE) = 0
from temperature t
```
6. CONCLUSIONS AND FUTURE WORKS

We have presented an object-relational model ORDimDB for uniform handling of dimensional data, of which spatial, temporal, spatio-temporal, and ordinary data are special cases. Different data dimensions are treated uniformly by the concept of dimension alignment. Our ORDimDB extends the relational model RelDimDB by incorporating object fragments and the type hierarchy. A query language ORDimSQL for spatio-temporal data is presented for associative navigation. Our ORDimSQL supports path expressions and places computed types appropriately in the existing type hierarchy to allow maximal use of existing methods. In this thesis, however, we do not consider natural join and full form of cross product. Certain issues on optimizing dimensional data have been discussed and a framework for algebraic optimization of the query language is suggested as a first step for further investigation. A pattern matching language is designed for complex querying of spatio-temporal data. Our pattern language is based on the concept of time cursor. We have shown how the temporal patterns and spatial patterns are constructed, and how spatio-temporal patterns can be formed by interleaving temporal patterns and spatial patterns. The pattern matching language recognizes special features of time and space providing an appropriate level of abstraction for application development compared to traditional languages. It also provides hooks for direct query of scientific data (observations). We now summarize the merits of our model as follows.

- Dimension alignment automatically allows lower dimensional data and queries to be used in a higher dimensional context, making our model dimensionally extensible.

- Our model ORDimDB seamlessly extends RelDimDB by incorporating object fragments and type hierarchy, where our query language ORDimSQL seamlessly extends RelDimSQL to support associative navigation through path expression.

- Key is factored out from the type hierarchy: it provides compatibility to classical model and allows identities such as \( O = I_K(I_K'(O)) \) to hold immediately in our model.
• Computed types are placed appropriately in the existing type hierarchy to allow maximal use of existing methods.

• Our pattern matching language seamlessly extends the associative navigation in our query language, and allows query of data with multiple granularities and continuous data.

• Temporal patterns and spatial patterns are special cases of spatio-temporal patterns. It allows migration to spatio-temporal model from either temporal or spatial model to be seamless. An advantage is that the user needs to know only one pattern matching language for a particular data dimension.

• Concepts of dimensionality and addition of oids and type hierarchy are mutually orthogonal. Thus starting from classical ordinary data, one may migrate to higher forms of relational or object-relational data in any sequence, without having to recode application software.

Currently our model is parameterization of classical Inf dbms, and complex objects need to be added to our model. We also plan to develop a full blown algebraic optimization, including a study of the complexity of dimensional expressions. As a first step, our pattern matching language is only for data with one time dimension and/or two spatial dimensions. In dimensional databases, however, data could have any number of time dimensions or space dimensions. Our future work is to develop a pattern matching language that can support data with any number of time/space dimensions. In active databases patterns could be used as triggers; therefore application of patterns in active databases should be investigated.

Seamlessness has been the main objective of our works. We have demonstrated that our model and query language can be extended substitutively without needing application software to be redesigned. This direction of thought should be reassuring to industry and user community as in our proposed framework, a gradual transition can be made to complex form of data.
BIBLIOGRAPHY


ACKNOWLEDGEMENTS

I would like to extend my most sincere thanks to my thesis advisor, Dr. Shashi K. Gadia, for his help during the course of this work. Not only did Dr. Gadia constantly provide me with direction and guidance for my thesis, he was readily available for any personal advice. I would also like to thank the other members of my Program of Study committee -- Dr. S. Kothari, Dr. L. Miller, Dr. A. Oldehoeft, Dr. M. Shelley, and Dr. J. Smith -- for their time and effort.

I also take this opportunity to thank my parents and especially my grandmother, who gave unconditional help whenever I was in need. I can never repay the debt I owe them. Finally, I like to express my sincere gratitude to my wife, Yuen Wong, for her patience and love during my study at Iowa State University.