An Extension of the Relational Data Model to Incorporate Ordered Domains

Wilfred Siu Hung Ng

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
of the
University of London

Department of Computer Science
University College London

1, July 1997

Abstract

In this thesis, we extend the relational data model to incorporate partial orderings into data domains and present a comprehensive formalisation of the extended model. The main contributions of the thesis are that we show how and why such an extension can considerably improve the capabilities of capturing semantics in a wide spectrum of advanced applications such as tree-structured information, temporal information, incomplete information, fuzzy information and spatial information, whilst preserving the elegance of the theoretical basis of the model.

Within the extended model, we extend the relational algebra to the Partially Ordered Relational Algebra (PORA) and the relational calculus to the Partially Ordered Relational Calculus (PORC), respectively. These two languages are shown to be equivalent. We then apply a generalized form of Paredaens' and Bancilhon's Theorem to justify our claim that the PORA is adequately expressive, i.e., non-uniformly complete. We also show that there is a one-to-one correspondence between three hierarchies of: computable queries, ordered domains and PORAs, according to the inherent structures of the underlying domains.

Moreover, we formally define Ordered Functional Dependencies (OFDs) and Ordered INclusion Dependencies (OINDs) for the extended model. We then present a sound and complete axiom system for OFDs and OINDs in the case of pointwise linear orderings. In addition, we establish a set of sound and complete chase rules for OFDs in the case of lexicographical linear orderings.

We extend SQL to OSQL as a query language for ordered databases. OSQL provides users with the capability to define partial orderings over data domains. In order to demonstrate the feasibility of OSQL, we have carried out an experimental implementation of the new language using the Oracle DBMS for low level data management. We have evaluated the implementation by conducting a user survey. From the results of the survey, we confirm that the essential features of OSQL which we have designed are easy to learn, understand and apply, and are useful in formulating queries involving order. Furthermore, we demonstrate that a wide range of queries in many advanced applications can be formualted in a unified manner by introducing the notion of an OSQL package.

Acknowledgements

Thanks to God. He allows me to possess the good health and the sound mind to complete this piece of demanding work.

Thanks to my supervisor, Mark Levene. He has given me the full attention and support since I started my Ph.D. programme three years ago. With his valuable advice and guidance on my work, I could carry out my research persistently even in the most difficult time. Also, I wish to express my gratitude to my second supervisor, Graham Roberts. He has given me many useful feedback at different stages.

Thanks to a number of people who have provided many helpful comments on individual chapters of this thesis. I would like to show my appreciation of their help by stating their names as follows, Ethan Collopy, Janet Mcdonnell, Nadav Zin, Tina Yu, Jonathan Poole, Alberto Souza and Adam Fagin.

Thanks to my wife, Marianne. She has never made any complaints with my anti-social working style during my thesis writing-up period. Furthermore, she contributed her best possible support and provided encouragement when giving up seems to be the best thing to do.

Thanks to the Department of Computer Science and the Graduate School at UCL. They have provided me with financial assistance to attend various workshops/conferences for presenting my papers.

Contents

List of Figures

Chapter 1

Introduction

This thesis describes an extension of the relational data model to incorporate partial orderings into data domains. The basic aim of the extension is to improve the capabilities of the model in capturing semantics of data. The thesis presents the theoretical investigation of the ordered relational model and the implementation of a minimal extension of SQL, called OSQL, which allows querying over ordered relational databases. We demonstrate that with such an extension and a package discipline, which allows a set of generic operations associated with a specific application to be grouped within a module, relational databases can considerably widen their applicability in the areas of tree-structured information [15], incomplete information [38], fuzzy information [17], temporal information [144] and spatial information[50].

In Section 1.1 we give some background material and motivation for defining the ordered relational model. In Section 1.2 we explain the goals of our research and the major contributions of the thesis. In Section 1.3 we briefly outline the main results obtained in the individual chapters of this thesis.

1.1 Background and Motivation

The development of DataBase Management Systems (DBMSs) has been a major research topic in computer science for nearly four decades. One of the most fundamental components in any information system is the database. In order to provide various facilities for users in an efficient manner, a database should be built according to a formal database model (or simply a data model) which defines its data structures, query languages and integrity constraints. Several data models have been proposed since the 1960s

[31, 10, 33, 30]. The most influential one is Codd's proposal for the relational data model [34]. Since then, database products have been developed in order to conform with this model and relational DBMSs have gradually come to dominate the commercial market. Relational database theory has been comprehensively developed during the past 27 years. The relational data model is unquestionably the most successful data model to date.

What factors make the relational model so successful ?

In our view, the above question could be answered by considering several different facets related to DBMSs. For example, the model provides a sharp and clear boundary between the logical and physical levels of DBMSs [7, 147]. Moreover, the relational data model offers the advantage of *physical data independence* to users, that is, changing the physical organisation of a database does not require alteration of its logical data structures. Other crucial factors to its success can be justified from three perspectives as follows:

1. From the point of view of *usability*, the model is natural and has a simple interpretation in terms of real world concepts. The essential data structure of the model is a relation, which can be visualised in a tabular format. Due to this simplicity, relational databases have gained acceptance from a broad range of users.

2. From the point of view of *applicability*, the model is flexible and general and can be used in many applications, especially in business-oriented ones such as accounting and payroll processing. As a result, the model has the advantage that it has gained popularity and credibility in a variety of application areas.

3. From the point of view of *formalism*, the model is elegant enough to support extensive research and analysis. Since the framework of the model is based on the well-established set-theoretic formalism, it facilitates better database theory research. Actually, it has inspired the development of many vital theoretical issues in databases such as query language and dependency theory, which have had a major impact on DBMS development.

As computing technology has been making steady progress, in the early 1980s there were new demands to apply database technology to handle different areas of applications such as computer aided design (CAD), image-processing, text retrieval, statistical databases and geographical analyses [2, 103]. Unfortunately, the relational data model is inadequate to meet these new demands since they usually require more structure in the data domains in order to capture their semantics than that provided in the standard model. Moreover, it is apparent that relational databases cannot be easily applied to the areas of tree-structured information, temporal information, incomplete information, fuzzy information and spatial information. The shortcomings of the relational data model resulted in two main trends of data model development in order to facilitate the better use of database technology.

One main trend in the development of data models, which began around 1986, was the attempt to combine the notion of an object into the data model [47, 102, 138], obviously as a result of the success of object-oriented programming in the 1980s. Object-oriented programming is commonly recognised as a powerful methodology to support the development and the maintenance of very large and complex applications. The use of the notion of objects in database systems seems interesting and promising, and thus following this direction, database researchers at that time opened a new realm of research concerning object-oriented databases. However, the research in the past decade involving the issues of incorporating objects into the data model indicated that it may be too optimistic regarding the success of the combination of objects and databases. For example, the commercial market for object-oriented databases has grown very slowly in terms of venture capitals and revenues ratio [140]. Some of the potential consumers of object-oriented databases such as CAD vendors have been slower than anticipated in using object database technology.

One difficulty for the object-oriented database paradigm is that the formalisation of object-orientation in the context of databases (c.f., see the interview of C.J. Date by Data Base Newsletter recorded in [44]) is not clearly understood and thus it is difficult to reach consensus. There are many interpretations of some fundamental concepts such as *object identity* and *object class* [25]. Although there are draft standards for an object language and a programming interface for object databases called ODMG-93 Release 1.2, which was released in 1996 [24], most object-oriented database vendors do not actually support all the features defined by the draft standard. There are still major differences between many claimed object-oriented databases in terms of query languages and programming interfaces. Moreover, there are many ways of allowing vendors to incorporate objects into database systems. Objects can be added to databases via an approach of radical transformation such as the systems Orion and O_2 [78, 46], which attempt to address all the features of objects, or a mild reform to the relational data model such as ADT-Ingres [139], which mainly extends the relational data model with user-defined abstract data types.

Another main trend is to develop specialised databases to cater for an individual class of application such as temporal and spatial databases. In contrast to object-oriented extensions, this approach does not maintain the spirit of finding an application independent data model as does the original relational data model. Some researchers even make further specialisation within the scope of a particular application. For example, in the area of spatial applications, some researchers have proposed a model for one kind of spatial information having two spatial dimensions and two temporal dimensions [153], whilst other researchers have proposed a model for another kind of spatial information having polygonal regions [71]. There are similar trends in the area of temporal databases [145, 98]. From the point of view of usability, this approach may quickly gain acceptance by the experts in the related discipline because they can easily understand the operations of the specialised database. In addition, the approach facilitates better understanding of the needs of the communities concerned. However, from the point of view of applicability, the disadvantage of this approach is that it loses the flexibility in adapting the model to other applications needing other specialised facilities.

We agree that some ideas from the object-oriented formalism such as *abstract data types* and *object encapsulation* can be very useful in the enhancement of the relational data model. However, we think that in the first place it is still necessary to clarify some fundamental concepts. We also recognise that the research into specialised databases can give deeper insights into the needs of the communities concerned. However, we think that the search for a unified model is necessary because it provides generic operations allowing the discovery of new possibilities in a wide variety of specialised areas. In this thesis we propose an extension of the relational model, in which we strive for a balance between these somewhat conflicting demands. On the one hand, our extended model unifies significant classes of different specialised applications. As a result, it provides a basis for the investigation of new possible applications. Moreover, it provides robustness and efficiency of the implemented generic operations. On the other hand, the extension we propose is as minimal as possible with respect to the relational data model, which may help to clarify the fundamental issue of user-defined data types in object-oriented databases, a key issue in the development of object-relational databases [140].

We bear in mind the successful factors of the relational data model that we have mentioned above. In our extension of the relational model we incorporate partial orderings into the data domains of the data model. There are several other reasons which motivate our extension. Firstly, there is strong evidence that ordering is inherent to the underlying structure of data in many database applications [20, 103, 130, 92, 129]. Furthermore, in many applications incomparability of data is a prominent phenomenon that must be captured explicitly. Secondly, the semantics of partial orderings is simple and well understood. A partially ordered domain serves as a bridge between an unordered domain and a linearly ordered domain. Thirdly, with this minimal extension we preserve the formal basis of the relational model which can be employed to study the effect of partial orderings on many well-known theoretical issues such as the expressiveness of query languages and the axiomatisation of new classes of data dependencies arising from the extension.

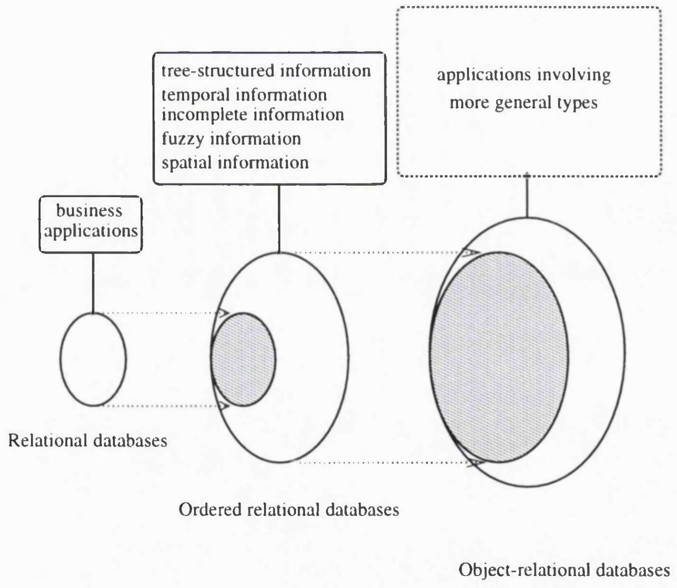

tree-structured information
temporal information
incomplete information
fuzzy information
spatial information
applications involving more general types
business applications
Relational databases
Ordered relational databases
Object-relational databases

Figure 1.1: Application areas with respect to extended relational databases

We use a highly simplified diagram that is shown in Figure 1.1 to illustrate how our extension relates to the current development of relational databases. On the one hand, the ordered relational data model is much stronger than the conventional model, since it is capable of capturing semantics in a wide spectrum of advanced applications. On the other hand, it captures an important part of object-relational databases, since ordered domains can be viewed as a general kind of type.

1.2 Main Goals of the Research and the Thesis Contribution

The main goal of this research is to investigate the effect of partial orderings on the data structures, query languages and integrity constraints of the relational data model. The scope of our investigation includes the examination of existing advanced applications related to partial orderings under the framework of ordered relational databases.

The main contribution of the thesis is to show how and why the extension of the relational data model to incorporate partial orderings into data domains can considerably improve the applicability of a DBMS. The ordered relational model is demonstrated to have the capability of capturing the semantics in a wide spectrum of advanced applications such as tree-structured information, temporal information, incomplete information, fuzzy information and spatial information, whilst preserving the elegance and simplicity of the conventional relational data model.

The potential benefits of the extension are threefold from the point of view of the development of relational databases.

1. The extension provides a viable alternative to other extensions of the relational data model. In practice, it can also be the optimised solution to an organisation somewhere between a specialised database, which may be too narrow a solution, and an object-oriented database, which may be too complex a solution.

2. As most major database vendors such as IBM, Informix and Oracle are putting their efforts in the direction of object-relational databases, our work on ordered domains serves as a good reference point for the development of the abstract data type facility for the object-relational database model.

3. We notice that in reality many large enterprises which use relational databases have just recently adapted to relational technology, and thus they may not be willing to take large commercial risks and shift to the paradigm of object-oriented databases. On the other hand, as long as object-relational databases are upwards

compatible they may accept such an extension. Our approach is a minimal extension of the relational model, which can reduce the risks of vendors in shifting to object-orientation and most important of all, there is convincing evidence of the benefits in our approach. Therefore, our work can provide significant impetus for the acceptance of object-relational database technology.

1.3 Outline of the Thesis

The thesis is divided into eight chapters, designated in the text by Chapter 1 to Chapter 8. We now give an overview of the thesis.

In this chapter we introduce the background and the overview of the extension of the relational model. The motivation, the objectives and the main contribution of the research have already been explained.

In Chapter 2 we formalise the ordered relational data model as an extension of the conventional relational data model to include partial orderings as an integral part of data domains. From the point of view of the standard three-level architecture of a DBMS, physical data independence includes the requirement that the physical ordering of data on a storage device cannot be accessed by users' application programs [34]. This can be achieved by the standard domain orderings at the logical level such as numerical orderings and alphabetical orderings provided by DBMSs. One important notion introduced here is that given a domain then, apart from the standard domain orderings, we can also declare new semantic orderings at the external level above the logical level, which override the standard domain orderings. The following example illustrates the use of semantic orderings in various domains.

Example 1.1 In Figure 1.2(a) we have the semantic domain EMP_RANK consisting of three employee names describing a hierarchy in a company of the employees Mark, Ethan and Nadav. The semantics are that both Ethan and Nadav are the subordinates of Mark. In Figure 1.2(b) we have the semantic domain SALARY_TIME consisting of four calendar years which simply follow the chronological ordering provided by a DBMS. Note that in this case we can use the standard system ordering. In Figure 1.2(c) we have the semantic domain INCOMPLETE which captures the semantics of different null values in a database which models various types of incomplete information. The known data value "programmer" is more informative than the null symbol UNK (UNKnown), and the null symbols UNK and DNE (Does Not Exist) are more informative than the null symbol NI (No Information), meaning either the data value does not exist or the data value exists but is not known. In Figure 1.2(d) we have another semantic domain called QUALIFY which captures the semantics of the fuzzy requirement "good science background" defined in a company.

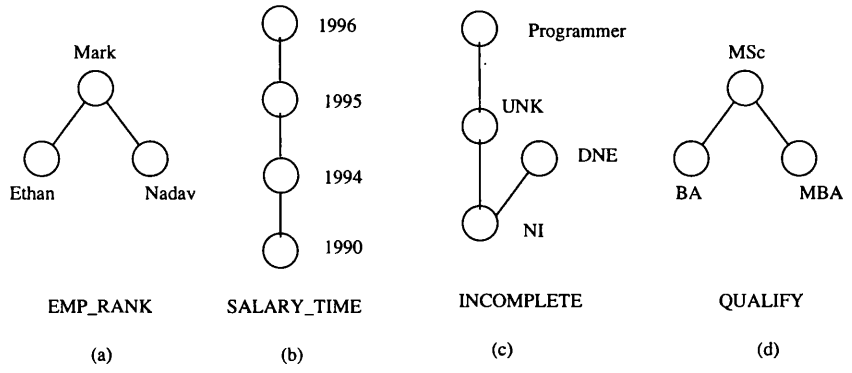


Figure 1.2: Using domains to capture the semantics in various information

In Chapter 3 we extend the relational algebra to the Partially Ordered Relational Algebra (the PORA) by allowing the ordering predicate , $\sqsubseteq$, to be used in the formulae of the selection operator (σ). Thereafter the relational calculus is extended to the Partially Ordered Relational Calculus (the PORC) in a similar manner. The PORA and the PORC are shown to be equivalent. This preserves the classical result of Codd's completeness, which is an important notion regarding the expressiveness of query languages proposed by Codd [35]. We then show that the PORA expresses exactly the set of all possible relations which are invariant under order-preserving automorphisms over databases. Informally, an order-preserving automorphism is a permutation of the values in the active domain of a database instance that does not alter the database and also preserves the ordering of the active domain. This preserves non-uniform completeness, which is an important notion concerning the expressiveness of query languages, also known as Paredaens' and Bancilhon's Theorem [123, 12]. Moreover, we investigate three hierarchies of: (1) computable queries, (2) query languages and (3) ordered domains, and show that there is a one-to-one correspondence between them. The implication of this result is that if the underlying data domains of an ordered database have more inherent structure, then a wider scope of queries is possible. In other words, the ordered relational model can provide more expressive query languages than those of the conventional one, and in this sense we can say that more meaningful queries are possible with respect to an ordered relational database.

In Chapter 4 we extend the notions of Functional Dependencies (FDs) [147, 9] and INclusion Dependencies (INDs) [109, 22] to be satisfied in an ordered database and call them Ordered Functional Dependencies (OFDs) and Ordered INclusion Dependencies (OINDs), respectively. FDs and INDs are commonly recognised as the most fundamental data dependencies that arise in practice. Informally speaking, OFDs can capture a monotonicity property between two set of attributes, and OINDs can capture the notion of a *Hoare ordering* [20] between two sets of tuples. For example, the Hoare ordering can represent the semantics that a relation is more informative than another relation. In the special case of an unordered set, the Hoare ordering simply reduces to a set inclusion. As an illustration of the new data dependencies we have mentioned, the OFD SALARY $\hookrightarrow$ SALARY_TIME over the relation EMP_DETAIL shown in Figure 1.3 states the fact that the SALARY of an employee increases with the SALARY_TIME. The OIND MANAGER[NAME, SALARY] $\hat{\subseteq}$ EMP_DETAIL[NAME, SALARY] states that the (complete or incomplete) information of the name and salary of a manager, represented by a relation MANAGER, should be consistent with and upper bounded by the information given by the relation EMP_DETAIL. For example, assuming the said OIND holds, then the tuple $\langle Mark, UNK\rangle$ is allowed in MANAGER, because it is consistent with and contains less information than the tuple $\langle Mark, 18K\rangle$ or the tuple $\langle Mark, 10K\rangle$ in EMP_DETAIL. However, we can check that the tuple $\langle Bill, UNK\rangle$ is not allowed in MANAGER because it is not consistent with any tuple in EMP_DETAIL.

NAME	SALARY	PREVIOUS_WORK	EDUCATION	SALARY_TIME
Ethan	12K	UNK	MSc	1994
Mark	10K	NI	MBA	1990
Mark	18K	NI	MBA	1996
Nadav	15K	Programmer	BA	1995

Figure 1.3: An employee relation EMP_DETAIL

We further classify OFDs and OINDs with respect to two kinds of orderings, namely, lexicographical and pointwise-orderings. Lexicographical orderings resemble the way in which words are arranged in a dictionary and pointwise-orderings require each component of a data value to be greater than its predecessors. For example, the tuple $\langle x_1, \dots, x_n \rangle$ is less than another tuple $\langle y_1, \dots, y_n \rangle$ according to a lexicographical ordering if there is an index $j \geq 1$ such that $x_j < y_j$ and for each $i < j$, $x_i = y_i$. The tuple $\langle x_1, \dots, x_n \rangle$ is less than another tuple $\langle y_1, \dots, y_n \rangle$ according to a pointwise-ordering if for all $1 \leq i \leq n$, $x_i \leq y_i$. We present sound and complete axiom systems for OFDs and OINDs, respectively, in the case of pointwise-orderings. In relational database theory, the *chase* is a fundamental theorem proving tool, whose main uses have been testing implication of data dependencies [101] and testing consistency of a relational database with respect to a set of data dependencies [61, 86, 87]. The intuitive idea behind the chase is that we start with the hypothesis that a relation in a generalised form satisfies a set of data dependencies F. Suppose we want to test whether a data dependency f follows from F. We then apply the chase rules with respect to F to "chase it down" for all the consequences of F that occur in this relation. If we can finally reach a state of the relation that represents the conclusion of f, then we have a proof that f follows from F. If we fail to draw the desired conclusion, the relation that results when we finish the chase is a counter example, i.e., it satisfies F but not f. We present a set of sound and complete chase rules for OFDs in the case of lexicographical orderings. The chase rules are also important for studying OINDs in the case of lexicographical orderings. The axiom systems we present provide us with a useful tool to infer additional data dependencies from a given set of OFDs or OINDs and the chase rules form the basis of both a theorem prover and an inference engine for OFDs and OINDs.

In Chapter 5 we describe OSQL, which is an extension of SQL for the ordered relational model, and show that OSQL combines the capabilities of standard SQL with the power of semantic ordering. Using OSQL users have the ability to define partial orderings over data domains which are implied by the underlying semantics of the data of an application. We also discuss the issues concerning the implementation of OSQL, which has been prototyped using Oracle for low level data management. The following running example demonstrates how OSQL can be applied to solve various problems that arise in DBMSs involving applications having tree-structured information, incomplete information, fuzzy information and temporal information under the unifying framework of the ordered relational model.

Example 1.2 Let us consider the relation EMP_DETAIL again shown in Figure 1.3.

- Tree-structured Information:

  Using the semantic domain EMP_RANK shown in Figure 1.2(a) for the attribute NAME, we can formulate the query of finding the name and salary of the common bosses of Nadav and Ethan as follows.

  ($Q_{1.1}$) *SELECT* (∗) (∗)
  FROM EMP_DETAIL
  WHERE (NAME > 'Nadav' *WITHIN* EMP_RANK)
  AND (NAME > 'Ethan' *WITHIN* EMP_RANK).

  With some knowledge of standard SQL, one can understand the meaning of the query $Q_{1.1}$ quite easily. The first clause (∗) after *SELECT* means that all attributes are selected and the second clause (∗) means that all tuples are selected. The keyword *WITHIN* specifies that the comparison NAME > 'Nadav' is interpreted according to the semantic ordering of the domain EMP_RANK.

- Temporal Information:

  We assume that SALARY_TIME is a time attribute whose values are timestamps of the tuples in the relation EMP_DETAIL (for simplicity in presentation, we also assume that the time_stamping denotes *valid time* [145]). For instance, we can see that Mark has salary 10K in 1990 and his salary increased in 1996. Note that we do not record Mark's salary if there had been no change since the year it was last updated. We can use the keyword LAST to find the last time the tuple was updated, since the domain of the attribute SALARY_TIME is linearly ordered as is shown in Figure 1.2(b). With the following query, we show how to find Mark's salary in 1993.

  ($Q_{1.2}$) *SELECT* (SALARY_TIME, SALARY) (*LAST*)
  FROM EMP_DETAIL
  WHERE NAME = 'Mark'
  AND SALARY_TIME <= 1993.

- Incomplete Information:

  Using the domain INCOMPLETE as shown in Figure 1.2(c) for the attribute PREVIOUS_WORK, we can formulate the query which finds the name and previous work of those employees whose previous work is more informative than NI, as follows.

($Q_{1.3}$) *SELECT* (NAME, PREVIOUS_WORK) (*)
FROM EMP_DETAIL
WHERE (PREVIOUS_WORK > 'NI' *WITHIN* INCOMPLETE).

- Fuzzy Information:

  Using the semantic domain QUALIFY shown in Figure 1.2(d), we can formulate the query of finding the name of an employee with "good science background" by way of academic qualification as follows.

  ($Q_{1.4}$) *SELECT* ((EDUCATION *WITHIN* QUALIFY), NAME) *DESC* (1)
  FROM EMP_DETAIL.

  The keyword *DESC* specifies that the employees in EMP_DETAIL are sorted according to the semantic ordering of the domain QUALIFY with the suitable ones put first.

In Chapter 6 we report on a survey that we have carried out in the Department of Computer Science at University College London. The main objective of the survey was to gain more insights into the usages and the acceptance of the extended features of the *attribute list*, the *tuple list* and the *comparison clause* of the OSQL SELECT command. We invited 70 students and 10 computer professionals to participate in an experiment to compare their performance between using OSQL and SQL in formulating a set of queries involving order. We also requested them to fill in a questionnaire for feedback purposes. From the attitudes of the subjects towards the use of OSQL, we can determine whether the various proposed features of OSQL are appropriate for database programmers to learn and apply. Our first finding is that the subjects in the OSQL survey formulated difficult queries involving order in an easier manner than in SQL. Our second finding is that the subjects confirmed that the extended features are easy to learn, understand and apply. From the subjects' feedback in the questionnaire, we also realise that a more comprehensive implementation of OSQL, which includes a custom built user interface, is needed so that further experiments can be carried out to test the viability of the OSQL extension to SQL.

In Chapter 7 we introduce the notion of an OSQL package, which informally is a collection of generic operations over an ordered domain. The incorporation of a package discipline into OSQL enhances the expressiveness of OSQL by utilising the capabilities of OSQL in a more systematic manner. We define in detail a variety of generic operations with respect to the mentioned advanced applications and classify them into four OSQL packages: OSQL_TREE, OSQL_TIME, OSQL_INCOMP and OSQL_FUZZY. Using these packages, we now demonstrate how the above mentioned queries can be formulated in a simpler manner by embedding the generic operations of these OSQL packages into OSQL.

Using the package OSQL_TREE, the query ($Q_{1.1}$) can be simplified as follows:

> ($Q_{1.5}$) *SELECT* (∗) (∗) *FROM* EMP_DETAIL
> *WHERE* NAME *IN* COM_ANCESTOR('Nadav', 'Ethan').
>
> The operation COM_ANCESTOR in ($Q_{1.5}$) returns the names of all common bosses of Nadav and Ethan.

Using the package OSQL_TIME, the query ($Q_{1.2}$) can be simplified as follows:

> ($Q_{1.6}$) *SELECT* (SALARY) (∗) *FROM* SNAPSHOT(EMP_DETAIL, 1993)
> *WHERE* NAME = 'Mark'.
>
> The operation SNAPSHOT in ($Q_{1.6}$) returns the employee records in 1993.

Using the package OSQL_INCOMP, the query ($Q_{1.3}$) can be simplified as follows:

> ($Q_{1.7}$) *SELECT* (NAME, PREVIOUS_WORK) *FROM* EMP_DETAIL
> WHERE MORE_INFO(PREVIOUS_WORK, 'NI').
>
> The operation MORE_INFO in ($Q_{1.7}$) checks whether the previous work of an employee is more informative than the null symbol 'NI'.

Using the package OSQL_FUZZY, the query ($Q_{1.4}$) can be simplified as follows:

> ($Q_{1.8}$) *SELECT* (IMPOSE_FUZZY(EDUCATION, QUALIFY), NAME) (1) *FROM*
> EMP_DETAIL.
>
> The operation IMPOSE_FUZZY in ($Q_{1.8}$) returns the most appropriate tuple such that it satisfies the imposed fuzzy requirement "good science background" by way of academic qualification.

In addition to the four packages mentioned above, we also develop the package OSQL_SPACE for handling spatial information for the special case of rectangular regions, which is one of the fundamental geometric regions in developing spatial databases [60, 98]. We assume that MIN_VERTEX and MAX_VERTEX are two spatial attributes used to specify a rectangular region. The following query shows how to use OSQL_SPACE to find Bill's neighbours. The operation PICK_REGION in ($Q_{1.9}$) is a graphical interface operation which converts the mouse pointed region on the screen (we assume it is Bill's room) to its corresponding spatial attributes. The MEET operation is one of the eight *Egenhofer-Franzosa topological relationships* [49].

($Q_{1.9}$) *SELECT* (OCCUPANT) (*) *FROM* FLOOR_PLAN
    WHERE MEET(MIN_VERTEX, MAX_VERTEX, PICK_REGION()).

In Chapter 8 we conclude our work with some final remarks and discuss future work resulting from the ordered relational model.

Four appendices which are relevant to OSQL are attached to the end of the thesis for the purpose of reference. In Appendix A we present the full reference of the BNF for OSQL. In Appendix B we give the declaration of all the OSQL packages discussed in Chapter 7 and their operations. In Appendix C we include the full documentation of the survey detailed in Chapter 6. In Appendix D we give a sample of the C code which implements OSQL.

Chapter 2

The Ordered Relational Model

In this chapter, we extend the conventional relational model and give the background and preliminary material needed throughout the thesis. In particular, the relational data model is extended to incorporate partial orderings into data domains.

In Section 2.1 we clarify the notions of order [65, 59, 133] and its relevance to the data domains used in existing information systems. In Section 2.2 we formally extend the relational data model [34] to include partial orderings into the structure of the model. In Section 2.3 we discuss the effect of orderings on the three DBMS levels of the conventional model [7, 147, 4]. In Section 2.4 we compare the features of our extension with the conventional relational data model. In Section 2.5 we review related work that has recognised the importance of ordering as a fundamental property in data modelling.

In the sequel, we employ the following mathematical notation for sets. Let S and T be sets, then $\mid S \mid$ denotes the *cardinality* of S, $S \subseteq T$ denotes *set inclusion*, $S \subset T$ denotes *proper set inclusion* and $\mathcal{P}(S)$ denotes the *finite powerset* of S. We denote the k term Cartesian product $S \times S \cdots \times S$ by S^k, and the singleton $\{A\}$ simply by A when no ambiguity arises. We also let *id* be the identity mapping on any set. A partially ordered set is depicted by a Hasse diagram [59] which is a graph where each node is an element of the base set, and each edge connects two distinct comparable elements such that one element is either an *immediate successor* or *immediate predecessor* [65] of another.

2.1 The Notion of Order

In this section we present the basic concepts and terminology of partial orderings. In Subsection 2.1.1, we define partial orderings and some important special cases. In Subsection 2.1.2, we introduce the extensions of orderings to a Cartesian product and a powerset, which are fundamental in the orderings of data domains.

2.1.1 Formal Definition of Partial Orderings

Definition 2.1 (Partial Ordering) A *partial ordering of the set S* is a binary relation on S, denoted by $\sqsubseteq$, satisfying the following conditions.
For all $x, y, z \in S$,

1. *Reflexivity*: $x \sqsubseteq x$.

2. *Anti-symmetry*: If $x \sqsubseteq y$ and $y \sqsubseteq x$, then $x = y$.

3. *Transitivity*: If $x \sqsubseteq y$ and $y \sqsubseteq z$, then $x \sqsubseteq z$.

There are two important special cases of partial orderings in our context, *linear orderings* and *unordering.* Informally, given a set S related by a partial ordering, in the former case all elements are related in a chain and in the latter case no two distinct elements are related. We formally define these two extreme cases as follows.

Definition 2.2 (Linear Ordering and Unordering) A *linear ordering* of the set S is a partial ordering $\sqsubseteq$, if it satisfies the *linearity* condition as follows, for all $x, y \in S$, $x \sqsubseteq y$ or $y \sqsubseteq x$. We denote this special case by $\leq$. An *unordering* of the set S is a partial ordering $\sqsubseteq$, if it satisfies the *incomparability* condition as follows, for all $x, y \in S$, if $x \neq y$, then neither $x \sqsubseteq y$ nor $y \sqsubseteq x$. In other words, when each element is only comparable with itself, S is unordered. We denote this special case by $=$, since $\sqsubseteq$ is just the equality predicate $=$.

We denote that x and y are incomparable by $x \parallel y$ and that $x \sqsubseteq y$ but $x \neq y$ by $x \sqsubset y$. Note that for any elements x and y in S, if $x \neq y$, then exactly one of the following holds: $x \sqsubseteq y$, $y \sqsubseteq x$, or $x \parallel y$. Furthermore, there is a very interesting case of linear orderings, *dense linear orderings*, which are very often used in topological spaces [122]. For example, a point object in a two Euclidean dimensional space can be represented as a pair of real values whose domains are densely linearly ordered.

Definition 2.3 (Dense Linear Ordering) A linear ordering $\leq$ is said to be *dense*, if for any two distinct elements $x, y \in S$ such that $x \leq y$, there is an element $z \in S$ which is distinct from x and y such that $x \leq z$ and $z \leq y$.

A *partially ordered set* (or simply an ordered set), denoted as $\mathcal{S}$, is a structure $\langle S, \sqsubseteq \rangle$. It consists of a set S which is partially ordered (or simply ordered) by the relation $\sqsubseteq$. In particular, the structure $\langle S, \leq \rangle$ is called a *linearly ordered set* and the structure $\langle S, = \rangle$ is called an *unordered set*. From now on, the term *ordered* will mean partially ordered, unless explicitly stated to the contrary.

Definition 2.4 (Subordering) For two ordered sets $\langle T, \sqsubseteq_T \rangle$ and $\langle S, \sqsubseteq_S \rangle$ satisfying $T \subseteq S$ and for all $a_1, a_2 \in T$, $a_1 \sqsubseteq_T a_2$ if and only if $a_1 \sqsubseteq_S a_2$, we call $\mathcal{T}$ a *subordering* of $\mathcal{S}$. In this case we may write $\langle T, \sqsubseteq_T \rangle$ as $\langle T, \sqsubseteq_S \rangle$.

Example 2.1 Let N and R be the set of natural numbers and real numbers, respectively. $\langle N, \leq \rangle$ and $\langle R, \leq \rangle$ with their usual ordering $\leq$ are typical examples of linearly ordered sets, in which R is a dense linear ordering. The orderings of natural numbers and real numbers are collectively called *numerical orderings*, which is an essential property of the primitive domains in existing database systems. A set of object names is an unordered set (assuming we choose to interpret alphabets without lexicographical orderings). The finite powerset $\langle \mathcal{P}(S), \subseteq \rangle$ is a partially ordered set.

2.1.2 Extension of Orderings

It is very desirable to define two extensions of the orderings of data domains in order to capture the semantics of data. One extension is on the Cartesian product of ordered sets. Another extension is on the powerset of an ordered set. We first discuss *lexicographical ordering* and *pointwise-ordering*, which are two common kinds of orderings on the Cartesian product of ordered sets. Then we discuss various kinds of orderings arising from the powerset of an ordered set.

Let $S_1, \ldots, S_n$ be n ordered sets, t be an element in the Cartesian product $S = S_1 \times \cdots \times S_n$ and $t[i]$ be the ith coordinate of t. We now define *lexicographical orderings* on the Cartesian product of ordered sets.

Definition 2.5 (Lexicographical Ordering) For all t_1, $t_2 \in S$, $t_1 \sqsubseteq t_2$ if either

1. there exists k with $1 \leq k \leq n$ such that $t_1[k] \sqsubset_{S_k} t_2[k]$, and for all $1 \leq i < k$, $t_1[i] = t_2[i]$, or
2. for all $1 \leq i \leq n$, $t_1[i] = t_2[i]$.

For example, we can construct the lexicographical ordering on N^n, which is an infinite lexicographical ordering. Another important example is the lexicographical ordering on alphabets. Let A be a linearly ordered set over a finite alphabet. Then we can easily construct a finite lexicographical ordering on A^n in the same way as N^n, which we call a *dictionary ordering* or an *alphabetical ordering*, since it resembles the ordering of words in a dictionary. Note that A^n is just a subset of the set of possible strings formed over A (those string of length n). The usual meaning of the domain $CHAR(n)$ in a DBMS should be interpreted by $CHAR(n) = \bigcup_{i=1}^{n} A^n$, and the lexicographical ordering on $CHAR(n)$ should be extended as follows, for any $x, y \in CHAR(n)$, $x \leq y$ if and only if either x is a prefix of y or x and y have a longest common prefix u such that $x = uv$, $y = uw$ and $head(v) \leq head(w)$, where the operator $head(u)$ returns the first character of a given string u.

Example 2.2 Let $A = \{a \leq b\}$, then $CHAR(2)$ forms the following ordering:

$$a \leq aa \leq ab \leq b \leq ba \leq bb.$$

Another common way to form an ordering on $CHAR(n)$, which we denote by $\leq_\alpha$, is to use a combination of length and the lexicographical ordering of a string. The ordering $\leq_\alpha$ is defined such that for any $x, y \in CHAR(n)$, $x \leq_\alpha y$ if and only if either $length(x) < length(y)$, or $length(x) = length(y)$ and $x \leq y$. The operator $length(u)$ returns the number of occurrences of characters in a given string u. We use the set $CHAR(2)$ in Example 2.2 to illustrate this ordering as follows:

$$a \leq_\alpha b \leq_\alpha aa \leq_\alpha ab \leq_\alpha ba \leq_\alpha bb.$$

We note that the ordering of the domain *DATE*, called *chronological orderings*, can be viewed as the lexicographical ordering of the domains $YEAR$, $MONTH$ and DAY, if (1) the domain $MONTH$ has the ordering as $\{JAN \leq FEB \leq \cdots \leq DEC\}$ and (2) the Cartesian product of the domains are taken in the following order: $YEAR \times MONTH \times DAY$.

We next define another kind of ordering, *pointwise-ordering*, on the Cartesian product of ordered sets.

Definition 2.6 (Pointwise-Ordering) For all t_1, $t_2 \in S$, $t_1 \sqsubseteq_S t_2$ if for all $1 \leq i \leq n$, $t_1[i] \sqsubseteq_{S_i} t_2[i]$.

For example, assume that a domain of constants, denoted as *Dom*, contains a distinguished symbol UNK, which means the data value exists but is UNKnown. We define a partial ordering in *Dom* as follows, for all $x, y \in Dom$, $x \sqsubseteq y$ if $x = y$ or $x =$ UNK. Then we can extend $\sqsubseteq$ to be a pointwise-ordering in a relation r over $\{A, B\}$ as follows, for all $t_1, t_2 \in r$, $t_1 \sqsubseteq t_2$ if $t_1[A] \sqsubseteq t_2[A]$ and $t_1[B] \sqsubseteq t_2[B]$. This extension naturally captures the meaning of t_1 being *less informative* than t_2, or alternatively t_2 being *more informative* than t_1. Actually, the relationship between incompleteness and orderings has been commonly used in studying the issues concerning incomplete information [154, 91, 84, 87, 88].

It is apparent that the notion of less informative can be further extended from tuples to relations in a similar manner. However, Buneman in [20] recognises that there are three possible extensions to the powerset of an ordered set, namely, *Hoare orderings* [62], *Smyth orderings* [142] and *Plotkin orderings* [126], all of which are essential to the semantics of incomplete information in databases. The notion of one relation r_1 being less informative than another relation r_2 can be captured by these three orderings given in Definition 2.7.

Definition 2.7 (Hoare, Smyth and Plotkin Orderings) Let $X, Y \in \mathcal{P}(S)$.

1. A *Hoare ordering*, denoted by $X \sqsubseteq^{\flat} Y$, is defined as for all $x \in X$, there exists $y \in Y$ such that $x \sqsubseteq_S y$.

2. A *Smyth ordering*, denoted by $X \sqsubseteq^{\sharp} Y$, is defined as for all $y \in Y$, there exists $x \in X$ such that $x \sqsubseteq_S y$.

3. A *Plotkin ordering*, denoted by $X \sqsubseteq^{\natural} Y$, is defined as $X \sqsubseteq^{\flat} Y$ and $X \sqsubseteq^{\sharp} Y$.

In the special case of unordered sets, Hoare and Smyth orderings become subsets and supersets, respectively. Thus, we can view a Hoare ordering as a generalised subset ordering and a Smyth ordering as a generalised superset ordering. Informally speaking, Hoare orderings can capture the concept that one relation contains more information than another. For example, the number of tuples in a temporal relation increases as time passes or the *union* of two relations carries the resulting relation to a "higher" order. On the other hand, Smyth orderings can capture the intuition that a relation in a "higher" order is a more precise description of a set of real world objects. For example, when querying a database, the answer obtained from the operations *natural join* and *selection* [147], obtains a more precise description than in the database. Let us give an example to illustrate Definition 2.7.

Example 2.3 In Figure 2.1, we show examples of Hoare, Smyth and Plotkin orderings. Obviously, all the tuples in r_1 are also in r_2 and thus we have $r_1 \sqsubseteq^{\flat} r_2$. Moreover, all the tuples in r_1 are more informative than the tuple $\langle UNK, CS, 30K \rangle$ in r_3. So we have $r_3 \sqsubseteq^{\sharp} r_1$. Finally, we have $r_3 \sqsubseteq^{\natural} r_2$, since it satisfies that $r_3 \sqsubseteq^{\flat} r_2$ and $r_3 \sqsubseteq^{\sharp} r_2$.

$r_1 =$

NAME	DEPT	SALARY
Bill	CS	30K
Mark	CS	30K

(a)

$r_2 =$

NAME	DEPT	SALARY
Bill	CS	30K
Mark	CS	30K
Ethan	EE	26K
Nadav	EE	26K

(b)

$r_3 =$

NAME	DEPT	SALARY
UNK	CS	30K
UNK	EE	26K

(c)

Figure 2.1: Examples of Hoare, Smyth and Plotkin orderings

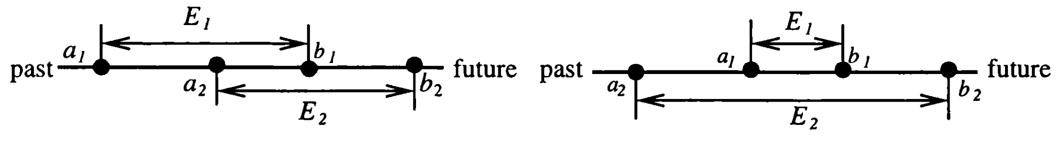
There are two extensions of ordering on the powerset of a linearly ordered set, which are especially useful in temporal domains. Let S be a linearly ordered set. We denote an interval I as $[a, b)$ where $a < b$ and define I by $I = \{t \in S \mid a \leq t < b\}$. Let *Int* be the collection of all finite intervals over S. We give the definition of *precedence orderings* and *containment orderings* as follows:

Definition 2.8 (Precedence and Containment Orderings) Let $I_1 = [a_1, b_1)$, $I_2 = [a_2, b_2) \in Int$.

1. A *precedence ordering*, denoted as $I_1 \sqsubseteq_{prec} I_2$, is defined as $a_1 \leq a_2$ and $b_1 \leq b_2$.

2. A *containment ordering*, denoted as $I_1 \sqsubseteq_{cont} I_2$, is defined as $a_2 \leq a_1$ and $b_1 \leq b_2$.

In temporal databases, we commonly use a time interval to time-stamp a tuple in a temporal relation and thus each interval can be used for specifying the period of an event in a temporal database. In Figure 2.2(a), we show the fact that the period of an event E_1 being specified by the interval I_{E_1} precedes the period of another event E_2 being specified by the interval I_{E_2} can be captured by $I_{E_1} \sqsubseteq_{prec} I_{E_2}$. Similarly, in Figure 2.2(b), we show that the fact that the period of an event E_1 is within the period of another E_2 can be captured by $I_{E_1} \sqsubseteq_{cont} I_{E_2}$.



(a) Precedence ordering (b) Containment ordering

Figure 2.2: Examples of precedence and containment orderings

2.2 Orderings in Databases

In this section we extend the relational data model to incorporate ordered domains. Within the extended model, we define ordered databases.

Let D be a countably infinite set of constant values and $\sqsubseteq_D$ be an ordering on D. Without loss of generality, we assume that all attributes share the same domain D. We now give the definition of an ordered database.

Definition 2.9 (Attributes and Ordered Domains) We assume a countably infinite linearly ordered set of attribute names, $\langle U, \leq_U \rangle$. For all attributes $A \in U$, the domain of A is $\langle D, \sqsubseteq_D \rangle$. We call $\sqsubseteq_D$ the *domain ordering* of D.

Definition 2.10 (Relation Schema and Database Schema) A relation schema (or simply a schema) R, is a subset of U consisting of a finite set of attributes $\{A_1, \ldots, A_m\}$

for some $m \geq 1$. A *database schema* is a finite set $\mathbf{R} = \{R_1, \ldots, R_n\}$ of relation schemas, for some $n \geq 1$.

Definition 2.11 (Tuple and Tuple Projection) Let $X = \{A_1, \ldots, A_m\}$ be a finite subset of U where $A_i \neq A_j$ for $i \neq j$ and $A_1 \leq_U \cdots \leq_U A_m$. A *tuple* t over X is a member of D^m. We let $t[A_i]$ denote the ith coordinate of t. The *projection* of a tuple t onto a set of attributes $Y = \{A_{i_1}, \ldots, A_{i_k}\}$, where $1 \leq i_1 < \cdots < i_k \leq m$, is the tuple $t[Y] = \langle t[A_{i_1}], \ldots, t[A_{i_k}] \rangle$.

Definition 2.12 (Ordered Relation and Ordered Database) An *ordered relation* (or simply a relation) r defined over a schema R is a finite set of tuples over R. An *ordered database* (or simply a database) over $\mathbf{R} = \{R_1, \ldots, R_n\}$ is a finite set $d = \{r_1, \ldots, r_n\}$ such that each r_i is a relation over R_i. We call r and d an *unordered relation* and an *unordered database*, respectively, if the underlying domain $\langle D, \sqsubseteq_D \rangle$ is unordered, i.e., it is $\langle D, = \rangle$. Similarly, we call r and d a *linearly ordered relation* and a *linearly ordered database*, respectively, if the underlying domain is linearly ordered.

Therefore, we can view a conventional database as a special case of ordered databases with unordered domains. We compare the similarities and differences between various essential properties of conventional relations and ordered relations in the table given in Figure 2.3.

Conventional Relations	**Ordered Relations**
No duplicate tuples are allowed.	Same as left.
Tuples are unordered.	Tuples are ordered according to the extension of the domain ordering.
Attributes in a schema are unordered.	Attributes in a schema are linearly ordered.
All domain elements are *atomic.*	Same as left but note the correct interpretation of such a phrase in Section 2.4.

Figure 2.3: Comparison between conventional relations and ordered relations

2.3 Orderings and Data Independence

Although ordering is a fundamental property of almost all primitive data types, existing database theory usually makes an implicit assumption that domains are either linearly ordered or unordered and thus the former case allows the less than predicate, $\leq$, to be used in selection formulae [147, 9]. In practice, all relational database systems support only the following three kinds of domain orderings considered to be essential in practical utilisation: (1) the *alphabetical ordering* over the domain of strings, (2) the *numerical ordering* over the domain of numbers, and (3) the *chronological ordering* over the domain of dates [41]. Let us call these orderings the *standard domain orderings.* There is strong evidence that ordering is inherent to the underlying structure of data in many database applications [20, 103, 130, 92, 129, 115, 116] and therefore the limited support of domain orderings results in loss of semantics of data. We call the ordering semantics in the context of a specific application *semantic orderings*, which will be addressed in detail in Chapter 5.

The ordering of tuples in a relation is useful information needed by the aggregate function *ORDER BY*, and thus in practice should be provided by the DBMS. We let a *system ordering*, denoted by $\leq_{sys}$, on a relation r be a linear ordering on r that is generated by a DBMS. Note that the concept of system orderings and domain orderings are different. The ordering $\leq_{sys}$ may or may not follow the extension of domain orderings on tuples due to the fact that different DBMSs have their own storage and retrieval strategy. The following example helps further to clarify this concept.

Example 2.4 Let r over $\{A\}$ be the relation $\{a, b, c\}$ (having 3 tuples) and the domain of A be alphabetically ordered. A system ordering, which is dependent on a particular DBMS, can take one of the six ways to arrange the tuples over r given in Figure 2.4.

A
a
b
c

A
a
c
b

A
b
a
c

A
b
c
a

A
c
a
b

A
c
b
a

Figure 2.4: Six possible system orderings of tuples in r

Although in most cases the choice of the ordering of r in the above example is done according to standard domain orderings (i.e., the first one in Figure 2.4), the ordering of tuples cannot be guaranteed to be alphabetically ordered if r is the answer to a complex query over the DBMS. This is because the choice of ordering of r is dependent on the implementation of a particular DBMS. It is worthwhile to consider how $\leq_{sys}$ affects the use of *cursors* in an embedded SQL statement [43]. For example, the result of selecting the nth tuple of r is dependent on the ordering of tuples in r. In such case there will be a risk of losing *physical data independence*, due to the fact that the returned tuples depend on $\leq_{sys}$, which in turn depends on the implementation of the system. This is rather undesirable and thus the current remedy is that we need to use the function *ORDER BY* to help "position" tuples when declaring a cursor (c.f., see chapter 10 in [43]). In other words, we need domain orderings to achieve physical data independence. We show in Figure 2.5 the differences between the various notion of orderings introduced so far.

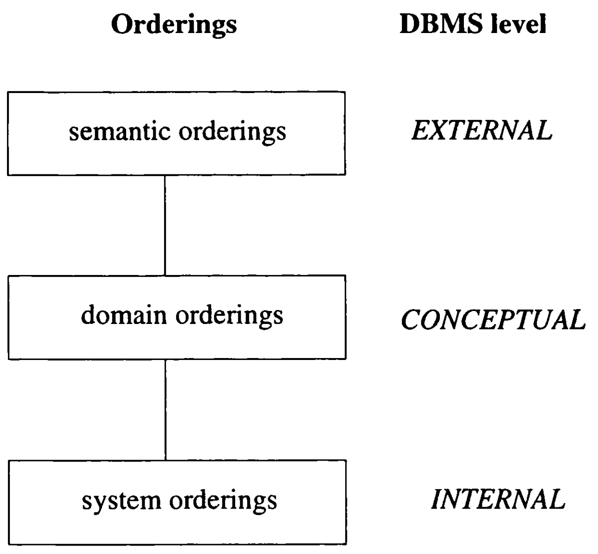


Figure 2.5: Orderings at different DBMS levels

We now define an operator called a *domain ordering operator* whose aim is to help present the relationship between domain orderings and data dependencies.

Definition 2.13 (Domain Ordering Operator) Let r be a relation over R and X be a sequence of attributes which is a subset of R. Then a domain ordering operator over r, denoted by ω_X, is defined as the set of linear orderings $\mathcal{L}$ on r such that for each $\leq_r \in \mathcal{L}$ and for any tuples $t_1, t_2 \in r$, if $t_1[X] \leq_X t_2[X]$, then $t_1 \leq_r t_2$ (recall that $\leq_X$ is a lexicographical ordering on the Cartesian products of data domains associated with X).

Furthermore, we call ω_X SOI (System Ordering Independent) if $\mathcal{L}$ is a singleton.

Example 2.5 Let $D_1 = \{1, 2\}$ and $D_2 = \{a, b, c\}$ and a given relation $r = \{\langle 2, a\rangle, \langle 1, c\rangle, \langle 1, b\rangle\}$. Then the following ordered relations given in Figure 2.6 exhibit two different domain orderings in $\omega_{A_1}(r)$, because there are two choices of ordering for the tuples $\langle 1, b\rangle$ and $\langle 1, c\rangle$ by the system.

A_1	A_2
1	b
1	c
2	a

A_1	A_2
1	c
1	b
2	a

Figure 2.6: Two possible domain orderings of tuples on r

Thus we can understand from the above example that the ordering of r is still partially system dependent although it is ordered according to the domain ordering of A_1 only. It is also clear that if X is equal to the schema of r, then ω_X should be SOI. If X is a proper subset of the schema of r, then it is desirable for ω_X to be SOI, since we can save some computation resources of the system to achieve the independence of system orderings. This is because the system does not have to perform the sorting over every attribute in the relational schema in order to maintain ordered relations (recall that we assume that in an ordered relation tuples are ordered).

We conclude this subsection by the following Lemma describing an interesting relationship between Functional Dependencies (FDs) and SOI. The detailed study of the impact of order on FDs will be given in Chapter 4.

Lemma 2.1 Let r be a relation over R, $X, X' \subseteq R$ and $\pi_X(r)$ be the projection of the tuples in r onto X.

1. A relation r satisfies a FD $X \rightarrow Y$ if and only if $\omega_X \pi_{XY}(r)$ is SOI.

2. X is a superkey of a relation r if and only if $\omega_X(r)$ is SOI.

3. X is key of a relation r if and only if X is a superkey and for no proper subset $X' \subset X$, $\omega_{X'}(r)$ is SOI. $\square$

2.4 Relationship to the Conventional Model

As we have discussed in Chapter 1, the relational data model was introduced by E.F. Codd in 1970 [34], resulting in the development of relational DBMSs. His work is solidly founded on the concept of a relation in set theory. The set-oriented model offers three main advantages. Firstly, it provides the tabular format of a relation which is simple enough to be understood by all users including non-programmers. Secondly, it is flexible enough to be useful in a wide spectrum of applications, especially in the area of commercial applications. Thirdly, it is elegant enough to support the development of many theoretical issues such as query languages and dependency theory.

These three essential ingredients of the relational data model have made it the most successful data model to date and should serve as guidelines for any further extensions of the relational data model. Partial orderings, which are binary relations on a set, are based on the set-theoretic formalism. We consider the effects on the following three components of the conventional model of incorporating partial orderings.

1. *Structural*: The structure of the relational data model represents information at three different levels stated below.

    (a) Data elements in a domain.

    (b) Tuples in a relation.

    (c) Relations in a database.

    We impose a partial ordering on all data domains of attributes. There follows an induced lexicographical ordering on tuples as the relation schema is assumed to be linearly ordered. This serves as a minimal extended model which incorporates orderings.

2. *Operational*: We extend the relational algebra and the relational calculus to the Partially Ordered Relational Algebra (which we call the PORA) and the Partially Ordered Relational Calculus (which we call the PORC), respectively, by allowing the use of the ordering predicate $\sqsubseteq$ in both languages. We apply Paredaens' and Bancilhon's Theorem [12, 123] to examine the expressiveness of the PORA. Paredaens' and Bancilhon's Theorem is a fundamental result in query language theory, which characterises the expressiveness of the Relational Algebra (which we

call the RA) in terms of automorphisms. An automorphism is a renaming of the occurrences of data values in a database such that it leaves the database invariant. Informally, the theorem states that a conventional relation can be obtained as the result of a RA expression on a database if and only if it is invariant by every automorphism of the database. Based on the PORA (or equivalently, the PORC) we extend SQL to Ordered SQL (which we call OSQL) which combines the capabilities of SQL with the power of semantic orderings.

3. *Constraints*: We consider Ordered Functional Dependencies (which we call OFDs) and Ordered Inclusion Dependencies (which we call OINDs) which are generalized forms of Functional Dependencies (which we call FDs) and inclusion dependencies (which we call INDs), respectively.

Therefore, the ordered relational data model is designed in an upwards compatible manner and ordered domains are the fundamental structure in our extension. Note that the notion of a domain in our model still obeys *the principle of atomicity* and an ordered relation satisfies the so-called *first normal form* criterion [147]. We note that such a restriction does not necessarily mean that a domain must be as simple as numbers or strings. The principle of atomicity should be interpreted as the restriction that the internal structure of a domain element is not decomposable as far as the DBMS concerned [37, 41]. Using the terminology of objects, a domain element is *encapsulated* [155].

For example, the data type $DATE$ obviously has three components, year, month and day, but can still be considered as an atomic domain. It may seem that a user could use the functions associated with $DATE$ such as $YEAR$ to violate the encapsulation principle. For instance, $YEAR$("1-July-1997") returns the year 1997. This apparent paradox has been explained by Date in [41]. The internal structure of a domain element is not accessible by users, but it is perfectly permissible to access its internal structure through the standard functions associated with the domain. We would like to point out that the advantage in adopting this view of atomicity is that a domain can capture more complex data types. A domain can be a simple data type such as *Int* or a compound data type such as the temporal domain $DATE$ as we have mentioned. If the domain support is fully implemented to include more data types, relational databases can meet the demand of many advanced applications such as multimedia without violating the first normal form criterion.

2.5 Other Related Work

In this section we discuss some related research work that has recognised the importance of ordering in information systems. However, we know of no similar attempt to incorporate partial orderings into the data domains of the relational data model as we have done.

2.5.1 Partial Order Databases

A recent proposal which notes the lack of awareness of partial ordering in data modelling models can be found in [129]. Raymond proposes that partial orderings should be a basic component in a database model and illustrates the potential of using partial orderings with some application examples such as textual information and software information. The conclusion of Raymond's thesis is endorsed in this research that partial orderings are a fundamental property of data that needs to be captured in a data model. Moreover, it also aims at defining a unified data model in order to widen the applicability of databases.

However, there are three basic differences between his work and in this thesis. First, the idea of generalisation in Raymond's work is based on the idea of object-oriented inheritance. As a result, a more generalized class does not necessarily support all the operations of its specialised classes. In contrast, our notion of generalization is based on the idea of upwards compatibility: a conventional database is a special case of ordered databases and the ordered relational data model generalises the structures, operations and integrity constraints of the conventional relational data model. Second, Raymond's partial order model is defined to be a collection of algebraic operators for manipulating partially ordered sets. However, we present a thorough investigation of the impact of partial orderings on all the fundamental components of the conventional relational data model including the extensions of data dependencies and SQL. Third, our work is justified by the capability of unifying a wide spectrum of applications, the result of a survey result carried out to evaluate the prototype of the extension, and some theoretical metrics reported herein such as the soundness and completeness of the axiom systems for data dependencies, none of them being considered in Raymond's work.

Ginsburg and Hull [55] have introduced the term *order dependencies* and examined the issue of the extension of functional dependencies to incorporate information involving order. Their work mainly focuses on the implication problem of order dependencies and the application of such dependencies in the area of physical implementation of relational databases. On the theoretical issues, the authors establish a formalism which is analogous to propositional calculus for analysing order dependencies. Moreover, they exhibit a sound and complete set of inference rules for order dependencies, whose implication problem is shown to be co-NP complete [54]. The central notion of order dependencies is similar to that of our definition of ordered functional dependencies arising from pointwise-orderings (POFDs), except that the involved domain orderings in order dependencies are further divided into total order, empty order and general partial-order. This finer classification of a partial ordering requires more sophisticated mathematical tools to explore the axiom systems for order dependencies. On the practical issues, the authors show that indexing space can be reduced substantially if order dependencies are present. For example, given a check account database and an order dependency stating that the check number of checks increases as the date that the check was written, a file which holds the check data will be automatically sorted by date if the file is sorted by the check number. This allows a substantial savings in storage space because without the knowledge of the order dependency, a dense index would have to be used.

2.5.2 Multi-Resolution Data Model

The notion of a multi-resolution set adopted in [130] is equivalent to a special case of a partially ordered domain which can be defined as follows, a multi-resolution set is an ordered set with a unique minimal element and some maximal elements. It has been shown in this work that multi-resolution domains have very strong connections with the notion of approximation such as incompleteness or impreciseness of data. Actually, resolution is a necessary means of managing a very large amount of data transmission, since it is generally true that lower resolution data needs less space than higher resolution data, and thus takes less time to retrieve. For example, in the case of hypermedia information it normally consists of a very large amount of image data and thus resolution is an effective means of managing the size of data domain elements. Let us illustrate this concept with the following simplified multi-resolution domain:

{ 'Null'<'Black and white icon'<'Black and white raster'<'8-bit Colour raster'<'24-bit Colour raster' }.

In the above domain we have five distinct levels of resolution so that the users can select the appropriate level to save the transmission time for downloading a hypermedia document.

Based on the assumption of multi-resolution sets, the author extends the relational data model to support a construct that is called a *sandbag*, which essentially combines cardinality constraints of the scope of approximation into a multi-resolution set. For example, the sandbag "1⟨ Ferrari, Red ⟩3" means that there are one to three red Ferrari cars. A sandbag is a powerful construct to model incomplete or imprecise information, which can be viewed as a generalised form of histograms. The main result of this work is that it provides a formal framework to study the concept of multi-resolution data retrieval and presents some useful algorithms to implement sandbags. Moreover, it extends the conventional relational algebra to incorporate the notion of sandbags so that a relational DBMS can progressively refine the answer to a query. Overall speaking, a sandbag is a complex structure developed from the notion of *sandwiches* in [19], which has a rather complex definition. Although it is, to our knowledge, a novel attempt to unify incomplete information and multimedia information, it is basically a tailor-made model for manipulating incomplete information. In most real life situations, the exact bounds of cardinalities will be unknown when the information is incomplete. Any artificial estimates would cause unnecessary burden upon the DBMS.

2.5.3 List or Sequenced Data

There has been a fair amount of research to extend the relational data model to include lists or sequences as data types [63, 149, 57, 136]. A list can arrange objects in some pre-defined order. Thus it can be defined as a mapping between a collection of similarly structured real world objects and a linearly ordered domain. From this point of view, a linearly ordered set can be regarded as a non-repeating list. However, an ordered set is not allowed to contain duplicates. A list cannot, in general, represent a partial ordering and thus in this sense a list (or a sequence) and an ordered set are two incomparable entities. Richardson in [131] describes a way to incorporate lists into a data model and defines a collection of operators to manipulate a list. However, the expressive power of such operations is not clear and there is a lack of theoretical justification of such extensions.

Wang [152] proposes two useful operators, called *rs-operations*, which are based on *regular languages* [54] and which define a family of list merging and extracting operations. Each operator takes a regular expression as an argument, and the words generated by the expression serve as patterns that direct how lists should be shuffled together or picked apart. A simple example is that the regular expression $(x_1 x_1 x_2 x_1 x_2)$ merges the lists *abd* and *ce* and then generates *abcde*. Another example is that by using the symbol $*$, the regular expression $Q = (x_1 x_2)*$ can generate the set of word patterns $\{x_1 x_2, x_1 x_2 x_1 x_2, \ldots\}$. In a merge operation, Q can be used for producing the *perfect shuffle* of two equal-length lists, meaning that the two lists are evenly and maximally shuffled. For example, the perfect shuffle of two lists *ac* and *bd* is *abcd*. In an extraction, Q can be used for producing the sublist of elements in odd (or even) numbered positions. These operators add considerable power to the user's ability to manipulate lists.

2.5.4 Other Unified Models

A related approach is to extend the relational data model to incorporate abstract data types in domains, which have their associated operations as an integral part of each data type [138, 121]. As discussed in Chapter 1, this approach is basically an object-oriented extension of the relational model (which is usually called the object-relational data model), resulting from the strong trend of object-oriented programming in the 1980s. Although research into the object-relational data model is still on-going, the abstract data type is an extremely powerful and established facility to have in a DBMS. In principle, we can use this facility to simulate ordered domains. The ordered relational model is related to this approach in two main areas. Firstly, our work helps to explore ordered data types, which is a fundamental but relatively unexplored territory of abstract data types in the object-relational data model. Secondly, our work provides some impetus for the acceptance of object-relational databases, since the current relational DBMSs can much more easily be upgraded to conform to our extension.

There is an attempt in [83] to extend the relational data model in order to unify various kinds of incomplete information, fuzzy information and temporal information. Although the approach is based on fuzzy theory rather than orderings, it brings out the important fact that these different types of information systems have some fundamental common property which can be unified in the relational data model. It also brings out another important fact that in practice, fuzziness is usually embedded in temporal information. However, the author does not attempt to develop a complete fuzzy temporal data model in his work. It is also interesting to mention the work in [153] which illustrates that spatial information is closely associated with temporal information in many real life applications. All these observations indicate that there is an essential need to adopt a uniform approach in order to handle these three kinds of information in an efficient manner.

Buneman and his colleagues have extensively investigated the generalisation of relational databases in the context of domain theory [20, 90, 72]. As discussed in Definition 2.7, Buneman has studied three possible orderings on powerdomains (i.e., the powersets of domains) considered to be useful in incomplete information. His work on this generalisation utilises the Smyth orderings to provide a method of representing databases as typed objects in programming languages. He demonstrates that the proposed framework can be used for generalising the two useful operators *natural join* and *projection* [20]. Moreover, he also characterises many important concepts such as that of relational schema in databases, FDs and *nested relations* in terms of powerdomains. Although it was very convenient to use the Smyth orderings to obtain these fruitful results, it may lead to some counter-intuitive observations in actual databases. For example, a conventional relation is of a lower order than its subsets according to Smyth orderings. In fact, Hoare orderings also play an important role in in the theory of incomplete databases. Many database researchers still use Hoare orderings to capture the semantics of incompleteness in studying different issues concerning incomplete information [154, 89, 87, 88]. Libkin [91] presents an update semantics in incomplete information and proposes that Hoare orderings correspond to the natural orderings of sets, whereas Smyth orderings lead to the orderings of *or-sets*, which are basically sets of disjunctive facts (c.f., [68, 134]). There are still many interesting extensions of the notion of orderings in powerdomains such as *mixes, sandwiches, snacks* and *scones* [90]. They are all used for providing different semantics of approximations.

Chapter 3

Query Languages for the Ordered Relational Model

In this chapter we extend the *relational algebra* (the RA) to the *partially ordered relational algebra* (the PORA) by allowing the ordering predicate $\sqsubseteq$ to be used in the formulae of the *selection* operator (σ). Thereafter the *relational calculus* (the RC) is extended to the *partially ordered relational calculus* (the PORC) in a similar manner. The extension is justified by the following four different facets related to query language theory. Firstly, it preserves the robustness of the PORA and the PORC, since these two languages can be shown to be equivalent. Secondly, it is consistent with the two important extreme cases of unordered and linearly ordered domains. In one special case of unordered domains (i.e., where each data element is only comparable with itself under ordering), the PORA reduces to the standard RA and gives the same result as Paredaens' and Bancilhon's Theorem [123, 12]. In another special case of linearly ordered domains (i.e., where any two data elements are comparable under ordering), the PORA expresses exactly the countably infinite set of all possible ordered relations on the active domain of a given ordered database. Thirdly, we show that the PORA is *non-uniform complete*, since it expresses exactly the set of ordered relations which are invariant under order-preserving automorphisms over databases. Fourthly, we demonstrate that there is a one-to-one correspondence between three well-defined hierarchies of: (1) computable queries, (2) query languages and (3) ordered domains.

As an illustration of the usefulness of the PORA, consider a partially ordered domain consisting of three names where Nadav $\sqsubseteq$ Mark and Ethan $\sqsubseteq$ Mark, capturing the semantics of both Nadav and Ethan being under the supervision of Mark. Suppose we would like to find the names of all members in Mark's research group. This query can be formulated in the PORA as $\sigma_{NAME \sqsubseteq 'Mark'}$(STAFF), where STAFF is a relation over $\{NAME\}$. We note that such semantics cannot easily be captured without imposing an order on the underlying domain.

In Section 3.1 we give the definitions of the PORA and the PORC. We also present two useful operators, *initial segment* (γ) and *horizontal projection* (τ), in order to manipulate tuples in linearly ordered relations more effectively. We show that the expressive power are equivalent for the three languages of: (1) RA$\cup\{\gamma\}$, (2) RA$\cup\{\tau\}$, and (3) RA$\cup\{\leq\}$ under the assumption that the cardinalities of relations are smaller than or equal to a fixed natural number. In Section 3.2 we demonstrate that the PORA and the PORC are equivalent and discuss some effects of this equivalence on the design of Ordered SQL, which extends SQL to the context of ordered databases. In Section 3.3 we investigate the expressive power of the PORA and show that it is complete in the sense that it satisfies a generalised Paredaens' and Bancilhon's Theorem [12, 123] (which we call BP-complete). In Section 3.4 we investigate three hierarchies of: (1) computable queries, (2) query languages and (3) ordered domains, and demonstrate that there is a one-to-one correspondence between them. In Section 3.5 we investigate the issues concerning updating ordered domains and ordered databases. In Section 3.6 we briefly discuss an open problem of finding a syntactical characterisation of the concept of *more ordered domains.*

Throughout this chapter we use the term *active domain*, denoted by $adom(d)$, to represent the set containing those values that appear in a database instance d. Thus, $\langle adom(d), \sqsubseteq \rangle$ is a subordering of the underlying domain of d (recall Definition 2.4 for subordering).

Definition 3.1 (Active Domain) The *active domain* of a relation r over R, denoted as $adom(r)$, is defined by $adom(r) = \{v \mid \exists A \in R, \exists t \in r \text{ such that } t[A] = v\}$. The *active domain* of a database instance $d = \{r_1, \ldots, r_n\}$ over $\mathbf{R}$ is defined by

$$adom(d) = \bigcup_{i=1}^{n} adom(r_i).$$

3.1 Query Languages: the PORA and the PORC

In this section we introduce an extension of the conventional *relational algebra* (RA) and the conventional *relational calculus* (RC) [34, 147], which are called the *partially ordered relational algebra* (PORA) and the *partially ordered relational calculus* (PORC), respectively. These two languages are essentially those conventional ones with the ordering predicate added to deal with ordered domains.

3.1.1 The PORA: an Algebraic Query Language

The PORA consists of a collection of six operators, each of which takes as input a set of relations and returns as output the relation resulting from applying the operator to them.

Definition 3.2 (Partially Ordered Relational Algebra). The PORA is a collection of the following six operators.

1. *Union* ($\cup$).
2. *Cartesian product* ($\times$).
3. *Difference* ($-$).
4. *Vertical projection* (π_X), where $X \subseteq U$ is a finite set of attributes.
5. *Renaming* ($\rho_{X \to Y}$), where $X \to Y$ is a bijective function from a finite set of attributes $X \subseteq U$ to a finite set of attributes $Y \subseteq U$.
6. Extended *selection* (σ_F), where the *selection formula* F is restricted to be one of the forms: $A = B$, $A \neq B$, $A \sqsubseteq B$ or $A \not\sqsubseteq B$, where $A \in U$, and either $B \in U$ or B is a constant.

The six operators given in Definition 3.2 are the standard ones (see [147, 9] for their formal definitions and semantics) and the meaning of the selection over the formula $A \sqsubseteq B$ is also as expected, i.e., given a relation r, $\sigma_{A \sqsubseteq B}(r) = \{t \in r \mid t[A] \sqsubseteq t[B]\}$. We choose to interpret the *union compatibility* as follows, the union is applicable only to two relations with the same schema and the orderings of the domains of the corresponding attributes are the same. Let $X = \{A_1, \ldots, A_n\}$ and $Y = \{B_1, \ldots, B_n\}$ be finite subsets of U. We use the shorthand notations $\sigma_{X=Y}(r)$ to represent the expression

$\sigma_{A_1=B_1}(\cdots(\sigma_{A_n=B_n}(r))\cdots)$, and $\sigma_{X \neq Y}(r)$ to represent the expression $\sigma_{A_1 \neq B_1}(r) \cup \cdots \cup \sigma_{A_n \neq B_n}(r)$, respectively. As discussed in Chapter 2, the ordering of a relation r is the *lexicographical ordering* defined over r, which is an extension of domain orderings. We use $\sigma_{X \tilde{\sqsubset} Y}(r)$ to mean the comparison according to lexicographical orderings, which is also a short hand notation representing the expression $\sigma_{A_1 \sqsubset B_1}(r) \cup (\sigma_{A_1=B_1}(\sigma_{A_2 \sqsubset B_2}(r))) \cup \cdots \cup (\sigma_{A_1=B_1} \cdots (\sigma_{A_{n-1}=B_{n-1}}(\sigma_{A_n \sqsubset B_n}(r)))\cdots)$.

We now define PORA expressions using the six operators mentioned.

Definition 3.3 (Partially Ordered Relational Algebra Expression) A PORA expression is a well-formed expression composed of a finite number of operators in the PORA whose operands are relation schemas. We denote by E_{PORA} the set of all PORA expressions. For the sake of clarity, when no ambiguity arises we may omit some parentheses in E_{PORA} expressions.

A query over an ordered database is formulated by means of a PORA expression and the answer to an expression is dependent on a database instance.

Definition 3.4 (Answer to an Expression) Let $d = \{r_1, \ldots, r_n\}$ be a database over $\mathbf{R} = \{R_1, \ldots, R_n\}$. The *answer to an expression* $e \in E_{PORA}$ with respect to an ordered database d over $\mathbf{R}$ is obtained by substituting the relation r_i for every occurrence of R_i in e and computing the result by invoking the operators present in e. The answer is undefined, if there is some operand R of an expression which is not in $\mathbf{R}$. We denote the answer to e with respect to d by $e(d)$.

An expression represents a query over a database. We need the notion of *equivalence of expressions* in order to compare the expressiveness of different queries.

Definition 3.5 (Equivalence of Expressions) Let $DB(\mathbf{R})$ denote the set of all databases over $\mathbf{R}$. Two expressions e_1 and e_2 are said to be *equivalent*, denoted by $e_1 \equiv e_2$, if for all $d \in DB(\mathbf{R})$, $e_1(d) = e_2(d)$. Two sets of expressions E_1 and E_2 are said to be *equivalent*, denoted by $E_1 \equiv E_2$ if

1. $\forall\, e_1 \in E_1$, $\exists\, e_2 \in E_2$ such that $e_1 \equiv e_2$, and

2. $\forall\, e_2 \in E_2$, $\exists\, e_1 \in E_1$ such that $e_1 \equiv e_2$.

Informally, two sets of expressions being equivalent means that they represent the same set of queries. Using this concept we say an operator *op* is *uniformly simulated* by an expression e, if *op* is equivalent to e. If *op* is a PORA operator and e is a PORA expression which does not contain *op*, then the *op* is not primitive with respect to the PORA. In such cases it does not add any extra expressiveness into the PORA, except that it would help to simplify PORA expressions. One example is that we do not include the operator *intersection* ($\cap$) in the PORA, since we can uniformly simulate it as the following expression: $r_1 \cap r_2 \equiv r_1 - (r_1 - r_2)$. Another good example is that we do not include the operator *natural join* ($\bowtie$) in the PORA, since we can also uniformly simulate it as the following expression: $r_1 \bowtie r_2 \equiv \pi_{R_1.A_1,\ldots,R_1.A_m,B_1,\ldots,B_n}\ (\sigma_{R_1.A_1=R_2.A_1} \cdots (\sigma_{R_1.A_m=R_2.A_m}(r_1 \times r_2)) \cdots)$, where r_1 and r_2 are over R_1 and R_2, respectively, $A_1, \ldots, A_m$ are the common attributes of R_1 and R_2, and $B_1, \ldots, B_n$ are the attributes in either R_1 or R_2 except those common attributes.

A weaker notion of simulation of operators called *non-uniform simulation* requires only that for any given database there exists a PORA expression equal to the result of the PORA operator on the database. A typical example is the *difference* operator ($-$), which can be non-uniformly simulated by the other operators in the PORA for given relations r_1 and r_2 as will be shown in Lemma 3.1. However, this refers to given instances of r_1 and r_2 only; the difference operator cannot be uniformly simulated by a PORA expression that involves only the other five operators of the PORA. Therefore, the difference operator is still primitive, which we should include in Definition 3.2.

Lemma 3.1 For every pair of relations r_1 and r_2 over W, the relation $(r_1 - r_2)$ can be obtained as the result of a PORA expression whose operands are r_1 and r_2.

Proof. Let $w = r_1 \cap r_2$ contain k tuples $\{t_1, \ldots, t_k\}$ defined over W. First, we construct a relation that is essentially the Cartesian product of w with itself k times with suitable renaming as follows: $w^k = \rho_{W \to T_1}(w) \times \cdots \times \rho_{W \to T_k}(w)$ (k times). Now consider the expression $e = \sigma_{T_1 \neq T_2}(\cdots(\sigma_{T_i \neq T_j}(\cdots(\sigma_{T_{k-1} \neq T_k}\ (w^k))\cdots))\cdots)$ for all distinct $i, j \in \{1, \ldots, k\}$ which returns a relation with k factorial tuples; each of them is a juxtaposition of the k tuples in w up to renaming of the attributes. We now have $r_1 - r_2 = \pi_W(\sigma_{W \neq T_1} \cdots (\sigma_{W \neq T_k}(r \times e))\cdots)$ since t_i will be eliminated by $\sigma_{W \neq T_i}$ for $i \in \{1, \ldots, k\}$. Thus, the remaining tuples are in the relation $r_1 - r_2$ after taking the projection onto W. $\square$

The following example can help to clarify the above lemma.

Example 3.1 Let $r_1 = \{a, b, c\}$ and $r_2 = \{b, c, d\}$ be the relations over $\{A\}$. Thus $r_1 \cap r_2 = \rho_{B \to A}(\pi_B(\sigma_{B=C}(\rho_{A \to B}(r_1) \times \rho_{A \to C}(r_2))))$. So we have $w = r_1 \cap r_2 = \{b, c\}$ over A. Now $e = \sigma_{B \neq C}(\rho_{A \to B}(w) \times \rho_{A \to C}(w)) = \{bc, cb\}$ over $\{B, C\}$. Finally, $r_1 - r_2 = \pi_A(\sigma_{A \neq B}(\sigma_{A \neq C}(r_1 \times e))) = \{a\}$ as expected.

We observe that $\sigma_=$ is not primitive, since for any relation r it can be uniformly simulated as the following expression: $\sigma_{A=B}(r) \equiv \sigma_{A \sqsubseteq B}(\sigma_{B \sqsubseteq A}(r))$. Similarly, $\sigma_{A \neq B}(r)$ can be simulated as follows, $\sigma_{A \neq B}(r) \equiv \sigma_{A \not\sqsubseteq B}(r) \cup \sigma_{B \not\sqsubseteq A}(r)$. In the extreme case of unordered domain $\sigma_\sqsubseteq$ becomes $\sigma_=$, i.e., for any given relation r, $\sigma_{A \sqsubseteq B}(r) \equiv \sigma_{A=B}(r)$. Therefore, our definition of the PORA is consistent with the standard relational algebra used in [123]. Let UORA $= \{\rho, \times, -, \cup, \pi, \sigma_=, \sigma_{\neq}\}$ be the unordered relational algebra and LORA $= \{\rho, \times, -, \cup, \pi, \sigma_\leq, \sigma_{\not\leq}\}$ be the linearly ordered relational algebra for a given linear ordering of D. We formalise our observations as follows.

Proposition 3.2 Let $\langle D, \sqsubseteq_D \rangle$ be the underlying domain. Then

$E_{PORA} \equiv E_{UORA}$ if $\langle D, \sqsubseteq_D \rangle$ is unordered, and

$E_{PORA} \equiv E_{LORA}$ if $\langle D, \sqsubseteq_D \rangle$ is linearly ordered. $\square$

Note that those relations which can be generated by E_{PORA} involve only relations in d and contain values solely in *adom*(d). We denote by $e_{ad}(d)$ the PORA expression that generates *adom*(d). The following proposition will be repeatedly used in many formal proofs subsequently in this chapter.

Proposition 3.3 Let $e_{ad}(d) = \bigcup_{i,j} \pi_{A_j}(r_i)$, $\forall A_j \in R_i$ where $R_i \in \mathbf{R}$, and $\forall r_i \in d$. Then *adom*$(d) = e_{ad}(d)$. $\square$

The *possible information* of d is the countably infinite set of all relations that can be derived from the *adom*(d).

Definition 3.6 (Possible Information) The *possible information* of d, denoted by $Poss(d)$, is defined by

$$Poss(d) = \bigcup_{i=0}^{\infty} \mathcal{P}(adom(d)^i).$$

Although the selection operator $\sigma_{\leq}$ in the LORA can be employed to make comparison of tuples in linearly ordered relations, it is still not clear whether it has sufficient power to retrieve tuples according to their orderings. For example, we would like to know whether the LORA can express some common queries involving order, such as retrieving the first three lowest sales figures in a sales record. We now introduce an operator called *initial segment* (γ), which should help to manipulate tuples in a linearly ordered relation r over schema R. This operator allows us to select the first n tuples according to the linear ordering $\leq_r$, where n is a positive integer.

Definition 3.7 (Initial Segment) The *initial segment* of a linearly ordered relation r over R, denoted by γ_n whose parameter n is an integer, is defined by $\gamma_n(r) = \{t \mid t \in r$ and t is the kth tuple t according to the linear ordering $\leq_r$ with $1 \leq k \leq n\}$ if $1 \leq$ n; otherwise, $\gamma_n(r)$ is defined to be $\emptyset$.

We note that $\gamma_n(r) = r$ if and only if $n \geq \mid r \mid$. We give the following simple example to illustrate the usefulness of the initial segment operator.

Example 3.2 Consider a linearly ordered relation $r = \{111, 212, 221\}$ (3 tuples). Then $\gamma_1(r) = \{111\}$, $\gamma_2(r) = \{111, 212\}$, $\gamma_n(r) = r$ for $n \geq 3$ and $\gamma_n(r) = \emptyset$ for $n \leq 0$.

An interesting fact of γ_n is that a weaker operator γ_1 (i.e., returning a first tuple) together with the UORA is sufficient to uniformly simulate the effect of γ_n in a linearly ordered relation r. For instance, $\gamma_3(r) = \gamma_1(r - \gamma_1(r - \gamma_1(r))) \cup \gamma_1(r - \gamma_1(r)) \cup \gamma_1(r)$. The following proposition can be easily proved by using induction on the parameter n of γ.

Proposition 3.4 $\gamma_n(r) \equiv e$ for some expression e defined over $\{\gamma_1, \cup, -\}$. □

It is clear that the operator γ_n is not equivalent to any UORA expression, since there is no operator in the UORA to be defined over linearly ordered relations. We now make a simple extension of the UORA as follows, $UORA^{\gamma}$ = UORA $\cup$ $\{\gamma_n\}$ and assume that $\gamma_n(r)$ is undefined over unordered relations. Then the expressive power of the $UORA^{\gamma}$ is equivalent to that of the UORA if the domains are unordered, since in this case no extra power can be gained by γ_n. On the other hand, $UORA^{\gamma}$ can be applied to linearly ordered relations due to the fact that in general, all operators in the UORA can be applied to ordered databases. We now define a similar operator called *horizontal projection* (τ), which allows us to select a particular tuple by specifying its position according to the ordering of a linearly ordered relation.

Definition 3.8 (Horizontal Projection). The *horizontal projection* of a linearly ordered relation r over R, denoted by τ_n whose parameter n is an integer, is defined by $\tau_n(r) = \{t \mid t \in r \text{ and } t \text{ is the } n\text{th tuple according to } \leq_r\}$ if $1 \leq$ n; otherwise, $\tau_n(r)$ is defined to be $\emptyset$.

Note that $\tau_n(r)$ is a singleton if and only if $1 \leq n \leq \mid r \mid$, otherwise $\tau_n(r) = \emptyset$. The horizontal projection operator τ has the similar property as stated in Proposition 3.4.

Proposition 3.5 $\tau_n(r) \equiv e$ for some expression e defined over $\{\tau_1, \cup, -\}$. $\Box$

We now let $UORA^\tau$ = UORA $\cup \{\tau_n\}$ and denote by E^τ_{UORA} the set of all $UORA^\tau$ expressions. Similarly, we denote by E^γ_{UORA} the set of all $UORA^\gamma$ expressions. The simple relationships between γ_n, τ_n, E^γ_{UORA}, and E^τ_{UORA} can be formally stated as follows.

Lemma 3.6 The following statements over linearly ordered relations are true.

1. The operator τ_n can be uniformly simulated by an E^γ_{UORA} expression.
2. The operator γ_n can be uniformly simulated by an E^τ_{UORA} expression.

Proof. Let r be a linearly ordered relation over R.

1. We claim that $\tau_1(r) = \gamma_1(r)$ and for $n \geq 2$, $\tau_n(r) = \gamma_n(r) - \gamma_{n-1}(r)$. This claim can be easily proven by using induction on n. Thus, it follows that $\tau_n(r)$ can be expressed in E^γ_{UORA}.
2. This part can be easily established by noting that $\gamma_n(r)$ can be expressed as $\tau_1(r) \cup \ldots \cup \tau_n(r)$. $\Box$

We impose a restriction on the cardinalities of r over R in order to define a subclass of linearly ordered relations called *bounded relations* as follows, a relation r is said to be *bounded* if $\mid r \mid \leq k$ for some fixed natural number k. Furthermore, we call a database a *bounded* database if all relations in the database are bounded. Note that if r satisfies the condition that $\mid \pi_A(r) \mid \leq k$ for $A \in R$, then r is bounded. The restriction of $\mid \pi_A(r) \mid \leq k$ is known as a domain constraint [74, 51, 32], which is also a basic kind of constraint in conventional databases. The assumption that a relation is bounded is certainly practical, since it is necessary to restrict the cardinalities of a relation in the implementation of a DBMS due to the limited space resources of a platform environment.

Lemma 3.7 The following statements over bounded and linearly ordered relations are true:

1. The operator τ_n can be uniformly simulated by an E_{LORA} expression.
2. The operator $\sigma_{A \leq B}$ can be uniformly simulated by an E^{τ}_{UORA} expression.

Proof.

Let $R_i = \{R_i.A_1, \dots, R_i.A_m\}$ for $i \in \{1, 2\}$.

1. By Proposition 3.5, it suffices to show that $\tau_1(r)$ is equivalently to an E_{LORA} expression. It can be checked that $\tau_1(r) \equiv s \cup (r - w)$, where $s =$ $\rho_{R_1 \to R}(\pi_{R_1}(\sigma_{R_1 \tilde{<} R_2}(r \times r))) - \rho_{R_2 \to R}(\pi_{R_2}(\sigma_{R_1 \tilde{<} R_2}(r \times r)))$ and $w =$ $\rho_{R_1 \to R}(\pi_{R_1}(\sigma_{R_1 \tilde{<} R_2}(r \times r))) \cup \rho_{R_2 \to R}(\pi_{R_2}(\sigma_{R_1 \tilde{<} R_2}(r \times r)))$, respectively. We recall that the notation $\sigma_{R_i \tilde{<} R_j}(r) = \sigma_{R_i.A_1 < R_j.A_1}(r) \cup (\sigma_{R_i.A_1 = R_j.A_1}(\sigma_{R_i.A_2 < R_j.A_2}(r)))$ $\cup \cdots \cup (\sigma_{R_i.A_1 = R_j.A_1} \cdots (\sigma_{R_i.A_{m-1} = R_j.A_{m-1}}(\sigma_{R_i.A_m < R_j.A_m}(r))) \cdots)$. Note that the subexpression $(r - w)$ in the above formula is necessary to cater for the case of r being a singleton, since in this case $\sigma_{R_1 \tilde{<} R_2}(r \times r) = \emptyset$.

2. Note that there are at most km (k tuples $\times$ m attributes) distinct elements in r. The selection operator ($\sigma_{A \leq B}$) is equivalently to the following expression: $\sigma_{A \leq B}(r)$ $\equiv \pi_R(\sigma_{AB=CD}(r \times s))$, where s is a relation over $\{C, D\}$, which is defined by $s =$ $\bigcup_{i=1}^{km} \bigcup_{j=i}^{km} (\tau_i(e_{ad}(r)) \times \tau_j(e_{ad}(r)))$. $\square$

The following example helps to clarify Lemma 3.7.

Example 3.3 We use the same relation r as given in Example 3.2, whose schema is $R = \{A, B, C\}$. Let $R_i = \{R_i.A, R_i.B, R_i.C\}$ for $i \in \{1, 2, 3\}$. It can be checked that $\tau_1(r) \equiv s = \rho_{R_1 \to R}(\pi_{R_1}(\sigma_{R_1 \tilde{<} R_2}(r \times r))) - \rho_{R_2 \to R}(\pi_{R_2}(\sigma_{R_1 \tilde{<} R_2}(r \times r)))$. (We ignore $(r - w)$ for the sake of clarity but it can be checked that $r - w = \emptyset$ since r is not a singleton.) We show below the process of evaluating this expression by stepwise computation.

1. Let $s_1 = r \times r$ over $\{R_1.A, R_1.B, R_1.C, R_2.A, R_2.B, R_2.C\}$.
   Then we have $s_1 =$ {111111, 111212, 111221, 212111, 212212, 212221, 221111, 221212, 221221} (9 tuples).

2. Let $s_2 = \sigma_{R_1 \tilde{<} R_2}(s_1)$.
   Then we have $s_2 = \sigma_{R_1.A < R_2.A}(s_1) \cup (\sigma_{R_1.A = R_2.A}(\sigma_{R_1.B < R_2.B}(s_1))) \cup$

$(\sigma_{R_1.A=R_2.A}(\sigma_{R_1.B=R_2.B}(\sigma_{R_1.C<R_2.C}(s_1))))$. Now s_2 over R_1, which consists of three tuples as given by $\{111212, 111221, 212221\}$.

3. Let $s = \rho_{R_1 \to R}(\pi_{R_1}(s_2)) - \rho_{R_2 \to R}(\pi_{R_2}(s_2)) = \{111, 212\} - \{212, 221\}$. Then we have $s = \{111\}$ over R as the required answer.

As an illustration of part 2 of Lemma 3.7, the selection operator $\sigma_{A \leq B}$ can be uniformly simulated as follows, $\sigma_{A \leq B}(r) \equiv \pi_R(\sigma_{AB=CD}(r \times s))$, where $s = (\tau_1(e_{ad}(r)) \times \tau_1(e_{ad}(r))) \cup (\tau_1(e_{ad}(r)) \times \tau_2(e_{ad}(r))) \cup (\tau_2(e_{ad}(r)) \times \tau_2(e_{ad}(r))) = \{11, 12, 22\}$. (We do not show τ_n for $n \leq 3$ since it is equal to $\emptyset$.) Then we have $\pi_R(\sigma_{AB=CD}(r \times s)) = \{111, 221\}$.

We now summarise the expressiveness of $E^{\mathcal{T}}_{UORA}$, E^{γ}_{UORA} and E_{LORA} as the following theorem.

Theorem 3.8 $E^{\mathcal{T}}_{UORA}$, E^{γ}_{UORA} and E_{LORA} are equivalent over bounded and linearly ordered databases.

Proof.

The result follows by Definition 3.5, Lemma 3.6 and Lemma 3.7. □

3.1.2 The PORC: a Calculus Query Language

In this subsection we make a simple extension of the conventional tuple relational calculus [147, 9] called the *partially ordered relational calculus* (PORC). We first define the set of *symbols* which are allowed in formulas of the PORC.

Definition 3.9 (Symbols of the PORC) The *symbols* of the PORC have the following six items:

1. *Constant values* (or simply constants) are $a, a_1, a_2, \ldots$, which are elements of the domain D.

2. *Tuple variables* (or simply variables) are $t, t_1, t_2, \ldots$, which are members of a countably infinite set of variables $\mathcal{V}$ such that $\mathcal{V} \cap D = \emptyset$.

3. *Relational symbols* are $R, R_1, R_2, \ldots$, which are drawn from a countably infinite set of variables $\mathcal{R}$ such that $\mathcal{R} \cap \mathcal{V} \cap D = \emptyset$.

4. The *operators* are $=$ and $\sqsubseteq$.

5. The *logical connectives* are $\exists, \vee$ and $\neg$.

6. *Delimiters* are () (parentheses), and ,(comma).

Note that we consider the negation, disjunction and existential quantifier only since they are sufficient to generate other connectives such as $\forall$ (*universal quantifier*), $\wedge$ (*conjunction*), $\Rightarrow$ (*implication*) and $\Leftrightarrow$ (*equivalence*). For example, $R_1 \wedge R_2 = \neg((\neg R_1) \vee (\neg R_2))$ and $\forall R = \neg\exists(\neg R)$. Although $x = y$ can be represented as $(x \sqsubseteq y \wedge y \sqsubseteq x)$, we include the equality predicate $=$ for the sake of clarity in expressions. We denote by $t[i]$ the ith component of a tuple t. We now define *atomic formulas* over symbols, which are the basic components of a PORC expression.

Definition 3.10 (Atomic Formulas) The *atomic formulas* of the PORC have the following two forms.

1. $R(t)$, where R is a relation symbol and t is a variable.

2. $x\theta y$, where $\theta \in \{=, \sqsubseteq\}$, x is a *component reference* of a variable of the form $t[i]$, where t is a variable and i is an index, and y is either a component reference or a constant.

We write $x \not\sqsubseteq y$ as an abbreviation $\neg(x \sqsubseteq y)$.

Definition 3.11 (Well-formed Formulas) The *well-formed formulas* (or simply formulas) are defined recursively as follows.

1. An atomic formula is a formula.

2. If F is a formula, then so are $\neg F$ and (F).

3. If F_1 and F_2 are formulas, then so is $F_1 \vee F_2$.

4. If F is a formula, then so is $\exists t : R(F)$ (or simply $\exists t(F)$ if no ambiguity arises), where t is a variable and R is a set of attributes.

5. No other formulas are formulas.

We omit parentheses in formulas if no ambiguity arises as to the meaning of a formula. In addition, we assume that all the relation schemas corresponding to the relation symbols that are mentioned in F are included in a database schema $\mathbf{R}$. We call a variable defined by item (4) of Definition 3.11 a *bound variable*, otherwise we call a variable a *free variable.* We write $F(t, t_1, \ldots)$ for a formula F to indicate that $t, t_1, \ldots$ are the free variables occurring in F.

We now define *PORC expressions* using well-formed formulas and free variables.

Definition 3.12 (Partially Ordered Relational Calculus Expression) A *PORC expression*, consisting of a free variable t, a bijective function g from a finite set of attributes $R \subseteq U$ to a finite set of attributes $S \subseteq U$, and a well-formed formula F, is defined as $\{t : g(R) \mid F(t)\}$.

Note that in the above definition there can be only one free variable in F. If g is an identity, i.e., no renaming of attributes is required, we just omit g and write the PORC expression as $\{t : R \mid F(t)\}$. We denote by E_{PORC} the set of all PORC expressions, and by E_{LORC} and E_{UORC} the set of those PORC expressions for the cases of linearly ordered domains and unordered domains, respectively (c.f., E_{LORA} and E_{UORA} in Proposition 3.2). We now define the semantics of PORC expressions.

Definition 3.13 (Satisfaction of a Formula by a Tuple) Let $d = \{r_1, \ldots, r_n\}$ be a database over the schema $\mathbf{R} = \{R_1, \ldots, R_n\}$ and consider the PORC expression $\{t : R \mid F(t)\}$. A tuple $u = \langle a_1, \ldots, a_m \rangle$ *satisfies* the formula F with respect to d, if $u \in D^m$, and one of the following conditions is satisfied:

1. If F is the atomic formula $R(t)$, then $R \in \mathbf{R}$ and the tuple u satisfies $u \in r$, where $r \in d$ which is over R.

2. If F is the atomic formula $x\theta y$, then $a_i \theta a_j$ is satisfied, where a_i substitutes x and either $y = t[j]$ and a_j substitutes y, or y is a constant and $a_j = y$.

3. If F is the formula (F_1), then u satisfies the formula F if u satisfies F_1.

4. If F takes on one of the forms: $\neg F_1$, $F_1 \wedge F_2$, $F_1 \vee F_2$, then u satisfies F is defined according to the semantics of the corresponding Boolean connectives on the truth values of F_1 and F_2 [53].

5. If F is the formula, $\exists t_1 : R_1(F_1(t, t_1))$, where the arities of t and t_1 are m and m_1, respectively, then u satisfies F if there exists a tuple $\langle b_1, \ldots, b_{m_1}\rangle \in D^{m_1}$ such that when $\langle b_1, \ldots, b_{m_1}\rangle$ is substituted for t_1, u satisfies $F_1(t)$.

Informally, an answer to a PORC expression with respect to a database d is the set of all tuples satisfying F.

Definition 3.14 (Answer to a PORC expression) The *answer to a PORC* expression $\{t : R \mid F(t)\}$ with respect to a database d over $\mathbf{R}$, denoted as $\{t : R \mid F(t)\}(d)$, is a relation r over schema R, which is defined by $r = \{t \mid t \text{ satisfies } F\}$. The answer is undefined if there is some relation symbol R of an expression which is not in $\mathbf{R}$.

An important detail to be considered is that the answer to a PORC expression is supposed to be finite and is dependent only on a given database d. However, PORC expressions allow us to define the formulas as shown in the following example, whose result is an infinite set of tuples [147].

Example 3.4 Consider the following two PORC expressions.

1. $\{t : R \mid \neg R(t)\}$.
2. $\{t : R \mid \exists t_1((R_1(t_1) \wedge (t[1] = t_1[1])) \vee (R_1(t_1) \wedge (t[2] = t_1[2])))\}$.

It is clear that the answer to the first expression in the above example depends, not only on the database instance d, but also on the domain D. Let us call such formulas *domain-dependent formulas.* If D is infinite, then the first expression results in an infinite relation as an answer, which is undesirable, since we can only have a finite number of tuples in relations. As the second example shows, a domain-dependent formula does not necessarily have negation. It can be checked that the answer to this expression depends on D and thus results in an infinite number of tuples as an answer. So it is not trivial to deduce from the syntax of the formula whether it is domain-dependent or not. In fact, it has been shown that this problem in the context of the conventional RC is undecidable [148], i.e., the following problem is undecidable:

Is the formula in an E_{UORC} expression a domain-dependent formula?

Therefore, it follows that the problem of domain-dependence of E_{PORC} is also undecidable, due to the fact that E_{UORC} is just the special case of E_{PORC} (i.e., when D is unordered). We formalise our discussion by the following proposition.

Proposition 3.9 Given a formula F in a PORC expression, the problem of whether F is domain-dependent is undecidable. $\square$

There are two possible approaches to tackle this problem.

The first approach is to impose certain restrictions on the syntax of all *subformulas* to ensure that a PORC expression is *safe* [147], in the sense that the answer of the expression depends on d only. (By a *subformula* of a formula F we mean a substring F that is also a formula.) For example, one restriction is that we only allow the negation operator to apply to a formula which is in a conjunction with some safe formulas. Another restriction is that we do not allow different free variables occurring in F_1 and F_2 when they are connected by disjunction. It is clear that the first and the second expressions in Example 3.4 violate these two restrictions respectively.

The second approach is to impose a restriction on the semantics of PORC expressions. Note that a PORC expression is domain-dependent mainly because the variables in the expression are allowed to vary freely over the domain. Therefore, we can resolve this problem by assuming that all substitutions of variables are chosen from a subset of $adom(d)$. For example, we can assume that each variable in a well-formed formula is associated with a *declaration of range* [9], and thus the answer of a PORC expression cannot contain any new values apart from those in the declared ranges. We adopt this approach and from now on assume that all the constants in the first item in Definition 3.9 should be chosen from $adom(d)$, and that the set of all constants appearing in a PORC expression is a subset of $adom(d)$.

3.2 Equivalence between the PORA and the PORC

In this section we compare the expressive power of the PORA and the PORC. We show that they are actually equivalent. The equivalence can be established by the next two lemmas. The method that we use is standard (c.f., see Chapter 3 in [147]), whose basic idea is to carry out induction on the number of occurrences of operators in PORA expressions (or PORC expressions in the reverse direction).

Lemma 3.10 For every PORC expression e_c, there is an equivalent PORA expression e_a.

Proof. We prove it by induction on the number of occurrences of the connectives and the quantifiers in F. Assume an e_c expression to be of the form $\{t : R \mid F(t)\}$.

(*Basis*). There are two cases of atomic formulas.

1. The formula is $R_1(t)$; the corresponding PORA expression is $e_a = r_1$, where r_1 is a relation over the schema R_1.

2. The formula is $x\theta y$; the corresponding PORA expression is $e_a = \sigma_{A_i\theta A_j}(e_{ad})^n$ if $x = t[i]$ and $y = t[j]$, or $e_a = \sigma_{A_i\theta a}(e_{ad})^n$ if $y = a$, where e_{ad} is the expression to generate the active domain of the relation r (see Proposition 3.3 for the definition of e_{ad}).

(*Induction*). We consider the negation, disjunction and existential quantifier as follows.

1. Negation: $F = \neg F_1$.
   $e_a = (e_{ad})^n - e_{F_1}$, where e_{F_1} is the corresponding PORA expression for F_1.

2. Disjunction: $F = (F_1) \vee (F_2)$.
   $e_a = e_{F_1} \cup e_{F_2}$, where e_{F_1} and e_{F_2} are the corresponding PORA expressions for F_1 and F_2, respectively.

3. Existential quantifier: $F = \exists t_1 : R_1(F_1)$.
   $e_a = \pi_{R_1}(e_{F_1})$, where e_{F_1} is the corresponding PORA expression for F_1.

This completes the induction. □

Lemma 3.11 For every PORA expression e_a, there is an equivalent PORC expression e_c.

Proof. We use similar method as the previous lemma. The proof is by induction on the number of occurrences of the six operators that are defined in Definition 3.2.

(*Basis*). There are two cases of relations.

1. For the case of a constant relation, without loss of generality, we assume that it is a unary relation which has only one tuple, $e_a = \{\langle a \rangle\}$; $e_c = \{t : R \mid t[1] = a\}$ where $R = \{A\}$.

2. For the case of a relation $r \in d$, over $R = \{A_1, \ldots, A_n\}$; $e_c = \{t : R \mid R(t)\}$.

(*Induction*). We consider various primitive operators, *op* $\in$ PORA, at the top level of the algebraic expression. Let $e_a, e'_a, e_1, e_2 \in E_{PORA}$. For the cases of unary operators of projection, selection and renaming, we let the corresponding calculus expression for e'_a be $e'_c = \{t : R \mid F(t)\}$, where $R = \{A_1, \dots, A_n\}$. Thus we have $e_a = op(e'_a)$. For the cases of binary operators of Cartesian product, union and difference, we let the corresponding calculus expressions for e_1 be $\{t : R_1 \mid F_1(t)\}$ and e_2 be $\{t : R_2 \mid F_2(t)\}$, respectively. Thus we have $e_a = (e_1)op(e_2)$.

1. Projection: without loss of generality, we assume that only the attribute A_n is projected out: $e_a = \pi_{A_1,\dots,A_{n-1}}(e'_a)$; the calculus expression is $e_c = \{t : R \mid \exists t_1 : R_1(F(t) \wedge (t[1] = t_1[1]) \wedge \cdots \wedge (t[n-1] = t_1[n-1]))\}$, where $R_1 = R - \{A_n\}$.

2. Selection: $e_a = \sigma_{A_i \theta B}(e'_a)$; then the calculus expression is given by $e_c = \{t : R \mid F(t) \wedge (t[i]\theta t[j])\}$ if $B = A_j$, or $e_c = \{t : R \mid F(t) \wedge (t[i]\theta a)\}$ if $B = a$.

3. Renaming: $e_a = \rho_{X \to Y}(e'_a)$ with $X = \{A_1, \dots, A_m\}$ and $Y = \{B_1, \dots, B_m\}$, respectively. Let g be a bijective function from X to Y, which is defined by $g(A_i) = B_i$. Then the calculus expression is given by $e_c = \{t : g(R) \mid F(t)\}$ where $g(R) = \{g(A_1), \dots, g(A_n)\}$.

4. Cartesian product: $e_1 \times e_2$; we let $R = R_1 \cup R_2$. Then the calculus expression is given by $e_c = \{t : R \mid \exists t_1 \exists t_2 (F_1(t_1) \wedge F_2(t_2) \wedge (t[1] = t_1[1]) \wedge \cdots \wedge (t[m] = t_1[m]) \wedge \cdots \wedge (t[m+1] = t_2[1]) \wedge \cdots \wedge (t[n] = t_2[n-m]))\}$.

5. Union: $e_1 \cup e_2$; the calculus expression is given by $e_c = \{t : R \mid F_1(t) \vee F_2(t)\}$.

6. Difference: the proof is similar to the case of union except we replace the formula in the calculus expression e_c by $(F_1(t) \wedge \neg F_2(t))$.

This completes the induction. □

We now give the main theorem in this section.

Theorem 3.12 E_{PORA} and E_{PORC} are equivalent.

Proof. By Definition 3.5, Lemma 3.10 and Lemma 3.11, we can readily establish the equivalence. □

The above theorem shows that the PORA and the PORC actually express the same set of queries. It also shows the robustness of these two languages and thus they can be adopted as a benchmark for evaluating expressiveness of a query language in ordered databases. From this point of view, we can say that the PORA (or the PORC) is *Codd-complete* [35], since Codd suggests that the RC is adopted to be the standard to measure completeness of a query language in conventional databases. Moreover, SQL, which is the most common query language for commercial DBMSs, is developed to conform to the standard of the RC. We adopt the conventional approach in developing SQL and extend SQL to Ordered SQL (OSQL). Our extension is also based upon the essential features of PORC. For example, we extend the *WHERE* predicate in OSQL to implement the atomic formula $t[i] \sqsubseteq_D t[j]$ in the PORC as follows, $A_i \leq A_j$ *WITHIN* D, where the keyword *WITHIN* specifies the ordered domain D, and A_i and A_j are attributes corresponding to the tuple components $t[i]$ and $t[j]$, respectively.

An important feature of OSQL is the implementation of a *tuple list*, which is basically the listing of different levels of an *internal hierarchy* of a relation. We can view an internal hierarchy as a generalisation of the position of a tuple in a linearly ordered relation. We need some terminology to explain the underlying idea of this extension. Let $\langle r, \sqsubseteq_r \rangle$ be an ordered relation. We denote by $part(r)$ a *partition* of r, which is a set of pairwise disjoint non-empty subsets of r such that $\bigcup_{T \in part(r)} T = r$, and call an element $T \in part(r)$ a *tuple level* of r. An *internal hierarchy* of a relation r is a linearly ordered partition induced by $\sqsubseteq_r$.

Definition 3.15 (Internal Hierarchy of a Relation) An *internal hierarchy* of r is a linearly ordered set $\langle part(r), \leq \rangle$, such that

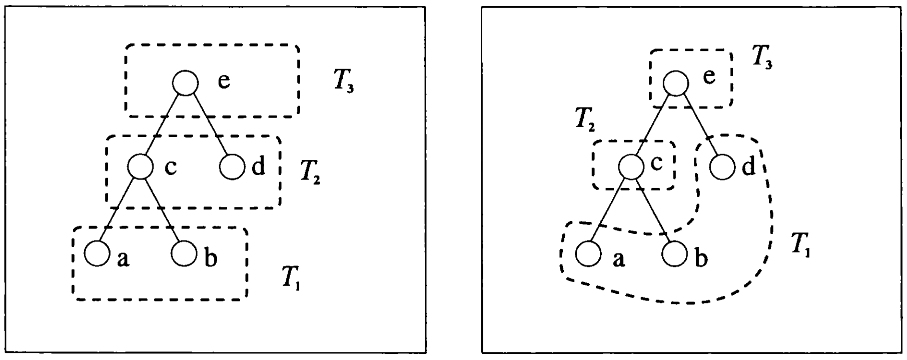
1. $\forall T \in part(r), \forall t_1, t_2 \in T$, either $t_1 = t_2$ or $t_1 \parallel t_2$ (i.e., T is unordered).

2. $\forall T_i, T_j \in part(r)$, $T_i < T_j \Rightarrow \forall t_1 \in T_i, \forall t_2 \in T_j$, $t_2 \not\sqsubseteq_r t_1$.

3. $\forall T_i, T_j \in part(r)$, $T_i < T_j \Rightarrow \exists t_1 \in T_i, \exists t_2 \in T_j$ such that $t_1 \sqsubset_r t_2$.

A tuple $u \in s$ is said to be *minimal*, where $s \subseteq r$, if for any $t \in s$, $t \sqsubseteq_r u$ implies that $t = u$. We remark that s may have more than one minimal tuple. In the special case of linearly ordered relations, s has a unique minimal tuple. In the other extreme case of an unordered relation, all tuples in s are minimal.

Example 3.5 Consider a unary relation having 5 tuples, $\langle r, \leq_r \rangle = \langle \{a, b, c, d, e\}, \{a \sqsubseteq c, b \sqsubseteq c, c \sqsubseteq e, d \sqsubseteq e\} \rangle$ (5 tuples). We show two possible internal hierarchies $part(r) = \{T_1, T_2, T_3\}$ given in Figure 3.1.



(a) (b)

Figure 3.1: Two possible internal hierarchies for a relation r

The following lemma shows that by successively collecting the sets of minimal tuples in the subsets of a relation we can construct an internal hierarchy as illustrated in Figure 3.1(b).

Lemma 3.13 Every relation contains an internal hierarchy.

Proof.

Let r be a given relation. We use the following algorithm to generate a partition.

Algorithm 3.1

```
1.  begin
2.     $r_0 = r$ and $T_0 = \emptyset$;
3.     do until $r_{i-1} = \emptyset$
4.        $T_i$ is the set of minimal tuples of $r_i = r_{i-1} - T_{i-1}$;
5.     return Result = $\{T_1, \dots, T_l\}$;
6.  end.
```

It is trivial that the above algorithm will terminate for a finite relation r. Let the last tuple level generated by the algorithm be T_l and $\{T_1 < T_2 < \cdots < T_l\}$ be a collection of subsets obtained by the above algorithm, where the linear ordering is according to the order of generation of T_i in the steps 3 and 4. Clearly, it is a partition of r such that for all $t_1, t_2 \in T_i$, if t_1 and t_2 are distinct, then we have $t_1 \parallel t_2$, since they are both the minimal tuples of r_i. So it satisfies part 1 in Definition 3.15. Now, we assume to the contrary that $\exists t_1 \in T_i, \exists t_2 \in T_j$ such that $t_2 \sqsubseteq_r t_1$ and $T_i < T_j$. Then it follows that $t_2 = t_1$, since t_1 is a minimal tuple and is less than t_2. However, this is impossible because

T_i and T_j are disjoint. Hence part 2 is also satisfied. Finally, part 3 can be established by noting that T_i is the set of all minimal elements of some superset of T_j. It follows that for any element $t_2 \in T_j$, there is an element $t_1 \in T_i$ such that $t_1 \sqsubset_r t_2$. □

The next lemma is immediately followed by the definition of Algorithm 3.1.

Lemma 3.14 The internal hierarchy generated by Algorithm 3.1 is unique.

Proof.

This can be easily established by using induction on T_i and the fact that T_i is the unique set of all minimal tuples of r_i. □

The following lemma can be regarded as a generalisation of part 1 of Lemma 3.7 to partially ordered relations. It shows that the tuple levels of the internal hierarchy generated by Algorithm 3.1 can be expressed by the PORA (or equivalently, the PORC) for a relation r over R.

Lemma 3.15 Any tuple level of the internal hierarchy generated by Algorithm 3.1 can be expressed by the PORA.

Proof.

We can generate T_i, where $1 \leq i \leq n$, recursively as follows.

$i = 1$: $T_1 = s \cup (r - w)$, where $s =$
$\rho_{R_1 \to R}(\pi_{R_1}(\sigma_{R_1 \hat{\sqsubset} R_2}(r \times r))) - \rho_{R_2 \to R}(\pi_{R_2}(\sigma_{R_1 \hat{\sqsubset} R_2}(r \times r)))$, and $w =$
$\rho_{R_1 \to R}(\pi_{R_1}(\sigma_{R_1 \hat{\sqsubset} R_2}(r \times r))) \cup \rho_{R_2 \to R}(\pi_{R_2}(\sigma_{R_1 \hat{\sqsubset} R_2}(r \times r)))$, respectively.

$i > 1$: $T_i = s \cup (r_i - w)$, where $r_i = (\cdots((r - T_1) - T_2) \cdots - T_{i-1})$, $s =$
$\rho_{R_1 \to R}(\pi_{R_1}(\sigma_{R_1 \hat{\sqsubset} R_2}(r_i \times r_i))) - \rho_{R_2 \to R}(\pi_{R_2}(\sigma_{R_1 \hat{\sqsubset} R_2}(r_i \times r_i)))$, and $w =$
$\rho_{R_1 \to R}(\pi_{R_1}(\sigma_{R_1 \hat{\sqsubset} R_2}(r_i \times r_i))) \cup \rho_{R_2 \to R}(\pi_{R_2}(\sigma_{R_1 \hat{\sqsubset} R_2}(r_i \times r_i)))$, respectively. □

Lemmas 3.13, 3.14 and 3.15 have practical significance as they indicate that a unique internal hierarchy can be generated by collecting the minimal tuples of a relation (or its subset) and in addition, using the PORA we can express a tuple level of such hierarchy for a given relation. The concept of tuple levels is very natural and easy to understand. In the special case of linearly ordered relations, T_i is the singleton containing the ith tuple. Thus, our choice of the *SELECT* statement in OSQL to include a *tuple list* specifying tuple levels in a relation can be justified by this formalism. (We will discuss OSQL in detail in Chapter 6.)

3.3 Non-Uniform Completeness of the PORA

In this section we present our result of a generalisation of Paredaens' and Bancilhon's Theorem [123, 12] to ordered databases. We begin this section by discussing the concept of order-preserving database automorphism and then we examine the expressive power of the PORA by Paredaens' and Bancilhon's Theorem.

3.3.1 Order-Preserving Database Automorphisms

We generalise the notion of automorphism [9] to the context of ordered databases. Informally, an *ordering automorphism* of an ordered set is a permutation of its elements such that the ordering of the set is preserved.

Definition 3.16 (Ordering Isomorphism and Ordering Automorphism) Let $\langle S, \sqsubseteq_S \rangle$ and $\langle T, \sqsubseteq_T \rangle$ be ordered sets. The function $f : S \longrightarrow T$ is an *ordering isomorphism* if f is bijective and f satisfies the condition that $a_1 \sqsubseteq_S a_2$ if and only if $f(a_1) \sqsubseteq_T f(a_2)$; $\langle S, \sqsubseteq_S \rangle$ and $\langle T, \sqsubseteq_T \rangle$ are *ordering isomorphic* if there exists an ordering isomorphism $f : S \longrightarrow T$. In particular, if $T = S$ and $\sqsubseteq_T = \sqsubseteq_S$, then we call f an *ordering automorphism* of $\langle S, \sqsubseteq_S \rangle$. If the set $\{a \in S \mid f(a) \neq a\}$ is finite, then we call f a *finite* ordering automorphism. We denote the set of all finite ordering automorphisms of an ordered set $\langle S, \sqsubseteq_S \rangle$ by $Aut(S, \sqsubseteq_S)$, or simply $Aut(S)$ when $\sqsubseteq_S$ is clear from the context.

We now define an order-preserving automorphism of a database. Informally, this is a permutation of the values in the active domain of a database instance that does not alter the database and also preserves the ordering of the active domain.

Definition 3.17 (Order-preserving Database Automorphism) Let h be a partial function from D to D such that it is an ordering automorphism of $\langle adom(d), \sqsubseteq \rangle$. The extension of h to tuples t, relations r and databases d is defined recursively as follows:

1. $h(t) = \langle h(a_1), \ldots, h(a_m) \rangle$, if $t = \langle a_1, \ldots, a_m \rangle$.
2. $h(r) = \{h(t_1), \ldots, h(t_k)\}$, if $r = \{t_1, \ldots, t_k\}$.
3. $h(d) = \{h(r_1), \ldots, h(r_n)\}$, if $d = \{r_1, \ldots, r_n\}$.

We call h an order-preserving database automorphism if its extension to d satisfies $h(d) = d$; by this we mean that $h(r_i) = r_i$ for $1 \leq i \leq n$. Furthermore, h can be regarded as an identity on $(D - adom(d))$ unless further specified. We denote the set of all order-preserving database automorphisms of database d by $Aut(\sqsubseteq, d)$, or simply $Aut(d)$ when $\sqsubseteq$ is clear from the context.

The following example should help to clarify the meaning of $Aut(d)$.

Example 3.6 Let d contain just a single relation having 4 tuples, $r = \{xz, yz, xw, yw\}$, and let $\langle adom(d), \sqsubseteq\rangle = \langle\{w, x, y, z\}, \{x \sqsubseteq y, x \sqsubseteq z, x \sqsubseteq w\}\rangle$. We define functions: h_1 by $h_1(x) = y$, $h_1(y) = x$, $h_1(z) = z$ and $h_1(w) = w$; h_2 by $h_2(x) = x$, $h_2(y) = z$, $h_2(z) = y$ and $h_2(w) = w$; and h_3 by $h_3(x) = x$, $h_3(y) = y$, $h_3(z) = w$ and $h_3(w) = z$. Then $h_1 \notin Aut(d)$ because, although it preserves the database instance, it does not preserve the ordering; and $h_2 \notin Aut(d)$ because, although it preserves the ordering, it does not preserve the database instance; however, $h_3 \in Aut(d)$ because it preserves both the ordering and the database instance.

It follows from Definition 3.17 that, for all partial orderings $\sqsubseteq$, $id \in Aut(\sqsubseteq, d) \subseteq Aut(=, d)$. Moreover, id is the only element of $Aut(\leq, d)$ for any linear ordering $\leq$. It also follows that $Aut(\sqsubseteq, d) = Aut(=, d) \cap Aut(adom(d), \sqsubseteq)$.

3.3.2 A Generalisation of Paredaens' and Bancilhon's Theorem

We now present our result of the generalisation of Paredaens' and Bancilhon's theorem in this subsection. The underlying principle in our approach is to view an ordered database as an unordered database together with a binary relation s representing $\langle adom(d), \sqsubseteq_D\rangle$. An ordered relation r derived from d is regarded as an unordered relation over $r \times \hat{s}$, where $\hat{s}$ is a binary relation representing $\langle adom(r), \sqsubseteq_D\rangle$. We assume from now on that (1) $adom(r) \subseteq adom(d)$ (we note that this is equivalent to assuming $r \in Poss(d)$), and (2) the set of constants using in selection formulas $C = \emptyset$. We need the following technical lemmas to establish our main theorem. The proofs of the next two lemmas follow from the definition of order-preserving automorphism.

Lemma 3.16 Let $d = \{r_1, \ldots, r_n\}$ be a database over $\{R_1, \ldots, R_n\}$, s be the unordered relation over S given by $s = \{\langle a, b\rangle \mid a \sqsubseteq b \text{ and } a, b \in adom(d)\}$, and let $d' = \{r_1, \ldots, r_n, s\}$ considered as an unordered database over $\{R_1, \ldots, R_n, S\}$. Then $Aut(=, d') = Aut(\sqsubseteq, d)$.
Proof.
We show $Aut(d') = Aut(d)$ by the following two parts.

($Aut(d') \subseteq Aut(d)$). Let $h \in Aut(d')$. It follows that $h(s) = s$ and thus $\langle a, b\rangle \in s$ if and only if $\langle h(a), h(b)\rangle \in s$. Hence, we have $\forall a, b \in adom(d)$, $a \sqsubseteq b$ if and only if $h(a) \sqsubseteq h(b)$. So $h \in Aut(d)$.

($Aut(d) \subseteq Aut(d')$). Let $h \in Aut(d)$. It follows from Definition 3.17 that h preserves the ordering of $adom(d)$ and $h(d) = d$. Thus, $\forall a, b \in adom(d)$, $a \sqsubseteq b$ if and only if $h(a) \sqsubseteq h(b)$. Hence, we have $h(s) = s$ and thus $h(d') = d'$. So $h \in Aut(d')$. □

For a relation r, we define $Aut(r) = Aut(\{r\})$.

Lemma 3.17 Let r be a relation over R, $\hat{s}$ be the unordered relation over S defined by $\hat{s} = \{\langle a, b\rangle \mid a \sqsubseteq b \text{ and } a, b \in adom(r)\}$, and let $r' = r \times \hat{s}$ considered as an unordered relation over RS. Then $Aut(=, r') = Aut(\sqsubseteq, r)$.

Proof.

We show $Aut(r) = Aut(r')$ by the following two parts.

($Aut(r) \subseteq Aut(r')$). Let $h \in Aut(r)$ and $t \in r'$. By the definition of r', it follows that $\exists t_1 \in r$ and $t_2 \in \hat{s}$ such that $t_1 = t[R]$ and $t_2 = t[S]$. By the definition of $Aut(r)$, we have $h(t_1) \in r$ and $h(t_2) \in \hat{s}$. It follows that $\exists t' \in r'$ such that $t'[R_1] = h(t_1)$ and $t'[S] = h(t_2)$. Thus, $h(t) \in r'$.

($Aut(r') \subseteq Aut(r)$). Let $h \in Aut(r')$ and $t_1 \in r$ and $t_2 \in \hat{s}$. Then $\exists t \in r'$ such that $t[R] = t_1$ and $t[S] = t_2$. By the definition of $Aut(r')$, we have $h(t) \in r'$. It follows that $h(t_1) \in r$ and $h(t_2) \in \hat{s}$. □

Defining d' and r' as the above two lemmas, the following result can be proved using induction on the number of relational operators together with some algebraic manipulation.

The next lemma follows from the previous two lemmas, where d' and r' are defined above.

Lemma 3.18 Let d be a database over $\mathbf{R}$ and r a relation over R. Then $e'(d') = r'$ for some $e' \in E_{UORA}$ if and only if $e(d) = r$ for some $e \in E_{PORA}$.

Proof.

IF: Let $e^k(d)$ be the answer to an expression in E_{PORA} with respect to d having k operators. We show by induction on the number of operators in e that $e(d') = r'$ for some $e \in E_{UORA}$. Let $e(d)$ be some relation r over R.

(*Basis*). We have $d = \{r\}$ and $d' = \{r, s\}$ where s is defined as in Lemma 3.16, it is trivial that $r' = r \times \hat{s}$ with $\hat{s} = s$.

(*Induction*). Assume that $e^k(d) = r$ and $e'(d') = r'$, where $k \geq 1$. For any operator $op \in$ PORA $-\{\sigma_{A \sqsubseteq B}\}$, we have $e'(d') = (op(\pi_R(r')))\times\hat{s}$ if op is unary and $e'(d') = (((\pi_R(r'_1))op\ ((\pi_R(r'_2)))\times\hat{s}$ if op is binary. For the operator $\sigma_{A \sqsubseteq B}$, we have $e'(d') = (\pi_R(\sigma_{AB=CD}(r')))\times\hat{s}$, where $\{C, D\}$ is the schema of $\hat{s}$. We note that $\hat{s}$ can be expressed by an UORA expression in all the above cases. Without loss of generality, assume op is unary, we have $\hat{s} = \pi_{CD}\sigma_{AB=CD}(e_{ad}(op(\pi_R(r'))) \times e_{ad}(op(\pi_R(r'))) \times s)$ (recall the definition of e_{ad} in Proposition 3.3).

ONLY IF: Similarly, we prove this part by induction.

(*Basis*). We have $d' = \{s\}$ and $d = \emptyset$. Thus, r and $s = \emptyset$. It is trivial that $e(d) = r$.

(*Induction*). Assume that $(e')^k(d') = r'$ and $e(d) = r$, where $k \geq 1$. We first show that s and $\hat{s}$ can be expressed by the PORA expressions, $s = \sigma_{A \sqsubseteq B}(e_{ad}(d) \times e_{ad}(d))$ and $\hat{s} = \sigma_{A \sqsubseteq B}(e_{ad}(r) \times e_{ad}(r))$, respectively. For any operator $op \in UORA - \{\sigma_{A=B}\}$, we have $e(d) = \pi_R(op(r \times s))$ if op is unary and $e(d) = \pi_{R_1 R_2}((r_1 \times s)op(r_2 \times s))$ if op is binary, where R_1 and R_2 are the schemas of r_1 and r_2, respectively. For the operator $\sigma_{A=B}$, we have $e(d) = \pi_R(\sigma_{A \sqsubseteq B}(\sigma_{B \sqsubseteq A}(r \times s)))$. □

In order to compare $Aut(d)$ and $Aut(r)$, we interpret $Aut(d) \subseteq Aut(r)$ as follows, for all $h \in Aut(d)$, $h(r) = r$ and h is a permutation of the elements in $(adom(d) - adom(r))$. Note that this interpretation is consistent with the usual meaning of set inclusion $\subseteq$ when $adom(d) = adom(r)$. Using our notation, we can state Paredaens' and Bancilhon's theorem in [123] as follows.

Lemma 3.19 Let d be an unordered database. Then $e(d) = r$ for some $e \in E_{UORA}$ if and only if $Aut(=, d) \subseteq Aut(=, r)$. □

We now show that this can be generalised to ordered databases.

Theorem 3.20 Let d be an ordered database over $\mathbf{R}$ and r an ordered relation over R. Then $e(d) = r$ for some $e \in E_{PORA}$ if and only if $Aut(\sqsubseteq, d) \subseteq Aut(\sqsubseteq, r)$.

Proof.

From Lemma 3.16, $Aut(=, d') = Aut(\sqsubseteq, d)$, and from Lemma 3.17, $Aut(=, r') = Aut(\sqsubseteq, r)$. The result then follows from Lemma 3.19, with d' substituted for d and r' for r, together with Lemma 3.18. □

We note that Theorem 3.20 can be straightforwardly extended to data domains having any specified binary relation, but in this case $\sigma_=$ may be primitive. Moreover, our result can be easily generalised to the case of $C \neq \emptyset$ by replacing $Aut(d)$ in Theorem 3.20 by the so-called *C-fixed* $Aut(d)$ (see section 2.3 in [9]), which is defined as $\{h \in Aut(d) \mid h$ is an identity on $C\}$.

We close this section with the following corollary, which is an interesting result that follows from Theorem 3.20. Informally, in the case of linearly ordered domains, the LORA expresses exactly the countably infinite set of all possible relations generated by the active domain of a given database.

Corollary 3.21 Let d be a linearly ordered database. Then $e(d) = r$ for some $e \in E_{LORA}$ if and only if $r \in Poss(d)$.

Proof.

IF: As we have observed, $Aut(d) = \{id\}$. Now, since $id \in Aut(r)$, the statement $Aut(d) \subseteq Aut(r)$ holds for all relations r in $Poss(d)$. Furthermore, LORA $\equiv$ PORA for linearly ordered databases. By Theorem 3.20, it follows that there exists $e \in E_{LORA}$ such that $e(d) = r$.
ONLY IF: It is trivial that for all $e \in E_{LORA}$, $e(d) \in Poss(d)$. $\square$

3.4 Hierarchy of Computable Queries with Ordered Domains

In this section we investigate the relationship between computable queries, ordered domains and partially ordered relational algebras. We first define a hierarchy for each of them and then we show that there exists a one-to-one correspondence between these three hierarchies.

We now use an index subscript to denote different orderings over D, i.e., $\mathcal{D}_i = \langle D, \sqsubseteq_i \rangle$ where i is a positive integer. We also use $Aut(\mathcal{D}_i)$ and $PORA_i$ to represent the set of ordering automorphisms and the PORA in which $\sigma_\sqsubseteq$ is $\sigma_{\sqsubseteq_i}$, respectively. The semantics of "more ordered" domains can be defined in terms of ordering automorphisms of the subsets of domains.

Definition 3.18 (More Ordered Domain) A domain $\mathcal{D}_2$ is said to be *more ordered* than another domain $\mathcal{D}_1$, denoted by $\mathcal{D}_1 \preceq \mathcal{D}_2$, if for all $T \subseteq D$, $Aut(T, \sqsubseteq_2) \subseteq Aut(T, \sqsubseteq_1)$.

The informal reason for allowing $T \subseteq D$ in the above definition is that we take into account the fact an active domain of a database can be defined on any subset of D. As a consequence of the definition, $Aut(d)$ would not be affected by the automorphisms induced from outside the active domain. Let us consider the following example.

Example 3.7 In Figure 3.2(a) we use Hasse diagrams representing ordered domains. Obviously, we have for all $T \subseteq D = \{a, b, c\}$, $Aut(T, \sqsubseteq_3) \subseteq Aut(T, \sqsubseteq_2) \subseteq Aut(T, \sqsubseteq_1)$ and thus the relationship $\mathcal{D}_1 \leq \mathcal{D}_2 \leq \mathcal{D}_3$ can be captured by Definition 3.18 in a natural manner.

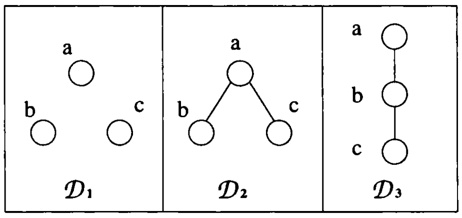


Figure 3.2: Hasse diagrams of ordered domains

Now we consider the expressiveness of the PORA for different orderings. Let the set of relations generated from the information contained in a given database d, denoted by $Gen(\sqsubseteq_i, d)$, be defined as $\{r \mid r = e(d) \text{ for some } e \in E_{PORA_i}\}$.

Definition 3.19 (More Powerful Relational Algebra) A relational algebra $PORA_2$ is *more powerful* than another $PORA_1$, denoted by $PORA_1 \preceq PORA_2$, if for all databases d, $Gen(\sqsubseteq_1, d) \subseteq Gen(\sqsubseteq_2, d)$.

If $PORA_2$ is a more powerful language than $PORA_1$, then we can retrieve more relations from a given database instance using $PORA_2$. We still need to make an extension of the notion of computable query for ordered databases, but we take a different approach from [28]. The motivation for our definition is to include those queries which are meaningful with respect to the ordered domain concerned. The criteria for being meaningful over an ordered database d is that the query must be invariant under all order-preserving database automorphisms over d.

Let $DB(\mathbf{R})$ be the countably infinite set of all databases defined over a database scheme $\mathbf{R}$ and let $\chi = \bigcup_{i=0}^{\infty} \mathcal{P}(D^i)$. (Recall that we assume that D is a common domain for all attributes.)

Definition 3.20 (Meaningful Computable Query) A *meaningful computable query* with respect to a given domain $\mathcal{D}_i$, denoted by δ, is a partial recursive function from $DB(\mathbf{R})$ to χ such that for all $d \in DB(\mathbf{R})$,

1. if $\delta(d)$ is defined, then $\delta(d) \in Poss(d)$, and
2. for all $h \in Aut(\sqsubseteq_i, d)$, $h(\delta(d)) = \delta(d)$.

We denote the set of all meaningful computable queries by Q_i.

Note that our definition of a meaningful computable query is the same as the conventional one if we restrict ourselves to unordered domains. Now we state two technical lemmas and then present our main theorem. The first lemma follows easily from Theorem 3.20 and Lemma 3.17. It can be regarded as a generalisation of Lemma 3.17 to databases. The second lemma is useful when we compare different ordered databases. Basically it allows us to consider ordering automorphisms on the underlying domain instead of automorphisms on databases.

Lemma 3.22 Let $d = \{r_1, \ldots, r_n\}$ be a database over $\{R_1, \ldots, R_n\}$, s be the unordered relation over S given by $s = \{\langle a, b\rangle \mid a \sqsubseteq b \text{ and } a, b \in adom(d)\}$, and let $r = r_1 \times \cdots \times r_n \times s$, considered as an unordered relation over $R_1 \cdots R_n S$. Then $Aut(\sqsubseteq, d) = Aut(=, r)$.

Proof.

We show $Aut(\sqsubseteq, d) = Aut(=, r)$ by the following two parts.

$(Aut(\sqsubseteq, d) \subseteq Aut(=, r))$. We let $r' = r_1 \times \cdots \times r_n$, which is an ordered relation obtained by the PORA expression as shown. By Theorem 3.20, it follows that $Aut(\sqsubseteq, d) \subseteq Aut(\sqsubseteq, r')$ and also by Lemma 3.17, it follows that $Aut(\sqsubseteq, r') \subseteq Aut(=, r)$. So we have $Aut(\sqsubseteq, d) \subseteq Aut(=, r)$.

$(Aut(=, r) \subseteq Aut(\sqsubseteq, d))$. Let $h \in Aut(=, r)$. We claim that $h(r_i) = r_i$ and $h(s) = s$. Assume to the contrary that this claim does not hold. Then we have either $h(r_i) \neq r_i$ or $h(s) \neq s$. Assume that $h(r_i) \neq r_i$, then $\exists t \in r_i$ such that $h(t) \notin r_i$. Let $t' \in r$ such that $t'[R_i] = t$. Thus it follows that $h(t') \notin r$, which leads to contradiction, since we assume $h \in Aut(=, r)$. The argument is similar to the case of $h(s) \neq s$. We now have $h \in Aut(=, d')$, where $d' = \{r_1, \ldots, r_n, s\}$. By Lemma 3.16, it follows that $Aut(=, d') = Aut(\sqsubseteq, d)$. Thus, $h \in Aut(\sqsubseteq_D, d)$. $\square$

Lemma 3.23 $\mathcal{D}_1 \preceq \mathcal{D}_2$ if and only if $Aut(\sqsubseteq_2, d) \subseteq Aut(\sqsubseteq_1, d)$ for all databases d over $\mathbf{R}$.

Proof.

IF: Consider any $h \in Aut(T, \sqsubseteq_2)$ with $T \subseteq D$. Let $X = \{a \in T \mid a \neq h(a)\}$ and, since h is a finite automorphism, suppose $X = \{a_1, \ldots, a_k\}$. Define a database d over $\mathbf{R}$ as follows, for all $r \in d$, r consists of exactly k tuples $\{t_1, \ldots, t_k\}$ for some finite natural number k, where $t_i = \langle a_i, \ldots, a_i \rangle$ for $1 \leq i \leq k$. Obviously, we have that $h \in Aut(\sqsubseteq_2, d)$. By hypothesis, this implies $h \in Aut(\sqsubseteq_1, d)$ and thus $h \in Aut(T, \sqsubseteq_1)$.

ONLY IF: This follows easily by using the fact that $Aut(\sqsubseteq_i, d) = Aut(=, d) \cap Aut\ (T, \sqsubseteq_i)$ for any database d, where $T = adom(d)$. □

We now present our main result stating the association between domains, queries and languages. This allows us to establish hierarchies for these entities.

Theorem 3.24

1. $\mathcal{D}_1 \preceq \mathcal{D}_2$ if and only if $Q_1 \subseteq Q_2$,

2. $\mathcal{D}_1 \preceq \mathcal{D}_2$ if and only if $PORA_1 \preceq PORA_2$.

Proof.

(1) *IF:* Assume $\mathcal{D}_1 \not\preceq \mathcal{D}_2$, by Definition 3.18 and Lemma 3.23, this implies that there exists a database d' such that $h_2 \notin Aut(\sqsubseteq_1, d')$ for some $h_2 \in Aut(\sqsubseteq_2, d')$. Let $d' = \{r_1, \ldots, r_n\}$. We now construct an instance of a query that is in Q_1 but not in Q_2. We substitute $d = d'$ and $r = r'$ in Lemma 3.22. Then we have that for all $h \in Aut(\sqsubseteq_1, d')$, $h(r') = r'$. On the other hand, $h_2(r') \neq r'$ since $h_2 \notin Aut(\sqsubseteq_1, d')$. We define a query δ as follows: $\delta(d) = r'$ when $d = d'$ and $\delta(d)$ is equal to $\emptyset$ otherwise. By part (2) of Definition 3.20, $\delta \in Q_1$ but $\delta \notin Q_2$.

ONLY IF: Let $\delta \in Q_1$ and $d \in DB(\mathbf{R})$. From part (1) of Definition 3.20, $\delta(d) \in Poss(d)$ and from part(2) of Definition 3.20, for all $h \in Aut(\sqsubseteq_1, d)$, $h(\delta(d)) = \delta(h(d))$. By the assumption $\mathcal{D}_1 \preceq \mathcal{D}_2$ and Lemma 3.23, $Aut(\sqsubseteq_2, d) \subseteq Aut(\sqsubseteq_1, d)$. Therefore, for all $h \in Aut(\sqsubseteq_2, d)$, $h(\delta(d)) = \delta(h(d))$ and thus $\delta \in Q_2$.

(2) *IF:* Assume $\mathcal{D}_1 \not\preceq \mathcal{D}_2$, by Definition 3.18 and Lemma 3.23, there exists a database $d' = \{r_1, \cdots, r_n\}$ such that $Aut(\sqsubseteq_2, d') \not\subseteq Aut(\sqsubseteq_1, d')$. It suffices to exhibit a database d and a relation r such that $r \in Gen(\sqsubseteq_1, d)$ but $r \notin Gen(\sqsubseteq_2, d)$. We let $d = d'$ and

$r = r_1 \times \cdots \times r_n \times s$ and $s = \{\langle a, b\rangle \mid a \sqsubseteq_1 b \text{ and } a, b \in adom(d')\}$, respectively. Clearly, s can be derived from d by some $e \in PORA_1$ and thus $r \in Gen(\sqsubseteq_1, d)$. It remains to show $r \notin Gen(\sqsubseteq_2, d)$. Suppose $r \in Gen(\sqsubseteq_2, d)$. By Theorem 3.20, $Aut(\sqsubseteq_2, d') \subseteq Aut(\sqsubseteq_2, r) = Aut(=, r) \cap Aut(adom(r), \sqsubseteq_2)$. It follows that $Aut(\sqsubseteq_2, d') \subseteq Aut(=, r)$. By Lemma 3.22 it follows that $Aut(\sqsubseteq_2, d') \subseteq Aut(\sqsubseteq_1, d')$, which leads to a contradiction.

ONLY IF: Let $r \in Gen(\sqsubseteq_1, d)$. We need to show that $r \in Gen(\sqsubseteq_2, d)$. By Theorem 3.20, $Aut(\sqsubseteq_1, d) \subseteq Aut(\sqsubseteq_1, r)$. Thus $Aut(adom(d), \sqsubseteq_2) \cap Aut(\sqsubseteq_1, d) \subseteq Aut(adom(d), \sqsubseteq_2) \cap Aut(\sqsubseteq_1, r)$. Moreover, we have $Aut(\sqsubseteq_1, d) = Aut(adom(d), \sqsubseteq_1) \cap Aut(=, d)$ and $Aut(\sqsubseteq_1, r) = Aut(adom(d), \sqsubseteq_1) \cap Aut(=, r)$. It follows that $Aut(adom(d), \sqsubseteq_2) \cap Aut(adom(d), \sqsubseteq_1) \cap Aut(=, d) \subseteq Aut(adom(d), \sqsubseteq_2) \cap Aut(adom(d), \sqsubseteq_1) \cap Aut(=, r)$. By the assumption of $\mathcal{D}_1 \preceq \mathcal{D}_2$ and by Definition 3.18, we have $Aut(adom(d), \sqsubseteq_2) \subseteq Aut(adom(d), \sqsubseteq_1)$. It follows that $Aut(adom(d), \sqsubseteq_2) \cap Aut(=, d) \subseteq Aut(adom(d), \sqsubseteq_2) \cap Aut(=, r)$. Hence we have $Aut(\sqsubseteq_2, d) \subseteq Aut(\sqsubseteq_2, r)$. By Theorem 3.20 again, we have $r \in Gen(\sqsubseteq_2, d)$. □

The following corollary states that there is a correspondence between the set of meaningful computable queries and the relational algebra. Informally the relational algebra $PORA_i$ (non-uniformly) expresses the result of Q_i on a fixed database instance. Therefore in this sense we can say that the language $PORA_i$ is *non-uniformly complete.*

Corollary 3.25 $Q_1 \subseteq Q_2$ if and only if $PORA_1 \preceq PORA_2$.

Queries	$Q_= \subseteq$	$\cdots$	$\subseteq Q_i \subseteq$	$\cdots$	$\subseteq Q_\leq$
	$\updownarrow$		$\updownarrow$		$\updownarrow$
Domains	$(D, =) \preceq$	$\cdots$	$\preceq (D, \sqsubseteq_i) \preceq$	$\cdots$	$\preceq (D, \leq)$
	$\updownarrow$		$\updownarrow$		$\updownarrow$
Algebras	$PORA_= \preceq$	$\cdots$	$\preceq PORA_i \preceq$	$\cdots$	$\preceq PORA_\leq$

Figure 3.3: A correspondence between hierarchies of queries, domains and languages

We present the diagram in Figure 3.3, which summarises the relationship between the hierarchies of (1) meaningful computable queries, (2) partially ordered domains, and (3) partially ordered relational algebras we have discussed. The implications of this result are that if the underlying data domains of an ordered database have more inherent structure, then a wider scope of queries are possible. In other words, the ordered relational model can provide more expressive query languages than those of the conventional one, and in this sense we can say that more meaningful queries are possible with respect to an ordered relational database. There is still an open problem to find a syntactic characterisation that is equivalent to the definition of a more ordered domain.

3.5 Updating Ordered Databases

There has been a fair amount of research on the topic of updates in conventional databases [4]. The problem of updating databases can be further partitioned by considering three perspectives related to the relational data model, which are listed as follows.

1. Updating views at the external level of a DBMS.

2. Updating relations at the conceptual level of a DBMS.

3. Updating the underlying domain of attributes.

Each of these three kinds of updates is related to the others. The first category of updates is still an on-going research issue, which concerns achieving *logical data independence* of conventional databases or their extensions such as incomplete databases [58]. Basically, a view in a database tailors the database to different requirements of a variety of database users. For example, using the view facilities provided by a DBMS, the DataBase Administrator (DBA) can choose to hide certain information in a database for some security reasons, or to *materialise* a view so as to facilitate the very recent strategy of *data warehousing* used in the commercial sector [69, 107]. However, the relational data model does not provide the users with full support of logical data independence. For example, if a view is obtained by projection on a relation r, then deleting a tuple from the view may lead to ambiguity in deleting the corresponding tuple in r, since a projected tuple may come from many possible tuples in r. Thus it results in the so-called *view update* problem as follows, given a view and an update against this view, how to translate the given update into an appropriate update against its underlying relational database without causing unnecessary loss of information.

The view update problem occurs similarly in ordered databases. There is no difference in the ordered relational model in this case of updating unless it affects the underlying domain. One approach to solve this problem is to restrict view updates in certain types [27] in order to prevent an inconsistent database occurring at the *conceptual level.* Another approach is to provide users with a *universal relation interface* [147], which can achieve logical database independence by allowing users to view the database as if it were composed of a single relation.

The problem of updating relations in the context of ordered databases is related to a more fundamental question of updating domains. Suppose a tuple involving a value which is associated with some new semantics of its domain is inserted into an ordered relation, then the semantic domain may need to be updated. For example, when a new manager is employed in a company, the boss/subordinate relationship may be changed and in such a case the domain EMP_NAME should be updated to reflect the new hierarchy of employees. It is interesting to note that some semantic domains are relatively static and they can be regarded as *intensional data* [9] such as relation schemas. For example, the semantic domain of the post ranks in a university, {lecturer $\leq$ senior lecturer $\leq$ professor} is invariant with respect to database instances. Suppose a tuple involving a value which associates with some semantics of its domain is deleted from a relation, then this value may need to be deleted from the semantic domains. For example, when a manager has left a company, the boss/subordinate relationship may be changed and the domain EMP_NAME should be updated accordingly.

We now address the issue of updating ordered domains. By updating we mean a sequence of delete or insert operations. Let us begin with two basic assumptions regarding ordered domains. First, we assume that the domain $\mathcal{D}$ we intend to update is finite. Second, we assume the uniqueness name axiom, i.e., each domain value is required to have a unique name which is distinguishable from other names of values.

We first consider the special case of $\mathcal{D}$ being unordered. Suppose we want to delete an element a from $\mathcal{D}$. If this value does not occur in r (i.e., $a \notin adom(r)$), then in principle, there should be no restriction on such a delete on $\mathcal{D}$, since it does not affect r. Otherwise, a delete operation should be carried out on the affected tuple (i.e., those tuples containing a) in r prior to the delete being carried out on $\mathcal{D}$. In such a case there are three approaches we can use to eliminate the occurrences of a from r.

1. To replace the occurrence of a with *null values* in the affected tuples if null symbols such as UNK are provided (i.e., an incomplete relation is defined).

2. To remove all the affected tuples from r.

3. To reject such a delete operation on $\mathcal{D}$.

We believe that all the above approaches are reasonable in practice. Thus the semantics of deleting an element from a domain should be further clarified. For example, an employee record $\langle$*Jose*, *junior_programmer*, $12K$ $\rangle$ in a relation EMP_RECORD over the schema {EMP_NAME, POST_TITLE, SALARY} represents the fact that the employee Jose, who is a junior programmer, has salary 12K. Then there are several possible semantics for deleting the value "junior programmer" in the corresponding domain of POST_TITLE. One scenario is that the post title is being changed but the company has not yet formerly approved a new title. Then the first approach is appropriate. Another scenario is that all junior programmers have left their jobs and thus the company changes its management structure and decides that this post title will not be used in future. Then the second approach is appropriate. The third scenario is that the DBA wants to delete the post title only if there is no employee still possessing such a post title in the EMP_RECORD. It is reasonable to expect that the DBMS should provide the users with further guidance to carry out this process.

The issue of maintenance of the partial ordering in a domain is also essential when updating a domain. The strategy we adopt is to keep the change to be minimal with respect to the ordering on the domain. We formalise this concept as the following definition.

Definition 3.21 (Order-Preserving Updates) Let $\mathcal{D} = \langle D, \sqsubseteq_D \rangle$ be a domain such that after it has been updated becomes $\mathcal{D}_1 = \langle D_1, \sqsubseteq_{D_1} \rangle$. We call the update an *order-preserving update* if $a \sqsubseteq_D b$ implies that $a \sqsubseteq_{D_1} b$ for any pair of elements $a, b \in D_1$.

We observe that when deleting an element a from $\mathcal{D}$ the ordering between its *successor* and *predecessor* should also be removed. If a is a minimal element in $\mathcal{D}$, then the delete is an order-preserving update; otherwise, we have many possible ways to define the new ordering of the elements that are previously connected to a in $\mathcal{D}$ and the delete may not be an order-preserving update.

Definition 3.22 (Delete Operator) Let *del*(a) denote the deletion of the element a from a domain $\mathcal{D}$. We call the process of invoking *del*(a) on $\mathcal{D}$ the *delete operation.* The delete operator can be classified into the following three modes of α-*del*, β-*del* and γ-*del*, respectively.

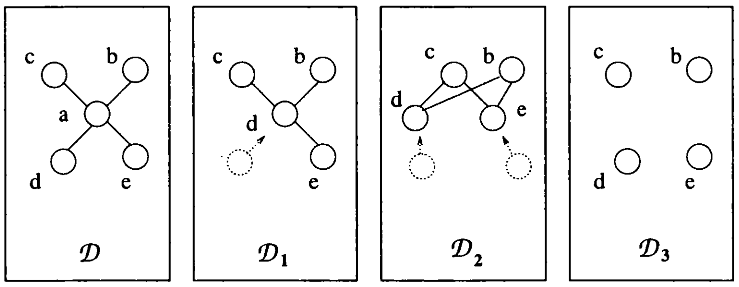
1. The delete operator in α *mode*, denoted as α-*del*(a), represents the process of deleting an element a from $\mathcal{D}$ and then promoting one of its successors to its original position.

2. The delete operator in β *mode*, denoted as β-*del*(a), represents the process of deleting an element a from $\mathcal{D}$ and then connecting all its successors to all of its predecessors (if they exist).

3. The delete operator in γ *mode*, denoted as γ-*del*(a), represents the process of deleting an element a from $\mathcal{D}$ without adjusting the ordering of its successors and predecessors, i.e., only removing $x \sqsubseteq_D y$ whenever $x = a$ or $y = a$.

The following example illustrates the delete of an element a from domain $\mathcal{D}$ via α-*del*, β-*del* and γ-*del* to become $\mathcal{D}_1$, $\mathcal{D}_2$ and $\mathcal{D}_3$, respectively.

Example 3.8 Figure 3.4(a) shows the domain $\mathcal{D}$ before the delete operation *del*(a). Figure 3.4(b) shows that the domain $\mathcal{D}_1$, which is the result of α-*del*(a). Note that the choice of a successor (elements d or e) is system dependent. Figure 3.4(c) shows that the domain $\mathcal{D}_2$, which is the result of β-*del*(a). Again, the choice of a successor is system dependent. Figure 3.4(d) shows that the domain $\mathcal{D}_3$, which is the result of γ-*del*(a). Thus, the domain is more fragmented than the mentioned ones.



(a) before delete (b) α-*del*(a) (c) β-*del*(a) (d) γ-*del*(a)

Figure 3.4: Various modes of deleting an element a from $\mathcal{D}$

The choice of deletion modes depends on the underlying semantics of the delete operation. For example, if $\mathcal{D}$ is a semantic domain representing the hierarchy in a company, then the operator *α-del* may be used to promote an employee to replace a retired manager. The operator *β-del* may be used when a manager is retired, and his subordinates are all assigned to report to his bosses. The last operator *γ-del* may be used to represent a step prior to the process of re-defining all ranks of employees. It is reasonable to expect that a DBMS should provide the users with further guidance to carry out this process.

We now consider inserting an element into $\mathcal{D}$. When an element a is added into $\mathcal{D}$, we should define the new ordering relationships between a and the existing elements in $\mathcal{D}$, which can be either a *successor* or a *predecessor* relationship. We formalise this concept as follows.

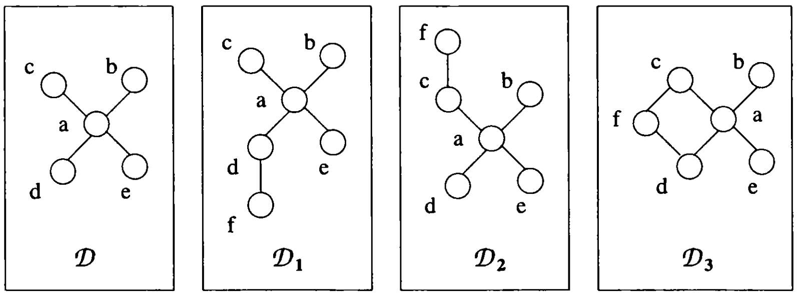
Definition 3.23 (Insert Operator) Let x be an element in $\mathcal{D} = \langle D \sqsubseteq_D \rangle$ and a be a new element which after adding it to $\mathcal{D}$ becomes $\mathcal{D}_1 = \langle D_1 \sqsubseteq_{D_1} \rangle$. The *successor operator*, denoted as $succ(x, a)$, creates an ordering $x \sqsubseteq_{D_1} a$. The *predecessor operator*, denoted as $pred(a, x)$, creates an ordering $a \sqsubseteq_{D_1} x$. The *insert operator*, denoted as $ins(a)$, consists of a finite set of operations $succ(x, a)$ and $pred(a, x)$, such that if $x_1 \sqsubseteq_D x_2 \sqsubseteq_D \cdots \sqsubseteq_D x_n$, then either $pred(a, x_1) \notin ins(a)$ or $succ(x_n, a) \notin ins(a)$. We call the process of invoking the insert operator on $\mathcal{D}$ the *insert operation*.

Note that the restriction $pred(a, x_1) \notin ins(a)$ or $succ(x_n, a) \notin ins(a)$ in the above definition is to prevent a "cyclic ordering" occurring in $\mathcal{D}_1$, for example if $b \sqsubseteq_D c$, then $ins(a) = \{pred(a, b), succ(c, a)\}$ results in $a \sqsubseteq_{D_1} b \sqsubseteq_{D_1} c \sqsubseteq_{D_1} a$, which violates the anti-symmetric criteria of a partial ordering (see Definition 2.1). Moreover, it can be checked that $\sqsubseteq_D \subseteq \sqsubseteq_{D_1}$. Thus the original ordering of $\mathcal{D}$ is preserved by the insert operation. We state this fact as the following observation.

Observation 3.1: The operator *ins* is an ordering-preserving update.

Example 3.9 Let us consider an element f inserted into the domain $\mathcal{D}$ as given in Figure 3.5(a). The following are three possible insert operations: $ins_1(f) = \{pred(f, d)\}$, $ins_2(f) = \{succ(c, f)\}$ and $ins_3(f) = \{succ(d, f), pred(f, c)\}$, which result in the domains $\mathcal{D}_1$, $\mathcal{D}_2$, and $\mathcal{D}_3$, as given in Figure 3.5(b), Figure 3.5(c) and Figure 3.5(d), respectively.

We recall that updating is considered to be a sequence of insert and delete operations. From this point of view, we can first use the delete operator *γ-del* repeatedly to remove all



(a) Before insert (b) $ins_1(f)$ (c) $ins_2(f)$ (d) $ins_3(f)$

Figure 3.5: Inserting an element f into $\mathcal{D}$

the existing elements in $\mathcal{D}$. Thereafter we use the insert operator *ins* to form an arbitrary new domain $\mathcal{D}_2$ by successively inserting elements to $\mathcal{D}_2$ with the desired ordering defined by an appropriate set of the *succ* and *pred* operators. The expressiveness of insert and delete operations in updating a domain can be informally stated as the following observation.

Observation 3.2: The two operators $ins(a)$ and $\gamma\text{-}del(a)$ are sufficient to transform an ordered domain $\mathcal{D} = \langle D, \sqsubseteq_D \rangle$ into another ordered domain $\mathcal{D}_1 = \langle D_1, \sqsubseteq_{D_1} \rangle$, where $D_1 = D \cup \{a\}$.

Note that if $\mathcal{D}$ is infinite, then the operator *ins* may not be capable of defining the new ordering of $\mathcal{D}_1$, since it is a finite set of *prec* and *succ*. In such a case the above observation is not applicable. For example, if $\mathcal{D}$ is unordered, we cannot have an insert operation to obtain a new domain $\mathcal{D}_1$ such that $\mathcal{D}_1 = \mathcal{D} \cup \{a\}$ and $b \sqsubseteq_{D_1} a$ for all $b \in D$.

3.6 Discussion

In this section we briefly discuss the open problem of finding a syntactic characterisation of "more ordered" that is equivalent to Definition 3.18. Recall that $\mathcal{D}_1 = \langle D, \sqsubseteq_1 \rangle$ and $\mathcal{D}_2 = \langle D, \sqsubseteq_2 \rangle$. We first state the problem as follows, given two domains $\mathcal{D}_1$ and $\mathcal{D}_2$, is there a characterisation of their structures such that $\mathcal{D}_1 \preceq \mathcal{D}_2$ if and only if $Aut(\mathcal{D}_2) \subseteq Aut(\mathcal{D}_1)$?

A possible attempt of defining "more ordered domains", which corresponds to the intuitive view of "more ordered" can be described as follows, if $\mathcal{D}_2$ is more ordered than $\mathcal{D}_1$, then those pairs of elements ordered by $\sqsubseteq_1$ are also ordered by $\sqsubseteq_2$. From this point of view, we now give the following definition.

Definition 3.24 (More Ordered Domains) $\mathcal{D}_1 \preceq \mathcal{D}_2$ if for all elements $a_1, a_2 \in D$, if $a_1 \sqsubseteq_1 a_2$, then $a_1 \sqsubseteq_2 a_2$.

Although this definition seems to be very natural, it is far too simple to be a complete solution to the problem. There are many cases happening that $\mathcal{D}_1 \not\preceq \mathcal{D}_2$ according to Definition 3.24 but $Aut(\mathcal{D}_2) \subseteq Aut(\mathcal{D}_1)$. The following is one of the counter examples.

Example 3.10 In Figure 3.6, the component $b \sqsubseteq_1 a$ in $\mathcal{D}_1$ but $b \not\sqsubseteq_2 a$ in $\mathcal{D}_2$, thus $\mathcal{D}_1 \not\preceq \mathcal{D}_2$. However, we still have that $Aut(\mathcal{D}_2) \subseteq Aut(\mathcal{D}_1)$ because $Aut(\mathcal{D}_2) = \{id, h\}$, where h is an automorphism defined by $h(a) = a, h(b) = b, h(c) = d$ and $h(d) = c$, which is equal to $Aut(\mathcal{D}_1)$.

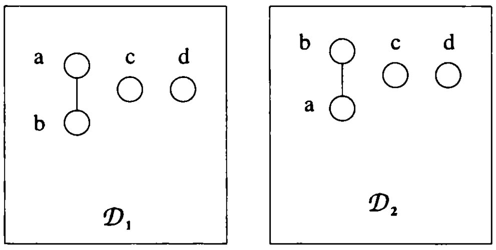


Figure 3.6: A counter example of Theorem 3.24 arising from Definition 3.24

From our further investigation, we find that there may be a dichotomy between the notion of automorphism and ordering of domains. Informally speaking, if we focus ourselves mainly on the ordering structures of domains as stated in Definition 3.24, we cannot establish a simple relationship between automorphisms of two domains.

Chapter 4

Data Dependencies and Database Design Issues for the Ordered Relational Model

Functional dependencies (FDs) [147, 9] and *inclusion dependencies* (INDs) [109, 22] are commonly recognised as the most fundamental data dependencies that arise in practice in conventional relational databases. In this chapter we examine these data dependencies in the context of ordered relational databases.

We extend the notions of FDs and INDs to hold in an ordered database and call them *ordered functional dependencies* (OFDs) and *ordered inclusion dependencies* (OINDs), respectively. Informally speaking, OFDs can capture a monotonicity property between two sets of values projected onto some attributes in a relation, and OINDs can capture the notion of a *Hoare ordering* (recall Definition 2.7) between two sets of values projected onto some attributes in a database. We also discuss the interactions between OFDs and OINDs.

The semantics of OFDs and OINDs are defined by means of two possible extensions of the domain orderings: *pointwise-orderings* and *lexicographical orderings*, whose semantics have been discussed in Chapter 2. We classify OFDs and OINDs according to whether we use pointwise-orderings or lexicographical orderings in their definitions. Altogether there are four categories of data dependencies, whose short forms are written as POFDs, LOFDs, POINDs and LOINDs, respectively. A summary of our classification of OFDs and OINDs with examples of their notations is given as the table in Figure 4.1

Data Dependencies	**Pointwise-Orderings**	**Lexicographical Orderings**
Ordered Functional Dependencies (OFDs)	POFD $X \hookrightarrow Y$	LOFD $X \rightsquigarrow Y$
Ordered Inclusion Dependencies (OINDs)	POIND $R[X] \hat{\subseteq} S[Y]$	LOIND $R[X] \tilde{\subseteq} S[Y]$

Figure 4.1: OFDs and OINDs arising from different extensions of domain orderings

In the relational database literature, the *implication problem* is an important issue arising from investigating data dependencies, which we now state as follows: given a relation r which satisfies a set of data dependencies F, is it also true that r satisfies a data dependency f? If the answer to the above question is positive, then we say F *logically implies* f and denote this fact by F $\models f$. There are two approaches to tackle this problem.

One approach is to establish a set of *inference rules* which constitutes the *axiom system* $\mathcal{A}$. Hence, we can use the rules of $\mathcal{A}$ to *derive* f from F and denote this process by F $\vdash f$. We call $\mathcal{A}$ *sound and complete*, if we can prove that F $\vdash f$ if and only if F $\models f$. A sound and complete axiom system for F is very desirable, since it guarantees the implication problem for F is *recursively enumerable.* This is due to the fact that in principle, we can exhaustively apply the rules of $\mathcal{A}$ to generate all data dependencies that can be logically implied by F. In addition, the axiom system $\mathcal{A}$ provides us with a basis to find a more efficient algorithm for solving the implication problem.

Another approach is to develop a *chase* procedure which consists of a set of *chase rules* as a theorem proving tool. We choose an appropriate chase rule to apply to a relation r until a fixpoint is attained in order to test whether r satisfies F [101, 9, 87]. Moreover, the chase procedure operates on a relation containing variables as data values, known as a *tableau* [5, 9], which is basically the template for those relations that could possibly violate f. Suppose we can prove that using the chase procedure we can transform a tableau that satisfies F into a tableau that also satisfies f, and this holds if and only if F $\models f$. Then we are able to use the chase procedure to confirm or refute that F logically implies f. We call the chase procedure possessing this properly *sound and complete.*

We assume that the domains are linearly ordered in discussing the issues of OFDs and OINDs. In Section 4.1 we present FDs in the context of ordered databases. In Section 4.2 we adopt the first mentioned approach to show that the axiom system comprising the inference rules for POFDs, which is a superset of Armstrong's axiom system for FDs, are sound and complete. We adopt the second mentioned approach to extend the chase rules for the case of LOFDs. We investigate the properties of a relation r being chased with respect to a set of LOFDs F (which we denote as $CHASE(r,\mathrm{F})$) and then show that the procedure $CHASE(r,\mathrm{F})$ terminates and satisfies F. Moreover, using an extended notion of *tableaux* for LOFDs, we show that the chase is sound and complete for LOFDs. Hence, the implication problems for POFDs and LOFDs are *decidable* and it is *linear time* for POFDs. We also present a set of inference rules for LOFDs, which are shown to be sound but we do not know if they are complete. In Section 4.3 we generalise the definition of conventional inclusion dependencies to Ordered INclusion Dependencies (which we call OINDs). We present a set of inference rules for POINDs and LOINDs, respectively, which are both shown to be sound. We show that the axiom system comprising the inference rules for POINDs is also complete. The interaction between OINDs and OFDs is also discussed. In Section 4.4 we discuss some database design issues for ordered databases in the presence of OFDs and OINDs. In Section 4.5 we give our concluding remarks for this chapter.

Throughout this Chapter we refer to a sequence of attributes as a short hand for a sequence of distinct attributes. (In other words, we assume that sequences of attributes do not contain any repeated attributes.) We use the common notation for both sequences and sets, i.e., X and Y are used to denote sequences of attributes, whereas A and B are used to denote single attributes. When no ambiguity arises we refer to a sequence of attributes as a set of attributes. However, we remark that the sequence AB and BA are different, whereas the sets AB and BA are the same. We take $A \in \langle A_1, \dots, A_n \rangle$ to mean $A \in \{A_1, \dots, A_n\}$ and $\langle A_1, \dots, A_n \rangle \subseteq \langle B_1, \dots, B_m \rangle$ where $n \leq m$, to mean $\{A_1, \dots, A_n\} \subseteq \{B_1, \dots, B_m\}$. We may also write $A_1 \cdots A_n$ instead of $\langle A_1, \dots, A_n \rangle$ to describe a sequence when convenient.

Let $X = \langle A_1, \dots, A_m \rangle$ and $Y = \langle B_1, \dots, B_n \rangle$. We denote the fact that two sequences have the same elements, i.e., $\{A_1, \dots, A_m\} = \{B_1, \dots, B_n\}$, by $X \sim Y$. The difference between two sequences of attributes, denoted as $X - Y$, is defined by the sequence resulting from removing all the common attributes in X and Y from X while maintaining the original order of the remaining attributes in X. Y is said to be a *subsequence* of X if $Y \subseteq X$ and the attributes of Y maintain the original order of X. We also denote by XY the *concatenation* of two sequences X and Y, where X and Y are *disjoint*, i.e., they have no common attributes. If X and Y are not disjoint, then XY is defined to be $X(Y-X)$. A *prefix* of X, denoted by *pre*(X), is a sequence of the form $\langle A_1, \dots, A_{m_1}\rangle$ where $1 \leq m_1 \leq m$. A *shuffle* of X and Y, denoted by $shu(X,Y)$, is defined as a sequence of the form $\langle C_1, \dots, C_{m+n}\rangle$, where there exists two subsequences of attributes $\langle C_{i_1}, \dots, C_{i_m}\rangle = X$ and $\langle C_{j_1}, \dots, C_{j_n}\rangle = Y$. For example, let $X = \langle a, b, c\rangle$ and $Y = \langle 1, 2, 3\rangle$. Then both $\langle a, 1, 2, 3, b, c\rangle$ and $\langle 1, a, 2, b, c, 3\rangle$ are $shu(X,Y)$. However, $\langle b, c, a, 1, 2, 3\rangle$ is not because the ordering of X is not maintained.

4.1 Functional Dependencies (FDs) in Ordered Databases

Bearing in mind that the implication problem is an important issue arising in developing the theory of data dependencies and FDs are the most natural data dependencies arising in practice, we first formalise the notion of *logical implication* and *an axiom system*, and then review *Armstrong's axiom system* for FDs, which is a classical example of *axiom systems* in the literature of relational database theory [147, 9].

Definition 4.1 (Logical Implication and Axiom System) We say that a set of data dependencies F *logically implies* a data dependency f over R, written F $\models f$, whenever for all relations r over R, if for all $f' \in$ F, $r \models f'$ holds, then $r \models f$ also holds. An *axiom system* $\mathcal{A}$ for F is a set of inference rules (or simply rules) that can be used to *derive* data dependencies from F over R. We say that f is *derivable* from F by $\mathcal{A}$, if there is a finite sequence of data dependencies over R, whose last element is f, and where each data dependency in the said sequence is either in F or follows from a finite number of previous data dependencies in the sequence by one of the inference rules. We denote by F $\vdash f$ the fact that f is *derivable* from F by a specified axiom system.

We remark that Definition 4.1 will be repeatedly used in different contexts of data dependencies. For example, in this section the set of data dependencies F is restricted to the scope of FDs. However, when discussing OFDs in the next section, we will use F

$\vdash f$ to mean that a set of OFDs F logically implies an OFD f. Similarly, we will also use F $\vdash f$ to mean that a set of OINDs F logically implies an OIND f when discussing OINDs in Section 4.3.

Armstrong's axiom system provides a set of inference rules which can infer new FDs from given ones. It is also well-known that Armstrong's axiom system is sound and complete for FDs being satisfied in conventional relations. This result is very significant in database design, since using this axiom system we can derive some efficient algorithms to confirm whether or not a given FD holds in a relation schema [9]. Moreover, it provides us with a basis to further develop FDs in the context of other advanced applications which have fuzzy, incomplete or imprecise information [128, 86, 87].

Definition 4.2 (Armstrong's Axiom System) Let X, Y, Z be subsets of R, $A \in R$ and F be a set of FDs. *Armstrong's axiom system* constitutes the following inference rules for FDs.

(FD1) *Reflexivity*: if $Y \subseteq X$, then F $\vdash X \rightarrow Y$.

(FD2) *Augmentation*: If F $\vdash X \rightarrow Y$, then F $\vdash XA \rightarrow YA$.

(FD3) *Transitivity*: if F $\vdash X \rightarrow Y$ and F $\vdash Y \rightarrow Z$, then F $\vdash X \rightarrow Z$.

There are two possible views of FDs in the context of ordered databases. The first one is straightforward, that is, a FD in conventional relational databases can be viewed as a special case of an OFD when a database is unordered.

Another view of FDs in the context of ordering is more interesting. We recall that we have discussed the notion of SOI (System Ordering Independence) in Chapter 2, which basically means that the ordering of tuples in a relation is not affected by the implementation of the system. We have also defined the domain ordering operator ω_X which governs the ordering of a relation r over R by imposing the lexicographical ordering over $X \subseteq R$. The operator ω_X is a useful tool to study the relationship between domain orderings and data dependencies. When combining ω_X with projection we have the following interesting properties.

Proposition 4.1 Let $X, Y, Z \subseteq R$ and r be a relation over R. The following statements are true.

1. $\omega_R(r)$ is SOI.

2. if $\omega_X \pi_{XY}(r)$ is SOI, then $\omega_{XZ} \pi_{XY}(r)$ is SOI.

3. if $\omega_X \pi_{XYZ}(r)$ is SOI, then $\omega_X \pi_{XY}(r)$ is SOI.

4. if $\omega_X \pi_{XY}(r)$ is SOI, then $\omega_{XZ} \pi_{XYZ}(r)$ is SOI.

5. if $\omega_X \pi_{XY}(r)$ is SOI and $\omega_Y \pi_{YZ}(r)$ is SOI, then $\omega_X \pi_{XZ}(r)$ is SOI. $\square$

We can now use ω_X to define FDs via the notion of SOI as follows.

Definition 4.3 (Alternative Definition of FD in Ordered Relations) An ordered relation r over R *satisfies* a functional dependency $X \to Y$ if $\omega_X \pi_{XY}(r)$ is SOI.

The operator ω_X can be further used to define a subclass of relations called *object relations* [14, 85]. We need the following definition to illustrate this concept.

Definition 4.4 (Meta-attribute in Ordered Relations) An attribute $M \in R$ is said to be a *meta-attribute* for an ordered relation r over R, if it satisfies that $\omega_{MX}(r) = r$ for all $X \subseteq R$, where X can be empty.

We call a relation schema R an *object relational schema* if it contains a distinguished attribute being a meta attribute. Furthermore, we call a subclass of relations *object relations*, if it consists of relations that are defined over object relational schemas. Meta-attributes in object relational schemas are maintained by the system only, and thus they can be hidden from users. The definition of meta-attributes formalises the use of *tuple identifiers.* For example, the relational DBMS Oracle employs an attribute called *ROWID* (ROW IDentifier) to manipulate tuples but this attribute is normally hidden from users [82]. The following proposition states that meta-attributes possess the desirable property of SOI.

Proposition 4.2 $\omega_M(r)$ is SOI.

We now extend Armstrong's axiom system for FDs to the class of object relations by adding the following inference rule.

(FD4) *Meta-attribute:* $\mathrm{F} \vdash M \to R$.

We need the following inference rule, which can be derivable from FD1 to FD3, to prove next theorem.

(FD5) *Union:* if $\mathrm{F} \vdash X \to Y$ and $\mathrm{F} \vdash X \to Z$, then $\mathrm{F} \vdash X \to YZ$.

The *closure* of a set of attributes, $X \subseteq R$, with respect to a given set of FDs F, denoted as X^+, is given by $X^+ = \{A \mid \text{F} \vdash X \to A\}$. We now show that the axiom system comprising inference rules from FD1 to FD4 is also sound and complete for FDs, holding in the class of object relations. The method that we use is standard (c.f., see Chapter 7.3 in [147]), whose idea is first assuming that $X \to Y$ cannot be inferred from the axiom system, and then presenting a relation as a counterexample in which all the dependencies of F hold except $X \to Y$. In other words, we obtain the result that F does not logically imply $X \to Y$.

Theorem 4.3 The axiom system comprising inference rules from FD1 to FD4 is sound and complete for a set of FDs F, holding in the class of object relations.

Proof.

By Proposition 4.1, it follows that the inference rules from FD1 to FD3 are sound. FD4 is also sound by Definition 4.4 and Proposition 4.2. We prove completeness by showing that if $\text{F} \nvdash X \to Y$, then $\text{F} \not\models X \to Y$. Equivalently for the latter, it is sufficient to exhibit a relation r such that $r \models \text{F}$ but $r \not\models X \to Y$. Let r be the relation shown in Figure 4.2, where M, X^+ and Z denote pairwise disjoint sets of attributes such that $Z = R - MX^+$. Note that $M \notin X^+$, otherwise, it is trivial that $X \to Y$ by FD4.

M	X^+	Z
1	$1 \cdots 1$	$1 \cdots 1$
0	$1 \cdots 1$	$0 \cdots 0$

Figure 4.2: An object relation r showing that $r \not\models X \to Y$

We first show that $r \models \text{F}$. Suppose to the contrary that r $\not\models$ F and thus there exists a FD, $C \to D \in \text{F}$ such that $r \not\models C \to D$. It follows by the construction of r that $C \subseteq X^+$ and there exists $A \in (D \cap ZM)$ such that $A \notin X^+$. Suppose $A \in Z$. By FD1, it follows that $C \to A$ and by FD3 again, it follows that $X \to A$. This leads to contradiction, since it follows that $A \in X^+$. Suppose $A = M$. By FD4, it follows that $M \to R$, by FD1, it follows that $M \to Y$, by FD3, it follows that $X \to M$, and finally by FD3 again, it follows that $X \to Y$. This leads to contradiction, since we have derived $\text{F} \vdash X \to Y$.

We conclude the proof by showing that $r \not\models X \to Y$. Suppose to the contrary that $r \models X \to Y$; by the construction of r, $Y \subseteq X^+$ since $M \notin X^+$. It implies that for all

$A \in Y$, F $\models X \to A$. Therefore, for all $A \in Y$, F $\vdash X \to A$. By FD5, it follows that F $\vdash$ $X \to Y$. This leads to contradiction, since we have derived F $\vdash X \to Y$. □

4.2 Ordered Functional Dependencies (OFDs)

An OFD in the ordered relational data model involves comparing the orderings between two sets of data items. We find that OFDs arise naturally in many applications, especially in those that consist of temporal data. A typical example is that an OFD can capture the constraint that the salary of an employee increases every year. Another good example (c.f., see [55]) is the constraint that in a bank account the chronological ordering of date increases as does the numerical ordering of check numbers. Moreover, OFDs can be applied to maintain the "sum of data values" relative to a set of attributes. For instance, the total production for a manufacturing plant should increase every month or the commission earned by an insurance salesperson should increase as the total number of policies he can make from his/her customers.

The semantics of an OFD with two or more attributes on either the left hand side or right hand side are defined according to lexicographical orderings and pointwise-ordering on the Cartesian product of the underlying domains of the attributes in the OFD, which gives rise to POFDs and LOFDs, respectively. From now on, OFDs means either POFDs or LOFDs. We remark also that they are exactly the same data dependencies in the special case of unary attributes, which means that only one attribute is allowed on both the left and right hand sides of an OFD.

To illustrate the usage of OFDs, we show in Figure 4.3 a relation called EMP_RECORD over the set of attributes {EMP, POST_TITLE, YEARS, SALARY}. The semantics of SALARY_RECORD are: an EMPloyee with a given POST TITLE, who has been working in a company for some YEARS, has the present SALARY.

EMP	POST_TITLE	YEARS	SALARY
Mark	Senior Programmer	15	35K
Nadav	Junior Programmer	7	25K
Ethan	Junior Programmer	6	22K

Figure 4.3: An employee relation SALARY_RECORD

We assume there is a semantic ordering in POST_TITLE as represented by the following domain {'Junior Programmer' < 'Senior Programmer'}. Then the relation SALARY_RECORD given in Figure 4.3 satisfies the POFD, {POST_TITLE, YEARS} $\hookrightarrow$ SALARY, which states the fact that the SALARY of an employee is greater than other employees with junior post titles and less experience in the company, and the LOFD, {POST_TITLE, YEARS} $\rightsquigarrow$ SALARY, which states the fact the SALARY of an employee is greater than other employees with junior post titles, or with the same post title but less experience in the company. Note that the semantics of the POFD and the LOFD mentioned above are different. For instance, in the former case, an employee has higher salary than another one only if he/she has both a senior post title and more experience than another, whereas in the latter case, it requires only that he/she has a senior post title than another. Furthermore, if Mark leaves his post and the promotion of Ethan to replace Mark's position is carried out by updating his record to be $\langle Ethan, SeniorProgrammer, 6, 26K\rangle$ (i.e., updating the third tuple), then this updating violates neither the POFD nor the LOFD. However, if his record is updated to be $\langle Ethan, SeniorProgrammer, 6, 24K\rangle$, then it violates the LOFD, since Ethan now has a more senior title but less salary than Nadav. On the other hand, the POFD still holds in this updating, since Nadav still has more experience than Ethan. The appropriateness for the choice between the POFD or the LOFD in this case depends entirely on the semantics of the promotion policy adopted by the company.

We let X be $\langle A_1, \ldots, A_m\rangle$ and, without loss of generality, assume that D is the underlying domain of all the attributes in X. We recall that the pointwise-ordering $t_1 \sqsubseteq_X t_2$ means that for all $A_i \in X$, $t_1[A_i] \sqsubseteq_D t_2[A_i]$, and the lexicographical ordering $t_1 \sqsubseteq_X t_2$ means that either (1) there exists k with $1 \leq k \leq m$ such that $t_1[A_j] = t_2[A_j]$ with $1 \leq j < k$ and $t_1[A_k] \sqsubset_D t_2[A_k]$, or (2) $t_1[A_i] = t_2[A_i]$ for all $A_i \in X$. We use the common notation, $\sqsubseteq$, for both pointwise-ordering and lexicographical ordering whenever the meaning is understood from context. We first discuss OFDs having domains with pointwise-orderings and then OFDs having domains with lexicographical orderings.

4.2.1 OFDs Arising from Pointwise-Orderings

We give the definition of a POFD as follows.

Definition 4.5 (Ordered Functional Dependency Arising from Pointwise-Orderings) An *ordered functional dependency arising from pointwise-orderings* (or simply a POFD) over a relation schema R, is a statement of the form $R : X \hookrightarrow Y$ (or simply $X \hookrightarrow Y$ whenever R is understood from the context), where $X, Y \subseteq R$ are sequences of attributes. The POFD $X \hookrightarrow Y$ is said to be *standard* if $X \neq \emptyset$.

Hereinafter we will assume that all POFDs are standard. We now give the definition of the semantics of a POFD.

Definition 4.6 (Satisfaction of a POFD) A POFD, $R : X \hookrightarrow Y$, is satisfied in a relation r over R, denoted by $r \models X \hookrightarrow Y$, if for all $t_1, t_2 \in r$, $t_1[X] \sqsubseteq_X t_2[X]$ implies that $t_1[Y] \sqsubseteq_Y t_2[Y]$, where $\sqsubseteq_X$ and $\sqsubseteq_Y$ are pointwise-orderings on the Cartesian products of the domains of X and Y, respectively.

We next give a set of inference rules for POFDs and show that Armstrong's axiom system carries over to ordered relations with respect to POFDs.

Definition 4.7 (Inference Rules for POFDs) Let X, Y, Z, W be subsets of R and F be a set of POFDs over R. The inference rules for POFDs are defined as follows:

(POFD1) *Reflexivity*: if $Y \subseteq X$, then $\mathrm{F} \vdash X \hookrightarrow Y$.

(POFD2) *Augmentation*: if $\mathrm{F} \vdash X \hookrightarrow Y$ and $Z \subseteq R$, then $\mathrm{F} \vdash XZ \hookrightarrow YZ$.

(POFD3) *Transitivity*: if $\mathrm{F} \vdash X \hookrightarrow Y$ and $\mathrm{F} \vdash Y \hookrightarrow Z$, then $\mathrm{F} \vdash X \hookrightarrow Z$.

(POFD4) *Permutation*: if $\mathrm{F} \vdash X \hookrightarrow Y$, $W \sim X$ and $Z \sim Y$, then $\mathrm{F} \vdash W \hookrightarrow Z$.

We remark that POFD4 is introduced because we are dealing with sequences of attributes rather than the usual sets of attributes in FDs. The following lemma can be readily proved by induction on the number of steps in the inference of $X \hookrightarrow Y$ from a set of POFDs.

Lemma 4.4 Let F be a set of POFDs, $f = X \hookrightarrow Y$ be a POFD and $f^* = X \rightarrow Y$ be a FD corresponding to f. We define $F^* = \{f^* \mid f \in F\}$. Then f^* is derivable from F^* using Armstrong's axiom if and only if $\mathrm{F} \vdash f$. □

The above lemma is useful because it suggests that we can apply existing algorithms for FDs to determine whether a POFD f can be inferred from a given set of POFDs using the inference rules from POFD1 to POFD4. For example, Beeri and Bernstein's algorithm [13] can be used to compute the closure of a set of attributes with respect to a set of POFDs. We need the following rules derivable from Definition 4.7 to establish the soundness and completeness of the axiom system for POFDs.

Lemma 4.5 The following inference rules can be derived from the inference rules in Definition 4.7.

(POFD5) *Decomposition*: if F $\vdash X \hookrightarrow Y$, then F $\vdash X \hookrightarrow Z$, where $Z \subseteq Y$.

(POFD6) *Union*: if F $\vdash X \hookrightarrow Y$ and F $\vdash X \hookrightarrow Z$, then F $\vdash X \hookrightarrow YZ$. □

The closure of a set of attributes X^+ in the context of POFDs is given by $X^+ = \{A \mid$ F $\vdash$ X $\hookrightarrow$ A $\}$. We now show in the following theorem that the above axiom system is sound and complete for POFDs, holding in ordered databases. The underlying idea in this proof is standard [147] and similar to Theorem 4.3. Moreover, we need to assume that each domain has at least two named elements. We believe that this assumption is reasonable in practice.

Theorem 4.6 The axiom system comprising from POFD1 to POFD4 is sound and complete for POFDs.

Proof.

It is easy to show that the inference rules from POFD1 to POFD4 are sound. We prove completeness by showing that if F $\nvdash X \hookrightarrow Y$, then F $\not\models X \hookrightarrow Y$. Equivalently for the latter, it is sufficient to exhibit a relation, say r, such that $r \models$ F but $r \not\models X \hookrightarrow Y$. Let r be the relation consisting of two tuples t_1 and t_2 shown in Figure 4.4, where $Z = R - X^+$.

	X^+	Z
t_1	$1 \cdots 1$	$1 \cdots 1$
t_2	$1 \cdots 1$	$0 \cdots 0$

Figure 4.4: A relation r showing that $r \not\models X \hookrightarrow Y$

Assuming that $0 \sqsubset 1$, we have $1 \not\sqsubseteq 0$ (i.e. $t_1[Z] \not\sqsubseteq t_2[Z]$). We first show that $r \models$ F. Suppose to the contrary that r $\not\models$ F and thus there exists a POFD, $C \hookrightarrow D \in$ F such that $r \not\models C \hookrightarrow D$. It follows by the construction of r and by POFD-5 that $C \subseteq X^+$ and that $\exists A \in Z$ such that $A \notin X^+$. By POFD1 and by POFD5, it follows that $C \hookrightarrow A$, and by POFD3, it follows that $X \hookrightarrow A$. This leads to contradiction, since it follows that $A \in X^+$. We conclude the proof by showing that $r \not\models X \hookrightarrow Y$. Suppose to the contrary that $r \models X \hookrightarrow Y$. By the construction of r, $Y \subseteq X^+$. This leads to contradiction, since by POFD6 we have $X \hookrightarrow Y'$, where $Y' \sim Y$. Then by POFD4 we have derived F $\vdash$ $X \hookrightarrow Y$. □

4.2.2 OFDs Arising from Lexicographical Orderings

We give the definition of a LOFD as follows.

Definition 4.8 (Ordered Functional Dependency Arising from Lexicographical Orderings) An *ordered functional dependency arising from lexicographical orderings* (or simply a LOFD) over a relation schema R, is a statement of the form $R : X \rightsquigarrow Y$ (or simply $X \rightsquigarrow Y$ whenever R is understood from the context), $X, Y \subseteq R$ are sequences of attributes.

Similar to POFDs, we assume that all LOFDs are standard. We now give the definition of the semantics of a LOFD.

Definition 4.9 (Satisfaction of a LOFD) A LOFD, $R : X \rightsquigarrow Y$, is satisfied in a relation r over R, denoted by $r \models X \rightsquigarrow Y$, if for all $t_1, t_2 \in r$, $t_1[X] \sqsubseteq_X t_2[X]$ implies that $t_1[Y] \sqsubseteq_Y t_2[Y]$, where $\sqsubseteq_X$ and $\sqsubseteq_Y$ are the lexicographical orderings on the Cartesian product of the domains of X and Y, respectively.

We observe that the concept of POFDs and LOFDs are incomparable. A relation satisfying the POFD $X \hookrightarrow Y$ may not necessarily satisfy the LOFD $X \rightsquigarrow Y$ and conversely, a relation satisfies the LOFD $X \rightsquigarrow Y$ may not necessarily satisfy the POFD $X \hookrightarrow Y$. The following example helps to illustrate this point.

Example 4.1 Consider the relations r_1 and r_2 over $R = \{A, B, C\}$ shown in Figure 4.5. It is clear that in (a) $r_1 \models A \rightsquigarrow BC$ but $r_1 \not\models A \hookrightarrow BC$. On the other hand, in (b) $r_2 \models AB \hookrightarrow C$ but $r_2 \not\models AB \rightsquigarrow C$.

The chase is a fundamental theorem proving tool in relational database theory. The main uses of the chase have been testing implications of data dependencies [101] and

$r_1 =$

A	B	C
1	3	6
2	4	5

(a)

$r_2 =$

A	B	C
1	4	6
2	3	5

(b)

Figure 4.5: Relations r_1 and r_2 showing that POFDs and LOFDs are incomparable.

testing consistency of a relational database with respect to a set of data dependencies [61, 86]. We now extend the classical chase defined over conventional relations with respect to FDs [101, 9] to ordered relations with respect to LOFDs. The extended chase will be used as a sound and complete inference tool for LOFDs in Theorem 4.9. We need two operations *equate* and *swap* to manipulate values in ordered domains before presenting our chase rules.

Definition 4.10 (Equate and Swap Operations) We denote $min(a, b)$ and $max(a, b)$ the minimum and maximum of the values a and b, respectively. For any two distinct tuples $t_1, t_2 \in r$ over R and some $A \in R$, the *equate* of t_1 and t_2 on A, denoted as *equate*($t_1[A]$, $t_2[A]$), is defined by replacing both $t_1[A]$ and $t_2[A]$ by $min(t_1[A], t_2[A])$; the *swap* of t_1 and t_2 on A, denoted as *swap*($t_1[A]$, $t_2[A]$), is defined by replacing $t_1[A]$ by $min(t_1[A], t_2[A])$ and $t_2[A]$ by $max(t_1[A], t_2[A])$, respectively.

We demonstrate how to use the equate and swap operations with the following example.

Example 4.2 Consider a relation r shown in Figure 4.6 (a), which consists of two tuples $t_1 = \langle 2 \rangle$ and $t_2 = \langle 1 \rangle$, respectively. We apply the equate operation of t_1 and t_2 on A resulting in the relation shown in Figure 4.6 (b). We apply the swap operation of t_1 and t_2 on A resulting in the relation shown in Figure 4.6 (c).

We now give the chase rules, which are applied to two tuples in a relation with respect to a set of LOFDs.

Definition 4.11 (Chase Rules for LOFDs) Let t_1 and t_2 be two tuples in r such that $t_1[X] \sqsubseteq_X t_2[X]$ but $t_1[Y] \not\sqsubseteq_Y t_2[Y]$, A be the first attribute in X such that $t_1[A] \neq t_2[A]$,

	A
t_1	2
t_2	1

(a) $r = \{t_1, t_2\}$

	A
t_1	1
t_2	1

(b) $equate(t_1[A], t_2[A])$

	A
t_1	1
t_2	2

(c) $swap(t_1[A], t_2[A])$

Figure 4.6: An example of using the equate and swap operations

if such an attribute exists, and B be the first attribute in Y such that $t_1[B] \neq t_2[B]$, then the *chase rules* for the LOFD $X \rightsquigarrow Y$, is defined by the following two rules:

Equate rule: if $t_1[X] = t_2[X]$ but $t_1[B] \neq t_2[B]$, then $equate(t_1[B], t_2[B])$;

Swap rule: if $t_1[A] \sqsubset t_2[A]$ but $t_2[B] \sqsubset t_1[B]$, then $swap(t_1[B], t_2[B])$, or if $t_2[A] \sqsubset t_1[A]$ but $t_1[B] \sqsubset t_2[B]$, then $swap(t_1[A], t_2[A])$.

The said chase rules cater for all the possible cases when there are two tuples in a relation violating $X \rightsquigarrow Y$. We note that in applying the chase rules we need a fixed ordering on the tuples t_1 and t_2. If we choose different orderings on t_1 and t_2 in different applications of the rules, then the chase procedure may result in a non-terminating process. We further clarify this point by the following example.

Example 4.3 Let F $= \{A \rightsquigarrow B, C \rightsquigarrow B\}$ and the tuples $t_p = \langle 146 \rangle$ and $t_q = \langle 235 \rangle$, respectively, as shown in Figure 4.7 (a). First we let $t_1 = t_p$ and $t_2 = t_q$, then apply the

	A	B	C
t_p (as t_1)	1	4	6
t_q (as t_2)	2	3	5

(a) before the chase

	A	B	C
t_p (as t_2)	1	*3*	6
t_q (as t_1)	2	*4*	5

(b) chase for $A \rightsquigarrow B$ on (a)

	A	B	C
t_p	1	*4*	6
t_q	2	*3*	5

(c) chase for $C \rightsquigarrow B$ on (b)

Figure 4.7: An example showing that the chase procedure never terminates

swap rule with respect to $A \rightsquigarrow B$ and thus obtain the result as shown in Figure 4.7 (b). Now we let $t_1 = t_q$ and $t_2 = t_p$ (i.e., change the ordering of t_p and t_q). Then we apply the swap rule with respect to $C \rightsquigarrow B$ and thus obtain the result as shown in Figure 4.7 (c), which is the beginning relation that we have shown in Figure 4.7 (a).

Fortunately, this undesirable property can be removed if we impose a fixed linear ordering on r and assign t_1 to be the smaller tuple and t_2 to be the larger tuple with respect to this ordering. We will show in Lemma 4.7 that under such a condition the chase procedure always terminates. Therefore, in Example 4.3 if we assume the ordering of t_p and t_q is fixed as given in Figure 4.7 (a) throughout the chase procedure, then the process terminates and it can be checked that the final relation is obtained as shown in Figure 4.8.

	A	B	C
t_p (as t_1)	1	*3*	*5*
t_q (as t_2)	2	*4*	*6*

Figure 4.8: The chase procedure terminates in Example 4.3 with a fixed ordering

Let $r = \{t_1, \ldots, t_n\}$ be an ordered relation over R and F be a set of LOFDs with $\mid R \mid = m$. We now give the pseudo-code of an algorithm designated, $CHASE(r,\text{F})$, which applies the chase rules given in Definition 4.11 to R as long as possible and returns the resulting relation r over R, also denoted as $CHASE(r,\text{F})$.

Algorithm 4.1 ($CHASE(r$, F$)$)

```
1.   begin
2.      Result := $r = \langle t_1, \ldots, t_n \rangle$ ;
3.      Tmp:= $\emptyset$;
4.      while Tmp $\neq$ Result do
5.         Tmp := Result;
6.         if $\exists X \leadsto Y \in$ F, $\exists\ t_p, t_q \in$ Result such that
           $t_p[X] \sqsubseteq_X t_q[X]$ but $t_p[Y] \not\sqsubseteq_Y t_2[Y]$ then
7.         Apply the appropriate chase rule to Result with
           $t_1 = t_{min(p,q)}$ and $t_2 = t_{max(p,q)}$;
8.      end while
9.      return Result;
10.  end.
```

Lemma 4.7 $CHASE(r,\text{F})$ in Algorithm 4.1 terminates and satisfies F.

Proof.

Let P_j with $1 \leq j \leq m$ be the sequence $\langle a_{1j}, \ldots, a_{nj} \rangle$, where $a_{ij} = t_i[A_j]$ (i.e., $P_j = \pi_{A_j}(Result)$), a_j^{min} be the minimum value in P_j, and P_j^{min} be the sequence $\langle a_j^{min}, \ldots, a_j^{min} \rangle$ (a sequence of n identical values). Suppose an application of a chase rule changes P_j to $P_j' = \langle a_{1j}', \ldots, a_{nj}' \rangle$. Since the chase rules neither change the value a_j^{min} nor introduce any new values into the variable *Result*, P_j^{min} is unchanged throughout the process of the chase. In order to prove that $CHASE(r,\text{F})$ terminates, it suffices to show that $P_j^{min} \leq_{lex} P_j' <_{lex} P_j$, where $\leq_{lex}$ is a lexicographical ordering. There are two cases to consider.

In the first case the change to P_j is due to an application of the equate rule. Then by Algorithm 4.1 we have $a_{pj} \neq a_{qj}$. It follows that $a_{pj}' = min(a_{pj}, a_{qj})$, $a_{qj}' = min(a_{pj}, a_{qj})$ and $a_{ij}' = a_{ij}$ for $i \notin \{p, q\}$. Thus, $P_j' <_{lex} P_j$.

In the second case the change to P_j is due to an application of the swap rule. Without loss of generality we assume $p < q$. Then by Algorithm 4.1 $a_{qj} < a_{pj}$. It follows that $a_{pj}' = min(a_{pj}, a_{qj})$, $a_{qj}' = max(a_{pj}, a_{qj})$ and $a_{ij}' = a_{ij}$ for $i \notin \{p, q\}$. Thus, $P_j' <_{lex} P_j$.

It is also trivial that in both cases $P_j^{min} \leq_{lex} P_j'$, since the minimum of any two values in P_j is greater than or equal to the minimum of all values in P_j.

Due to the consideration above, it follows that $CHASE(r,\text{F})$ satisfies F, otherwise, we can apply one of the chase rules given in Definition 4.11 to $CHASE(r,\text{F})$, thus leading to contradiction, since $CHASE(r,\text{F})$ has not yet terminated. □

Lemma 4.8 $CHASE(r,\text{F})$ in Algorithm 4.1 can be computed in time polynomial in the size of r and F.

Proof.

By Definition 4.11, we observe that the lines 6 to 7 in Algorithm 4.1 can be executed at most $O(m)$ times for a LOFD in F, where m is the number of distinct symbols in r. Thus, there is at most $O(m)$ application of chase rules to r. So each execution of the while loop beginning in line 4 and ending at line 8 can be computed in polynomial time in the size of r and F. □

Example 4.4 Let $\text{F} = \{A \leadsto B, B \leadsto C\}$ and a relation r consisting of three tuples $t_1 = \langle 332 \rangle$, $t_2 = \langle 221 \rangle$ and $t_3 = \langle 313 \rangle$, respectively, as shown in Figure 4.9 (a). First, we carry out the chase rules to eliminate the violation of $A \leadsto B$ as follows, apply the chase rule $swap(t_2[B], t_3[B])$ since $t_2[A] \sqsubset t_3[A]$ but $t_3[B] \sqsubset t_2[B]$, and then apply the chase rule $equate(t_1[B], t_3[B])$ since $t_1[A] = t_3[A]$ but $t_1[B] \neq t_3[B]$. We thus obtain the intermediate result as shown in Figure 4.9 (b), which satisfies $A \rightsquigarrow B$. Second, we carry out the chase rules to eliminate the violation of $B \rightsquigarrow C$ as follows, apply the chase rule $equate(t_1[C], t_3[C])$ since $t_1[B] = t_3[B]$ but $t_1[C] \neq t_3[C]$. The chase procedure now terminates and the final result $CHASE(r,\mathrm{F})$ is given in Figure 4.9 (c), which satisfies F.

	A	B	C
t_1	3	3	2
t_2	2	2	1
t_3	3	1	3

(a) r prior to the chase

	A	B	C
t_1	3	*2*	2
t_2	2	*1*	1
t_3	3	*2*	3

(b) chase for $A \rightsquigarrow B$ on (a)

	A	B	C
t_1	3	2	2
t_2	2	1	1
t_3	3	2	*2*

(c) chase for $B \rightsquigarrow C$ on (b)

Figure 4.9: An example of obtaining $CHASE(r,\mathrm{F})$

We note that the result of the chase is not necessarily unique. For instance, in the above example we can apply $equate(t_1[B], t_3[B])$ first to eliminate the violation of $A \rightsquigarrow B$, then we have at least two '1's under the column of attribute B, this leads to a final result different from that given in Figure 4.9 (c). Although the final result of the chase may not be unique, we still can apply it in tackling the implication problem of LOFDs. This point is illustrated by the results shown in next theorem and Theorem 4.12.

Theorem 4.9 Let r be a relation over R and F be a set of LOFDs over R. Then $r \models \mathrm{F}$ if and only if $r = CHASE(r,\mathrm{F})$.

Proof.

(*IF:*) Assume to the contrary that $r \not\models \mathrm{F}$ and thus there exists a LOFD, $X \rightsquigarrow Y \in \mathrm{F}$ such that $r \not\models X \rightsquigarrow Y$. It follows that there must be two rows, $t_1, t_2 \in r$, such that $t_1[X] \sqsubseteq_X t_2[X]$ but $t_1[Y] \not\sqsubseteq_Y t_2[Y]$; so the chase rule for $X \rightsquigarrow Y$ can be applied to r resulting in a different relation. Hence $r \neq CHASE(r, F)$.

(*ONLY IF:*) It follows from Definition 4.11 that a chase rule for F can be carried out only if r violates some LOFD in F. □

Lemma 4.7 and Theorem 4.9 are fundamental because they allow the chase procedure to be employed in order to test the satisfaction of r with respect to a set of F in a finite number of steps; many similar results for different kinds of data dependencies such as FDs, INDs and JDs (Join Dependencies) can be found in [101, 73, 104]. These results provide us with a theorem proving tool to test consistency of a database with respect to a set of LOFDs. Furthermore, the chase can be used for maintaining consistency by applying the rules in Definition 4.11 to fix the violation of a LOFD in relations. This is also found to be important in the case of fuzzy or imprecise relations [86]. For example, assuming that a relation r is updated with information obtained from several different sources in a mobile computing environment [11], it may be the case that at any given time the relation violates some LOFDs. Thus we can modify the relation by using the chase rules.

In order to provide a proof procedure for LOFDs, we now define the notion of *ordered variables.* Such variables afford us the ability to infer orderings between attribute values and to set up a set of *templates for relations*, which are essentially the same concept as tableaux used in [101, 9, 87].

Definition 4.12 (Ordered Variables and Variable Domain) The *variable domain* of a relation schema R, denoted by $vdom(R)$, is the finite set $\{l_1, \ldots, l_m, h_1, \ldots, h_m,\}$, where $m = \mid R \mid$. The variables l_i and h_i with $i \in \{1, \ldots, m\}$ are called *low ordered variables* and *high ordered variables*, respectively. We call them collectively *ordered variables*, whose ordering is given by $l_i \sqsubset h_i$.

We now define a set of relations defined over variable domains with respect to a given LOFD, which basically enumerate all the possible cases for two tuples violating the LOFD.

Definition 4.13 (Template Relations for a LOFD) Let f be the LOFD $X \rightsquigarrow Y$ over R with $\mid X \mid= n$ and $\mid R \mid= m$. We use two short hand symbols u_i and v_i to represent one of the following three cases: (1) $u_i = l_i$ and $v_i = l_i$, (2) $u_i = l_i$ and $v_i = h_i$ or (3) $u_i = h_i$ and $v_i = l_i$. A *template relation* (or simply a template) with respect to f, denoted as r_f, is a relation consisting of two tuples, t_1 and t_2, whose underlying domain is $vdom(R)$, such that it is equal to either T_0 or T_k, where $Pre(X) = \langle x_1, \ldots, x_k \rangle$ for $1 \leq k \leq n$.

We remark that in Definition 4.13 the symbols u_i and v_i represent three possibilities of combinations of l_i and h_i. Therefore, it is easy to verify that there are 3^{m-n} templates

$T_0 =$

	X	$R - X$
t_1	$l_1 \cdots l_n$	$u_{n+1} \cdots u_m$
t_2	$l_1 \cdots l_n$	$v_{n+1} \cdots v_m$

$T_k =$

	$x_1 \cdots x_{k-1}$	x_k	$R - Pre(X)$
t_1	$l_1 \cdots l_{k-1}$	l_k	$u_{n+1} \cdots u_m$
t_2	$l_1 \cdots l_{k-1}$	h_k	$v_{n+1} \cdots v_m$

Figure 4.10: Template relations for a LOFD

defined by T_0 and 3^{m-k} templates defined by T_k for each k. Altogether there are $3^{m-n} + (3^{m-n} + \cdots + 3^{m-1}) = 3^{m-n} + \frac{3^m - 3^{m-n}}{2} = \frac{3^m + 3^{m-n}}{2}$ templates. Note that there are some redundant templates in both T_0 and T_k, if we take into account the fact that there are two possible orderings for t_1 and t_2, but it does not affect the order of the upper bound of the number of templates, which is shown to be $O(3^m)$.

We apply the chase rules to a template relation using the ordering defined on a variable domain $vdom(R)$. The following proposition gives the result corresponding to Theorem 4.9.

Proposition 4.10 Let r_f be a template relation over R and F be a set of LOFDs over R. Then $r_f \models$ F if and only if $r_f = CHASE(r_f, F)$. □

A template relation can be viewed as a relation instance consisting of two tuples by using an ordering isomorphism mapping values in D to low ordered variables and high ordered variables, respectively. We formalise this idea by the following definition.

Definition 4.14 (Valuation Mapping) Let $R = \{A_1, \ldots, A_m\}$ and $vdom(R) = \{l_1, \ldots, l_m, h_1, \ldots, h_m, \}$. A *valuation mapping* ρ is a mapping from $vdom(R)$ to D such that $\rho(l_i) < \rho(h_i)$ for all $1 \leq i \leq m$. We extend ρ to a tuple t by $\rho(t) = \langle \rho(t[A_1]), \ldots, \rho(t[A_m]) \rangle$. Furthermore, we extend ρ to a template relations r_f by $\rho(r_f) = \{\rho(t_1), \rho(t_2)\}$.

The next proposition states that if there is a valuation mapping relating a template relation to a relation having two tuples, then they satisfy the same set of LOFDs.

Proposition 4.11 Let $\rho(r_f) = r$, where r is a relation over R having two tuples. Then $r_f \models X \leadsto Y$ if and only if $r \models X \leadsto Y$.

Proof.

The result immediately follows by Definition 4.14 since r_f is isomorphic to r and the ordering of data values in the ith column of r corresponds to the ordering of the ordered variables l_i and h_i. □

The following example shows how to apply a valuation mapping to a template relation.

Example 4.5 Consider the template relation r_f over $\{A, B, C\}$ with respect to the LOFD f, $A \rightsquigarrow BC$, which is shown in Figure 4.11 (a). We define the valuation mapping ρ by $\rho(l_1) = 1$, $\rho(l_2) = 2$, $\rho(h_2) = 3$, $\rho(l_3) = 4$ and $\rho(h_3) = 5$. Then we have $\rho(r_f)$ shown in Figure 4.11 (b). Note that in this example r_f is one of the templates defined by T_0 in Definition 4.13.

$r_f =$

A	B	C
l_1	l_2	h_3
l_1	h_2	l_3

(a)

$\rho(r_f) =$

A	B	C
1	2	5
1	3	4

(b)

Figure 4.11: An example showing the application of a valuation mapping

We now extend the notion of tableaux for a LOFD f to be a set of templates. The tableaux in our case is different from that for FDs, which just requires a single template for FDs (see Theorem 4.2 in [9]). We define *tableaux*, denoted by T_f, to be the set of all template relations in Definition 4.13.

Definition 4.15 (Satisfaction and a Valuation Mapping of Tableaux) The chase of T_f, denoted as $CHASE(T_f, F)$, is defined by $CHASE(T_f, F) = \{CHASE(r_f, F) \mid r_f \in T_f\}$. $CHASE(T_f, F)$ satisfies $X \rightsquigarrow Y$, denoted by $CHASE(T_f, F) \models X \rightsquigarrow Y$, if for all $r_f \in T_f$, $CHASE(r_f, F) \models X \rightsquigarrow Y$. Furthermore, $CHASE(T_f, F)$ satisfies F, denoted by $CHASE(T_f, F) \models$ F, if for all $X \rightsquigarrow Y \in$ F, $CHASE(T_f, F) \models X \rightsquigarrow Y$. A valuation mapping of T_f is a valuation mapping of some r_f in T_f.

The following theorem shows that the chase rules can be also viewed as a sound and complete inference procedure for LOFDs.

Theorem 4.12 Let F be a set of LOFDs over R and f be a LOFD $X \rightsquigarrow Y$. Then $CHASE(T_f, F) \models f$ if and only if F $\models f$.

Proof.

IF: Assume $CHASE(T_f, F) \not\models f$. By Definition 4.15, there exists $r_f \in T_f$ such that $CHASE(r_f, F) \not\models f$ but $CHASE(r_f, F) \models$ F. Note that $CHASE(r_f, F)$ is a template which can be viewed as an instance. Therefore, we have a valuation mapping ρ to generate a relation instance $\rho(CHASE(r_f, F))$ and by Proposition 4.11, $\rho(CHASE(r_f, F)) \models$ F but $\rho(CHASE(r_f, F)) \not\models f$. This leads to contradiction.

ONLY IF: We let w_1, w_2 be any two tuples in a relation r such that $w_1 \sqsubseteq_X w_2$. We claim $w_1 \sqsubseteq_Y w_2$. Let $s_f \in T_f$ be the template relation such that $\rho(t_1) = w_1$ and $\rho(t_2) = w_2$. We can always find such an s_f because T_f exhausts all possibilities of two tuples which satisfy the condition $w_1 \sqsubseteq_X w_2$. Thus we have $\rho(s_f) = \{w_1, w_2\}$ and $\rho(s_f) \models$ F. By Proposition 4.11, we have $s_f \models$ F. It follows by Proposition 4.10 that $s_f = CHASE(s_f, F)$. Since we have assumed that $CHASE(T_f, F) \models f$, we have $CHASE(s_f, F) \models f$. Thus, $\rho(CHASE(T_f, F)) = \rho(s_f) = \{w_1, w_2\}$, which implies that $w_1 \sqsubseteq_Y w_2$ as required. $\square$

The following corollary is an immediate result of Theorem 4.12.

Corollary 4.13 Let F be a set of LOFDs over R. The chase procedure is a decidable, sound and complete inference algorithm for LOFDs.

The above corollary shows that the chase rules together with tableaux can be used to provide a systematic way to solve the implication problem for LOFDs. Furthermore, it provides a basis for further investigation in examining the completeness of the following axiom system for LOFDs [87].

Definition 4.16 (Inference Rules for LOFDs) Let F be a set of LOFDs over R. The inference rules for LOFDs are defined as follows:

(LOFD1) *Prefix Decomposition*: if F $\vdash X \rightsquigarrow Y$, then F $\vdash X \rightsquigarrow pre(Y)$.

(LOFD2) *Right Augmentation*: if F $\vdash X \rightsquigarrow Y$ and $Z \subseteq R$, then F $\vdash XZ \rightsquigarrow Y$.

(LOFD3) *Pseudo Transitivity*: if F $\vdash X \rightsquigarrow WY$ and F $\vdash Y \rightsquigarrow Z$, then F $\vdash X \rightsquigarrow WZ$.

(LOFD4) *Right Union and Shuffle*: if F $\vdash X \rightsquigarrow Y$ and F $\vdash X \rightsquigarrow Z$, then F $\vdash X \rightsquigarrow shu(Y, pre(Z))$.

(LOFD5) *Left Union and Shuffle*: if F $\vdash X \rightsquigarrow Z$ and F $\vdash Y \rightsquigarrow Z$, then F $\vdash shu(X, pre(Y)) \rightsquigarrow Z$.

(LOFD6) *Right Contraction I*: if F $\vdash W_1 \rightsquigarrow W_2$ and F $\vdash X \rightsquigarrow Y shu(W_1, W_2) Z$, then F $\vdash X \rightsquigarrow Y W_1 Z$.

(LOFD7) *Right Contraction II*: if F $\vdash W_1 \leadsto W_2$ and F $\vdash X \leadsto YW_1Z_1W_2V$, then F $\vdash X \leadsto YW_1ZV$.

(LOFD8) *Left Contraction I*: if F $\vdash W_1 \leadsto W_2$ and F $\vdash Yshu(W_1, W_2)Z \leadsto X$, then F $\vdash YW_1Z \leadsto X$.

(LOFD9) *Left Contraction II*: if F $\vdash W_1 \leadsto W_2$ and F $\vdash YW_1ZW_2V \leadsto X$, then F $\vdash YW_1ZV \leadsto X$.

The following lemma can be easily obtained from Definition 4.16.

Lemma 4.14 The axiom system comprising the inference rules shown in Definition 4.16 is sound for LOFDs. □

We remark that the reflexivity rule is just a special case of LOFD1. In addition, we note that the augmentation rule (this should not be confused with LOFD2), which is sound for POFDs, is not sound as an inference rule for LOFDs. Consider the counterexample, where r is the ordered relation shown in Figure 4.12; it is clear that $r \models A \leadsto B$ but $r \not\models AC \leadsto BC$.

A	B	C
1	3	5
2	3	4

Figure 4.12: A counterexample for the augmentation rule for LOFDs

The next lemma shows the interesting result that the converse of rules of LOFD6 to LOFD9 can be derived from Definition 4.16.

Lemma 4.15 The following rules can be derived from the inferences rule LOFD1 to LOFD9.

(LOFD10) *Right Expansion I*: if F $\vdash X \leadsto YW_1Z$ and F $\vdash W_1 \leadsto W_2$, then F $\vdash X \leadsto Yshu(W_1, W_2)Z$.

(LOFD11) *Right Expansion II*: if F $\vdash X \leadsto YW_1ZV$ and F $\vdash W_1 \leadsto W_2$, then F $\vdash X \leadsto YW_1ZW_2V$.

(LOFD12) *Left Expansion I*: if F $\vdash YW_1Z \rightsquigarrow X$ and F $\vdash W_1 \rightsquigarrow W_2$, then $Y shu(W_1, W_2)Z \rightsquigarrow X$.

(LOFD13) *Left Expansion II*: if F $\vdash YW_1ZV \rightsquigarrow X$ and F $\vdash W_1 \rightsquigarrow W_2$, then F $\vdash YW_1ZW_2V \rightsquigarrow X$. □

The following lemma proves the inference rules defined in Definition 4.16 are sound for LOFDs.

Lemma 4.16 The axiom system comprising rules from LOFD1 to LOFD9 is sound for LOFDs holding in ordered databases. □

The following proposition summarises the relationships between the satisfaction of POFDs, LOFDs and FDs in a relation r.

Proposition 4.17 Let r be a relation.

1. if $r \models X \hookrightarrow Y$, then $r \models X \rightarrow Y$.
2. if $r \models X \rightsquigarrow Y$, then $r \models X \rightarrow Y$. □

From the above proposition, we can deduce that the set of relations which satisfy a set of POFDs (or LOFDs) is a subset of relations which satisfy the corresponding set of FDs F^*, where F^* is defined as $\{X \rightarrow Y \mid X \hookrightarrow Y \in$ F (or $X \rightsquigarrow Y \in$ F)$\}$.

4.3 Ordered Inclusion Dependencies (OINDs)

In this section we continue our investigation of data dependencies in the ordered relational model by introducing ordered inclusion dependencies. Inclusion dependencies (INDs) are fundamental data dependencies that arise in practice, which can express the set inclusion between the projections of two relations. A well-known example is that an IND can capture the fact that the set of managers' names is contained in the set of employees' names of a company.

Definition 4.17 (Inclusion Dependency) An *inclusion dependency* (or simply an IND) over a database schema **R** is a statement of the form $R[X] \subseteq S[Y]$, where $R, S \in$ **R** and $X \subseteq R$, $Y \subseteq S$ are sequences of distinct attributes such that $\mid X \mid = \mid Y \mid$. An IND is said to be trivial, if it is of the form $R[X] \subseteq R[X]$. An IND $R[X] \subseteq S[Y]$ over **R**

is satisfied in d, denoted by $d \models R[X] \subseteq S[Y]$, whenever $\pi_X(r) \subseteq \pi_Y(s)$, where $r, s \in d$ are the relations over R and S, respectively.

We first review the inference rules constituting an axiom system which was shown to be sound and complete for INDs by Casanova, Fagin and Papadimitrious [22]. For the sake of simplicity, we call this axiom system *Casanova's axiom system.* This result will be useful in proving our OINDs axiom systems.

Definition 4.18 (Casanova's Axiom System) Let I be a set of INDs over $\mathbf{R}$ and $R, R_1, R_2, R_3 \in \mathbf{R}$. *Casanova's axiom system* constitutes the following inference rules for INDs.

(IND1) *Reflexivity*: if $X \subseteq R$, then $\mathrm{I} \vdash R[X] \subseteq R[X]$.

(IND2) *Projection and Permutation*: if $\mathrm{I} \vdash R_2[Y] \subseteq R_1[X]$, where $X = \langle A_1, \ldots, A_m \rangle \subseteq R_1$, $Y = \langle B_1, \ldots, B_m \rangle \subseteq R_2$ and $i_1, \ldots, i_k$ is a sequence of distinct integers from $\{1, \ldots m\}$, then $\mathrm{I} \vdash R_2[B_{i_1}, \ldots, B_{i_k}] \subseteq R_1[A_{i_1}, \ldots, A_{i_k}]$.

(IND3) *Transitivity*: if $\mathrm{I} \vdash R_3[Z] \subseteq R_2[Y]$ and $\mathrm{I} \vdash R_2[Y] \subseteq R_1[X]$, then $\mathrm{I} \vdash R_3[Z] \subseteq R_1[X]$.

An OIND in the ordered relational model can capture the notion of Hoare orderings between two sets of values projected onto some attributes of two relations in a database, which arise naturally in those applications that consist of incomplete information, which we have discussed in Chapter 2. Similar to OFDs, the semantics of an OIND with two or more attributes on each side are also defined according to lexicographical orderings and pointwise-orderings on the Cartesian products of the underlying domains of the attributes in the OIND, which give rise to POINDs and LOINDs, respectively. Thus, from now on OINDs means either POINDs or LOINDs. We note that POINDs and LOINDs are exactly the same data dependencies in the special case of unary attribute on the left and right hand sides.

To illustrate the usage of POINDs and LOINDs, we show in Figure 4.13 that a database consisting of two relations, called CURRENT_RECORD and HISTORY, which are both over the set of attributes {EMP, SALARY, YEAR, MONTH}. We allow the null value symbol UNK, which means "value at present unknown", to be used in the

EMP	SALARY	YEAR	MONTH
Mark	35K	1997	May
Nadav	UNK	1996	Dec
UNK	24K	1995	Oct

(a) CURRENT_RECORD

EMP	SALARY	YEAR	MONTH
Mark	30K	1996	Sep
Mark	35K	1997	May
Nadav	25K	1996	Dec
Ethan	24K	1995	Oct

(b) HISTORY

Figure 4.13: Relations CURRENT_RECORD and HISTORY

attributes EMP and SALARY only (recall that all data values are assumed to be larger than UNK).

The semantics of this database are: in CURRENT_RECORD an EMPloyee has the most recently updated SALARY starting from the date specified by the YEAR and the MONTH, and in HISTORY an EMPloyee has had the given SALARY starting from the date specified by the YEAR and the MONTH. If we assume that each EMP in CURRENT_RECORD is included in the set of EMP of HISTORY and each SALARY in CURRENT_RECORD is included in the set of SALARY in HISTORY, then this semantics can be captured by the POIND, CURRENT_RECORD[EMP, SALARY] $\hat{\sqsubseteq}$ HISTORY[EMP, SALARY]. As these relations are time-stamped by the attributes YEAR and MONTH, we can use the LOIND, HISTORY[YEAR, MONTH] $\tilde{\sqsubseteq}$ CURRENT_RECORD[YEAR, MONTH] to state the fact that CURRENT_RECORD contains the most recently updated record in HISTORY.

We note that the POIND and the LOIND in this case have different implications in updating. For example, if we want to update Mark's record to be $\langle Mark, 40K, 1997, Oct\rangle$, then assuming the POIND, we should insert this tuple into HISTORY prior to updating CURRENT_RECORD. On the other hand, when assuming the LOIND we should update CURRENT_RECORD prior to inserting the tuple in HISTORY. Note also that we cannot use the POIND, HISTORY[YEAR, MONTH] $\hat{\sqsubseteq}$ CURRENT_RECORD[YEAR, MONTH] to capture this semantics of the mentioned LOIND (recall that DATE is a lexicographical ordering).

We first discuss OINDs having domains with pointwise-orderings and then OINDs having domains with lexicographical orderings.

4.3.1 OINDs Arising from Pointwise-Orderings

We now give the definition of a POIND as follows.

Definition 4.19 (An Ordered Inclusion Dependency Arising from Pointwise-Orderings) An *ordered inclusion dependency arising from pointwise-orderings* (or simply a POIND) over a database schema $\mathbf{R}$, is a statement of the form $R[X] \hat{\sqsubseteq} S[Y]$, where $R, S \in \mathbf{R}$ and $X \subseteq R$, $Y \subseteq S$ are sequences of attributes such that $| X |=| Y |$.

We now give the definition of the semantics of a POIND.

Definition 4.20 (Satisfaction of a POIND) A POIND, $R[X] \hat{\sqsubseteq} S[Y]$, over $\mathbf{R}$ is satisfied in an ordered database d over $\mathbf{R}$, denoted by $d \models R[X] \hat{\sqsubseteq} S[Y]$, if $\forall t_1 \in r$, $\exists t_2 \in s$ such that $t_1[X] \sqsubseteq t_2[Y]$, where $\sqsubseteq$ is a pointwise-ordering, $r \in d$ is the relation over $R \in \mathbf{R}$ and $s \in d$ is the relation over $S \in \mathbf{R}$.

From now on we will assume that when $R[X] \hat{\sqsubseteq} S[Y] \in \mathrm{I}$ and d is a database over $\mathbf{R}$, then $r \in d$ is the relation over $R \in \mathbf{R}$ and $s \in d$ is the relation over $S \in \mathbf{R}$.

We observe that an IND in conventional relational databases can be viewed as a POIND in the special case of unordered databases. The following proposition gives a simple relationship between POINDs and INDs.

Proposition 4.18 If $d \models R[X] \subseteq S[Y]$, then $d \models R[X] \hat{\sqsubseteq} S[Y]$. □

We note that the converse of the above proposition does not hold as shown in the following counterexample.

Example 4.6 Let $d = \{r\}$ be a database over $\mathbf{R} = \{R, S\}$ where $r = \{0\}$ (1 tuple) and $s = \{1\}$ (1 tuple) are the relations over R and S, respectively, with $R = \{A\}$ and $S = \{B\}$. Then $d \models R[A] \hat{\sqsubseteq} S[B]$ but $d \not\models R[A] \subseteq S[B]$.

We note that Casanova's axiom system [22] can be carried over to be the set of *inference rules for POINDs*, simply by replacing the symbols $\subseteq$ in an IND as $\hat{\sqsubseteq}$ in an POIND for the inference rules given in Definition 4.18. The following lemma is immediately followed by the inference rules for POINDs (c.f., see the similar result in Lemma 4.4).

Lemma 4.19 Let I be a set of POINDs, $\alpha = R[X] \hat{\sqsubseteq} S[Y]$ be a POIND, $\alpha^* = R[X] \subseteq S[Y]$ be an IND corresponding to α and $I^* = \{\alpha^* \mid \alpha \in \mathrm{I}\}$. Then $I^* \vdash \alpha^*$ if and only if I $\vdash \alpha$. $\square$

We now show Casanova's axiom system is also sound and complete for POINDs.

Theorem 4.20 The axiom system comprising from IND1 to IND3 is sound and complete for POINDs.

Proof.

It is easy to show that the inference rules from IND1 to IND3 are sound for POINDs. Let $\alpha = R[X] \hat{\sqsubseteq} S[Y]$ with $X = \langle A_1, \ldots, A_n \rangle$ and $Y = \langle B_1, \ldots, B_n \rangle$. We prove completeness by showing that if I $\not\vdash \alpha$, then I $\not\models \alpha$. Thus, we need to exhibit a database d such that $d \models$ I but $d \not\models \alpha$.

By the assumption that I $\not\vdash \alpha$ and thus by Lemma 4.19, we have $I^* \not\vdash \alpha^*$. From the completeness of the axiom system for INDs, it follows that $I^* \not\models \alpha^*$. So there exists a database d^* such that $d^* \models I^*$ but $d^* \not\models \alpha^*$. By Proposition 4.18, we now have $d^* \models$ I. If $d^* \not\models \alpha$, then the result is immediately followed, since d^* is the required database d. So assume that $d^* \models \alpha$. Since $d^* \not\models \alpha^*$, we have a tuple $t \in r$ such that $t[X] \in \pi_X(r)$ but $t[X] \notin \pi_Y(s)$.

We now let $t[X] = \langle a_1, \ldots, a_n \rangle$ and b_i^{max} be any one of the maximal values in $\pi_{B_i}(s)$. Now we claim that there must be some i such that $a_i \sqsubset b_i^{max}$. Otherwise, it leads to contradiction as follows, $\langle a_1, \ldots, a_n \rangle = \langle b_1^{max}, \ldots, b_n^{max} \rangle$ since we assume $d^* \models \alpha$. So there must be a tuple $t_s \in s$ such that $t[X] \sqsubseteq t_s[Y]$. Since all a_i in $t[X]$ are maximal values, it follows that $t_s[Y] = t[X]$ and thus $t[X] \in \pi_Y(s)$.

We define a permutation h on $adom(d^*)$ to swap the elements a_i and b_i^{max} in d^* as follows, $h(a_i) = b_i^{max}$, $h(b_i^{max}) = a_i$ and h is an identity for others. We apply h to exchange the occurrences of a_i and b_i^{max} in d^*. Let the resulting database be d', the tuple t become t' and s become s'. Clearly, $d' \not\models \alpha$, since for all values $b_i \in \pi_{B_i}(s')$, $t'[A_i] \not\sqsubseteq b_i$. It remains to show that $d' \models I$ and then d' is the required database d. Since d' and d^* are identical to each other up to the renaming of a_i and b_i^{max} only, i.e., d' is isomorphic to d^*. Thus, $d' \models I^*$ and it follows by Proposition 4.18 again, $d' \models I$. $\square$

4.3.2 OINDs Arising from Lexicographical Orderings

We now give the definition of a LOIND as follows.

Definition 4.21 (An Ordered Inclusion Dependency Arising from Lexicographical Orderings) An *ordered inclusion dependency arising from lexicographical orderings* (or simply a LOIND) over a database schema $\mathbf{R}$, is a statement of the form $R[X] \tilde{\sqsubseteq} S[Y]$, where $R, S \in \mathbf{R}$ and $X \subseteq R$, $Y \subseteq S$ are sequences of attributes such that $| X |=| Y |$.

We now give the definition of the semantics of a LOIND.

Definition 4.22 (Satisfaction of a LOIND) A LOIND, $R[X] \tilde{\sqsubseteq} S[Y]$, over $\mathbf{R}$ is satisfied in an ordered database d over $\mathbf{R}$, denoted by $d \models R[X] \tilde{\sqsubseteq} S[Y]$, if $\forall t_1 \in r$, $\exists t_2 \in s$ such that $t_1[X] \sqsubseteq t_2[Y]$, where $\sqsubseteq$ is a lexicographical ordering, $r \in d$ is the relation over $R \in \mathbf{R}$ and $s \in d$ is the relation over $S \in \mathbf{R}$.

Similar to POINDs, we observe that an IND in conventional relational databases can be viewed as a LOIND in the special case of unordered databases. The following proposition gives a simple relationship between LOINDs and INDs (c.f., see Proposition 4.18).

Proposition 4.21 If $d \models R[X] \subseteq S[Y]$, then $d \models R[X] \tilde{\sqsubseteq} S[Y]$. $\square$

We note that the converse of the above proposition does not hold. The counterexample can be shown by using the same database d given in Example 4.6; we then have $d \models R[A] \tilde{\sqsubseteq} S[B]$ but $d \not\models R[A] \subseteq S[B]$.

We now give a set of inference rules for LOINDs.

Definition 4.23 (Inference Rules of LOINDs) Let I be a set of LOINDs. The inference rules for LOINDs are defined as follows:

(LOIND1) *Reflexivity*: if $X \subseteq R$, then $\mathrm{I} \vdash R_1[X] \tilde{\sqsubseteq} R_1[X]$.

(LOIND2) *Prefix Projection*: if $\mathrm{I} \vdash R_2[Y] \tilde{\sqsubseteq} R_1[X]$ where $| Pre(X) |=| Pre(Y) |$, then $\mathrm{I} \vdash R_2[pre(Y)] \tilde{\sqsubseteq} R_1[pre(X)]$.

(LOIND3) *Transitivity*: if $\mathrm{I} \vdash R_2[Y] \tilde{\sqsubseteq} R_1[X]$ and $\mathrm{I} \vdash R_3[Z] \tilde{\sqsubseteq} R_2[Y]$, then $\mathrm{I} \vdash R_3[Z] \tilde{\sqsubseteq} R_1[X]$.

It is easy to check that the above inference rules are sound for LOINDs.

Lemma 4.22 The axiom system comprising from LOIND1 to LOIND3 is sound for LOINDs. $\square$

The following proposition shows that there is a simple relationship between POINDs and LOINDs.

Proposition 4.23 If $d \models R[X] \hat{\sqsubseteq} S[Y]$, then $d \models R[X] \tilde{\sqsubseteq} S[Y]$. $\square$

We note that the converse of the above proposition does not hold. For example, let $d = \{r, s\}$ be a database over $\mathbf{R} = \{R, S\}$ where $r = \{01\}$ (1 tuple) and $s = \{10\}$ (1 tuple) are the relations over R and S, respectively, with $R = \{AB\}$ and $S = \{CD\}$. Then $d \models R[AB] \tilde{\sqsubseteq} S[CD]$ but $d \not\models R[AB] \hat{\sqsubseteq} S[CD]$.

We have not been able to establish the completeness for the axiom system comprising from LOIND1 to LOIND3. A further approach to tackle this problem is to establish a chase procedure similar to that of LOFDs; in such case Definition 4.11 may be useful.

4.3.3 Interactions between OFDs and OINDs

There are two important interaction rules between FDs and INDs [109] in conventional databases: (1) the *pullback rule*, which derives a new FD from a FD and an IND, and (2) the *collection rule*, which derives an IND from two INDs and a FD. We first review these two rules and then examine their semantics in the context of OFDs and OINDs.

Definition 4.24 (Pullback Rule and Collection Rule for FDs and INDs) Let Σ be a set of INDs and FDs.

(Pullback): if $\Sigma \vdash R[WZ] \subseteq S[XY]$, with $|X| = |Y|$, and $\Sigma \vdash S : X \rightarrow Y$, then $\Sigma \vdash R : W \rightarrow Z$.

(Collection): if $\Sigma \vdash R[UV] \subseteq S[XY]$, $R[UW] \subseteq S[XZ]$ and $\Sigma \vdash S : X \rightarrow Y$, then $R[UVW] \subseteq S[XYZ]$.

As the next example illustrates, in ordered databases the pullback rule is unsound for OFDs and OINDs.

Example 4.7 Consider the database d given in Figure 4.14. Let $\Sigma = \{R[AB] \hat{\sqsubseteq} S[CD], S : C \hookrightarrow D\}$. Then it can be checked that $d \models \Sigma$ but $d \not\models S : A \hookrightarrow B$. Note that this result still holds even we replace $R[AB] \hat{\sqsubseteq} S[CD]$ by $R[AB] \tilde{\sqsubseteq} S[CD]$, $C \hookrightarrow D$ by $C \rightsquigarrow D$ and $S : A \hookrightarrow B$ by $S : A \rightsquigarrow B$, respectively.

$r =$

A	B
0	0
0	1

$s =$

C	D
1	1

Figure 4.14: A database shows that the pullback rule is unsound for OFDs and OINDs.

We next show that there also exists a counterexample to the soundness of the collection rule for POINDs and OFDs. However, in contrast to the pullback rule in the case of LOINDs, the collection rule is sound for LOINDs and OFDs.

Example 4.8 Consider the database d given in Figure 4.15. Let $\Sigma = \{R[AB] \hat{\sqsubseteq} S[DE], R[AC] \hat{\sqsubseteq} S[DF], S : D \hookrightarrow E\}$. Then it can be checked that $d \models \Sigma$ but $d \not\models R[ABC] \hat{\sqsubseteq} S[DEF]$. Note that this result still holds even we replace $S : D \hookrightarrow E$ by $S : D \rightsquigarrow E$.

$r =$

A	B	C
0	1	1

$s =$

D	E	F
1	1	0
0	0	1

Figure 4.15: A database shows that the collection rule is unsound for OFDs and POINDs

We now give a formal proof of the soundness of the collection rules for LOINDs when interacting with OFDs.

Lemma 4.24 Let Σ be a set of LOINDs and OFDs. The following inference rules are sound for LOINDs and OFDs.

(POFD-LOIND) *Collection I*: if $\Sigma \vdash R[UV] \tilde{\sqsubseteq} S[XY]$, $\Sigma \vdash R[UW] \tilde{\sqsubseteq} S[XZ]$ and $\Sigma \vdash S : X \hookrightarrow Y$, then $\Sigma \vdash R[UVW] \tilde{\sqsubseteq} S[XYZ]$.

(LOFD-LOIND) *Collection II*: if $\Sigma \vdash R[UV] \tilde{\sqsubseteq} S[XY]$, $\Sigma \vdash R[UW] \tilde{\sqsubseteq} S[XZ]$ and $\Sigma \vdash S : X \rightsquigarrow Y$, then $\Sigma \vdash R[UVW] \tilde{\sqsubseteq} S[XYZ]$.

Proof. We show that $\forall t \in r$, $\exists t' \in s$ such that $t[UVW] \sqsubseteq t'[XYZ]$, where $\sqsubseteq$ is a lexicographical ordering. Let $t \in r$. By $R[UV] \tilde{\sqsubseteq} S[XY]$, we have $t_1 \in s$ such that

$t[UV] \sqsubseteq t_1[XY]$. If $t[UV] \sqsubset t_1[XY]$, then the result immediately follows, since we have $t[UVW] \sqsubset t_1[XYZ]$ and thus t is the required t'. Otherwise, $t[UV] = t_1[XY]$ and then we have $t[U] = t_1[X]$ and $t[V] = t_1[Y]$.

By $R[UW] \mathrel{\tilde{\sqsubseteq}} S[XZ]$, we have $t_2 \in s$ such that $t[UV] \sqsubseteq t_2[XY]$ (t_1 and t_2 may not be distinct). If $t[U] \sqsubset t_2[X]$, then the result immediately follows, since we have $t[UVW] \sqsubset t_2[XYZ]$. Otherwise, $t[U] = t_2[X]$ and $t[W] \sqsubseteq t_2[Z]$. So we have $t_2[X] = t_1[X]$. By S: $X \rightsquigarrow Y$ (or S: $X \hookrightarrow Y$), we have $t_2[Y] = t_1[Y]$. Thus, $t[V] = t_2[Y]$. Then the result also follows, since $t[UVW] \sqsubseteq t_2[XYZ]$ and t_2 is the required t'. □

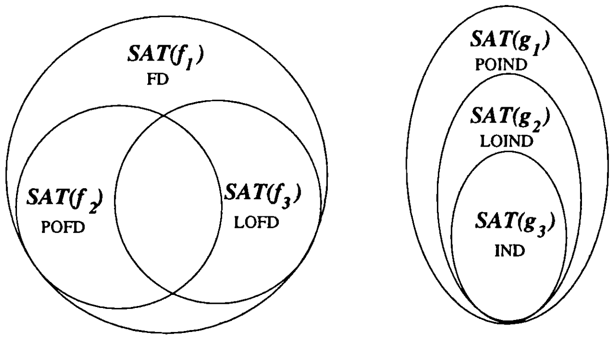


Figure 4.16: Satisfaction of various OFDs and OINDs in databases

We compare the satisfaction of an OFD and an OIND in ordered databases introduced so far as the diagram given in Figure 4.16 (the scale here is irrelevant). We let $SAT(f)$ be a set of database instances that satisfy a data dependency f, and $f_1 = X \dashrightarrow Y$, $f_2 = X \hookrightarrow Y$, $f_3 = X \rightsquigarrow Y$, $g_1 = R[X] \mathrel{\tilde{\sqsubseteq}} S[Y]$, $g_2 = R[X] \mathrel{\hat{\sqsubseteq}} S[Y]$ and $g_3 = R[X] \subseteq S[Y]$.

We remark that in the special case of unordered domains, there are no differences between FDs, POFDs and LOFDs, and between INDs, LOINDs and POINDs. In such a case we have $SAT(f_1) = SAT(f_2) = SAT(f_3)$ and $SAT(g_1) = SAT(g_2) = SAT(g_3)$. Besides, if X and Y are unary, then in general we have $SAT(f_2) = SAT(f_3)$ and $SAT(g_1) = SAT(g_2)$.

4.4 Database Design Issues with respect to OFDs and OINDs

Relational database design plays an important role in relational database theory and thus it is extensively covered in most database textbooks [147, 105, 9, 44]. Relational database design can be viewed as the process of replacing a relation schema R, together with a set of data dependencies over R by a set of relational schemas $\mathbf{R}$, which we call a *decomposition* of R.

Definition 4.25 (Decomposition) A set $\mathbf{R} = \{R_1, \ldots, R_n\}$ is said to be a *decomposition* of a relational schema R (or simply a decomposition whenever R is understood from the context) if $\bigcup_{i=1}^{n} R_i = R$ and $R_i \subseteq R$ for all $R_i \in \mathbf{R}$.

The motivations behind the process of a decomposition are twofold. First, an appropriate decomposition can remove the *redundancy* of data in a relation over R [150]. Second, an appropriate decomposition can remove the problem caused by *insertion and deletion anomalies* [36]. There are many criteria suggested in the literature to capture the notion of appropriateness in conventional databases [105].

One desirable property is that a decomposition possesses the property of *lossless join* (or simply is lossless) [5]. This is because in practice a query usually involves the join of many relations and thus this property guarantees that a relation can be recovered from its projections. We next give the formal definition of this concept.

Definition 4.26 (Lossless Join) Let $\mathbf{R}$ be a *decomposition* of R. The *project-join mapping* with respect to $\mathbf{R}$ is a mapping of relations r over R, denoted by $\mathbf{m_R}$, is defined by $\mathbf{m_R}(r) = \pi_{R_1}(r) \bowtie \cdots \bowtie \pi_{R_n}(r)$. This decomposition is a *lossless join* with respect to a set of OFDs F, if for every ordered relation r over R that satisfies these OFDs, then the condition $r = \mathbf{m_R}(r)$ holds.

Two other desirable properties which leads to good database design are BCNF and key-based INDs. These properties take into consideration the importance of FDs and INDs in conventional databases. FDs generalise the notion of *entity integrity* and *keys* [37] and INDs generalise the notions of *referential integrity* and *foreign keys* [37, 41]. We now extend the definitions of key, superkey and Boyce-Codd normal form into the context of ordered databases [147, 9].

Definition 4.27 (Key and Superkey) Let F be a set of POFDs (or LOFDs) over $\mathbf{R}$ and let $R \in \mathbf{R}$. A sequence of attributes $X \subseteq R$ is a *superkey* for R with respect to F (or simply a *superkey* for R if F is understood from the context) if $\mathrm{F} \models R : X \rightsquigarrow R$ (or $\mathrm{F} \models R : X \hookrightarrow R$). A sequence of attributes $X \in R$ is a *key* for R with respect to F (or correspondingly simply a *key* for R if F is understood from the context) if X is a superkey for R and there does not exist a proper subset Y of X such that Y is a superkey for R.

Definition 4.28 (Boyce-Codd Normal Form) A database schema $\mathbf{R}$ is in *Boyce-Codd normal form* (BCNF) with respect to a set of OFDs F over $\mathbf{R}$ (or simply in BCNF if F is understood from context) if for every OFD, X is a superkey for R.

We now examine a basic result in database design in ordered databases, which states that if a FD $Y \rightarrow X$ holds in a database over a schema $R = \{XYZ\}$, then the decomposition $\mathbf{R} = \{XY, YZ\}$ of R is lossless [147]. This property of FDs forms the basis of an algorithm to obtain a BCNF database schema for obtaining the lossless join of a decomposition having two components. We present the similar result of lossless decomposition for OFDs as follows:

Theorem 4.25 Given a relation scheme $R = \{XYZ\}$ with an OFD, either $Y \hookrightarrow X$ or $Y \rightsquigarrow X$, then the relation scheme R has a lossless decomposition into two components $R_1 = \{XY\}$ and $R_2 = \{YZ\}$.

Proof.

By Proposition 4.17, we have $Y \hookrightarrow X$ or $Y \rightsquigarrow X$ implies $Y \rightarrow X$. Thus it is a lossless join. $\square$

The converse of the above theorem holds [132] in the context of conventional FDs. However, we observe that a similar result does not hold for OFDs, even when we consider unary OFDs. Let us consider the following counterexample.

Example 4.9 Consider a relation r over $R = \{ABC\}$ decomposed into r_1 over $R = \{AB\}$ and r_2 over $R_2 = \{BC\}$, all of which are given in Figure 4.17. It is clear that neither of the following holds in r: $B \hookrightarrow A$, $B \hookrightarrow C$, $B \rightsquigarrow A$ or $B \rightsquigarrow C$.

$r =$

A	B	C
1	4	5
2	3	6

$r_1 =$

A	B
1	4
2	3

$r_2 =$

B	C
4	5
3	6

Figure 4.17: A decomposition of r into r_1 and r_2

We now extend the definition of of key-based INDs [104, 105] for OINDs in ordered databases as follows.

Definition 4.29 (Key-Based OINDs) An OIND, $R[X] \hat{\sqsubseteq} S[Y]$ (or $R[X] \tilde{\sqsubseteq} S[Y]$), over $\mathbf{R}$ is *key-based* if Y is a key for S. If $R[X] \hat{\sqsubseteq} S[Y]$ is a key-based OIND, then the set of attributes X is called a *foreign key* of R with respect to S (or simply a foreign key of R if S is understood from context). A set I of OINDs is key-based if every OIND in I is key-based.

The concept of key-based OINDs is closely related to that of referential integrity [37, 41]. Assume that for each relation schema in $\mathbf{R}$ we designate one of its keys as being a *primary* key [37]. Then a referential constraint can be defined as a key based OIND of the form $R[X] \hat{\sqsubseteq} S[Y]$ (or $R[X] \tilde{\sqsubseteq} S[Y]$), where Y is the primary key of S.

We show another desirable property in that the interactions between OFDs and LOINDs are reduced by showing that F and I have no interaction with respect to the collection rule if $\mathbf{R}$ is in BCNF and I is key-based.

Lemma 4.26 Let α be $R[X] \tilde{\sqsubseteq} S[Y]$ derivable from the inference rules for LOINDs and OFDs, F be a set of POFDs (or LOFDs) such that R is in BCNF and I be a set of key-based LOINDs. Then the collection rule POFD-LOIND (or LOFD-LOIND) is not used in any proof of α from F $\cup$ I.

Proof.

We use induction on the number of inference rules, k, used to derive α from F $\cup$ I.

(*Basis*): If $k = 0$, the result follows since $\alpha \in$ I. So I $\vdash \alpha$.

(*Induction*): Assume the result holds when the number of inferences rules used to derive I $\vdash \alpha$ is k, where $k > 0$. We then prove that the result of the number of inferences rule used to derive α from F $\cup$ I.

By inductive hypothesis, it follows that the collection rule must have been the last inference rule used in the proof of F $\cup$ I $\vdash \alpha$. Let α be the LOIND $R[UVW] \tilde{\sqsubseteq} S[XYZ]$. Thus, it must be the case that α follows from $R[UV] \tilde{\sqsubseteq} S[XY]$, $R[UW] \tilde{\sqsubseteq} S[XZ]$ and $S : X \hookrightarrow Y$ (or $S : X \leadsto Y$). By assumption we have the keys XY, XZ and X. Suppose $V \subseteq U$ and $W \subseteq U$. Then it is clear that $R[UVW] \tilde{\sqsubseteq} S[XYZ]$ can be obtained without using $S : X \hookrightarrow Y$. So we have either $V \not\subseteq U$ or $W \not\subseteq U$. Without loss of generality we assume $V \not\subseteq U$. By the assumption that $R[UV] \tilde{\sqsubseteq} S[XY]$ is key-based, we have XY is a proper superset of X. It follows that X is not a superkey of S, which violates the assumption that $\mathbf{R}$ is in BCNF. $\Box$

As a consequence of above Lemma 4.26 we can see that by assuming key-based LOINDs we can reduce the interactions arising from the collection rule. This is desirable because the implication problem for LOINDs and OFDs reduces to separate implication problems for LOINDs and OFDs. The result is consistent with the suggestion [105] for conventional databases that a good design principle in the presence of FDs and INDs is to obtain a BCNF database schema together with a set of key-based INDs.

4.5 Discussion

We have introduced OFDs and OINDs in ordered databases and studied their implication problems. Moreover, we have also investigated the interactions between OFDs and OINDs by examining the pullback rule and the collection rule. We classify OFDs and OINDs into two categories according to whether they arise from lexicographical orderings or pointwise-orderings on the Cartesian products of underlying domains. In the special cases of unary OFDs and OINDs, these two categories are identical. We have presented a sound and complete axiom system for POFDs. We have also presented a set of sound and complete chase rules for LOFDs, which can be employed as a theorem proving tool for LOFDs. Our results suggest that a good design principle is to obtain a BCNF database schema together with a set of key-based OINDs, since it gives rise to no interactions between OFDs and OINDs with respect to the pullback rule and collection rule.

The chase procedure given in Definition 4.11 can be further utilised to prove the completeness (or otherwise) of the axiom system comprising the inference rules for LOFDs given in Definition 4.16, which is a more refined axiom system for LOFDs. Specifically, we need to show that given a set of LOFDs F and a LOFD f, $CHASE(r,\mathrm{F})$ implies that $\mathrm{F} \vdash f$. The main techniques we use in the proof is to carry out induction on the number of chase steps in $CHASE(r,\mathrm{F})$. If the result can be obtained, then by Theorem 4.12, we have $\mathrm{F} \models f$ implies that $CHASE(r,\mathrm{F})$. Hence we can conclude that $\mathrm{F} \models f$ implies that $\mathrm{F} \vdash f$, which is the completeness of the axiom system. The proof is rather complex since it involves the technicalities in examining many possible cases arising in the inductive step (c.f., see Theorem 5.7 in [87]).

There is an open problem to find a chase procedure for LOINDs, that is, $CHASE(d,\mathrm{I})$, where d is a database and I is a set of LOINDs. Our preliminary idea is that a chase rule for LOINDs can be defined as follows, if $R[X] \tilde{\sqsubseteq} S[Y] \in \mathrm{I}$ and $\exists t_1 \in r$ such that $\nexists t_2 \in s$

with $t_1[X] \tilde{\sqsubseteq} t_2[Y]$, then add a tuple t_2 over S to s, with $t_2[Y] = t_1[X]$ and $\forall A \in S - Y$, $t_2[A] = min$, where min is a minimal value in $adom(d)$ (recall that the active domain $adom(d)$ is defined in Chapter 3). The appropriateness of this definition needs to be further examined. Moreover, we still have to investigate the computational complexity of the implication problems for LOFDs, LOINDs, and mixed OFDs and OINDs.

Finally, we have examined the collection rule and the pullback rule. A further problem following this line is to study the issue concerning the completeness of the axiomatisation of a mixed set of OFDs and OINDs. In other words, we still need to research whether there are any new rules for the interactions of OFDs and OINDs. An intuitive approach is to combine both the chase procedures for OFDs and OINDs in order to build the chase procedure $CHASE(d$,F,I) (c.f., [9, 87]), which is then used to test whether a database satisfies a set of OFDs and OINDs. Meanwhile, the chase procedure $CHASE(d$,F,I) can provide insight into the detailed mechanism of the interactions of OFDs and OINDs.

Chapter 5

An Extension of SQL to the Ordered Relational Model

In this chapter we describe OSQL, which is an extension of SQL for the ordered relational model, and show that OSQL combines the capabilities of standard SQL with the power of semantic ordering. By using OSQL users have the ability to define partial orderings over data domains which are implied by the underlying semantic of the data of an application. The syntax of OSQL is a minimal extension of SQL and thus it should be easy for current SQL users to adapt to OSQL. Although it is a minimal extension, OSQL allows the users to formulate a wide range of queries, such as fuzzy or temporal, which are either very awkward or impossible to formulate in standard SQL. We also discuss the experimental implementation of OSQL that we have carried out using Oracle for low level data management.

We emphasise that OSQL is not just an ad-hoc extension of SQL solely to remedy a few problems with conventional SQL queries. It has a very wide range of applicability which can assist in coping with the recent growing demand of support for advanced database applications such as tree-structured information, incomplete information, fuzzy information and temporal information.

The remainder of this chapter is organized as follows. In Section 1 we discuss the relationship between OSQL and SQL and the motivation for the extension to SQL. In Section 2 we describe the OSQL syntax. In Section 3 we outline the architecture of the implementation of OSQL over Oracle, which is one of the popular relational DBMSs. The prototype, which can be easily adapted to other relational DBMSs, provides us with the platform for gaining feedback from users. In Section 4 we demonstrate the benefits of OSQL through examples of how it can be used to support advanced applications having tree-structured information, incomplete information, fuzzy information and temporal information.

5.1 Comparing OSQL with the SQL Standard

Current relational Database Management Systems (DBMSs) are based on Codd's relational data model and their query languages are specified by the SQL2 standard [43] which is an extension of the relational algebra to incorporate some useful features such as aggregate functions and arithmetic capability [147].

We now demonstrate some of the inadequacies of SQL2 for certain types of fairly common queries. Let us consider the following three queries in the example below:

Example 5.1

1. Obtain the third and sixth lowest rainfalls from a rainfall record.

2. Obtain exactly five vacant seats from a theatre booking system.

3. Obtain the names of all John's bosses.

None of the queries in the above example can be written as a simple SQL statement. The first query shows a common problem that both naive and mature SQL users have. There is no straightforward way to answer this query in SQL. The best way to do this, is to formulate it into a nested query aided with the aggregrate functions *COUNT* and *MAX* (see section 25.1 in [27]). Moreover, it is not easy to avoid mistakes in formulating such SQL statements. It was also discovered in the survey reported in [99] that proper use of aggregate functions and nesting in SQL is difficult for many SQL users.

The second query cannot be answered satisfactorily due to the lack of output control of the number of tuples returned by SQL. As for the last query, in Oracle's SQL we may use the *CONNECT BY* clause to answer it but the same problem as before arises again. The use of this clause is non-trivial and, in addition, we must have the boss/subordinate relationship of each person explicitly stored in the database.

Another option in solving the above problems is to use a database programming approach. The common solutions are: embedded SQL programming such as Oracle's

Pro*C and a procedural language extension to SQL such as PL/SQL. But there are at least three drawbacks in the programming solution. Firstly, this approach requires a very competent level of programming skill, which is definitely a hindrance to the majority of users who are not professional programmers. Secondly, most queries do not need the full power of a fully-fledged programming language and, in fact, we will have to pay the performance penalty if there are too many calls from the programming level to the relational level. Thirdly, ordering is a fundamental property of information. It would be extremely inefficient to embed this property into an application program rather than to capture it in a database model.

SQL3, the most recent version of SQL, has the provision for new data types such as the list type [81, 108], which involves the notion of ordering. In addition, SQL3 has some powerful capabilities for defining abstract data types such as user-defined functions, which can simulate partial ordering of domains. However, as we have discussed in chapter 2.5.3, lists and ordered sets are two incomparable concepts and in addition, the issue of ordering abstract data type instances in SQL3 is a non-trivial issue [108]. Overall, SQL3 is much more complex than SQL2, and the process of adding to SQL3 the facility of managing objects has proved to be extremely difficult. Some design problems have already been found in SQL3 due to incompatible features arising from the integration of object orientation into SQL [106]. In any case the publication of SQL3 as an official standard which will replace SQL2 is predicted to be no sooner than 1998. It is then reasonable to anticipate, from the size of the proposed SQL3 standard documents, that the process of upgrading existing relational database systems in order to comply with the SQL3 standard will take even a longer time.

Based on the above consideration, we suggest OSQL as a suitable intermediate language to fill in the gap between SQL2 and SQL3. There are still two good reasons for the desirability of OSQL. First, due to the relative simplicity of the OSQL extension we do not anticipate many problems in upgrading SQL2 to comply with OSQL's ordering mechanism. Second, OSQL's notion of ordering on domains has the advantage of being easier to comprehend by users than the corresponding notion in SQL3.

We now show how the queries in Example 5.1 can be formulated in a simpler manner in OSQL:

($Q_{5.1}$) *SELECT* (RAIN_FALL) (3,6) *FROM* RAIN_RECORD_TABLE.

($Q_{5.2}$) *SELECT* (SEAT_NUMBER) (1..5) *FROM* THEATRE_BOOKING_TABLE *WHERE* STATUS = 'vacant'.

($Q_{5.3}$) *SELECT* (EMPLOYEE_NAME) (*) *FROM* EMPLOYEE_TABLE *WHERE* EMPLOYEE_NAME > 'John' *WITHIN* EMP_RANK.

Although we have not yet formally introduced OSQL, the meaning of the above statements is quite easy to understand, assuming that the reader has some knowledge of standard SQL. For instance, the clause (3,6) in the query ($Q_{5.1}$) means that the third and sixth tuples, according to the order of RAIN_FALL, are output and the clause (1..5) in the query ($Q_{5.2}$) means that the first to fifth tuples, according to the order of SEAT_NUMBER, are output. The keyword *WITHIN* in the query ($Q_{5.3}$) specifies that the comparison EMPLOYEE_NAME > 'John' is interpreted according to the semantic ordering of the domain EMP_RANK.

We conclude this section by summarising the features of OSQL below:

1. OSQL is based on the PORC (or equivalently, the PORA), which was formally defined in Chapter 3. The PORC is an extension of the relational calculus in the context of the ordered relational model. The expressiveness of the PORA has also been investigated in Chapter 3 and has been shown to be BP-complete [123, 12].

2. OSQL needs few syntactical modifications to the basic form of SQL2 and can also be interfaced to existing relational DBMSs to enhance their expressive power and usability.

3. OSQL incorporates some of the suggestions put forward by Date to improve SQL-type query languages, mainly concerning the support of the wider use of the "<" operator (see chapter 2 in [41]).

4. OSQL can be used to support the modelling and querying of hierarchical data domains such as tree-structured data, which we refer to from now on simply as tree-structured information. OSQL can also be widely applied to areas demanding advanced applications such as the querying of incomplete and temporal databases.

5. OSQL provides an easy way to control the number of output tuples without having to do low level programming. This facility is both necessary and convenient for

database users, especially for those who are non-programmers, when querying over large relational databases.

5.2 OSQL Specification.

In this section, we describe the extensions of OSQL to the Data Manipulation Language (DML) and Data Definition Language (DDL) of the standard SQL. In addition to the extended DML and DDL, OSQL provides a package definition language (PDL), which is detailed in Chapter 7. The full reference of the syntax of OSQL in Backus-Naur Form (BNF) can be consulted in Appendix A.

5.2.1 Data Manipulation Language

Simple queries in OSQL have their general form as:

SELECT ⟨ lists of attributes ⟩ [*ANY* | *ALL*] ⟨ levels of tuples ⟩ [*ASC* | *DESC*]
FROM ⟨ lists of ordered relations ⟩
WHERE ⟨ comparison clause ⟩

Here an *attribute list* is a list of attributes similar to the usual one, except that it provides us with an option that an attribute can be associated with a semantic domain by the syntax *attribute name WITHIN domain name.* The purpose of declaring a *WITHIN* clause is to override the system ordering with the semantic ordering of the semantic domain specified by the domain name. When the *WITHIN* clause is missing then the system ordering will be assumed.

A *tuple level*, which is a set of positive numbers, with the usual numerical ordering, can also be written in some short forms (see Appendix A.2). As a set of tuples in a linearly ordered relation $r = \{t_1, \dots, t_n\}$ is isomorphic to a set of linearly ordered tuples, we interpret each number i in a tuple level as an index to the position of the tuple t_i, where $i = 1, \dots, n$ and $t_1 < \cdots < t_n$. In the case of a partially ordered relation, we generalise the notion of a tuple level, meaning that a tuple level is the set of all *minimal tuples* of (or a subset of) a relation. Recall that we have proved in Chapter 3 how an internal hierarchy can be generated by a successive extracting of tuple levels from a relation.

An interesting situation of an internal hierarchy to consider is when the output of a relation is partially ordered as a tree, having tuple levels $\{l_1, \ldots, l_m\}$. In such a case we choose to interpret each number j in a tuple level as an index to a corresponding tree level l_j, where $j = 1, \ldots, m$ and $l_1 < \cdots < l_m$. Hence, a user can specify the retrieve of *ALL* the tuples or *ANY* one of the tuples in a specified level l_j. We note that in the case of a linearly ordered relation, the choice of using *ALL* or *ANY* has the same effect on the output since there is only one tuple in each level.

Let us examine the following example of an employees relationship of an organisation as shown in Figure 5.1.

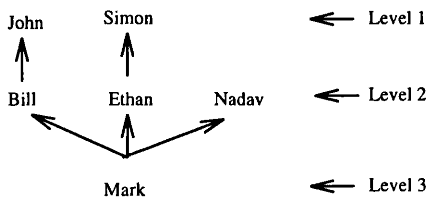


Figure 5.1: Relationship between employees in an organisation

Example 5.2 We can see from this figure that if a user specifies *ALL*(1) in the tuple list, the system returns 'Simon' and 'John'. Alternatively, if a user specifies *ANY*(1) or simply (1) in the tuple list, the system returns only 'John' (the system uses alphabetical ordering to choose the first tuple in this level).

A *comparison clause* follows the *FROM* keyword and a list of all relations used in a query. The meaning of the usual comparators $<, >, <=, >=$ is extended to include semantic comparison as we have mentioned earlier. A typical form of a semantic comparison is:

$\langle$ attribute $\rangle$ $\langle$comparator$\rangle$ $\langle$ attribute $\rangle$ *WITHIN* $\langle$ semantic domain $\rangle$

Without the optional *WITHIN* clause, the comparison is just the conventional one and is based on the relevant system ordering.

The following examples help further to clarify the semantic of the *SELECT* command.

Example 5.3 Let us examine at the following OSQL statements:

($Q_{5.4}$) *SELECT* (NAME, SALARY) (*) *FROM* EMPLOYEE.

($Q_{5.5}$) *SELECT* (SALARY, NAME) (*) *FROM* EMPLOYEE.

($Q_{5.6}$) *SELECT* ((NAME *WITHIN* EMP_RANK), SALARY) (*)
FROM EMPLOYEE.

Note that the ordering of tuples in an output relation depends on two factors. Firstly, on the ordering of domains of individual attributes and secondly on the order of the attributes in an attribute list. The attribute list of the query ($Q_{5.4}$) is (NAME, SALARY), and thus tuples in the output answer are ordered by NAME first and only then by SALARY (see Figure 5.2(a)). Therefore the ordering of tuples is, in general, different to that of query ($Q_{5.5}$), whose list is specified as (SALARY, NAME), since the output of ($Q_{5.5}$) is ordered by SALARY first and then by NAME (see Figure 5.2(b)). It will also be different from that of ($Q_{5.6}$) whose list is ((NAME *WITHIN* EMP_RANK), SALARY), where the ordering of NAME is given by the semantic domain EMP_RANK shown in Figure 5.1 (see Figure 5.2(c)). The standard aggregate functions [80] such as *COUNT, MIN, MAX, AVG, SUM* still apply to ordered relations.

NAME	SALARY
Bill	12K
Ethan	28K
John	14K
Mark	30K
Nadav	28K
Simon	12K

(a)

SALARY	NAME
12K	Simon
14K	Bill
14K	John
28K	Ethan
28K	Nadav
30K	Mark

(b)

NAME	SALARY
John	14K
Simon	12K
Bill	14K
Ethan	28K
Nadav	28K
Mark	30K

(c)

Figure 5.2: An employee relation EMPLOYEE with different ordering

5.2.2 Data Definition Language

The syntax of OSQL allows users to define semantic domains using the *CREATE DOMAIN* command as follows:

CREATE DOMAIN ⟨ domain name ⟩ ⟨ data types ⟩
ORDER AS ⟨ ordering specification ⟩

The first part of the command is similar to the SQL standard statement that declares a domain. Following the *ORDER AS* keywords is a specification of the ordering of a semantic domain. The basic syntax of the *ordering-specification* is: (⟨data-pair⟩, ⟨data-pair⟩,...) where *data-pair* is of the form, *data-item* B < *data-item* A, if and only if *data-item* A is greater than *data-item* B in the semantic domain. For example the definition of the semantic domain shown in Figure 5.1 can be written as follows:

($Q_{5.7}$) *CREATE DOMAIN* EMP_RANK *CHAR*(5) *ORDER AS*
('Simon'<'Ethan', 'John'<'Bill', 'Ethan'<'Mark', 'Bill'<'Mark', 'Nadav'<'Mark').

For a large and complex domain, this syntax may be tedious. Thus OSQL provides two useful short forms to make the task of formulating queries easier. First we allow the use of set notation, {}, to represent a set of data items with common predecessor (or successor). So the previous example can be rewritten as follows:

($Q_{5.8}$) *CREATE DOMAIN* EMP_RANK *CHAR*(5) *ORDER AS*
({'Bill','Nadav','Ethan'}<'Mark', 'John'<'Bill', 'Simon'<'Ethan')

Second we allow the use of the keyword *OTHER* for those data items not mentioned explicitly, with two options *OTHER SYO* and *OTHER UNO* meaning that those data values not mentioned are treated as *SYstem Ordered* or *UNOrdered.* Note that by default we assume other data items are unordered unless there is an explicit declaration that orders these items. We will see more examples later which demonstrate how this keyword can be useful in some applications.

To conclude this section, we show examples of using OSQL in formulating some queries over ordered databases. The reader may refer to the relation EMP1 in Figure 5.9.

($Q_{5.9}$) Find four names of employees whose salaries are greater than 10,000.
SELECT (NAME) (1..4) *FROM* EMP1 *WHERE* SALARY > 10K.

($Q_{5.10}$) Find the first and the fourth lowest salaries.
SELECT (SALARY) (1,4) *FROM* EMP1.

($Q_{5.11}$) Find the highest salary.
SELECT (SALARY) (*LAST*) *FROM* EMP1,
or equivalently,

($Q_{5.12}$) *SELECT* (SALARY) (1) *DESC FROM* EMP1.

5.3 Implementation of OSQL

In this section we present an overview of the design of the system architecture and the implementation strategy of OSQL. The implementation of OSQL provides us the following benefits.

1. It allows us to experiment with the capabilities of OSQL. Hence, we obtain a better insight into the actual benefit of ordering, allowing us to explore further research topics concerning ordered databases.
2. It provides us with a prototype which has enabled us to carry out a user survey, whose findings and feedback are reported in Chapter 6.
3. It offers a basis for us to build packages of different applications in order to enhance the utilities of ordered databases. The use of different packages in many advanced applications is detailed in Chapter 7.

We have implemented OSQL by building a layer on top of the Oracle DBMS. The advantage of this approach is that the prototype of the OSQL system could be implemented fairly quickly by making use of the existing database functionality of Oracle such as storage and transaction management. Moreover, using such a strategy, the OSQL system is very flexible in the sense that it could be easily transported to other platforms. The disadvantage of this approach is that the potential of OSQL cannot be fully analysed. For example, the actual performance of queries over ordered databases is not known to us, since we do not have access to the working of the query optimiser. Nevertheless, further research is needed at the physical level to investigate how best it can support ordered databases. Such an investigation is beyond the scope of this thesis.

5.3.1 The System Architecture

The query language OSQL which operates over ordered relational databases has been prototyped on a SUN machine Unix platform, using Oracle for low level data management. Oracle was chosen since it is both a typical and the world's most popular relational DBMS, which is readily available at UCL. The system allows the user to enter both

OSQL and SQL statements via the front end of a Unix interface. Thereafter, the OSQL precompiler generates a corresponding program consisting of a sequence of Oracle SQL statements and calls a dynamic SQL handling routine. This routine pipes the program into the back end Oracle server for execution. The overview of the system architecture is depicted in Figure 5.3.

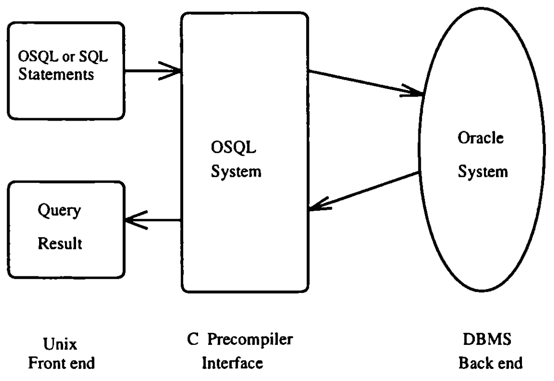


Figure 5.3: Architecture of the OSQL system

The current implementation does not take up many additional resources from the back end relational DBMS. The information about a semantic domain can be realised in a standard relation and thus it can be maintained by the Oracle DBMS. We illustrate that how the semantic domain EMP_RANK described in Figure 5.1 can be easily maintained by using a relation called ORDERING_EMP_RANK created in a database. After executing the *CREATE DOMAIN* statement written as $(Q_{5.7})$, the OSQL system generate an internal relation ORDERING_EMP_RANK to represent the semantic domain EMP_RANK. There are two approaches to construct this internal relation.

One approach is to use *transitive reduction* as the representation of the semantic domain, which is shown in Figure 5.4.

In this approach, the binary relation ORDERING_EMP_RANK, consisting of two attributes over ORDERING_SMALL and ORDERING_LARGE, implements the orderings between pairs of elements. For example, the first tuple in the relation, which is $\langle Bill, Mark\rangle$, means that Bill is under Mark in the organisation. Note that the relation ORDERING_EMP_RANK is a *transitive reduction* in the sense that it contains no tuple derivable from *transitive closure*. This approach caters for space reduction, i.e., we use the minimal numbers of tuples describing the semantic ordering of a given domain. The transitive closure can be easily obtained by the command *CONNECT BY* in Oracle,

ORDERING_SMALL	ORDERING_LARGE
Bill	Mark
Ethan	Mark
Nadav	Mark
John	Bill
Simon	Ethan

ORDERING_EMP_RANK

Figure 5.4: An internal relation to maintain the semantic domain EMP_RANK

which essentially performs a closure operation.

Another possible approach which uses the transitive closure as the representation of semantic ordering shown in Figure 5.5. This approach has the advantage of minimising the cost of query execution time. Let us consider the following semantic comparison in OSQL: STAFF < 'Mark' WITHIN EMP_RANK. In such case the semantic comparison can be done in a *linear time* if we use the transitive closure of ORDERING_EMP_RANK.

ORDERING_SMALL	ORDERING_LARGE
Bill	Mark
Ethan	Mark
Nadav	Mark
John	Bill
Simon	Ethan
John	Mark
Simon	Mark

ORDERING_EMP_RANK

Figure 5.5: Using a transitive closure to maintain the semantic domain EMP_RANK

Although these two approaches, the *transitive reduction* representation and the *transitive closure* representation, are equivalent in the sense that they represent the same partial ordering of a semantic domain, they have different implications in updating semantic domains. If we delete the tuple $\langle John, Bill\rangle$ of ORDERING_EMP_RANK in Figure 5.4 (i.e., the transitive reduction representation), then in the meantime it *implicitly* removes the ordering relationship between John and Mark. We also note that in this approach we have freedom to delete any tuple. In contrast, if we delete the same tuple $\langle John, Bill\rangle$ of ORDERING_EMP_RANK in Figure 5.5 (i.e., the transitive closure representation), it preserves the semantics of orderings of other elements in the domain. However, it may not possible to delete a particular tuple in such approach. For example, we cannot delete the tuple $\langle John, Mark\rangle$ only, otherwise the relation would be an invalid representation of a partial ordering, since we have $John \sqsubseteq Bill$ and $Bill \sqsubseteq Mark$ but $John \not\sqsubseteq Mark$, which violates the *transitivity* criteria of a partial ordering (see Definition 2.1).

We remark that in most cases it is not necessary that all the values in a semantic domain be explicitly stored in the database because many of these values are unordered relative to each other (recall the keyword *OTHER* to represent those values which are not mentioned). Moreover, we use the Oracle SQL command *CREATE VIEW* to form the necessary intermediate relations, and thus should not burden the system with large space usage overheads. Moreover, the dynamic SQL routine guarantees that the translated SQL program runs efficiently.

5.3.2 The Implementation Method

The implementation of the OSQL system is divided into four main modules as follows: *Tokens Checker*, *Syntax Checker*, *SQL Handler* and *SQL Translator*. We describe their functions in the table given in Figure 5.6.

We use Lex, Yacc and Pro*C (Oracle's C compiler allowing embedded SQL statements) to implement the OSQL system. This system is complied and linked together to perform the translation of OSQL statements into SQL statements. The diagram in Figure 5.7 illustrates the implementation of the system with respect to its modules.

The on-line running of the system performs four basic processes: (1) OSQL lexical and syntax checking, (2) Translating from OSQL into SQL, (3) Interacting with Oracle, (4) Exception Handling of the different stages. Firstly, the system check the correctness of an input OSQL statement (see Appendix A for the formal syntax of OSQL) and separate it from system control commands such as "quit". This process includes checking whether the symbols (or tokens) and the syntactical structure of an input statement are

	Modules	Functions
1.	Token Checker	Checking for the correctness of symbols (or tokens) of an input statement or command
2.	Syntax Checker	Checking the correctness of the syntactical structure of an input statement
3.	SQL Handler	Communicating between the OSQL interface and Oracle
4.	SQL Translator	Translating a valid OSQL statement into a SQL program

Figure 5.6: Brief description of the modules in the implementation of the OSQL system

valid. We invoke the module *tokens checker* and the module *syntax checker* to perform the tasks, which are implemented by the programs called *osql.l* and *osql.y*, respectively. Secondly, the module *OSQL translator*, which is implemented by the program *translation.c*, translates a valid OSQL statement into a SQL program, i.e., a sequence of standard SQL statements. Another common implementation approach is to combine the process of lexical and syntax checking and code generation. However, we find that in our case the separation of these modules is easier to manage in practice. Thirdly, the module *SQL handler*, which is implemented by the program *dynamic.pc*, communicates with Oracle in order to execute the translated SQL program. The communication between the C programs and Oracle is done by embedded Pro*C calls, which is the Oracle C programming interface. In our case, the SQL program is not fully known until runtime. For example, the tables or columns to be referenced in a *SELECT* statement may only be known as a result of data itself retrieved when the system is running. Therefore, we use *dynamic SQL Method 4* (i.e., SQL Descriptor Areas) [119], which can handle such statements with an unknown number of select-list items or input host variables to be constructed at run time and then executed dynamically. Fourthly, at different stages there are many possible kinds of errors occurring, which are dealt by the SQL Handler during the exception handling process. The rollback mechanisms becomes very complex

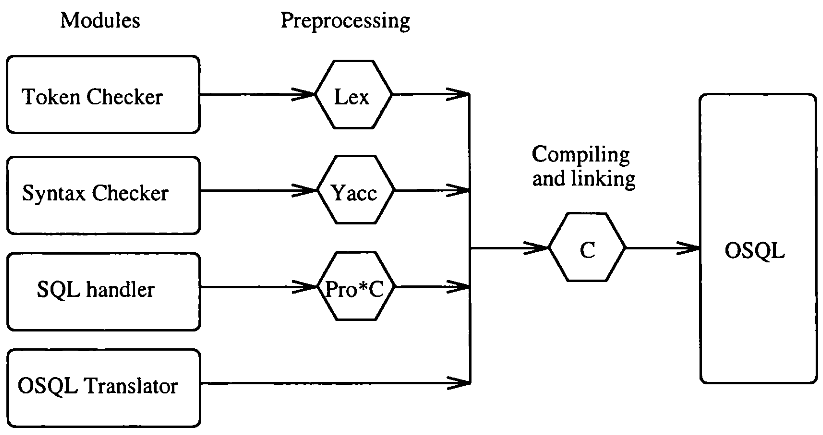


Figure 5.7: The implementation of the OSQL system

and our system is still immature in this respect. Thus, there is much room for improving in the exception handling process. As we discuss in Chapter 6, one common feedback from the user survey is that the system can be improved by providing them with more error diagnosis and on-line help.

The implementation was achieved using Oracle SQL statements and is currently operational; all the OSQL statements mentioned in this Chapter have been successfully tested on the system. Some sample code for the mentioned programs are listed in Appendix D. The diagram given in Figure 5.8 illustrates the stages in the on-line running of the system.

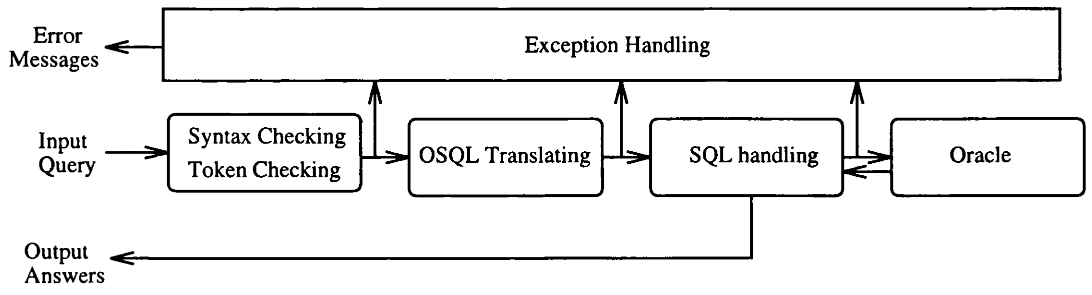


Figure 5.8: The stages of on-line running of the OSQL system

5.4 Application Examples of OSQL

There is a growing demand for support in relational DBMSs of applications involving tree-structured information [15], incomplete information [38], fuzzy information [17] and temporal information [144]. There has already been a fair amount of research on how to incorporate such applications into the framework of the relational model; see for instance [37, 145, 111]. The drawback of the solutions that have been proposed so far is that they do not provide a unified treatment for all the above mentioned applications. In this section, we demonstrate how OSQL can be applied to solve various problems that arise in all of these applications under the unifying framework of the ordered relational model.

We remark that many tailor-made systems have been developed for specific applications [16], representing an approach which is quite different from what we suggest. This can arise from a perceived need to support specialised domains. For example, some temporal researchers [145] claim that the time domain is fundamentally different from other relational model domains. One even claims that his particular time domain is fundamentally different from other temporal domains [75].

Unquestionably, a system tailored to a particular application has the advantage of making it easier for users of that application to understand the operations supported. However, a tailor-made system lacking a general algebraic query capability may be less powerful than one based on the unified approach we present. Furthermore, such specialised approaches either make substantial changes to standard SQL or extend the conventional relational model, this forming a barrier to their widespread adoption.

5.4.1 Tree-Structured Information

Tree data is very common in practice, for example, the manager/subordinate relationship and the parent/child relationship. In standard SQL the support for tree-structured information is poor due to the fact that there is no trivial way to handle tree like structures when the data elements are unordered [27]. We now consider the following relation EMP1 as shown in Figure 5.9.

Suppose that the hierarchy of the employees in EMP1 is as in the diagram shown in Figure 5.1 and that the domain EMP_RANK is declared as by the statement ($Q_{5.7}$) (or equivalently the statement ($Q_{5.8}$)). We consider the following queries over the relation EMP1.

($Q_{5.13}$) Find the most senior staff member.

SELECT ((NAME *WITHIN* EMP_RANK)) (1) *DESC FROM* EMP1.

NAME	SALARY
Bill	12K
Ethan	29K
John	14K
Mark	30K
Nadav	28K
Simon	10K

Figure 5.9: An employee relation EMP1

($Q_{5.14}$) Find a member of staff at the most junior level.

SELECT ((NAME *WITHIN* EMP_RANK)) (1) *FROM* EMP1.

($Q_{5.15}$) Find all the members of staff at the most junior level.

SELECT ((NAME *WITHIN* EMP_RANK)) *ALL*(1) *FROM* EMP1.

($Q_{5.16}$) Find the name and salary of all the bosses of "John".

SELECT (∗) (∗) *FROM* EMP1 *WHERE* (NAME > 'John' *WITHIN* EMP_RANK).

($Q_{5.17}$) Find the name and salary of the common bosses of "John" and "Simon".

SELECT (∗) (∗) *FROM* EMP1 *WHERE* (NAME > 'John' *WITHIN* EMP_RANK) *AND* (NAME > 'Simon' *WITHIN* EMP_RANK).

Various semantic domains could be defined on employees, which depend on the needs of specific applications. For example, in a typical organisation we can define a domain EMP_QUALIFICATION or EMP_CONNECTION which orders the names of employees according to their academic qualifications and community connections, respectively.

5.4.2 Incomplete Information

In reality, we do not expect a database containing large volumes of data to have perfect information on the enterprise it is modelling, due to the fact that information may be missing or imprecise. We call the former type of information *incomplete information* [38] and the latter type of information *fuzzy information* [17]. We expect an upgraded relational DBMS to handle incomplete and fuzzy information and provide reasonable answer for queries over them.

Consider the relation EMP2, shown in Figure 5.10, which contains some records with incomplete information. We classify the incompleteness into three unmarked *null symbols* whose semantics is given in [154, 38, 37], respectively:

UNK: Value exists but is UNKnown at the present time, for example some employees do not want to disclose their ages, this kind of incompleteness is presented as the symbol "UNK".

DNE: Value Does Not Exist, for example a fresh graduate does not have any previous work experience.

NI: No Information is available for the value, for example we may not have any information available as to whether an employee has previous working experience. The employee either has no previous working experience or it is unknown at the present time.

NAME	SALARY	PREVIOUS_WORK	ACADEMIC_ATTAINMENT
Mark	30K	UNK	PhD
Ethan	29K	DNE	MSc
Nadav	28K	administrator	MBA
Bill	20K	programmer	MSc
John	14K	NI	BSc
Simon	10K	NI	A Level

Figure 5.10: An employee relation EMP2

We now introduce the notion of *more informative* values, which allows us to deduce useful information available from the relation having incomplete data. The diagram in Figure 5.11 shows a partial ordering, say $\leq$, based upon the relative information content in a domain augmented with the three null values we have introduced. We can extend this partial order to tuples by defining a tuple t_1 to be less informative than another tuple t_2 if for all attributes A in the relational schema, $t_1[A] \leq t_2[A]$.

In other words, UNK and DNE are more informative than NI, and any values which are not unmarked null symbols are more informative then UNK. Let us define a semantic domain called INCOMPLETE_DOMAIN for the attribute PREVIOUS_WORK:

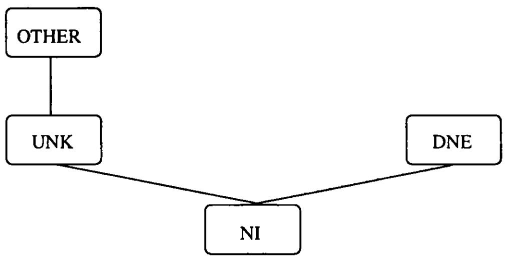


Figure 5.11: A partial ordering on a data domain

($Q_{5.18}$) *CREATE DOMAIN* INCOMPLETE_DOMAIN *CHAR*(10) *ORDER AS* ('NI'<'DNE','NI'<'UNK'< *OTHER*).

Now we can query the relation EMP2 as follows:

($Q_{5.19}$) Find the name and previous work of those employees whose previous work is more informative than or equal to UNK.
SELECT (NAME, PREVIOUS_WORK) (∗) *FROM* EMP2
WHERE (PREVIOUS_WORK >= 'UNK' *WITHIN* INCOMPLETE_DOMAIN).

5.4.3 Fuzzy Information

There is a strong correspondence between ordering and fuzziness. Assuming that the comparison, <, indicates linear ordering, the semantic comparison $x_1 < x_2$ can be used to represent the fact that the data value x_1 is fuzzier than the data value x_2 [29]. The smaller the value is with respect to an ordered domain, the fuzzier the value is relative to a given fuzzy requirement. For example, the more junior an employee is with respect to the ordered domain EMP_RANK, the "better chance" for this employee to be promoted.

The advantage of using such associations is that we do not need to define a *membership function* for a fuzzy set of data values as adopted by the traditional approach in fuzzy set theory [76]. Therefore, we can avoid measuring the fuzziness of data in terms of an exact number, which is in practice difficult and sometimes unnatural.

As a more detailed illustration, suppose that there is a project which requires an employee with a good science background and academic qualification. We can declare a semantic domain called PREFERRED_QUALIFICATION to capture the semantics of the requirement "good science background" as follows:

($Q_{5.20}$) *CREATE DOMAIN* PREFERRED_QUALIFICATION *CHAR*(10)
ORDER AS ({*OTHER UNO*}<'BSc', 'BSc'<{'Phd', 'MSc'}) .

We demonstrate our idea by a typical example of a fuzzy query.

($Q_{5.21}$) Find three names of those employees with good science background and academic qualification preferred.

SELECT ((ACADEMIC_ATTAINTMENT *WITHIN*
PREFERRED_QUALIFICATION), NAME) (1..3) *DESC FROM* EMP2.

Another advantage of the above approach is that, as pointed out by Chang in [29], the output will be a sorted list of tuples arranged in a manner so that the first will be the "most appropriate" one with respect to the fuzzy requirement "good science background and academic qualification". As a result, it supports a decision based on a fuzzy criterion.

5.4.4 Temporal Information

There is already a substantial amount of research on incorporating time into the framework of the relational model (see the collection of papers in [145]). This research is motivated by the many applications that need to make reference to past and/or future data. For example, storing historical data allows this data to be reviewed for forecasting purposes. One of the approaches to manipulating temporal data is to use an attribute, which we call a *time attribute* and to *timestamp* the attribute values of this attribute with *time-points* [144]. For simplicity, we assume that the timestamping denotes *valid time* [145]. Let us consider the example shown in Figure 5.12, where we make use of the time attribute SALARY_TIME to timestamp the attribute SALARY with the time-points of years. For instance, we can see that Mark had salary 20K in 1990 and his salary increased to 25K in 1992.

In contrast to [144] we do not use an ordered pair such as $\langle SALARY, SALARY\_TIME\rangle$ as a data item to represent temporal information, since it gives a Non-First Normal Form relation resulting in an added degree of complexity.

The following temporal queries are typical for a temporal database.

($Q_{5.22}$) What was the salary of Bill in 1990?

SELECT (SALARY_TIME, SALARY) (*LAST*) *FROM* EMP3 *WHERE*
NAME = 'Bill' *AND* SALARY_TIME <= 1990.

($Q_{5.23}$) What is the salary of Mark now (assume the current year is 1996)?

SELECT (SALARY_TIME, SALARY) (*LAST*) *FROM* EMP3 *WHERE*

NAME	SALARY	SALARY TIME
Mark	20K	1990
Mark	25K	1992
Mark	30K	1995
Ethan	21K	1994
Ethan	29K	1996
Nadav	28K	1995
Bill	10K	1988
Bill	15K	1991
Bill	18K	1995
Bill	20K	1996
John	14K	1996
Simon	10K	1996

Figure 5.12: An employee relation EMP3

NAME = 'Mark' *AND* SALARY_TIME <= 1996.

($Q_{5.24}$) What was the starting salary of Mark and when was it?

SELECT (SALARY_TIME, SALARY) (1) *FROM* EMP3 *WHERE*
NAME = 'Mark'.

($Q_{5.25}$) What is the salary history of Ethan?

SELECT (SALARY_TIME, SALARY) (∗) *FROM* EMP3 *WHERE*
NAME = 'Ethan'.

($Q_{5.26}$) What are the names of employees who earned more than 20K during the period 1992-1995 inclusive?

SELECT (NAME) (∗) *FROM* EMP3 *WHERE*
SALARY > 20K *AND* SALARY_TIME <= 1995 *AND* SALARY_TIME >= 1992.

Note that the relation EMP3 does not necessarily contain a tuple for every year, for example there is no change in salary for Bill in 1990. Thus, in the query ($Q_{5.22}$) OSQL will select the "most recently updated" tuple before 1990. This can be done by using the keyword *LAST* together with the comparison SALARY_TIME <= 1990 as shown in

($Q_{5.22}$). A similar remark also applies to query ($Q_{5.23}$).

Chapter 6

A User Survey on OSQL

In Chapter 5 we discussed OSQL, which is the extension of SQL for the ordered relational data model. OSQL provides the facilities of semantic orderings over domains in addition to the standard ones. From the point of view of usability, we believe that the human factors are important measures in order to justify our implementation of OSQL. In order to gain further insight into the usage and the acceptance of the semantic orderings provided by the OSQL SELECT command, we invited 70 students (or simply the *students sample*), who were studying a relational databases course, and 10 computer professionals (or simply the *professionals sample*) to participate in our user survey (or simply *the survey*). We call the students sample and the professionals sample collectively, *the subjects.* The subjects were asked to apply the OSQL SELECT statement to formulate a set of queries in the experiment designed for the survey (or simply *the survey queries*), which was done using our prototype of OSQL as detailed in Chapter 5. The survey queries are very common queries when using order in databases, and represent different degrees of difficulty involving the notion of order. The subjects were then requested to hand in the solution to the survey queries and to comment on the difficulty and the usefulness of the OSQL SELECT statement.

We present the results of the experiment as part of the report on the survey. From the attitudes of the subjects towards the use of OSQL, we have determined whether the various proposed features of OSQL are really easy to learn, understand and apply in practice. Furthermore, from our observations throughout the process of the survey and the communication with the subjects via different channels (e.g., email), we have obtained many valuable suggestions which should help to improve OSQL and convert the current

OSQL prototype into a fully-fledged product in the future. Amongst our conclusions we identify that (1) the subjects in the survey formulated difficult queries involving order in an easier manner than in SQL, and (2) the extended features in the OSQL SELECT statement were easy for the subjects to learn, understand and apply. The survey may also provide beneficial pointers to the teaching, presentation and evolution of OSQL and SQL in general.

The specific objectives of the study include the following five aspects.

1. To access the subjects' attitudes on the usefulness of the three extended features of OSQL which are (1) the attribute list, (2) the tuple list, and (3) the comparison clause of the OSQL SELECT command (explained in detail in Section 6.2).

2. To determine whether the said features of the OSQL SELECT command are easy to apply in formulating the survey queries.

3. To compare the accuracy of the solutions to the survey queries being formulated in OSQL and SQL, respectively.

4. To identify ways of improving the syntax of the OSQL SELECT command and the current implementation of OSQL from the feedback of the subjects.

5. To identify the related areas that can be helpful in the development of more practical and effective training courses for current and potential OSQL users.

In Section 6.1 we describe the method used in carrying out the survey and the general issues relating to the survey queries. In Section 6.2 we discuss the underlying ideas in designing the questionnaire of the survey. In Section 6.3 we present the results of the survey. The analysis is grouped under the following categories: (1) the knowledge profile of the subjects, (2) the result of the subjects' attempts in formulating the survey queries, and (3) the subjects' attitudes on the extended features of the OSQL SELECT command. In Section 6.4 we discuss the implications of the survey for the issues concerning the usages and development of OSQL.

6.1 The User Survey

The students sample (70 people) were all full-time students of which 39 were undertaking the conversion MSc in Information Technology or Computer Science at University College

London (or UCL). The other 31 students were final year undergraduates. They had completed the first 8 weeks (out of 11 weeks) lectures of a database management course and had just finished a piece of coursework on using SQL prior to the survey. Most of the students had not used Oracle before the course. However, a majority of them had other computer science knowledge and experience such as C++ programming.

The professionals sample (10 people) consisted of 5 computer science researchers and 5 experienced practitioners. The researchers were doing research on some areas related to database systems in the Department of Computer Science of UCL. The 5 experienced practitioners had been exposed to a relational database environment for at least 2 years and had been using SQL.

There were only very few unanswered questionnaires (5 people in the students sample and 1 person in the professionals sample), which were treated as invalid questionnaires, and were not taken into account in our survey. As a result, 74 questionnaires were completed totalling a response rate of 92.5%.

We targeted the extended features of the OSQL SELECT statement, which were classified into the following three aspects: the attribute list, the tuple list and the comparison clause. We presented the OSQL SELECT statement in the format as given below to the subjects in order to facilitate better comprehension.

SELECT ⟨ attribute list ⟩ ⟨ tuple list ⟩
FROM relation list
WHERE ⟨ comparison clause ⟩

The survey queries were designed to involve the use of the mentioned features. The subjects were required to attempt the survey queries listed in the experiment sheet (see the survey documents in Appendix C). There were a total nine queries which retrieved information from an Oracle database provided by the Department of Computer Science in UCL. The database consisted of two ordered relations containing the information that might be used by a frame company and a furniture company, respectively. The structure and content of these relations were straightforward enough to be easily understood. The subjects were required to use OSQL (as Task I in the experiment) and SQL (as Task II in the experiment), respectively, to formulate these queries over the given database.

Prior to the experiment, there was one meeting with students, which lasted up to 60 minutes. Before the meeting we had prepared the following documents the subjects needed to complete the survey (which we will refer to as *the documents*).

1. An OSQL mini-manual (or simply the *manual*) described the essential features of OSQL, the overall architecture of the system and the technical details to connect to the system.

2. An experiment sheet listed the survey queries and included the necessary instructions for returning their solutions.

3. A questionnaire consisting of 6 questions.

At the beginning of the meeting, we issued the manual, the experiment work sheet and the questionnaire to the subjects. Thereafter the meeting was devoted to introducing OSQL and the demonstration of the example queries in the manual (see Appendix C.1 for Queries 1 to 15 given in the manual). The subjects were requested to return within one month, their solutions to the survey queries and their output from these queries, and also the filled in questionnaires. Finally, we had approximately ten minutes to allow the subjects to ask questions about the survey.

In order to motivate the students sample to complete the survey queries, we notified the students that their work for the experiment sheet would contribute part of the marks for their coursework. On the other hand, for the sake of objectivity, we also told them that their questionnaire would not count towards their final mark but that its completion was compulsory. Finally, we assured the students that the questionnaire would be treated in an anonymous fashion. The aim of the said measures was to encourage the students sample to be honest when filling in their questionnaires.

We adopted a binary grading method, *correct* or *incorrect*, to simplify the process of deciding the correctness of the subjects' work on the survey queries. These two grades were classified according to the criteria as stated below.

1. A *correct solution* meant that one of the following three conditions was satisfied:

   (a) The solution to a query was completely correct.

   (b) The solution to a query had a minor error such as a missing attribute in an attribute list. This error was able to be corrected fairly easily and the solution yielded a reasonable result.

(c) The solution to a query was correct with respect to another possible interpretation of the query and this alternative interpretation was reasonable.

2. An *incorrect solution* meant that one of the following three conditions was satisfied:

   (a) The query was unattempted.

   (b) The solution to a query had a major error.

   (c) The solution was correct with respect to another possible interpretation of the query. However, this alternative interpretation was unreasonable.

If several errors in both categories were found in the solution of a query, the solution was treated as an incorrect solution. The order of queries in the experiment sheet was roughly based on our beliefs of their difficulty. Thus, Query 1 was assumed to be the easiest one and Query 9 the hardest one.

We used simple statistical analysis on the results of Task I and Task II of the experiment such as the percentage of correct answers to each query. We were cautious in comparing the solutions of SQL and OSQL because for some simple queries their results were remarkably close. The details of the statistics of correct solutions to the survey queries formulated in OSQL and SQL are summarised in tabular format and will be discussed in Section 6.4.

6.2 Questionnaire Design

The survey questionnaire, which was designed to be as concise as possible for the convenience of the subjects, contained six main questions (some questions had several parts). As we already stated before, the majority of the subjects were students in the Department of Computer Science at UCL. The background information most interesting to us was their experience of using SQL or having other related computer experience. Based on these considerations, we decided that Questions 1 and 2 of the questionnaire would solicit information about the profile of respondents' SQL knowledge and the programming languages known to them.

Question 3 collected the data about the OSQL experiment in a table, which recorded the number of attempts for each query in Task I of the experiment. The number of attempts to complete a query reflected the difficulty encountered by subjects when using OSQL to formulate it. We classified the number of attempts into four categories as below.

1. Less than three times.

2. Three to six times.

3. More than six times.

4. Unsuccessful.

We decided that the first category of the above classification was a reasonable range to show that the query was relatively easy to users. The next category implied that there were minor problems for the users when using OSQL. The subjects might experience minor frustration but making three to six times attempt did not seem to discourage them very much. The third category indicated that the subjects had problems in formulating a query using OSQL. The fourth category is self-explanatory, which implied that the subjects failed to formulate a query in OSQL.

Question 4 was another part (Task II) of the experiment which recorded the number of attempts at using standard SQL to formulate the survey queries, which were the same set of queries as in Task I. We used them as a reference to compare with the result of the corresponding queries formulated in OSQL.

Question 5 was the evaluation of the extended features of OSQL SELECT command given by the subjects. We adopted the five-point Likert scale, which is the most commonly used attitude scale type in Psychology [93], to seek comments on the usefulness and the difficulty of the OSQL statement. This question reflected the attitudes of the subject towards the OSQL SELECT commands. In order to evaluate the command in more detail, we specifically asked the subjects in this question for the usefulness of the three extended features of OSQL, i.e., the attribute list, the tuple list and the comparison clause as well as the difficulty they had encountered while using them (see the instructions to the subjects and the format of this question in Appendix C.3).

Finally, question 6 is designed to be divergent so that we can gain feedback on OSQL in a more open manner.

6.3 Summary of Results

6.3.1 Knowledge Profile of the Subjects

The following table summarises the general information of the subjects' experience of using SQL.

Experience of SQL	Percentage
Learned SQL on the course	85.00
Less than 2 years	6.25
2 to 4 years	6.25
More than 4 years	2.50

Figure 6.1: Experience of SQL of the subjects

The majority of the subjects (85%) were first exposed to SQL through the course in our department as shown in Figure 6.1. This is a reasonable result because most of the subjects are students. It could be argued that if the questionnaire was distributed more widely to include more SQL professionals in the subjects, the result would be more representative. However, in our case we have an advantage that the subjects acquired similar exposure to both SQL and OSQL, and thus the comparison of these languages can be achieved in a fairer manner. Furthermore, any negative responses found in the survey signify that they should be recognised at an early stage in the training of both using OSQL or SQL.

Other Programming Languages	Percentage
C or C++	92.50
Miranda	58.75
Microsoft software	37.50
HTML	25.00
Java, Basic and others	26.25

Figure 6.2: Computing knowledge and experience of the subjects

As for the computing knowledge and experience of the subjects, we find that they had a certain degree of exposure to other programming languages as shown in Figure 6.2. The most popular ones were C, C++ and Miranda, which were the languages the students sample learned from the basic courses provided by our department. However, it was very rare for the subjects to have knowledge and experience in other database programming languages such as those offered by Sybase or Informix. The Microsoft software here, refers to word processing software and similar packages.

6.3.2 The Result of Formulating the Survey Queries

In order to compare the performance of using OSQL and SQL, we aggregate the results of the number of the correct solutions and the attempts for all queries in Tasks I and II. We compare the result of Question 3 and 4 from two perspectives. Firstly, in Figure 6.3 we present a bar chart of the percentage of the total number of correct solutions obtained using OSQL and that of using SQL. Secondly, we present a table in Figure 6.4 showing the aggregated result for the number of attempts for each query recorded in Question 3 and Question 4 of the questionnaires. It seems reasonable to view both the number of correct solutions obtained and the number of attempts as pointers which indicate the difficulty of either using OSQL or SQL to formulate a query.

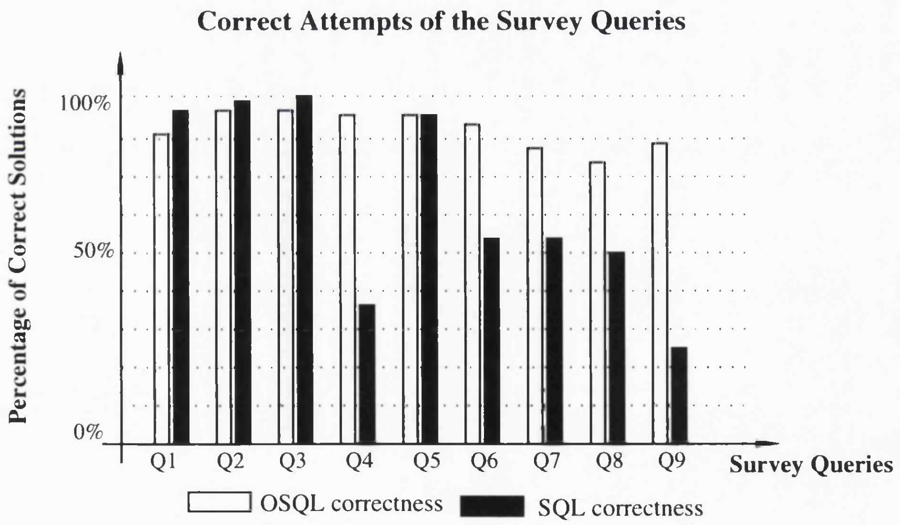


Figure 6.3: A bar chart to compare the correct solution of the survey queries

Figure 6.3 shows that over 90% of subjects formulated Query 1, 2 and 3 correctly both in OSQL and SQL. It is not difficult to verify that the subjects showed less than

Queries	Languages	Percentage			
		Less than 3 times	3 to 6 times	More than 6 times	Unsuccessful
1	OSQL	92.31	4.62	3.08	0.00
	SQL	96.92	3.08	0.00	0.00
2	OSQL	92.31	6.15	1.54	0.00
	SQL	96.92	3.08	0.00	0.00
3	OSQL	89.20	10.77	0.00	0.00
	SQL	93.85	4.62	1.54	0.00
4	OSQL	83.07	16.92	0.00	0.00
	SQL	7.69	35.38	33.85	23.08
5	OSQL	76.92	20.00	3.07	0.00
	SQL	75.38	18.46	4.62	1.53
6	OSQL	44.61	44.61	9.23	1.53
	SQL	6.15	43.08	27.69	23.08
7	OSQL	53.85	36.92	7.69	1.53
	SQL	18.46	40.00	15.38	26.15
8	OSQL	43.08	43.08	13.85	0.00
	SQL	33.85	32.31	10.77	23.08
9	OSQL	36.92	44.61	15.38	3.08
	SQL	15.38	40.00	23.08	21.54

Figure 6.4: Attempts of survey queries formulating in OSQL and SQL

5% difference in the correctness of formulating these queries, which were considered to be easy queries. It is also consistently shown in Figure 6.4 that over 90% subjects were able to finish these queries in less than 3 attempts. One further interesting point which can be deduced from Figures 6.3 and 6.4 is that there is a strong correlation between the number of attempts in formulating the survey queries and the percentage of correct solutions obtained for the survey queries. Thus, it implies the obvious conclusion that the less the number of attempts that are required to formulate a query, the higher the chance that the query is correct. It is also interesting to note that the performance of using SQL is slightly better than that use of OSQL in these three simple queries. This can be attributed to the simpler syntactical structure of the SQL SELECT statement and thus the subjects could formulate these queries more accurately.

A substantial difference in the performance of formulating queries in OSQL and SQL is found in Query 4 given as below.

What is the cost of the third and fifth cheapest part?

The query above involves the uses of many levels of nesting in SQL and it was a much harder query for the subjects as many researchers have already anticipated [39, 99]. Under this circumstance, the merits of using the tuple list in the OSQL SELECT command is fairly evident as indicated in Figure 6.4. It shows that over 80% of the subjects finished this query in less than three attempts when using OSQL whereas less than 10% finished within three attempts when using SQL to do it. Moreover, the unsuccessful attempts using SQL are up to 23%, which indicates some inadequacies of SQL in handling this type of query.

The performance in Query 5 using SQL gives a much better result than Query 4 because this query just requires the subjects to use the standard orderings provided by Oracle using comparison and sorting. However, when it comes to the queries requiring more semantics of data (a parts hierarchy and incomplete information) as in Queries 6, 7, 8 and 9, the result shows clear differences between using OSQL and SQL. In this case the number of attempts using SQL is significantly greater than that of OSQL. Also more subjects, between 20% to 50% more, managed to work out the correct solution when using OSQL. It shows that OSQL is very useful in the area of semantic comparison and confirms that our extension on semantic domains is helpful.

Considered above, the superiority of OSQL has been demonstrated in the survey queries because using it required less query attempts and also more accurate results were obtained in formulating the survey queries using OSQL. Moreover, OSQL is also a viable extension to SQL in practice if we take account into the fact that we just used one hour to introduce OSQL to the subjects for the experiment of the survey. Note that standard SQL queries will execute in the OSQL system, since the system is designed to be upwards compatible.

6.3.3 The Subjects' Attitudes to the Extended Features

We summarise the results of Question 5 by aggregating the results into percentages and then summarising the data into the table shown in Figure 6.5. The majority of the subjects, which is well over 80%, chose the scales 4 and 5, to rate the usefulness of the extended features. In particular, the subjects expressed their strong support for usefulness of the tuple list, in which over 50% of the subjects gave the scale 5. As for the difficulty of using the attribute list and the tuple list, more than 90% of the subjects' responses were spread fairly evenly amongst the scales 1 to 3. This indicates that these two features are basically not difficult for the subjects although they are not overly easy. Relatively speaking, the use of semantic domains in the comparison clause seems to be

Extension	Usefulness					Difficulty				
	1	2	3	4	5	1	2	3	4	5
Attribute list (%)	0.00	4.62	23.08	35.38	36.92	35.38	35.38	23.08	3.08	3.08
Tuple list (%)	0.00	4.62	9.23	30.77	55.38	35.38	27.69	32.31	4.62	0.00
Comparison (%)	3.08	3.08	13.85	44.62	35.38	16.92	29.23	41.54	7.69	4.62

Figure 6.5: Users' attitudes on using the extended features

more difficult than the attribute list and the tuple list, as less subjects (approximately 20% less than other features) chose the scales 1 and 2 for the difficulty of this feature.

As for Question 6, we summarise below the useful comments in three areas.

1. In general, the subjects showed appreciation of the extended features of the OSQL SELECT command. Many of their comments in this question expressed their support for our extension. They agreed that OSQL provided many benefits for them to formulate the survey queries. The tuple list was the most positive feature that the subjects frequently mentioned.

2. As far as the OSQL syntax is concerned, the subjects commented that the OSQL SELECT command was quite easy to use as it syntactically resembled the SQL SELECT statement. However, some pointed out that the use of brackets in semantic orderings such as the attribute list ((NAME *WITHIN* EMP_RANK), SALARY) was best avoided as too many brackets would easily cause typing errors.

3. As far as the implementation of OSQL is concerned, many subjects were dissatisfied with the inadequacy of the error recovery mechanism, the on-line help manual and the system facilities provided by our implementation. For example, the subjects expected more editing commands to be available to correct their queries.

There were also some interesting points raised by a few subjects. Some mentioned that the attribute list had imposed some restrictions on the format of the presentation of the query result. Also, some suggested that in the attribute list, more use of the asterisk symbol "*" to specify other attributes was necessary. An example raised by one of the subjects was that the attribute list (COST *) might mean that the tuples were sorted by the attribute COST in the first place but that the orderings of other attributes would be immaterial.

6.4 Evaluation of OSQL as a Result of the Survey

Assuming that our results are in fact representative, we find that there are many implications obtained from this survey for the extended features of the OSQL SELECT command as follows.

- With respect to the attributes list we should provide further optional facilities in the attribute list to aid the users when using semantic domains. In particular, it would be helpful to provide the users with the asterisk symbol "*" when they want to specify other attributes whose orderings are irrelevant.
- With respect to the tuple list, it has proved to be a powerful mechanism and a user-friendly facility for users to manipulate orderings. We should preserve this good feature in further design.
- With respect to the comparison clause, the semantic orderings are useful features but they may not be so trivial to some inexperienced users. More elaborations should be given for OSQL beginners in order to help them to acquire the use of the concept of semantic orderings.

We admit that there are not enough on-line facilities, such as a good user interface, for the present OSQL prototype. It affects the users' attitudes towards the extended features. Therefore, a more comprehensive implementation of OSQL, which includes more error diagnosis and a custom-built user interface, is needed so that further experiments can be carried out to test the viability of the OSQL extension to SQL. One method is to incorporate the SELECT command into a graphical user interface, for example, instead of typing the name of a semantic domain in the comparison, users may make their choices from a drop down menu in a window environment, which presents a series of available semantic domains. In fact, OSQL can also provide the facility to develop a library of custom-built semantic domains together with their relevant operations via the notion of package. We will discuss in Chapter 7 the full detail of the package facility in OSQL.

We have identified possible biases to our data, which may limit the findings in the survey. Firstly, the samples should not, in general, be restricted to students and computer professionals. People in other organisations may yield different results. However, in our case the students sample is more readily available. It is also convenient for us to manage the process of the survey within the time constraints. Secondly, the students collective viewpoint may be biased by perceiving OSQL as an obstacle to overcome to obtain a good mark for the coursework rather than as a means to assist SQL development. Despite the fact that we have taken some precautions to avoid this happening (recall that the measures were stated in Section 6.2), one still could argue that their opinions are, to some extent, guided by us or by the OSQL manual rather than their own experience. Thirdly, the survey queries adopted in the experiment sheet involve no nestings in OSQL statements. Thus, the full capabilities of OSQL over SQL have not yet been fully illustrated in the survey.

We have three short-term objectives and a long-term goal in order to continue and evolve the survey in future. Firstly, we plan to extend the scope of the survey outside of the current sample groups to include more computer professionals. Secondly, we plan to improve the interface of the OSQL system and the error diagnosis facilities as we mentioned before so that OSQL can be more fairly evaluated. Thirdly, we plan in the next experiment to include queries that involve more complex ordered domains and nested OSQL statements. As a more long-term goal, we are interested to know how much the performance of the survey queries also applies to SQL3, which is estimated to be released in 1998.

Chapter 7

The Development of OSQL Packages for Modelling Advanced Applications

In this chapter we demonstrate that OSQL aided with a package discipline is extremely powerful and has a very wide range of applicability. In particular we demonstrate that OSQL is very useful in managing the five advanced database applications of tree-structured information, temporal information, incomplete information, fuzzy information and spatial information, which are described in more detail in the table given in Figure 7.1.

Package Name	**Brief Description**
OSQL_TREE	Provides support for tree-structured information in ordered databases. For example, finding the common ancestors between two nodes.
OSQL_TIME	Provides support for temporal information in ordered databases. For example, finding the historical information pertaining to a relation for a given year.
OSQL_INCOMP	Provides support for incomplete information in ordered databases. For example, comparing two tuples in order to decide which one contains more information than another.
OSQL_FUZZY	Provides support for fuzzy requirement in ordered databases. For example, finding the most suitable tuples in a relation according to a given fuzzy requirement.
OSQL_SPACE	Provides support for spatial information in ordered databases. For example, finding all spatial objects on the left side of a selected region.

Figure 7.1: Brief description of the OSQL packages

The use of packages is very popular and successful in many existing software systems such as Mathematica [146], PL/SQL in Oracle [52] and most recently in $\LaTeX\,2_\varepsilon$ [66] and Java [8]. Similar to the usage of packages in other systems, OSQL packages, supported by OSQL language constructs, enjoy many of the benefits of using modularisation techniques as a management tool.

A related approach is to use abstract data types to define domains and their associated operations, which can be treated as an integral part of the data type. This approach was initiated by the ADT-Ingres project at Berkeley [118]. There followed many projects to further this line of development, such as Postgres [138], the EXODUS project [26] and the Starburst project [94]. Examples of commercial products which are now available are Illustra's DataBlades and IBM's Database Extenders. Following our analysis of the development of the relational model in Chapter 1, this approach is basically an object-oriented extension of the conventional model, resulting from the strong trend of object-oriented programming in the 80s. For instance, the system RAD [121] provides a language which enhances the relational algebra operations with external programs which allow the database programmer to introduce arbitrary new data types to a relational DBMS. As a result, RAD is *computationally query complete* [28]. However, optimisation of RAD programs would be rather difficult due to the fact that they can only be introduced to the execution engine at run time.

We emphasise that our approach is novel in the sense that we regard partial ordering as a fundamental property of data which is captured explicitly in the ordered relational model. Furthermore, our approach adheres to the principle of upwards compatibility, since OSQL packages are provided as additional utilities to be used rather than replacing any standard features of a relational DBMS. Thus, our approach provides maximum flexibility for users and allows the design of optimisation strategies for the execution engine of a relational DBMS.

We summarise other features of OSQL packages as below:

1. A top-down design approach is adopted for the grouping of related operations in an OSQL package where the semantics of the package follow from a set of core operations specified by the pre-defined requirements of the package.

2. Constraints within an OSQL package can be enforced and supported by a language construct called enforcement and thus operations in an OSQL package can

be controlled in a more coherent manner.

3. OSQL packages can hide the implementation details of the code of their operations. The database administrator has the flexibility to decide whether an operation is public or private.

The remainder of this chapter is organised as follows. In Section 7.1 we present the syntax of an OSQL package. From Section 7.2 to Section 7.6, we define the generic operations arising from tree-structured information, temporal information, incomplete information, fuzzy information and spatial information, respectively, in the form of five OSQL packages shown in Figure 7.1.

7.1 The Syntax of OSQL Packages

In this section we introduce the Package Definition Language (PDL), which defines the building blocks of an OSQL package; the full syntax of the PDL is given in Appendix A.3.

An OSQL package is defined by the following statement.

PACKAGE ⟨ package name ⟩
  ⟨package body⟩
END PACKAGE

where the package body consists of a parameter component, a function component and an enforcement component. These three components are specified by the following six basic PDL *language constructs*:

1. Parameter constructs.

2. Function constructs.

3. OSQL constructs.

4. Program constructs.

5. Enforcement constructs.

The parameter component in an OSQL package is organized as a sequence of *parameter constructs* followed by the keyword *PARAMETER* as follows:

PARAMETER: parameter construct [parameter construct]...

where a parameter construct is of the form *package data type*: *variable names*, declaring the global variables used in the function and enforcement components. For example, VARCHAR, INT and BOOL are package data types representing characters, integers and boolean values, respectively. After each package data type declaration there follows one or more variable names of the package data type. We use the symbol "$" to specify those variable names that are known to the system at compile time.

The function component in a package is organized as a sequence of *function constructs* followed by the keyword *FUNCTION*. A function construct is a block structure which is defined as follows:

```
⟨ function name ⟩ ⟨ input variables ⟩
    ⟨ parameter list ⟩
    DEFINE
    ⟨ function body ⟩
RETURN [⟨ output variables ⟩]
```

where *parameter list* is a sequence of parameter constructs and where the variables are local to the function. The *function body* describes the operation of the function consisting of an *OSQL construct* or a *program construct*. An OSQL construct is simply an OSQL statement such that its variables have been declared either within a function (i.e., local variables) or in the parameter component at the beginning of the package (i.e., global variables). A function in a package returns a list of zero or more values.

As the expressive power of OSQL is limited [112], we enhance OSQL with a *program construct* in OSQL, which is of the form *AS PROG program name*. The program name is the path location and the name of a program, which is written in the C programming language, which allows SQL statements to be embedded in it. This program performs the operation of the function. For example, the program construct "AS PROG \usr\Prog\tree.root" in a function body specifies that the C program *tree.root* found in the directory \usr\Prog\\ implements the function.

The enforcement component in a package is organized as a sequence of *enforcement constructs* followed by the keyword *ENFORCEMENT*. An enforcement construct, which is similar to a function construct, is also a block structure as follows:

⟨ enforcement name ⟩

DEFINE

⟨enforcement body⟩

END

where the body of an enforcement construct is formulated by a program construct which implements some constraints over the functions of an OSQL package. For example an enforcement construct can be implemented to ensure that the identified domain is indeed tree-structured. We reserve the enforcement, ENFORCE_INIT, to be used by the system for the initialization of an OSQL package.

We refer to all functions and enforcements collectively as *operations.* There are two categories of operations, which are also common in some programming languages: one is that of *public* operations, which are available to the users, and another is that of *private* operations, which are only accessible by calling them from other operations within the package that they belong to. We use the keywords *PUB* and *PRI* to label the operations as public and private, respectively. By default, whenever there is no such keyword labelling, we treat an operation as private. The public operations comprise the interface of a package to the database users, whilst the private operations are encapsulated and thus hidden from the users. For example, all enforcements are private because they are used by the system to ensure the integrity of the domain and the consistency of the functions in a package.

Note that there is an important difference between using an OSQL construct and a program construct in a function. The OSQL statement in an OSQL construct can be decomposed and restructured by the query execution engine of a relational DBMS for optimisation purposes. For instance, the query ($Q_{7.1}$) below, which uses the package function COM_ANCESTOR,

($Q_{7.1}$) *SELECT* (∗) (∗) *FROM* EMP_DETAIL
WHERE NAME *IN* COM_ANCESTOR('Nadav', 'Ethan'),

is equivalent to the query ($Q_{7.2}$) below,

($Q_{7.2}$) *SELECT* (∗) (∗) *FROM* EMP_DETAIL
WHERE (NAME > 'Nadav' *WITHIN* EMP_RANK)
AND (NAME > 'Ethan' *WITHIN* EMP_RANK),

which is an ordinary OSQL statement not using any functions. On the other hand, an external program specified in a program construct is "opaque" with respect to a relational DBMS, in the sense that its code can only be integrated into its associated OSQL statement at run time and thus allows no possibility of optimisation at compile time. As a result, operations defined by OSQL constructs are, in general, more efficient to implement than those defined by program constructs.

7.2 Example OSQL Packages I: Tree-Structured Information

OSQL_TREE provides support for queries having tree-structured information such as finding the root and the parent of a node [27]. The terminology concerning trees used in the OSQL package are the usual ones [6]. We restrict the term tree to mean a *rooted tree* and, without loss of generality, consider only the case of having one tree in a tree domain. We also ignore those operations that involve updating nodes and consider only the operations needed for data retrieval.

Definition 7.1 Requirements and Core Operations of OSQL_TREE

r_1: for each *node*, there is an operation c_1 which finds its *parent.*

r_2: for each node, there is an operation c_2 which finds its *children.*

r_3: for each tree, there is an operation c_3 which finds its *root.*

r_4: for each node, there is an operation c_4 which finds all its *leaf nodes.*

r_5: for each node, there is an operation c_5 which finds all its *ancestors.*

r_6: for each node, there is an operation c_6 which finds all the nodes in the *subtree* rooted at the node.

A requirement r_i is said to be *realised* by the *core operation* c_i (or c_i is the *realisation* of r_i), where $i \in \{1, \dots, 6\}$. We call the set of all core operations of OSQL_TREE the *core set* of OSQL_TREE and denote it by $CORE(OSQL\_TREE)$.

Note that it is *not* necessary that for every c_i in $CORE(OSQL\_TREE)$, there exists exactly one corresponding operation in OSQL_TREE such that it has the same effect as c_i; a similar remark also applies to other packages. We now present the description of the operations in OSQL_TREE in the table given in Figure 7.2.

Operations	Brief Description
IDENTIFY function	To IDENTIFY a given domain as the tree domain used in OSQL_TREE.
PARENT function	To find the PARENT of a node within the tree domain used in OSQL_TREE.
CHILDREN function	To find all CHILDREN of a node within the tree domain used in OSQL_TREE.
ANCESTOR function	To find all the nodes prior to a node within the tree domain used in OSQL_TREE.
COM_ANCESTOR function	To find all the COMmon ANCESTORs of two given nodes within the tree domain used in OSQL_TREE.
OFFSPRING function	To find the nodes in the subtree of a given node.
LEAVES function	To find all the leaf nodes of the tree domain used in OSQL_TREE.
ROOT function	To find the ROOT of the tree domain used in OSQL_TREE.
LEVEL function	To find the LEVEL of a node within the tree domain used in OSQL_TREE.
SWAP function	To SWAP two nodes in the tree domain used in OSQL_TREE.
VERIFY function	To VERIFY that the identified tree domain satisfies the semantics of a tree domain.
NODE_COUNT function	To count the number of nodes in a given subset of the tree domain used in OSQL_TREE.
ENFORCE_INIT enforcement	To enforce the initialization which identifies the domain TREE to be used as the tree domain of OSQL_TREE.
ENFORCE_IDENTIFY enforcement	To enforce the verification over the identified domain given by the function IDENTIFY.
ENFORCE_SWAP enforcement	To enforce the verification over the tree domain after performing the function SWAP.

Figure 7.2: The description of the operations in OSQL_TREE

The reader can consult Appendix B for the full reference of the code of the operations. A similar remark also applies to other OSQL packages.

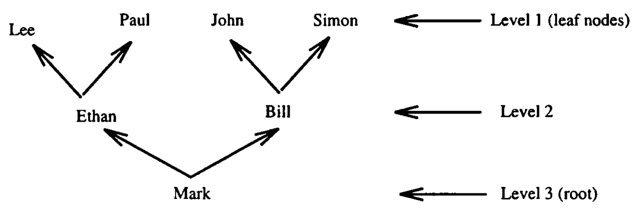
We assume that when OSQL_TREE is loaded into the system, there is a tree domain in the database. To enforce this assumption, the enforcement ENFORCE_INIT will search for the domain that is called TREE as the underlying domain to be used in the OSQL package. However, the users can still use the function IDENTIFY to declare other tree domains for the OSQL package. We use a relation called TREE_LEVEL, whose relational schema consists of two attributes LEVEL_NUMBER and NODE, to maintain the information of node levels in the tree domain. The function LEVEL can be used to access the relation TREE_LEVEL to find out the level of a node. The functions NODE_COUNT and VERIFY are private functions, which are only used by other operations in OSQL_TREE. The function SWAP_NODE is necessary so that we do not have to create a new tree domain in case some changes are required in the ordering of the nodes in the tree domain that is currently used. The declarations pertaining to OSQL_TREE are given in Figure 7.3.

```
PACKAGE OSQL_TREE
PARAMETER:
    VARCHAR: tree_node_1, tree_node_2, ext_domain,
        tree_domain, $ext_relation, $ext_att
    BOOL: bool_val,
    INT: node_level, count_nodes
    REL: nodes
FUNCTION:
    PUB  IDENTIFY(ext_domain) RETURN
    PUB  PARENT(tree_node_1) RETURN nodes
    PUB  CHILDREN(tree_node_1) RETURN nodes
    PUB  ANCESTOR(tree_node_1) RETURN nodes
    PUB  COM_ANCESTOR(tree_node_1, tree_node_2) RETURN nodes
    PUB  OFFSPRING(tree_node_1) RETURN nodes
    PUB  LEAVES() RETURN nodes
    PUB  ROOT() RETURN nodes
    PUB  LEVEL(tree_node_1) RETURN node_level
    PUB  SWAP(tree_node_1, tree_node_2) RETURN
    VERIFY(tree_domain) RETURN bool_val
    NODE_COUNT(nodes) RETURN count_nodes
ENFORCEMENT:
    ENFORCE_INIT()
    ENFORCE_IDENTIFY()
    ENFORCE_SWAP()
END PACKAGE
```

Figure 7.3: The package declaration for OSQL_TREE

Example 7.1 In this example, we present some typical tree-structured information queries on the relation EMP_TREE shown in Figure 7.4(a) in order to demonstrate how to apply the package operations within OSQL statements. The domain of employee names is depicted as a tree in Figure 7.4(b).

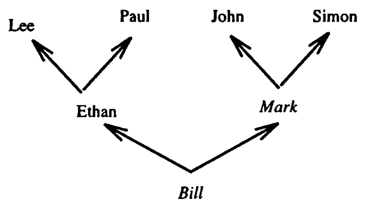
NAME	SALARY
Bill	12K
Ethan	29K
John	14K
Lee	25K
Mark	30K
Paul	23K
Simon	10K



(a) (b)

Figure 7.4: An employee relation EMP_TREE and the tree domain

1. The function IDENTIFY(EMP_RANK) will identify EMP_RANK to be the tree domain concerned instead of the default domain TREE; NODE_COUNT is a private function for internal use, for example NODE_COUNT(CHILD('Ethan')) = 2; the function SWAP('Mark', 'Bill') changes the ordering of the tree in Figure 7.4(b) as follows:



2. Find the most senior staff member.

   ($Q_{7.3}$) *SELECT* (∗) (∗) *FROM* EMP_TREE *WHERE* NAME *IN* ROOT().

3. Find the name and salary of the immediate boss of Bill.

   ($Q_{7.4}$) *SELECT* (∗) (∗) *FROM* EMP_TREE *WHERE* NAME *IN* PARENT('Bill').

4. Find the name and salary of the immediate subordinates of Bill.

   ($Q_{7.5}$) *SELECT* (∗) (∗) *FROM* EMP_TREE *WHERE* NAME *IN* CHILDREN('Bill').

5. Find the name and salary of the common bosses of David and Bill.

   ($Q_{7.6}$) *SELECT* (∗) (∗) *FROM* EMP_TREE *WHERE* NAME *IN*

COM_ANCESTOR('Bill','David').

6. Find the name and salary of all the subordinates of Bill.
   ($Q_{7.7}$) *SELECT* (*) (*) *FROM* EMP_TREE *WHERE* NAME *IN* OFFSPRING('Bill').

7. Find the name and salary of all the bosses of Bill.
   ($Q_{7.8}$) *SELECT* (*) (*) *FROM* EMP_TREE *WHERE* NAME *IN* ANCESTOR('Bill').

8. Find the name of the staff members who are in the same level as Bill.
   ($Q_{7.9}$) *SELECT* ((NAME WITHIN EMP_RANK)) (LEVEL('Bill')) *FROM* EMP_TREE.

7.3 Example OSQL Packages II: Temporal Information

The underlying semantics of time used in this OSQL package is that time is considered to be linearly ordered [97]. In our implementation an ordered relation is employed to maintain the data elements of a time domain, which are non-empty, finite, linearly ordered, and of the same data type. This relation can only be accessed by the operations of the package and the comparison of temporal data can be applied only over the time domain.

One of the many approaches [145] in the literature to manipulating temporal data is to use an attribute, which we call a *time attribute*, and to *timestamp* the attribute values of this attribute with either *time instants* or *time intervals* [144, 98]. We assume temporal data is timestamped with the time interval during which it is valid.

Let us consider the relation EMP_TIME in Figure 7.7, which uses the attributes FROM_TIME and TO_TIME to denote time intervals. We can see that, for instance, Mark had salary 20K in the time interval $1992 \leq \text{YEAR} < 1995$ (note that in our formalism the year 1995 is not included in the time interval).

The advantage of using time intervals in modelling time data is that it can save storage space. However, there are some complications arising from using time intervals in modelling time data. For example, they cannot directly support the update or retrieval of tuples at a particular time instant and some useful operations such as the *snapshot* operation obtaining the temporal relation in a particular year, cannot be carried out in a direct manner. To solve this problem, two operations *EXTEND* and *COALESCE* have been suggested in the literature [145]. It can be shown that these two operations can be formulated in OSQL, with the assumption that an ordered relation is maintained for the time domain used in OSQL_TIME. Therefore, in this sense, we can claim that the expressive power of OSQL_TIME is *temporally complete* (see Chapter 4 in [145]). We now introduce the following design requirements for OSQL_TIME, in which we use the terminology of [145].

Definition 7.2 Requirements and Core Operations of OSQL_TIME

r_1: for a given temporal relation, there is an operation c_1 which returns the snapshot relation for a given time instant of the current time domain.

r_2: there is an operation c_2 which provides a standard time domain to model the Gregorian calendar system (i.e. DAY-MONTH-YEAR).

r_3: there is an operation c_3 which allows users to define the time resolution of a certain granularity up to the unit time interval.

Similar to Definition 7.1, we call the set that consists of all the core operations of OSQL_TIME the core set of OSQL_TIME and denote it by *CORE(OSQL_TIME)*.

Note that we have not required that the set of core operations contain some of the common temporal operators [135], such as *overlaps* and *contains* (see Chapters 4, 5 and 6 in [145]), which can be explicitly defined in order to compare time intervals, since they can be quite easily formulated in OSQL comparison predicates. For instance, given two time intervals l_1 and l_2 specified by $\langle from_1, to_1\rangle$ and $\langle from_2, to_2\rangle$ respectively, l_1 *overlaps* with l_2 can be written as the predicate $((to_1 > from_2\ \mathit{WITHIN\ time\_domain})\ AND\ (to_2 > from_1\ \mathit{WITHIN\ time\_domain}))$ and l_2 *contains* l_1 can be written as the predicate $((from_1 > from_2\ \mathit{WITHIN\ time\_domain})\ AND\ (to_2 > to_1\ \mathit{WITHIN\ time\_domain}))$. Nevertheless, one can still argue that it would be useful to have the mentioned operations in the OSQL package, but that is another matter. We now present the following description of the operations in OSQL_TIME in the table given in Figure 7.5.

We assume that DATE (i.e. DAY-MONTH-YEAR) is the default domain to be used in the package unless the function IDENTIFY is used to specify another time domain.

Operations	**Brief Description**
IDENTIFY function	To IDENTIFY a given domain as the time domain used in OSQL_TIME.
CURRENT function	To return all the CURRENT tuples in a temporal relation.
HISTORY function	To return all tuples which are not valid at present.
SNAPSHOT function	To return all tuples which were valid at a given time instant.
SUCC function	To return the SUCCessor of a given time instant in the time domain used in OSQL_TIME.
PRED function	To return the PREDecessor of a given time instant in the time domain used in OSQL_TIME.
DURA function	To calculate the DURAtion between two time instants in the time domain used in OSQL_TIME.
EXPAND function	To convert interval-stamped tuples in a given relation into instant-stamped tuples.
COALESCE function	To convert instant-stamped tuples in a given relation into interval-stamped tuples, i.e. the reverse of the EXPAND function.
TIME_RES function	To create a time domain whose time scale is defined by the users.
VERIFY function	To VERIFY that the identified time domain satisfies the semantics of a time domain.
STRIP_TIME function	To project out the time attributes FROM_TIME and TO_TIME from the relational schema for a given relation and return the remaining attributes.
ENFORCE_INIT enforcement	To enforce the initialization which identifies the domain DATE to be used as the time domain of OSQL_TIME.
ENFORCE_IDENTIFY enforcement	To enforce the verification over the identified domain given by the function IDENTIFY.

Figure 7.5: The description of the operations in OSQL_TIME

Other standard domains available in OSQL_TIME include YEAR, MONTH, DAY, HOUR, MINUTE, SECOND. Furthermore, a user-defined time domain of an arbitrary resolution can be defined by the function TIME_RES. We use a relation called TIME_DOM_REL, whose relational schema consists of the attribute TIME_DATA, to maintain the standard time domains. If the time domain is user-defined, the package will prompt the user for the definition of the NOW variable. The function STRIP_TIME can be used to remove the time attributes of the schema of a temporal relation. The EXPAND function would be useful if users want to update a temporal relation. If we want to add a tuple into the relation EMP_TIME, then we have to first EXPAND the relation and then COALESCE the updated relation. We now show the declaration part of OSQL_TIME in Figure 7.6.

```
PACKAGE OSQL_TIME
PARAMETER:
    VARCHAR: time_domain, ext_relation, time_instant_1, time_instant_2, NOW
        non_time_schema, ext_domain
    INT: granularity, duration
    BOOL: bool_val
    REL: result_relation
FUNCTION:
    PUB  IDENTIFY(ext_domain)
    PUB  CURRENT(ext_relation) RETURN result_relation
    PUB  HISTORY(ext_relation) RETURN result_relation
    PUB  COALESCE(ext_relation) RETURN result_relation
    PUB  SUCC(time_instant_1) RETURN time_instant_2
    PUB  PRED(time_instant_1) RETURN time_instant_2
    PUB  DURA(time_instant_1, time_instant_2) RETURN duration
    PUB  SNAPSHOT(ext_relation, time_instant_1) RETURN result_relation
    PUB  EXPAND(ext_relation) RETURN result_relation
    PUB  TIME_RES(granularity, ext_domain) RETURN
    VERIFY(time_domain) RETURN bool_val
    STRIP_TIME(ext_relation) RETURN non_time_schema
ENFORCEMENT:
    ENFORCE_INIT()
    ENFORCE_IDENTIFY()
END PACKAGE
```

Figure 7.6: The package declaration for OSQL_TIME

Example 7.2 We use the relation EMP_TIME shown in Figure 7.7 whenever it is necessary.

1. IDENTIFY(YEAR) identifies the standard domain YEAR, which specifies the ordered set $\{1900 < \cdots < 2050\}$ and IDENTIFY(MONTH) identifies another standard domain $\{JAN < \cdots < DEC\}$. If the user has used the function TIME_RES(100, HUNDRED) to create a domain HUNDRED, then IDENTIFY (HUNDRED) identifies this user-defined domain, which specifies the ordered set $\{0 < \cdots < 99\}$.

2. Find the current salaries of all employees.

   ($Q_{7.10}$) *SELECT* (NAME, SALARY) (*) *FROM* CURRENT(EMP_TIME).

3. Find the salary history of Mark.

   ($Q_{7.11}$) *SELECT* (*) (*) *FROM* HISTORY(EMP_TIME).

   WHERE NAME = 'Mark'

4. Find the salary of Bill in 1994.

   ($Q_{7.12}$) *SELECT* (SALARY) (*) *FROM* SNAPSHOT(EMP_TIME, 1994)

   WHERE NAME = 'Bill'.

5. Find the names of those employees who have been worked for more than two years.

   ($Q_{7.13}$) *SELECT* (NAME) (*) *FROM* EMP_TIME

   WHERE DURA(FROM_TIME, TO_TIME) > 2.

NAME	SALARY	FROM_TIME	TO_TIME
Bill	15K	1991	1995
Bill	18K	1995	1996
Bill	20K	1996	1997
Mark	25K	1992	1995
Mark	30K	1995	1997

Figure 7.7: An employee relation EMP_TIME stamping with time intervals

7.4 Example OSQL Packages III: Incomplete Information

In this OSQL package, we classify the incompleteness into three unmarked *null symbols*, UNK, DNE and NI, whose semantics has already been discussed in Subsection 5.4.2. Recall that we use the notion of *more informative* values, which allows us to deduce useful information available from a relation having incomplete data.

The ordering of null values is captured by the standard incomplete domain called INCOMP provided by OSQL_INCOMP. Recall that the domain can be formulated by the OSQL statement in ($Q_{5.18}$) in Chapter 5. As we would like to make the domain INCOMP standard, we do not allow any user-defined incomplete domains in OSQL_INCOMP. The description of the operations is shown in the table in Figure 7.8 and the declaration part of OSQL_INCOMP is shown in Figure 7.9.

Definition 7.3 Requirements and Core Operations of OSQL_INCOMP

r_1: there is an operation c_1 which defines the standard domain describing the semantics of incompleteness such as the null values as shown in Figure 5.11 in Chapter 5.

r_2: for a given incomplete relation, there is an operation c_2 which returns all the tuples containing only known values.

r_3: for a given incomplete relation, there is an operation c_3 which returns tuples containing various degree of incompleteness.

r_4: there is an operation c_4 which checks whether one tuple is more informative than another with respect to some attributes.

Similar to Definition 7.1, we call the set that consists of all core operations of OSQL_INCOMP the core set of OSQL_INCOMP and denote it by *CORE(OSQL_INCOMP).*

Operations	**Brief Description**
COMPLETE_VAL function	To return all tuples which contain only known values of an attribute in an incomplete relation.
PARTIAL_VAL function	To return all tuples which contain a null value of an attribute in an incomplete relation.
DNE_VAL function	To return all tuples which contain the DNE value of an attribute in an incomplete relation.
NI_VAL function	To return all tuples which contain the NI value of an attribute in an incomplete relation.
UNK_VAL function	To return all tuples which contain the UNK value of an attribute in an incomplete relation.
MORE_INFO function	To check whether tuples are more informative than a given attribute value.
LESS_INFO function	To check whether tuples are less informative than a given attribute value.
IDENTIFY function	To IDENTIFY the domain INCOMP as the incomplete domain used in OSQL_INCOMP.
VERIFY function	To VERIFY that the domain INCOMP satisfies the semantics of an incomplete domain.
ENFORCE_INIT enforcement	To enforce the initialization which identifies the domain INCOMP as the incomplete domain used in the package.

Figure 7.8: The description of the operations in OSQL_INCOMP

Note that the function IDENTIFY in this OSQL package is private, since the users are not allowed to change the meaning of various null symbols. This shows that the package approach is very flexible in modelling versatile information. The functions COMPLETE_VAL, PARTIAL_VAL, DNE_VAL, NI_VAL and UNK_VAL provide users with the ability to manipulate various types of incomplete information based on the notion of being "more informative". The functions MORE_INFO and LESS_INFO provide users with the ability to semantically compare tuples in incomplete databases.

```
PACKAGE OSQL_INCOMP
PARAMETER:
   VARCHAR: ext_att, incomplete_domain, ext_relation, ext_val, predicate
   BOOL: bool_val
   REL: result_relation
   PUB  COMPLETE_VAL(ext_relation, ext_att) RETURN result_relation
   PUB  PARTIAL_VAL(ext_relation, ext_att) RETURN result_relation
   PUB  DNE_VAL(ext_relation, ext_att) RETURN result_relation
   PUB  NI_VAL(ext_relation, ext_att) RETURN result_relation
   PUB  UNK_VAL(ext_relation, ext_att) RETURN result_relation
   PUB  MORE_INFO(ext_att,ext_val) RETURN predicate
   PUB  LESS_INFO(ext_att,ext_val) RETURN predicate
   IDENTIFY() RETURN
   VERIFY(incomplete_domain) RETURN bool_val
ENFORCEMENT:
   ENFORCE_INIT()
END PACKAGE
```

Figure 7.9: The package declaration for OSQL_INCOMP

Example 7.3 We use the relation EMP_INCOMP in Figure 7.10 whenever it is necessary.

NAME	PREVIOUS_WORK
Mark	UNK
Ethan	DNE
Nadav	administrator
Bill	programmer
John	NI
Simon	NI

Figure 7.10: An employee relation EMP_INCOMP

1. Find the name and previous work of those employees whose previous work is less informative than unknown (i.e., UNK).

   ($Q_{7.14}$) *SELECT* (NAME, PREVIOUS_WORK) (∗) *FROM* EMP_INCOMP *WHERE* LESS_INFO(PREVIOUS_WORK, 'UNK').

2. Find the name and previous work of those employees whose information of previous work is not complete.
   ($Q_{7.15}$) *SELECT* (NAME, PREVIOUS_WORK) (*) *FROM*
   PARTIAL_VAL(EMP_INCOMP, PREVIOUS_WORK).

3. Find the name and previous work of those employees whose previous work does not exist (i.e., DNE).
   ($Q_{7.16}$) *SELECT* (NAME, PREVIOUS_WORK) (*) *FROM*
   DNE_VAL(EMP_INCOMP, PREVIOUS_WORK).

7.5 Example OSQL Package IV: Fuzzy Information

In OSQL_FUZZY we provide functions for users to impose fuzzy requirements on a relation. Users can obtain the most suitable information based on the defined requirements in the OSQL package. We assume that for each fuzzy requirement, there is a domain called fuzzy domain, which captures the semantics of the requirement, for example as we have shown in ($Q_{5.20}$) given in Chapter 5, that the fuzzy requirement "good science background and academic qualification" can be captured by the fuzzy domain QUALIFY. Therefore, the requirement can be referred to by the name of its corresponding fuzzy domain. If there are several fuzzy requirements to be imposed on a relation, then their priorities can be defined by the function ORDER_FUZZY and tuples can be ordered and then retrieved according to the priorities of fuzzy requirements.

Definition 7.4 Requirements and Core Operations of OSQL_FUZZY

r_1: for each fuzzy requirement, there is an operation c_1 which identifies a unique fuzzy domain associated with it.

r_2: there is an operation c_2 which specifies the relative priorities of different requirements.

r_3: there is an operation c_3 which retrieves tuples in a sorted list, in which the most suitable one is the first, from a relation according to a set of fuzzy requirements.

Similar to Definition 7.1, we call the set that consists of all core operations of OSQL_FUZZY the core set of OSQL_FUZZY and denote it by *CORE(OSQL_FUZZY)*.

We now present the description of the operations in the table in Figure 7.11.

Operations	**Brief Description**
IDENTIFY function	To IDENTIFY a fuzzy domain to be used to capture the semantic of a fuzzy requirement.
IMPOSE_FUZZY function	To IMPOSE a FUZZY requirement on an attribute.
ORDER_FUZZY function	To order the relative priorities of a set of fuzzy requirements which are currently used in OSQL_FUZZY.
LIST_REQ function	To list all the fuzzy requirements used in OSQL_FUZZY.
VERIFY function	To verify that the given domain satisfies the semantics of a fuzzy domain.
ENFORCE_INIT enforcement	To enforce the initialization which prepares an empty relation called FUZZY_DICT to maintain the fuzzy requirements.
ENFORCE_IDENTIFY enforcement	To enforce the verification over the identified fuzzy domain given by the function IDENTIFY.
ENFORCE_IMPOSE enforcement	To enforce the priorities of the identified fuzzy requirements.

Figure 7.11: The description of the operations in OSQL_FUZZY

The priorities of a set of fuzzy requirements are system defined (system ordered) if they are not specified. The function ORDER_FUZZY can be used to arrange the priorities of requirements. There is a parameter called order, which is a natural number describing the relative priority of the requirement defined in the second parameter fuzzy_domain. The information about the priorities is maintained by the relation called FUZZY_DICT, whose relational schema consists of the attributes FUZZY_REQ and PRIORITY, containing all the name information of the fuzzy requirements and their priorities. The users can use the function LIST_REQ, which returns the relation FUZZY_DICT, to check for the priorities of all fuzzy requirements. The declaration part of OSQL_FUZZY is shown in Figure 7.12.

Example 7.4 Let us consider the relation EMP_FUZZY in Figure 7.13 whenever it is necessary, and suppose that there is a project which requires an employee with a good science background in his/her academic qualification and strong connections in the research community. We use two fuzzy domains called QUALIFY and CONNECT to capture these semantics of the requirements. The fuzzy domain QUALIFY has been formulated in $(Q_{5.20})$ in Chapter 5 and the fuzzy domain CONNECT is given as the statement $(Q_{7.17})$ below.

```
PACKAGE OSQL_FUZZY
PARAMETER:
    VARCHAR: fuzzy_domain, ext_att, predicate
    INT: order
    BOOL: bool_val
    REL: result_relation
FUNCTION:
    PUB  IDENTIFY(fuzzy_domain) RETURN
    PUB  IMPOSE_FUZZY(ext_att, fuzzy_domain) RETURN predicate
    PUB  ORDER_FUZZY(fuzzy_domain, order) RETURN
    PUB  LIST_REQ() RETURN result_relation
    VERIFY(fuzzy_domain) it RETURN bool_val
ENFORCEMENT:
    ENFORCE_INIT()
    ENFORCE_IDENTIFY()
    ENFORCE_IMPOSE()
END PACKAGE
```

Figure 7.12: The package declaration for OSQL_FUZZY

NAME	EDUCATION
Bill	MSc
Ethan	MSc
John	BSc
Mark	PhD
Nadav	MBA
Simon	A-Level

Figure 7.13: An employee relation EMP_FUZZY

1. ($Q_{7.17}$) *CREATE DOMAIN* CONNECT *CHAR*(10)
   ORDER AS (*OTHER* < 'Mark' < 'Ethan').

2. Find the names of those employees with good science background in academic qualification and strong connection in the research community.
   ($Q_{7.18}$) *SELECT* (IMPOSE_FUZZY(NAME, CONNECT), IMPOSE_FUZZY (EDUCATION, QUALIFY) (1) *FROM* EMP_FUZZY.
   The employee Mark will be returned for this query.

3. We now use the function ORDER_FUZZY(CONNECT, 1) and ORDER_FUZZY(QUALIFY, 2) to change the priorities of the requirements, i.e., the requirement CONNECT should be considered first and then QUALIFY the second.

Then the employee Ethan will be returned for the query ($Q_{7.18}$).

4. Finally the fuzzy requirements can be listed as below by the function LIST_REQ() given in Figure 7.14.

FUZZY_REQ	PRIORITY
CONNECT	1
QUALIFY	2

Figure 7.14: A relation returned by LIST_REQ()

7.6 Example OSQL Package V: Spatial Information

In the past few years, applications that involve in computing geometric and pictorial objects are easier to implement due to the progress in processing capabilities and the computation of bitmap graphics. As a result, there are increasing demands in handling spatial data in many applications such as sophisticated user-interfaces, Computer Assisted Design (CAD), image processing, Geographical Information Systems (GIS) and the usages of pattern recognition in the areas of medicine, cartography and robot control.

Spatial data domains share many common features with time domains. Firstly, the underlying semantics of each dimension in space is considered to be linearly ordered. Secondly, the eight *Egenhofer-Franzosa topological relationships* (or simply EF-relationships) [49], of spatial data: *disjoint*, *meet*, *overlap*, *covers*, *covered by*, *inside*, *contains* and *equal*, [48, 122] are inherent to the generic operations on spatial data. Note that EF-relationships are also applicable in temporal data [145] (recall the operations overlap and contains in Section 7.2). Thirdly, spatial data needs a high level of abstraction to capture its semantics. For example, in temporal information we need a time interval to timestamp an event and in spatial information we need a rectangular region to model a room in the floor plane of a building. It has already been pointed out by [153] that in practice, many queries over GIS are related to both temporal and spatial concepts. Let us consider a simplified GIS relation in Example 7.5.

Example 7.5 The relation DESERT shown below records the snapshot of remotely sensed data captured by a satellite for a desert at a particular place and time. The

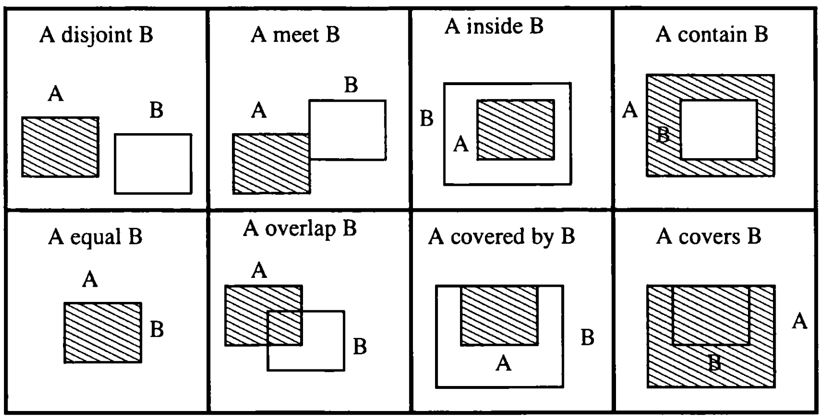


Figure 7.15: The eight binary topological relationships between rectangular regions

attribute LOCATION refers to a specific flight path of the satellite. The attribute FROM_TIME represents the receiving date when the data was captured by the remote sensing device. The accepting or rejecting of the CLASSIFICATION is determined by the image quality of satellite's photo.

LOCATION	FROM_TIME	TO_TIME	CLASSIFICATION	DESERT_AREA
001-007	9-1-92	1-12-93	Accepted	1000
001-007	1-12-93	23-1-94	Rejected	UNK
001-007	23-1-94	30-4-96	Rejected	UNK
001-007	30-4-96	30-4-97	Accepted	1100
001-007	30-4-97	NOW	Accepted	1150

Figure 7.16: A GIS relation DESERT to analyse desertification

The following queries over DESERT are typical. They are all easily formulated by making use of the package OSQL_TIME we presented in Section 7.2.

1. How big is the desert area in the region 001-007 now?

   ($Q_{7.19}$) *SELECT* (DESERT_AREA) (∗) *FROM* CURRENT(DESERT).

2. What are the failed classification (i.e., rejected) histories of 001-007?

   ($Q_{7.20}$) *SELECT* (FROM_TIME, TO_TIME) (∗) *FROM* HISTORY(DESERT).
   WHERE CLASSIFICATION = 'Rejected'

3. Find the desert growth information from 9-1-92 to 30-4-96.

($Q_{7.21}$) *SELECT* (FROM_TIME, TO_TIME, DESERT_AREA) (*)
FROM COALESCE(DESERT) *WHERE* FROM_TIME <= '9-1-92'
AND TO_TIME >= '30-4-96'.

The above queries show that temporal dimension plays an important role in a spatial information system. However, spatial data is much more complicated than temporal data in the following aspects.

1. Spatial data can be zero (e.g., a point), one (e.g., a line), two (e.g., an area) or three (e.g., a volume) dimensional and thus their interactions become very complex. In contrast, temporal data is just one dimensional if we consider time intervals as the timestamps for the data.

2. Within a fixed dimensional space, spatial data has lots of primitive regions, which require different parameters to characterise them. For example, in the special case of two dimensional spaces, we may have rectangular regions (being characterised by the four vertices), circular regions (being characterised by the centre and the radius) or algebraic curve regions (being characterised by the polynomial function) [122].

3. In order to manipulate spatial data more easily in practice, we normally employ a graphical environment to represent the data. The coupling of the language for spatial data queries and the language for the display control has been strongly advocated by many spatial database researchers (a good summary can be found in Table 1 of [50]). Therefore, it is vital to consider some appropriate graphical display facilitates, which should be included in any spatial extension of SQL.

In our opinion, the research on the issues of the expressiveness that a spatial query should possess and the essential operations that a spatial information system requires are still not resolved [124, 125]. Moreover, a definitive formalism of the semantics for spatial data representation is not yet available. Therefore, we only demonstrate some interesting operations that OSQL_SPACE can provide for handling two dimensional spatial objects. By no means does this package provide a comprehensive coverage of all the generic operations that are required in manipulating spatial information.

Definition 7.5 Requirements and Core Operations of OSQL_SPACE

r_1: for each EF relationship, there is an operation c_1 which specifies the relationship between spatial objects.

r_2: for each dimension in space, there is an operation c_2 which specifies the relative spatial order between spatial objects.

r_3: there is an operation c_3 which returns the area of a two dimensional spatial object.

r_4: there is an operation to represent a spatial object graphically by using its spatial attributes.

r_5: there is an operation to convert a selected object in a display interface into its spatial attributes.

Similar to Definition 7.1, we call the set that consists of all core operations of OSQL_SPACE the core set of OSQL_SPACE and denote it by *CORE(OSQL_SPACE)*.

The focus of this package will be exclusively on the operations that manipulate *rectangular regions* being parallel to X and Y axes of a two dimensional coordinates plan. We choose such a kind of primitive region because it is a very typical two dimensional spatial object. One example is the map of a city showing all buildings and roads. Moreover, in spatial database systems it is quite common to use multiple rectangles to approximate real spatial data [120, 60]. We need only two spatial attributes *MIN_VERTEX* and *MAX_VERTEX* to specify a rectangular region. These two attributes describe respectively the left lowest and right highest vertices of a rectangle. The advantage of using two vertices instead of using four vertices in modelling a rectangle is that it saves storage space. Let us illustrate this point with the diagram of the coordinates plane shown in Figure 7.17.

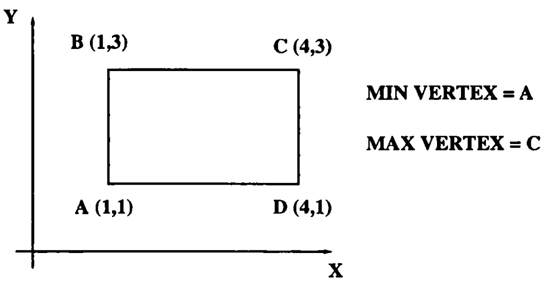


Figure 7.17: Using two vertices to specify a rectangular region

The rectangle on this plane has four vertices {A:(1,1), B:(1,3), C:(4,3), D:(4,1)}, which can be specified by MIN_VERTEX = (1,1) (i.e., point A) and MAX_VERTEX = (4,3) (i.e., point C).

We now present the description of the operations in OSQL_SPACE in the table given in Figure 7.18. Broadly speaking, there are four categories of operations in this package, which are classified according to the following concepts.

1. Topological relationships describe the different cases of intersections between two rectangular regions in space. They are DISJOINT, OVERLAP, MEET and CONTAIN, which represent the corresponding EF-relationships. Note that we have not required that OSQL_SPACE contain operations for the other EF-relationships *equal*, *inside* and *covers*, *covered_by*, since they can be quite easily formulated by using CONTAIN and AREA. For instance, the relationship equal between two rectangular regions $rect_1$ and $rect_2$ can be stated as $rect_1$ CONTAINs $rect_2$ and the AREA of $rect_1$ is equal to that of $rect_2$.

2. Orientation relationships describe the relative spatial order between rectangular regions in space. They are WEST, EAST, SOUTH and NORTH.

3. Display facilities provide the necessary operations to control the graphical environment. They are SET_DISPLAY, PICK_REGION, BOUNDARY and WHOLE.

4. Arithmetic parameters involve the measurement of regions such as AREA and PERI. Note that we may need a different set of arithmetic parameters for other primitive regions. For example, we may include some operations to calculate a sector area or an arc length in the case of circular regions.

We assume that the coordinates on a two dimensional space are captured by the standard domain 2DSPACE. In other words, 2DSPACE is a set of coordinates of points which are specified in the usual format of "(X-coordinate, Y-coordinate)", where the X-coordinate and the Y-coordinate are elements of an integer domain with the numerical ordering. To simplify the presentation of OSQL_SPACE, we do not consider three dimensional spatial domains. However, some operations in OSQL_SPACE can be generalised

Operations	**Brief Description**
IDENTIFY function	To IDENTIFY a given domain as the spatial domain used in OSQL_SPACE.
DISJOINT function	To check whether two rectangular regions satisfy the topological relationship DISJOINT.
OVERLAP function	To check whether two rectangular regions satisfy the topological relationship OVERLAP.
MEET function	To check whether two rectangular regions satisfy the topological relationship MEET.
CONTAIN function	To check whether two rectangular regions satisfy the topological relationship CONTAIN according to the convention that the first region in the parameter list of the function contains the second one.
EAST function	To return all tuples whose regions are on the EAST side of a given rectangular region.
WEST function	To return all tuples whose regions are on the WEST side of a given rectangular region.
SOUTH function	To return all tuples whose regions are on the SOUTH side of a given rectangular region.
NORTH function	To return all tuples whose regions are on the NORTH side of a given rectangular region.
PERI function	To calculate the PERIMETER of a given rectangular region.
AREA function	To calculate the AREA of a given rectangular region.
PICK_REGION function	To convert a REGION on the screen PICKed by a mouse to the corresponding spatial attributes MIN_VERTEX and MAX_VERTEX.
SET_DISPLAY function	To set the specification of a display environment to handle the spatial attributes of an object.
BOUNDARY function	To outline the boundary of all given rectangular regions specified by a spatial attribute.
WHOLE function	To display the whole region of all given rectangular regions specified by a spatial attribute.
XCOMP function	To extract the X-components of a given spatial attribute.
YCOMP function	To extract the Y-components of a given spatial attribute.
VERIFY function	To VERIFY that the identified spatial domain satisfies the semantics of a spatial domain.
ENFORCE_INIT enforcement	To enforce the initialization which identifies the domain 2DSPACE to be used as the space domain of OSQL_SPACE.
ENFORCE_DISPLAY enforcement	To activate the display environment for representing spatial data.

Figure 7.18: The description of the operations in OSQL_SPACE

or extended to the case of three dimensional space. For example, all the topological relationships can be generalised by incorporating one more coordinate (i.e., Z-coordinate) in their corresponding operations. The orientation relationships can be extended by adding the operations ABOVE/BELOW to handle the orientation in the extra dimension. We do not consider a user-defined arbitrary resolution in space domain because it relates to some technical details of the display environment such as the resolution power of the display. We now show the declaration part of OSQL_SPACE in Figure 7.19.

```
PACKAGE OSQL_SPACE
PARAMETER:
   VARCHAR: space_domain, ext_relation, region_parameter, x_comp, y_comp,
      min_vertex_1, max_vertex_1, min_vertex_2, max_vertex_2,
      ext_domain, display_predicate, topological_predicate
   INT: perimeter, area
   BOOL: bool_val
   REL: result_relation
FUNCTION:
   PUB  IDENTIFY(ext_domain) RETURN
   PUB  DISJOINT(min_vertex_1, max_vertex_1, min_vertex_2, max_vertex_2)
           RETURN topological_predicate
   PUB  OVERLAP(min_vertex_1, max_vertex_1, min_vertex_2, max_vertex_2)
           RETURN topological_predicate
   PUB  MEET(min_vertex_1, max_vertex_1, min_vertex_2, max_vertex_2)
           RETURN topological_predicate
   PUB  CONTAIN(min_vertex_1, max_vertex_1, min_vertex_2, max_vertex_2)
           RETURN topological_predicate
   PUB  EAST(ext_relation, min_vertex_1, max_vertex_1) RETURN result_relation
   PUB  WEST(ext_relation, min_vertex_1, max_vertex_1) RETURN result_relation
   PUB  NORTH(ext_relation, min_vertex_1, max_vertex_1) RETURN result_relation
   PUB  SOUTH(ext_relation, min_vertex_1, max_vertex_1) RETURN result_relation
   PUB  PERI(min_vertex_1, max_vertex_1) RETURN perimeter
   PUB  AREA(min_vertex_1, max_vertex_1) RETURN area
   PUB  PICK_REGION() RETURN region_parameter
   PUB  SET_DISPLAY(colour, pattern, mode) RETURN
   PUB  BOUNDARY(ext_relation) RETURN
   PUB  WHOLE(ext_relation) RETURN
   XCOMP(min_vertex_1) RETURN x_comp
   YCOMP(min_vertex_1) RETURN y_comp
   VERIFY(space_domain) RETURN bool_val
ENFORCEMENT:
   ENFORCE_INIT()
   ENFORCE_DISPLAY()
END PACKAGE
```

Figure 7.19: The package declaration for OSQL_SPACE

Example 7.6 We use the spatial relation FLOOR_PLAN shown in Figure 7.20 whenever it is necessary. The queries $(Q_{7.22})$, $(Q_{7.23})$, $(Q_{7.24})$ and $(Q_{7.27})$ are about spatial properties only. Other queries combine spatial and non-spatial properties.

PURPOSE	OCCUPANT	MIN_VERTEX	MAX_VERTEX
Staff room	Bill	(14,6)	(17,9)
Staff room	Lee	(17,6)	(19,9)
Staff room	Ethan	(19,6)	(22,9)
Staff room	Mark	(1,0)	(3,3)
Seminar room	DNE	(0,6)	(12,9)
Lift	DNE	(12,8)	(13,9)
Staircase A	DNE	(13,6)	(14,9)
Staircase B	DNE	(0,0)	(1,3)
Lecture room	DNE	(3,0)	(16,3)
Phd lab	DNE	(16,0)	(22,3)
Printing room	DNE	(18,0)	(22,3)

Figure 7.20: A spatial relation FLOOR_PLAN

1. ($Q_{7.22}$) Show the floor plan described by the relation FLOOR_PLAN.

   WHOLE(*SELECT* (MIN_VERTEX, MAX_VERTEX) (∗) *FROM* FLOOR_PLAN.

   The result is the desired map shown in Figure 7.21. Further features are also shown on the map due to the effect of the queries ($Q_{7.23}$ and $Q_{7.27}$).

2. Highlight the location of Bill's office with a shaded pattern.

   ($Q_{7.23}$) SET_DISPLAY(default, default, shaded).

   WHOLE(*SELECT* (MIN_VERTEX, MAX_VERTEX) (∗) *FROM* FLOOR_PLAN *WHERE* OCCUPANT = 'Bill').

3. What are the purposes of the space on the opposite side to Bill's office?

   ($Q_{7.24}$) *SELECT* (PURPOSE) (∗) *FROM* SOUTH(FLOOR_PLAN, PICK_DISPLAY()).

   Then the purpose "Lecture room" will be returned for this query.

4. Show if there is a lift on the left side of Bill's office?

   ($Q_{7.25}$) *SELECT*(MIN_VERTEX, MAX_VERTEX) (∗) *FROM* EAST(FLOOR_PLAN, MIN_VERTEX, MAX_VERTEX) *WHERE* PURPOSE = 'Lift'.

   Then the spatial attributes of Lift will be returned for this query.

5. Who are Bill's neighbours?

   ($Q_{7.26}$) *SELECT* (OCCUPANT) (∗) *FROM* FLOOR_PLAN

   WHERE MEET(MIN_VERTEX, MAX_VERTEX,PICK_DISPLAY()).

   Then "Lee" will be returned because his office is on the right side of Bill's.

6. Outline the partitions of the PhD laboratory by dotted lines.

   ($Q_{7.27}$) SET_DISPLAY(default, default, dotted line).

   BOUNDARY(*SELECT* (MIN_VERTEX, MAX_VERTEX) (∗) *FROM*FLOOR_PLAN

   WHERE PURPOSE = 'Printing room').

7. Find the names of the staff whose rooms have area greater than 10 square units.

   ($Q_{7.28}$) *SELECT* (OCCUPANT) (∗) *FROM* FLOOR_PLAN

   WHERE AREA(MIN_VERTEX, MAX_VERTEX > 10)

   AND PURPOSE = 'Staff room'.

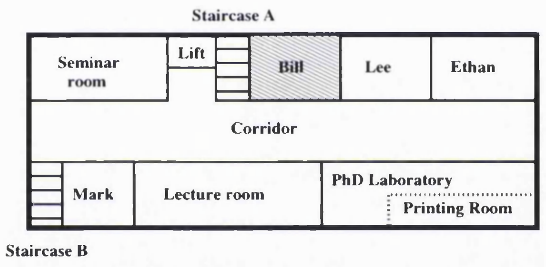


Figure 7.21: A graphical representation of the relation FLOOR_PLAN

7.7 Conclusions

We have presented a modularisation package discipline based on OSQL which supports a wide spectrum of applications. An OSQL package has the advantage that it integrates all of the useful operations with respect to a particular application in a more coherent and systematic way. OSQL provides us with new facilities to support the development of a package as well as to compare attributes according to semantic orderings, in addition to the usual system orderings. Thus, it allows us to capture the needed richer data semantics in advanced applications and it improves the expressive power of the standard SQL.

We are still in the process of implementing the PDL of OSQL using Oracle PL/SQL in order to define the mentioned OSQL packages and to make them available as built-in

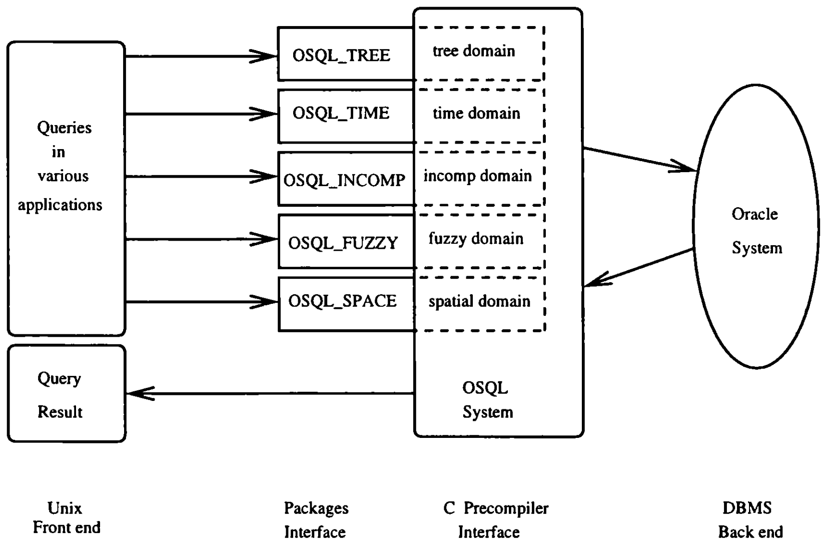


Figure 7.22: Architecture of the OSQL system

facilities. Our design of the system architecture is shown in Figure 7.22, which is built on top of OSQL system. We anticipate that it is possible to load more than one package into the system at the same time. All the functions of the loaded packages, which are qualified by their corresponding package names, can be applied directly in formulating a query. For example, the following query which involves the application having tree-structured information, temporal information and incomplete information can be formulated in a unified manner by using three OSQL packages. The relation EMP_DETAIL is shown in Figure 1.3 in Chapter 1.

($Q_{7.29}$) Find the name and salary of the common bosses of Nadav and Ethan in 1996, whose work is less informative than 'UNK'.

SELECT (NAME, SALARY) (∗)
FROM OSQL_TIME.SNAPSHOT(EMP_DETAIL, 1996)
WHERE OSQL_TREE.COM_ANCESTOR(Nadav, Ethan)
AND OSQL_INCOMP.LESS_INFO(PREV_WORK, 'UNK').

Chapter 8

Conclusions and Further Research

In this thesis we have presented the ordered relational model, which is a minimal extension of the relational data model. We have shown throughout the thesis that partial orderings in data domains have an important part to play in modelling data. The ordered relational model suggests that it is possible to unify a very large class of advanced real world applications in an efficient way. In Section 8.1 we review the main contributions of the thesis and evaluate our work from the points of view of usability, applicability and formalism. Finally, we discuss ongoing and further research in Section 8.2.

8.1 Summary of the Thesis Contribution

We have demonstrated that the extension of the relational data model to incorporate partial orderings into data domains can considerably improve the applicability of a relational DBMS. We now briefly recall the impact of partial orderings on the three fundamental components of the conventional relational data model.

With respect to its data structures, the relational data model is extended to incorporate partial orderings into data domains. Hence, it provides the flexibility to manipulate tuples in an ordered database according to the the semantics of underlying domains. We have shown that this extension serves as a good foundation to investigate the issues concerning query languages and data dependencies.

With respect to its query languages, we have extended the relational algebra and the relational calculus to the PORA and the PORC, respectively, by allowing the use of the ordering predicate, $\sqsubseteq$, in both languages. The PORA and the PORC are shown to be equivalent. Based on the PORA (or its counterpart the PORC), we have extended SQL to OSQL, which combines the capabilities of SQL with the power of semantic orderings. In order to gain more insights into the viability of OSQL, we have built a prototyped system of OSQL over the Oracle DBMS. The prototype was employed to perform a user survey in the department of computer science at UCL. From the survey we can confirm that the various extended features of the OSQL SELECT command are easy to learn, understand and apply, and are useful in formulating queries involving order.

With respect to its data dependencies, we have formally defined OFDs and OINDs, and have studied their semantics with respect to two categories of orderings: lexicographical orderings and pointwise orderings. In the case of pointwise orderings, we have presented sound and complete axiom systems for OFDs and OINDs. In the case of lexicographical orderings, we have presented a set of novel chase rules to OFDs, which are used to tackle the implication problem of OFDs. This set of chase rules is a useful tool for investigating other kind of data dependencies that require order.

Our work is best evaluated in the context of the three successful factors of the relational model, which we have discussed in Chapter 1.

1. From the point of view of usability, the ordered relational model is as natural and simple as the conventional relational model. Ordered domains are easily understood by non-specialist users due to the fact that partial orderings are structural truths about many types of data organisation in the real world. Our extension is done in a minimal and disciplined manner. The ordered database model we have defined is easily compatible with the syntax and semantics of the conventional relational database model.

2. From the point of view of applicability, the ordered relational model has been demonstrated to have the capabilities of capturing semantics in a wide spectrum of advanced applications such as tree-structured information, temporal information, incomplete information, fuzzy information and spatial information. In each of the above cases, the ordered relational model solves many interesting and common queries in a satisfactory manner. Moreover, it is the only data model known to us that combines all the above application capabilities under a single unified model.

3. From the point of view of formalism, the ordered relational model is elegant enough to support theoretical research in the areas of: data dependencies such as OFDs

and OINDs, the expressiveness of the ORA and the generic properties of queries over ordered databases. Moreover, we can build upon the rich mathematical research into the notion of order to investigate many important issues such as query completeness and axiomatisation of data dependencies.

8.2 Problems for Further Investigation

There is still a wide range of research issues that can be carried out on both the theoretical and the implementational aspects of the ordered relational model. We now discuss several areas that deserve our attention in further research.

1. Extension and specialisation of ordered domains:

   An interesting area for extension is to define some operators so that powerdomains can be derived from ordered domains (c.f., [20]). A useful ordering introduced by this extension is the containment ordering of a powerdomain. Apart from what we have discussed in textual databases, the concept of containment is closely associated with the object-oriented extension, since *inheritance* can be viewed as a containment ordering, in the sense that a derived type contains all the features of a base type. Containment is also related to the *Entity-Relationship approach* [30] because the ISA *relationship* can be viewed as a containment ordering. In addition to containment orderings, Hoare orderings and Smyth orderings [62, 143] are also important kinds of orderings arising from powerdomains since they generalise the notions of superset and subset, respectively.

   In contrast to the extension of domains to powerdomains, we may examine in more depth some restricted classes of partially ordered domains such as lattices and pre-orderings (or quasi-orderings) [59]. The restricted aspect of these extensions may bring out some interesting theoretical properties and provide more insights in optimisation of OSQL queries in practice.

2. OSQL implementation of packages and further user surveys:

   Although it is not our goal at this stage to develop a commercial version of the OSQL packages defined in Chapter 7, there are still many ways to employ our OSQL system as a basis to prototype certain package operations. One approach is to use Oracle PL/SQL facilitates to define functions that implement some operations, for example as SNAPSHOT and HISTORY in OSQL_TIME. Another approach is to use the C precompiler available under Oracle to deploy a package interfacing with OSQL. The prototype can serve as a basis to gain feedback from users as we did in the OSQL survey. The advantage of using precompiling C programs to develop the prototype of OSQL packages is that we can design and build a user interface more easily. A viable technique is to use embedded calls to the X windows system for creating and controlling an X-window interface (cf., [127]). With such an implementation we can extend the scope of our user survey by including a more comprehensive set of queries to verify the findings reported in Chapter 5.

3. Automorphisms and orderings:

   As we have pointed out in Chapter 3, there is an open problem to find a suitable syntactic definition of the notion "more ordered" in terms of the ordering automorphisms of an ordered domain. In a broader context, an ongoing research problem can be informally rephrased as follows, given the special class of abstract data types that are restricted to have generic binary relationships on data elements, is there a syntactic characterisation of the structure in terms of some generalised notion of automorphism. As the idea of automorphism represents some kinds of symmetry in domains, our intuitive feeling is that we may need to introduce some notion related to group theory in order to achieve a more general result.

4. Ordered data dependencies:

   We have developed the chase for LOFDs in Chapter 4 as a theorem proving tool, however, the maximal potential of this tool has not been fully developed. We are currently using the chase to prove the completeness of the set of inference rules of LOFDs proposed in Chapter 4. If we succeed in the proof, then the set of inference rules provide a more elegant axiom system for LOFDs. Otherwise, it implies that there may be some missing inference rules and in this case, the use of the chase in the proof may provide some insights to discover the missing inference rules. The chase is also a good reference point to design new inference procedures for LOINDs or for investigating the interaction of OFDs and OINDs. Besides, the chase supports the future work concerning the issues of time and space complexity of enforcing the data dependencies in ordered databases. Another important issue relevant to ordered data dependencies is the issue concerning database design. Although we have investigated the relationship between *lossless join* property [5] and OFDs in Chapter 4, we think that the effect of order on database design is not clearly known. For example, one question is how ordered data dependencies affect the *dependency preservation* property [13], and a more general question is whether there are any desirable properties of ordered data dependencies, or some useful special classes of them, in the context of relational database design.

5. Updating ordered databases:

   We have briefly discussed the issue of updating ordered domains in Chapter 3 but the problem of updating ordered databases has not been discussed in detail. It can be further investigated in terms of the algorithms and formal semantics of updating ordered domains, ordered databases and data dependencies. In particular, it is also important to consider how to enforce data dependencies to ensure that updates do not cause inconsistencies of data with respect to a set of OFDs or OINDs.

6. Notion of package completeness:

   As we know that OSQL packages, SQL3 and most SQL extensions are computationally complete, we feel that there is a need to establish a new framework to compare and contrast the operational completeness of different OSQL packages. In any case this problem also relates to the completeness of SQL3 and, in general, object-relational query languages.

To close this thesis, we would like to mention another large research area, that is, how to integrate the facilitates of user-defined orderings into the kernel of DBMSs at the physical level of a DBMS. A possible starting point is to examine a data structure called an *Ordered B-tree* [100, 139], which may serve as a basis to implement ordered relations. Roughly speaking, an *Ordered B-tree* stores data, for example tuple identifiers, in its leaf pages and a multi-level index is provided in each subtree to access data. In order to find a tuple identifier, the system is designed so that a scan can be performed from the root of the tree until a leaf page is encountered.

Bibliography

[1] S. Abiteboul and S. Ginsburg. Tuple Sequences and Lexicographical Indexes. *Journal of the Association for Computing Machinery* **33**(3), pp. 409-422, (1986).

[2] S. Abiteboul et al. Towards DBMSs for Supporting New Applications. In *Proceedings of the 3rd IEEE Conference on Data Engineering*, Los Angeles, pp. 580-589, (1987).

[3] S. Abiteboul and V. Vianu. Expressive Power of Query Languages. In J.D. Ullman (editor). *Theoretical Studies in Computer Science.* Academic Press, pp. 207-252, (1992).

[4] S. Abiteboul, R. Hull and V. Vianu. *Foundations of Databases.* Addison-Wesley, (1995).

[5] A.V. Aho and J.D. Ullman. Universality of Data Retrieval Languages. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pp. 110-120, (1979).

[6] A.V. Aho, J.E. Hopcroft and J.D. Ullman. *Data Structures and Algorithms.* Addison-Wesley, (1983).

[7] ANSI/X3/SPARC Study Group on Database Management Systems, Interim, Report. *FDT Bulletin of ACM SIGFIDET* **7**, (1975).

[8] E. Anuff. *The Java Sourcebook.* Wiley Computer Publishing John Wiley & Sons, Inc., (1996).

[9] P. Atzeni and V. De Antonellis. *Relational Database Theory.* Benjamin/Cummings Publishing Company, Inc., (1993).

[10] C.W. Bachman. Data Structure Diagrams. *Data Base* **1**(2), pp. 4-10, (1969).

[11] B.R. Badrinath and T. Imielinski. Replication and Mobility. In *Proceedings of the 2nd IEEE Workshop on Management of Replicated Data*, (1992).

[12] F. Bancilhon. On the Completeness of Query Languages for Relational Databases. In *LNCS 64: Mathematical Foundations of Computer Science*, Springer-Verlag, pp. 112-124, (1978).

[13] C. Beeri and P.A. Bernstein. Computational Problems Related to the Design of Normal Form Relational Schemas. *ACM Transactions on Database Systems* **4**(1), pp. 30-59, (1979).

[14] J. Biskup. Boyce-Codd Normal Form and Object Normal Forms. *Information processing Letters* **32**, pp. 29-33, (1989).

[15] J. Biskup. An Extension of SQL for Querying Graph Relations. *Computing Language* **15**(2), pp. 65-82, (1990).

[16] M.L. Brodie. On the Development of Data Models. In *On Conceptual Modelling: Perspectives from Artificial Intelligence, Databases, and Programming languages*, Springler-Verlag, pp. 19-47, (1984).

[17] B.P. Buckles and F.E. Petry. A Fuzzy Representation of Data for Relational Databases. *Fuzzy Sets and Systems* **7**, pp. 213-226, (1982).

[18] F. Buckley and F. Harary. *Distance in Graphs.* Redwood City, Ca.: Addison-Wesley, (1990).

[19] P. Buneman, S. Davidson and A. Watter. A Semantics for Complex Objects and Approximate Queries. In *7th Symposium on the Principle of Database Systems*, pp. 305-314, (1988).

[20] P. Buneman, A. Jung and A. Ohori. Using Powerdomains to Generalise Relational Databases. *Theoretical Computer Science* **91**, pp. 23-55, (1991).

[21] P. Buneman, S. Davidson, M. Fernandez and D. Suciu. Adding Structure to Unstructured Data. *Technical Report* MS-CIS 96-21, CIS Department, University of Pennsylvania, United States, (1996).

[22] M.A. Casanova, R. Fagin and C.H. Papadimitrious. Inclusion Dependencies and their Interaction with Functional Dependencies. *Journal of Computer and System Science* **28**, pp. 29-59, (1984).

[23] M.A. Casanova, A.L. Furtado and L. Tucherman. A Software Tool for Modular Database Design. *ACM Transactions on Database Systems* **2**, pp. 209-234, (1991).

[24] R. Cattell (editor). *The Object Database Standard: ODMG-93 (Release 1.2).* Morgan Kaufaman Publishers, (1996).

[25] M. Carey and D.J. DeWitt. Of Objects and Databases: A Decade of Turmoil. In *Proceedings of the 22nd VLDB Conference*, Mumbai, India, (1996).

[26] M. Carey et al. Storage Management in EXODUS. In *Object-Oriented Concepts, Databases, and Applications.* ACM Press and Addison-Wesley, pp. 341-369, (1989).

[27] J. Celko. *SQL For Smarties: Advanced SQL Programming.* Morgan Kaufmann Publishers, (1995).

[28] A.K. Chandra and D. Harel. Computable Queries for Relational Databases. *Journal of Computer System Science* **21**, pp. 156-178, (1980).

[29] C.L. Chang. Decision Support in an Imperfect World. *IBM Research Report* RJ3421, IBM, San Jose, Dec, (1982).

[30] P.P. Chen. The Entity-Relationship Model: Towards a Unified View of Data. *ACM Transactions on Database Systems* **1**(1), pp. 9-36, (1976).

[31] D.L. Child. Feasibility of a Set-Theoretical Data Structure – a General Structure Based on a Reconstitued Definition of Relation. *Proceeding IFIP Congress*, pp. 162-172, Amsterland, (1968).

[32] P. Ciaccia and D. Maio. On the Complexity of Finding Bounds for Projection Cardinalities in Relational Databases. *Information Systems* **17**(6), pp. 511-515, (1992).

[33] *CODASYL Data Base Task Group April 71 Report*, ACM, New York (1971).

[34] E.F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM* **13**(6), pp. 377-387, (1970).

[35] E.F. Codd. Relational Completeness of Data Base Sublanguages, In R. Rustin (editor). *Database Systems.* Prentice-Hall, Englewood Cliffs, NJ, pp. 65-98, (1972).

[36] E.F. Codd. Further Normalisation of the Data Base Relational Model. In R. Rustin (editor). *DataBase Systems*, Prentice-Hall, Englewood Cliffs, NJ, pp. 33-64, (1972).

[37] E.F. Codd. Extending Database Relational Model to Capture More Meaning. *ACM Transactions on Database Systems* **4**, pp. 397-434, (1979).

[38] E.F. Codd. Missing Information (Applicable and Inapplicable) in Relational Databases. *ACM SIGMOD record* **15**, pp. 53-78, (1987).

[39] E.F. Codd. *The Relational Model for Database Management*, Addison-Wesley, (1990).

[40] J. Conklin. Hypertext: An introduction and Survey. *IEEE Computer* **20**, pp. 17-41, (1987).

[41] C.J. Date. *Relational Database Writings 1985-1989.* Addison-Wesley, (1990).

[42] C.J. Date. *Relational Database Writings 1989-1991.* Addison-Wesley, (1992).

[43] C.J. Date. *A Guide to the SQL Standard.* Addison-Wesley, 3rd ed., (1993).

[44] C.J. Date. *Relational Database Writings 1991-1994.* Addison-Wesley, (1995).

[45] D. Denning et al. A Multilevel Relational Data Model. In *Proceedings of IEEE Symposium on Security and Privacy*, pp. 220-234, Oakland, Califonia, (1987).

[46] O. Deux et al. The O_2 system. *Communications of the ACM*, **34**(10), pp. 34-49, (1991).

[47] K. Dittrich and U. Dayal (editors). In *Proceedings of the 1st International Workshop on Object-Oriented Database Systems*, Pacific Grove, CA, (1986).

[48] M.J. Egenhofer. A Formal Definition of Binary Topological Relationships. In *LNCS 367: Foundations of Data Organization and Algorithms, 3rd International Conference, Proceedings*, Springer-Verlag, pp. 457-472, (1989).

[49] M.J. Egenhofer. Reasoning about Binary Topological Relations. In *LNCS 525: Advances in Spatial Databases, Second International Symposium, SSD'91*, Springer-Verlag, pp. 143-160, (1991).

[50] M.J. Egenhofer. Spatial SQL: A Query and Presentation Language. *IEEE Transaction on Knowledge and Data Engineering* **6**(1), pp. 86-95, (1994).

[51] R. Fagin. A Normal Form for Relational Databases that is Based on Domain and Keys. *ACM Transactions on Database Systems* **6**(3), pp. 310-319, (1981).

[52] S. Feuerstein. *Oracle PL/SQL.* O'Reilly & Associates, Inc., (1995).

[53] M. Fitting. *First-order Logic and Automated Theorem Proving.* Springer-Verlag, (1990)

[54] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman and Co., New York, (1979).

[55] S. Ginsburg and R. Hull. Order Dependency in the Relational Model. *Theoretical Computer Science* **26**, pp. 129-195, (1983).

[56] S. Ginsburg and R. Hull. Sort Sets in the Relational Model. *Journal of the Association for Computing Machinery,* **33**(3), pp. 465-488, (1986).

[57] S. Ginsburg and K. Tanaka. Computation-Tuple Sequences and Object Histories. *ACM Transactions on Database Systems* **11**(2), pp. 186-212, (1986).

[58] M. Gray. Views and Imprecise information in Databases. *Technical Report* No. 38, University of Cambridge, England, (1982).

[59] G. Gratzer. *General Lattice Theory.* NewYork : Academic Press, (1978).

[60] V. Gaede. Optimal Redundancy in Spatial Database Systems. In *LNCS 951: Proceedings of the 4th International Symposium in Advances in Spatial Databases SSD'95,* Springer-Verlag, pp. 96-116, (1995).

[61] G. Grahne. Dependency Satisfaction in Databases with Incomplete Information. In *Proceedings of the International Conference on Very Large Data Bases,* Singapore, pp. 37-45, (1984).

[62] G. Gunter. The Mixed Power Domain. *Theoretical Computer Science* **103**, pp. 311-334, (1992).

[63] R.H. Guting, R. Zicari and D.M. Choy. An Algebra for Structured Office Documents. *ACM Transactions on Office Information Systems* **7**(4), pp. 123-157, (1989).

[64] J. Haigh et al. The LDV Secure Relational DBMS Model. In S. Jajodia and C. Landwehr (editors). *Database Security, IV: Status and Prospects,* North-Holland, Amsterdam, pp. 265-279, (1991).

[65] P. Halmos. *Naive Set Theory.* Springer-Verlag, New York, (1974).

[66] H. Kopka and P. W. Daly. *A Guide to LaTeX 2_ε.* Addion-Wesley Publishing Company, Inc., (1995)

[67] E. Horowitz and S. Sahni. *Fundamental of Data Structures.* Computer Science Press, Inc., (1976)

[68] T. Imielinski, S. Naqvi and K. Vadaparty. Incomplete Objects - a Data Model for Design and Planning Applications. In *Proceedings of ACM-SIGMOD*, Denver, Colorado, pp. 288-297, (1991).

[69] W.H. Inmon. *Building the Data Warehouse.* John Wiley & Sons. Inc., (1996)

[70] S. Jajodia and R. Sandhu. A Novel Decomposition of Multilevel Relations into Single-Level Relations. *Proceedings of IEEE Symposium on Security and Privacy*, pp. 300-313, Oakland, Califonia, (1987).

[71] C.B. Jones, D.B. Kidner and J.M. Ware. The implicit Tringulated Irregular Network and Multiscale Spatial Databases. *The Computer Journal* **37**(1), pp. 43-57, (1994).

[72] A. Jung, L. Libkin and H. Puhlmann. Decomposition of Domains. In *LNCS 598: Proceedings of the Conference on Mathematical Foundations of Programming Semantics-91*, Springer-Verlag, pp. 235-258, (1992).

[73] D.S. Johnson and A. Klug. Testing Containment of Conjunctive Queries under Functional and Inclusion Dependencies. *Journal of Computer and System Sciences* **28**, pp. 167-189, (1984).

[74] P.C. Kanellakis. On the Computational Complexity of Cardinality Constraints in Relational Databases. *Information Processing Letters* **11**(2), pp. 98-101, (1980).

[75] R.H. Katz. Towards a Unified Framework for Version Modeling in Engineering Databases. *Computing Surveys* **22**(4), pp. 375-408, (1990).

[76] G. Klir and B. Yuan. *Fuzzy Sets and Fuzzy Logic: Theory and Application.* Prentice-Hall Inc., (1995).

[77] S. Kelly. *Data Warehousing: The route to Mass Customisation.* John Wiley & Sons. Inc., (1996)

[78] W. Kim, J.F. Garza, N. Ballou and D. Woelk. Architecture of Orion Next-Generation Database System. *IEEE Transactions on Knowledge and Data Engineering* **2**(1), pp. 109-124, (1990).

[79] W. Kim (editor). *Modern Database Systems: The Object Model, Interperability, and Beyond*, ACM Press, (1995).

[80] A. Klug. Equivalence of Relational Algebra and Relational Calculus Query Languages Having Aggregate Functions, *Journal of the Association for Computing Machinery*, **29**(3), pp. 699-717, (1982).

[81] K.G. Kulkarni. Object-Orientation and the SQL standard. *Computer Standards and Interfaces* **15**, pp. 287-300, (1993).

[82] G. Koch and K. Loney. *Oracle: The Complete Reference, Third Edition.* Osborne McGraw-Hill, (1995).

[83] W. Kurutach. *Analysis and Modelling of Aspects of Imperfection and Time in Databases.* Ph.D. Thesis, University of New South Wales, Australia, (1996).

[84] M. Levene and G. Loizou. Database Design of Incomplete Relations, *Research Note* RN/95/18, Department of Computer Science, University College London, United Kingdom, (1995).

[85] M. Levene and G. Loizou. A Corrspondence Between Variable Relations and Three-valued Propositional Logic. *International Journal Computer Mathematics* **55**, pp. 29-38, (1995).

[86] M. Levene and G. Loizou. Maintaining consistency of imprecise relations. *The Computer Journal* **39**, pp. 114-123, (1996).

[87] M. Levene and G. Loizou. Null Inclusion Dependencies in Relational Databases. To appear in *Information and Computation*, (1997).

[88] M. Levene and G. Loizou. The Additivity Problem for Functional Dependencies in Incomplete Relations. *Acta Informatica* **34**, pp. 135-149, (1997).

[89] L. Libkin. A Relational Algebra for Complex Objects Based on Partial Information. In *LNCS 495: Proceedings of Symposium on Mathematical Fundamentals of Database Systems-91*, Rostock, Springer-Verlag, pp. 36-41, (1992).

[90] L. Libkin. Algebraic Characterisation of Edible Powerdomains. *Technical Report MS-CIS-93-70*, University of Pennsylvania, United States, (1993).

[91] L. Libkin. *Aspects of Partial Information in Databases.* Ph.D. Thesis, University of Pennsylvania, United States, (1996).

[92] L. Libkin. A Semantics-based Approach to Design of Query Languages for Partial Information. In *Proceedings of the Workshop on Semantics in Databases*, pp. 63-80, (1995).

[93] R. Likert. A Technique for The Measurement of Attitudes. *Architecture Psychology* **140**, pp. 1-55, N.Y., (1932).

[94] B.G. Lindsay and L.M. Hass. Extensibility in the Starburst Experimental Database System. In *LNCS 466: Database Systems of the 90s, International Symposium, Proceedings,* pp. 217-248, Springer-Verlag, (1990).

[95] A. Loeffen. Text Databases: A Survey of Text Models and Systems. *SIGMOD Record* **23**(1), pp. 97-106, (1994).

[96] D. Lomet and E. Moss (Editors). Special Issue on Integration Text Retrieval and Database. *Bulletin of the Technical Committe on Data Engineering* **19**(1), (1996).

[97] N.A. Lorentzos. DBMS Support for Time and Totally Ordered Compound Data Types. *Information Systems* **17**(5), pp. 347-358, (1992).

[98] N.A. Lorentzos, A. Poulovassilis and C. Small. Manipulation Operations for an Interval-Extended Relational Model. *Data and Knowledge Engineering* **17**(1), pp. 1-29, (1995).

[99] H. Lu, H.C. Chan and K.K. Wei. A Survey on Usage of SQL. *SIGMOD Record* **22**(4), pp. 60-65, (1993).

[100] N. Lynn. *Implementation of Ordered Relations in a Data Base System,* Master Thesis, Department of Electrical Engineering and Computer Science, University of California, United States, (1982).

[101] D. Maier, A.O. Mendelzon and Y. Sagiv. Testing Implication of Data Dependencies. *ACM Transactions on Database Systems* **4**, pp. 455-469, (1979).

[102] D. Maier. Development of an Object-Oriented DBMS. *Proceedings ACM OOPSLA Conference*, Portland, OR, (1986).

[103] D. Maier and B. Vance. A Call to Order. In *Proceedings of the 12th ACM Symposium on Principles of Databases Systems*, pp. 1-16, (1993).

[104] H. Mannila and K-J Raiha. Generating Armstrong Databases for Sets of Functional and Inclusion Dependencies. *Research Report* A-1988-7, University of Tampere, Finland, (1988).

[105] H. Mannila and K-J Raiha. *The Design of Relational Databases.* Addision-Wesley, (1992).

[106] N. Mattos and L.G. DeMichiel. Recent Design Trade-offs in SQL3. *ACM SIGMOD Record* **23**(4), pp. 84-89, (1994).

[107] M.C. McCabe and D. Grossman. The Role of Tools in Development of a Data Warehouse. In *Proceedings of the 4th International Symposium on Assessment of Software Tools*, pp. 139-145, (1996).

[108] J. Melton. An SQL3 Snapshot. In *Proceedings of the International Conference on Data Engineering*, pp. 666-672, (1996).

[109] J.C. Mitchell. The Implication Problem for Functional and Inclusion Dependencies. *Information and Control* **56**, pp. 154-173, (1983).

[110] P. Mishra and M. Eich. Functional Completeness in Object-Oriented Databases. *ACM SIGMOD Record* **21**(1), pp. 71-83, (1992).

[111] H. Nakajima. Fuzzy Database Language and Library – Fuzzy Extension to SQL. *2nd IEEE International Conference Fuzzy Systems*, pp. 477-482, (1993).

[112] W. Ng and M. Levene. On the Expressive Power of the Relational Algebra with Partially Ordered Domains. *Research Note* RN/95/77, Department of Computer Science, University College London, United Kingdom, (1995).

[113] W. Ng and M. Levene. OSQL: An Extension to OSQL to Manipulate Ordered Relational Databases. In *Proceedings of the 3rd International Workshop on Next Generation Information Technologies and Systems*, Jerusalem, Israel, pp. 77-88, (1997).

[114] W. Ng and M. Levene. An Extension of OSQL to Support Ordered Domains in Relational Databases. In *IEEE Proceedings of the International Database Engineering and Applications Symposium*, Montreal, Canada, pp.358-367, (1997).

[115] W. Ng and M. Levene. The Development of Ordered SQL Packages for Modelling Advanced Applications. In *LNCS 1308: Database and Expert Systems Application, 8th International Conference, DEXA '97, Proceedings,* Springer-Verlag, Toulouse, France, pp. 529-538, (1997).

[116] W. Ng and M. Levene. The Development of Ordered SQL Packages to Support Data Warehousing. *Research Note* RN/97/27, Department of Computer Science, University College London. *Proceedings of the 8th International Database Workshop,* Hong Kong, pp. 208-235, (1997).

[117] W. Ng and M. Levene. A User Survey on Ordered SQL. *Research Note* RN/97/38, Department of Computer Science, University College London, United Kingdom, (1997).

[118] J. Ong, D. Fogg and M. Stonebraker. Implementation of Data Abstraction in the Relational Database System Ingres. *SIGMOD Record* **14**(1), pp. 1-14, (1984).

[119] *Programmer's Guide to the Oracle Precompilers Release 1.8.* Oracle Corporation, (1996).

[120] J. Orenstein and T.H. Merrett. A Class of Data Structures for Associative Searching. In *Proceedings of the 3rd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pp. 181-196, (1984).

[121] S.L. Osborn and T.E. Heaven. The Design of a Relational Database System with Abstract Data Types for Domains. *ACM Transactions on Database Systems* **11**, pp. 357-373, (1986).

[122] C.H. Papadimitriou, D. Suciu and V. Viamu. Topological Queries in Spatial Databases. In *Proceedings of PODs'96*, pp. 81-92, (1996).

[123] J. Paredaens. On the Expressive Power of the Relational Algebra. *Information Processing Letters* **7**(2), pp. 107-111, (1978).

[124] J. Paredaens, J. Bussche and D. Gucht. Towards a Theory of Spatial Database Queries. In *Proceedings of the 13th ACM symposium on Principles of Databases*, pp. 279-287, (1994).

[125] J. Paredaens. Spatial Databases, The Final Frontier. In *Proceedings of the 5th International Conference on Database Theory*, pp. 14-32, (1995).

[126] J. Plotkin. A Powerdomain Construction. *SIAM Journal of Computing* **5**, pp. 452-487, (1976).

[127] J. Poole. *A Graphical Oracle Database Browser.* Master Thesis, Department of Computer Science, Birkbeck College, University of London, United Kingdom, (1991).

[128] K.V.S.V.N. Raju and A.K. Majumdar Fuzzy Functional Dependencies and Lossless Join Decomposition of Fuzzy Relational Database Systems. *ACM Transactions on Database Systems* **13**(2), pp. 129-166, (1988).

[129] D. Raymond. *Partial Order Databases.* Ph.D. Thesis, University of Waterloo, Canada, (1996).

[130] R. Read. *Towards Multiresolution Data Retrieval via the Sandbag.* Ph.D. Thesis, University of Texas at Austin, United States, (1995).

[131] J. Richardson. Supporting Lists in a Data Model. *Proceedings of the 18th VLDB Conference*, Vancouver, Canada, (1992).

[132] J. Rissanen. Independent Components of Relations. *ACM Transactions on Database Systems* **2**(4), pp. 317-325, (1977).

[133] J. Rosenstein. *Linear orderings.* New York: Academic Press, (1982).

[134] B. Rounds. Situations-Theoretic Aspects of Databases. In *Proceedings of Conference on Situation Theory and Applications*, CSLI **26**, pp. 229-256, (1991).

[135] N.L. Sardra. Algebra and Query Language for a Historical Data Model. *The Computer Journal* **33**(1), pp. 11-18, (1990).

[136] P. Seshadri, M. Livny and R. Ramakrishnan. The Design and Implementation of a Sequence Database System. *Proceedings of the 22nd VLDB Conference*, Mumbai, India, (1996).

[137] I. Sommerville. *Software Engineering.* Addison-Wesley, (1992).

[138] M. Stonebraker. Inclusion of New Types in Relational Data Base Systems. In *Proceedings of the 2nd IEEE Data Engineering Conference,* Los Angeles, CA, (1986).

[139] M. Stonebraker (editor). *The INGRES Papers*, Reading, Mass, Addision Wesley, (1996).

[140] M. Stonebraker. *Interviewd by DBMS online*, URL Address: http://www.dbmsmag.com, (1994).

[141] M. Stonebraker. *Object Relational DBMSs: The Next Great Wave.* Morgan Kaufmann Publishers, Inc., (1996).

[142] M.B. Smyth. *Effectively Given Domains.* Theoretical Computer Science **5**, pp. 257-274, (1977).

[143] M.B. Smyth. Power Domains. *Journal of Computer and System Sciences* **16**, pp. 23-36, (1978).

[144] A. Tansel. Adding Time Dimension to Relational Model and Extending Relational Algebra. *Information Systems* **11**(4), pp. 343-355, (1986).

[145] A. Tansel et al. (editors). *Temporal Databases: Theory, Design and Implementation.* The Benjamin/Cummings Publishing Company, Inc., (1993)

[146] M. Trott. *Mathematica: A Detailed Introduction.* TELOS, Springer-Verlag, (1994).

[147] J.D. Ullman. *Principles of Database and Knowledge-Base Systems, Vol. I.* Rockville, MD., Computer Science Press, (1988).

[148] M.Y. Vardi. The Decision Problem for Database Dependencies. *Information Processing Letters* **12**(5), pp. 251-154, (1981).

[149] S.L. Vandenberg and D.J. DeWitt. Algebraic Support for Complex Objects with Arrays, Identity, and Inheritance. In *Proceedings of ACM SIGMOD Conference*, pp. 158-168, Denver, CO, (1991).

[150] M. Vincent. *The Semantic Justification for Normal Forms in Relational Database Design.* Ph.D. Thesis, Monash University, Australia, (1994).

[151] M. Winslett, K. Smith and X. Qian. Formal Query Languages for Secure Relational Databases. *ACM Transactions on Database Systems* **19**(4), pp. 626-662, (1994).

[152] X. Wang. *Pattern Matching by Rs-operations: Towards a Unified Approach to Querying Sequenced Data.* Ph.D. Thesis, University of Southern California, United States, (1992).

[153] M.F. Worboys. A Unified Model for Spatial and Temporal Information. *The Computer Journal* **37**(1), pp. 26-34, (1994).

[154] C. Zaniolo. Database Relations with Null Values. *Journal of Computer and System Science* **28**, pp. 142-166, (1984).

[155] Z. Zdonik and D. Maier (editors). *Readings in Object-Oriented Database Systems.* Morgan Kaufamann Publisher, (1990)

Appendix A

A Grammar of OSQL

Conventions:

- Key words are indicated by uppercase italicized characters.
- Non-terminal symbols are enclosed with "⟨⟩".
- Alternatives are separated by "|". If only one of the symbols is to be chosen out of several alternatives, then we enclose them with "{ }". In order not to cause confusion, we use "{{" and "}}" to represent the terminal symbols "{" and "}", respectively.
- Optional clauses are enclosed with "[ ]".
- "()" are just terminal symbols.
- Default keywords are underlined.
- A positive number begins with #.
- ... at the end if a subclause indicates that it may be repeated.

A.1 Data Definition Language

1. *CREATE DOMAIN* ⟨ domain-name ⟩ ⟨ data-type ⟩ [*ORDER AS*
⟨ ordering-specification ⟩]
⟨ ordering-specification ⟩ ::= (⟨ data-pair ⟩[, ⟨ data-pair ⟩]...)
⟨ data-pair ⟩ ::= [data-item | {{data-item,...}}] < [data-item | {{data-item,...}}]
2. *CREATE DOMAIN* ⟨ domain-name ⟩ *AS* ⟨ domain-name ⟩
3. *CREATE TABLE* ⟨ table-name ⟩
(⟨ column-specification ⟩ [,⟨ column-specification ⟩]...⟩ [*ORDER AS*
⟨ attribute-list ⟩]
⟨ column specification ⟩ ::= (attribute-name ⟨ data-type ⟩)
⟨ data-type ⟩::= {*CHAR*(integer) | *NUMBER*(integer)}

A.2 Data Manipulation Language

1. *SELECT* ⟨ attribute-list ⟩ [{<u>*ANY*</u> | *ALL*}] ⟨ tuple-list ⟩ [{<u>*ASC*</u> | *DESC*}] *FROM* ⟨ relation-list ⟩ [*WHERE* ⟨ condition ⟩]

⟨ attribute-list ⟩ ::= (⟨ extended-attribute ⟩ [,⟨ extended-attribute ⟩]...)
⟨ extended-attribute ⟩ ::= {attribute-name | (attribute-name *WITHIN*
⟨ domain-name ⟩ | *)}
⟨ tuple-list ⟩ ::= ({#n [, #n]) | *LAST* | #n1...#n2 | *})
⟨ condition ⟩ ::= ⟨ attribute-name | value⟩ ⟨ comparator ⟩
⟨{attribute-name | value}⟩ [*WITHIN* ⟨ domain-name ⟩]
⟨ comparator ⟩ ::= {<|>|>=|<=|<>}

2. *DELETE FROM* ⟨ table-name ⟩ [{ *WHERE* ⟨ condition ⟩ | *TUPLE*
⟨ tuple-list ⟩}]

3. *DELETE FROM* ⟨ table-name ⟩ [{ *WHERE* ⟨ condition ⟩ | *TUPLE*
⟨ tuple-list ⟩}]

4. *ALTER TABLE* ⟨ table-name ⟩ {*ADD* (⟨ column-list ⟩) | *MODIFY*
(⟨ column-list⟩) | *ORDER AS* (⟨ attribute-list ⟩)}

A.3 Package Definition Language

1. *PACKAGE* ⟨ package-name ⟩
⟨ package-body ⟩
END PACKAGE
⟨ package-body ⟩ :: = { *PARAMETER*: ⟨ parameter-list ⟩
FUNCTION:⟨ function-list ⟩ *ENFORCEMENT*:⟨ enforcement-list ⟩ }

2. ⟨ parameter-list ⟩ :: = { ⟨ parameter-construct ⟩[⟨ parameter-construct ⟩]...}
⟨ parameter-construct ⟩ :: = ⟨ package-data-type ⟩: variable-name [,variable-name]...
⟨ package-data-type ⟩:: = { *VARCHAR* | *INT* | *BOOL* | *REL* }

3. ⟨ function-list ⟩ :: = { ⟨ function-construct ⟩ [⟨ function-construct ⟩]...}
⟨ function-construct ⟩ :: =
[{*<u>PRI</u>* | *PUB*}] ⟨ function-name ⟩ variable-names ⟨ parameter-list ⟩
DEFINE
⟨ function-body ⟩
RETURN variable-names
⟨ function-body ⟩ :: = [⟨ program-construct ⟩ | ⟨ OSQL-construct ⟩]
⟨ program-construct ⟩ :: = *AS PROG* program-name pseudocode
⟨ OSQL-construct ⟩ :: = [DDL statements | DML statements]

4. ⟨ enforcement-list ⟩ :: = { ⟨ enforcement-construct ⟩ [⟨ enforcement-construct ⟩]...}
⟨ enforcement-construct ⟩ :: =
⟨ enforcement-name ⟩
DEFINE
⟨ program-construct ⟩
END

Appendix B

A Detailed Description of Built-In OSQL Packages

B.1 OSQL_TREE Package and Its Operation

```
PACKAGE OSQL_TREE
PARAMETER:
   VARCHAR: tree_node_1, tree_node_2, ext_domain,
      tree_domain, $ext_relation, $ext_att
   BOOL: bool_val,
   INT: node_level, count_nodes
   REL: nodes
FUNCTION:
   PUB  IDENTIFY(ext_domain) RETURN
   PUB  PARENT(tree_node_1) RETURN nodes
   PUB  CHILDREN(tree_node_1) RETURN nodes
   PUB  ANCESTOR(tree_node_1) RETURN nodes
   PUB  COM_ANCESTOR(tree_node_1, tree_node_2) RETURN nodes
   PUB  OFFSPRING(tree_node_1) RETURN nodes
   PUB  LEAVES() RETURN nodes
   PUB  ROOT() RETURN nodes
   PUB  LEVEL(tree_node_1) RETURN node_level
   PUB  SWAP(tree_node_1, tree_node_2) RETURN
   VERIFY(tree_domain) RETURN bool_val
   NODE_COUNT(nodes) RETURN count_nodes
ENFORCEMENT:
   ENFORCE_INIT()
   ENFORCE_IDENTIFY()
   ENFORCE_SWAP()
END PACKAGE
```

PUB IDENTIFY(ext_domain)

```
DEFINE
    CREATE DOMAIN tree_domain AS ext_domain
RETURN

PUB PARENT(tree_node_1)
DEFINE
    SELECT ($ext_att WITHIN tree_domain) (1) FROM $ext_relation
    WHERE ($ext_att > tree_node_1 WITHIN tree_domain)
RETURN nodes

PUB CHILDREN(tree_node_1)
DEFINE
    SELECT ($ext_att WITHIN tree_domain) (LAST)
    FROM $ext_relation
    WHERE ($ext_att < tree_node_1 WITHIN tree_domain)
RETURN nodes

PUB ANCESTOR(tree_node_1)
DEFINE
    SELECT ($ext_att) (*) FROM $ext_relation
    WHERE ($ext_att > tree_node_1 WITHIN tree_domain)
RETURN nodes

PUB OFFSPRING (tree_node_1)
DEFINE
    SELECT ($ext_att) (*) FROM $ext_relation
    WHERE ($ext_att < tree_node_1 WITHIN tree_domain)
RETURN nodes

PUB COM_ANCESTOR(tree_node_1, tree_node_2)
DEFINE
    SELECT ($ext_att) (*) FROM $ext_relation
    WHERE ($ext_att > tree_node_1 WITHIN tree_domain)
    AND ($ext_att > tree_node_2 WITHIN tree_domain)
RETURN nodes

PUB LEAVES()
DEFINE
    SELECT ($ext_att) (*) FROM $ext_relation
    WHERE NODE_COUNT(CHILDREN($ext_att)) = 0
RETURN nodes

PUB ROOT()
DEFINE
    SELECT ($ext_att) (1) DESC FROM $ext_relation
RETURN nodes
```

```
PUB LEVEL(tree_node_1)
REL: TREE_LEVEL
DEFINE
    SELECT (LEVEL_NUMBER) (*) FROM TREE_LEVEL
    WHERE NODE = tree_node_1
RETURN node_level

PUB SWAP(tree_node_1, tree_node_2)
DEFINE AS PROG tree.swap
    1. Replace tree_node_2 in tree_domain by tree_node_3 which is distinct from
       other nodes in the tree_domain.
    2. Replace tree_node_1 in tree_domain by tree_node_2.
    3. Replace tree_node_3 in tree_domain by tree_node_1.
RETURN

VERIFY (tree_domain)
DEFINE AS PROG tree.verify
    1. Define the boolean routine CHECK(domain) as follows.
        1.1 Remove the root from the given domain.
        1.2 Partition the remaining tree_nodes into n >= 0 disjoint ordered sets, $T_1, \ldots, T_n$,
            in which all nodes are connected.
        1.3 For each $T_i$, perform the routine CHECK($T_i$) recursively.
    2. CHECK(tree_domain).
RETURN bool_val

FUNCTION NODE_COUNT(nodes)
DEFINE
    SELECT (COUNT(*)) (*) FROM nodes
RETURN count_nodes

ENFORCE_INIT()
DEFINE AS PROG tree.enforce_init
    1. If there exists a domain called TREE, then IDENTIFY(TREE), else prompt
       for a tree domain.
END

ENFORCE_IDENTIFY()
DEFINE AS PROG tree.enforce_identify
    1. If the function IDENTIFY is called, then go to an error status if
       VERIFY(tree_domain) = false.
END

ENFORCE_SWAP()
DEFINE AS PROG tree.enforce_swap
    1. If the function SWAP is called, then go to an error status if
       VERIFY(tree_domain) = false.
END
```

B.2 OSQL_TIME Package and Its Operations

```
PACKAGE OSQL_TIME
PARAMETER:
    VARCHAR: time_domain, ext_relation, time_instant_1, time_instant_2, NOW
        non_time_schema, ext_domain
    INT: granularity, duration
    BOOL: bool_val
    REL: result_relation
FUNCTION:
    PUB  IDENTIFY(ext_domain)
    PUB  CURRENT(ext_relation) RETURN result_relation
    PUB  HISTORY(ext_relation) RETURN result_relation
    PUB  COALESCE(ext_relation) RETURN result_relation
    PUB  SUCC(time_instant_1) RETURN time_instant_2
    PUB  PRED(time_instant_1) RETURN time_instant_2
    PUB  DURA(time_instant_1, time_instant_2) RETURN duration
    PUB  SNAPSHOT(ext_relation, time_instant_1) RETURN result_relation
    PUB  EXPAND(ext_relation) RETURN result_relation
    PUB  TIME_RES(granularity, ext_domain) RETURN
    VERIFY(time_domain) RETURN bool_val
    STRIP_TIME(ext_relation) RETURN non_time_schema
ENFORCEMENT:
    ENFORCE_INIT()
    ENFORCE_IDENTIFY()
END PACKAGE
```

```
PUB IDENTIFY(ext_domain)
DEFINE
    CREATE DOMAIN time_domain AS ext_domain
RETURN

PUB CURRENT(ext_relation)
DEFINE
    SELECT (STRIP_TIME(ext_relation)) (*) FROM ext_relation
    WHERE TO_TIME = NOW
RETURN result_relation

PUB HISTORY(ext_relation)
DEFINE
    SELECT (*) (*) FROM ext_relation
    WHERE (TO_TIME < NOW  WITHIN time_domain)
RETURN result_relation

PUB SNAPSHOT(ext_relation, time_instant_1)
DEFINE
```

```
    SELECT (STRIP_TIME(ext_relation)) (*) FROM ext_relation
    WHERE (FROM_TIME <= time_instant_1 WITHIN time_domain)
    AND (TO_TIME > time_instant_1 WITHIN time_domain)
RETURN result_relation

PUB SUCC(time_instant_1)
REL: TIME_DOM_REL
DEFINE
    SELECT (TIME_DATA) (1) FROM TIME_DOM_REL
    WHERE TIME_DATA > time_instant_1
RETURN time_instant_2

PUB PRED(time_instant_1)
REL: TIME_DOM_REL
DEFINE
    SELECT (TIME_DATA) (LAST) FROM TIME_DOM_REL
    WHERE TIME_DATA < time_instant_1
RETURN time_instant_2

PUB DURA (time_instant_1, time_instant_2)
DEFINE AS PROG time.dura
    1. Convert time_instant_1 to the number of chronons, templ.
    2. Convert time_instant_2 to the number of chronons, temp2.
    3. Return the result of (templ - temp2).
RETURN duration

PUB EXPAND(ext_relation)
REL: TIME_DOM_REL
DEFINE
    SELECT (STRIP_TIME(ext_relation), TIME_DATA FROM_TIME,
    SUCC(TIME_DATA) TO_TIME) (*) FROM ext_relation, TIME_DOM_REL
    WHERE (FROM_TIME <= TIME_DATA WITHIN time_domain)
    AND (TO_TIME > TIME_DATA WITHIN time_domain)
RETURN result_relation

PUB COALESCE(ext_relation)
DEFINE
    SELECT (R.STRIP_TIME(ext_relation), R.FROM_TIME, MIN(S.TO_TIME)
    TO_TIME) (*) FROM ext_relation R, ext_relation S WHERE
    R.FROM_TIME NOT IN (SELECT (TO_TIME) (*) FROM ext_relation)
    AND TO_TIME NOT IN (SELECT (FROM_TIME) (*) FROM ext_relation)
    AND (R.FROM_TIME < S.TO_TIME WITHIN time_domain)
    AND R.STRIP_TIME(ext_relation) = S.STRIP_TIME(ext_relation)
    GROUP BY (R.STRIP_TIME(ext_relation), R.FROM_TIME)
RETURN result_relation

PUB TIME_RES(granularity, ext_domain)
DEFINE AS PROG time.res
```

```
    1. Create a relation to maintain the time domain.
    2. Populate the relation from 0 to (granularity − 1).
RETURN

FUNCTION VERIFY(time_domain)
DEFINE AS PROG time.verify
    1. Check that the time_domain is finite and linearly ordered.
    2. Check that all elements $t$ in the time_domain satisfy that $t <=$ NOW.
RETURN bool_val

PUB STRIP_TIME(ext_relation)
DEFINE AS PROG time.strip
    1. Obtain the relational schema of the ext_relation.
    2. Project out the attributes FROM_TIME and TO_TIME from the schema.
    3. Return the remaining attributes.
RETURN non_time_schema

ENFORCE_INIT()
DEFINE AS PROG time.enforce_init
    1. IDENTIFY(DATE)
    2. Using the relation TIME_DOM_REL to maintain DATE.
END

ENFORCE_IDENTIFY()
DEFINE AS PROG time.enforce_identify
    1. If the function IDENTIFY is called, then go to an error status if
       VERIFY(time_domain) = false.
    2. If the given time_domain is not in { DATE, YEAR, MONTH, DAY, HOUR,
       MINUTE, SECOND }, then prompt for the definition of NOW.
    3. Update the relation TIME_DOM_REL to maintain the given time domain.
END
```

B.3 OSQL_INCOMP Package and Its Operations

```
PACKAGE OSQL_INCOMP
PARAMETER:
    VARCHAR: ext_att, incomplete_domain, ext_relation, ext_val, predicate
    BOOL: bool_val
    REL: result_relation
    PUB  COMPLETE_VAL(ext_relation, ext_att) RETURN result_relation
    PUB  PARTIAL_VAL(ext_relation, ext_att) RETURN result_relation
    PUB  DNE_VAL(ext_relation, ext_att) RETURN result_relation
    PUB  NI_VAL(ext_relation, ext_att) RETURN result_relation
    PUB  UNK_VAL(ext_relation, ext_att) RETURN result_relation
    PUB  MORE_INFO(ext_att,ext_val) RETURN predicate
    PUB  LESS_INFO(ext_att,ext_val) RETURN predicate
    IDENTIFY() RETURN
```

```
   VERIFY(incomplete_domain) RETURN bool_val
ENFORCEMENT:
   ENFORCE_INIT()
END PACKAGE



PUB COMPLETE_VAL(ext_relation, ext_att)
DEFINE
   SELECT (*) (*) FROM ext_relation
   WHERE (ext_att > 'UNK' WITHIN incomplete_domain)
RETURN result_relation

PUB PARTIAL_VAL(ext_relation, ext_att)
DEFINE
   SELECT (*) (*) FROM ext_relation
   WHERE ext_att = 'DNE' OR (ext_att <= 'UNK' WITHIN
   incomplete_domain)
RETURN result_relation

PUB DNE_VAL(ext_relation, ext_att)
DEFINE
   SELECT (*) (*) FROM ext_relation
   WHERE ext_att = 'DNE'
RETURN result_relation

PUB NI_VAL(ext_relation, ext_att)
DEFINE
   SELECT (*) (*) FROM ext_relation
   WHERE ext_att = 'NI'
RETURN result_relation

PUB UNK_VAL(ext_relation, ext_att)
DEFINE
   SELECT (*) (*) FROM ext_relation
   WHERE ext_att = 'UNK'
RETURN result_relation

PUB MORE_INFO(ext_att, ext_val)
DEFINE
   SELECT '(ext_att > ext_val WITHIN incomplete_domain)'
RETURN predicate

PUB LESS_INFO(ext_att, ext_val)
DEFINE
   SELECT '(ext_att < ext_val WITHIN incomplete_domain)'
RETURN predicate
```

```
PUB IDENTIFY()
DEFINE
    CREATE DOMAIN incomplete_domain AS INCOMP
RETURN

VERIFY()
DEFINE AS PROG incomp.verify
    1. Check that the null values UNK, NI and DNE are included in the
       incomplete_domain.
    2. Check that NI < UNK, NI < DNE and UNK < all values except those null
       values.
RETURN bool_val

ENFORCE_INIT()
DEFINE AS PROG incomp.enforce_init
    1. IDENTIFY().
    2. If VERIFY() = false, then go to an error_status.
END
```

B.4 OSQL_FUZZY Package and Its Operations

```
PACKAGE OSQL_FUZZY
PARAMETER:
    VARCHAR: fuzzy_domain, ext_att, predicate
    INT: order
    BOOL: bool_val
    REL: result_relation
FUNCTION:
    PUB  IDENTIFY(fuzzy_domain) RETURN
    PUB  IMPOSE_FUZZY(ext_att, fuzzy_domain) RETURN predicate
    PUB  ORDER_FUZZY(fuzzy_domain, order) RETURN
    PUB  LIST_REQ() RETURN result_relation
    VERIFY(fuzzy_domain) it RETURN bool_val
ENFORCEMENT:
    ENFORCE_INIT()
    ENFORCE_IDENTIFY()
    ENFORCE_IMPOSE()
END PACKAGE


PUB IDENTIFY(fuzzy_domain)
REL: REQ_DICT
DEFINE
    INSERT INTO REQ_DICT  VALUES (fuzzy_domain, 1)
RETURN

PUB IMPOSE_FUZZY(ext_att, fuzzy_domain)
```

```
REL: REQ_DICT
DEFINE
    SELECT '(ext_att WITHIN fuzzy_domain)'
RETURN predicate

PUB ORDER_FUZZY(fuzzy_domain, order)
REL: REQ_DICT
DEFINE
    UPDATE REQ_DICT SET PRIORITY = order
    WHERE FUZZY_REQ = fuzzy_domain
RETURN

PUB LIST_REQ()
REL: REQ_DICT
DEFINE
    SELECT (*) (*) FROM REQ_DICT
RETURN result_relation

VERIFY(fuzzy_domain)
DEFINE AS PROG fuzzy.verify
    1. Check that REQ_DICT does not contain the same name of the given fuzzy
       domain.
    2. Check that the value of PRIORITY in REQ_DICT is a positive integer.
RETURN

ENFORCE_INIT()
DEFINE
    DELETE FROM REQ_DICT
END

ENFORCE_IDENTIFY()
DEFINE AS PROG fuzzy.enforce_identify
    1. If the function IDENTIFY is called, then go to an error status if
    VERIFY(fuzzy_domain) = false.
END

ENFORCE_IMPOSE()
DEFINE AS PROG fuzzy.enforce_impose
    1. If the function IMPOSE is called, then save the result in a specified list and
       assign a number to it according the priorities of the defined fuzzy domains.
    2. Arrange the list according to the number assigned.
    3. Replace the existing attribute list with the extended attribute list.
END
```

B.5 OSQL_SPACE Package and Its Operations

```
PACKAGE OSQL_SPACE
PARAMETER:
    VARCHAR: space_domain, ext_relation, region_parameter, x_comp, y_comp,
       min_vertex_1, max_vertex_1, min_vertex_2, max_vertex_2,
       ext_domain, display_predicate, topological_predicate
    INT: perimeter, area
    BOOL: bool_val
    REL: result_relation
FUNCTION:
    PUB  IDENTIFY(ext_domain) RETURN
    PUB  DISJOINT(min_vertex_1, max_vertex_1, min_vertex_2, max_vertex_2)
            RETURN topological_predicate
    PUB  OVERLAP(min_vertex_1, max_vertex_1, min_vertex_2, max_vertex_2)
            RETURN topological_predicate
    PUB  MEET(min_vertex_1, max_vertex_1, min_vertex_2, max_vertex_2)
            RETURN topological_predicate
    PUB  CONTAIN(min_vertex_1, max_vertex_1, min_vertex_2, max_vertex_2)
            RETURN topological_predicate
    PUB  EAST(ext_relation, min_vertex_1, max_vertex_1) RETURN result_relation
    PUB  WEST(ext_relation, min_vertex_1, max_vertex_1) RETURN result_relation
    PUB  NORTH(ext_relation, min_vertex_1, max_vertex_1) RETURN result_relation
    PUB  SOUTH(ext_relation, min_vertex_1, max_vertex_1) RETURN result_relation
    PUB  PERI(min_vertex_1, max_vertex_1) RETURN perimeter
    PUB  AREA(min_vertex_1, max_vertex_1) RETURN area
    PUB  PICK_REGION() RETURN region_parameter
    PUB  SET_DISPLAY(colour, pattern, mode) RETURN
    PUB  BOUNDARY(ext_relation) RETURN
    PUB  WHOLE(ext_relation) RETURN
    XCOMP(min_vertex_1) RETURN x_comp
    YCOMP(min_vertex_1) RETURN y_comp
    VERIFY(space_domain) RETURN bool_val
ENFORCEMENT:
    ENFORCE_INIT()
    ENFORCE_DISPLAY()
END PACKAGE
```

```
PUB IDENTIFY(ext_domain)
DEFINE
    CREATE DOMAIN space_domain AS ext_domain
RETURN

PUB DISJOINT(min_vertex_1, max_vertex_1, min_vertex_2, max_vertex_2)
DEFINE
    SELECT 'XCOMP(max_vertex_1) < XCOMP(min_vertex_2) OR
    XCOMP(max_vertex_2) < XCOMP(min_vertex_1) OR
    YCOMP(max_vertex_1) < YCOMP(min_vertex_2) OR
    YCOMP(max_vertex_2) < YCOMP(min_vertex_1)'
```

```
RETURN topological_predicate

PUB OVERLAP(min_vertex_1, max_vertex_1, min_vertex_2, max_vertex_2)
DEFINE
    SELECT 'XCOMP(max_vertex_1) > XCOMP(min_vertex_2) AND
    XCOMP(max_vertex_2) > XCOMP(min_vertex_1) AND
    YCOMP(max_vertex_1) > YCOMP(min_vertex_2) AND
    YCOMP(max_vertex_2) > YCOMP(min_vertex_1)'
RETURN topological_predicate

PUB MEET(min_vertex_1, max_vertex_1, min_vertex_2, max_vertex_2)
DEFINE
    SELECT
    '((XCOMP(max_vertex_1) = XCOMP(min_vertex_2) OR
    XCOMP(max_vertex_2) = XCOMP(min_vertex_1)) AND
    YCOMP(max_vertex_1) > YCOMP(min_vertex_2) AND
    YCOMP(max_vertex_2) > YCOMP(min_vertex_1))
    OR
    ((YCOMP(max_vertex_1) = YCOMP(min_vertex_2) OR
    YCOMP(max_vertex_2) = YCOMP(min_vertex_1)) AND
    XCOMP(max_vertex_1) > XCOMP(min_vertex_2) AND
    XCOMP(max_vertex_2) > XCOMP(min_vertex_1))'
RETURN topological_predicate

PUB CONTAIN(min_vertex_1, max_vertex_1, min_vertex_2, max_vertex_2)
DEFINE
    SELECT
    'XCOMP(max_vertex_1) >= XCOMP(max_vertex_2) AND
    XCOMP(min_vertex_1) <= XCOMP(min_vertex_2) AND
    YCOMP(max_vertex_1) >= YCOMP(max_vertex_2) AND
    YCOMP(min_vertex_1) <= YCOMP(min_vertex_2)'
RETURN topological_predicate

PUB EAST(ext_relation, min_vertex_1, max_vertex_1)
DEFINE
    SELECT (*) (*) FROM ext_relation
    WHERE XCOMP(MAX_VERTEX) <= XCOMP(min_vertex_1)
RETURN result_relation

PUB WEST(ext_relation, min_vertex_1, max_vertex_1)
DEFINE
    SELECT (*) (*) FROM ext_relation
    WHERE XCOMP(MIN_VERTEX) >= XCOMP(max_vertex_1)
RETURN result_relation

PUB SOUTH(ext_relation, min_vertex_1, max_vertex_1)
DEFINE
    SELECT (*) (*) FROM ext_relation
    WHERE YCOMP(MAX_VERTEX) <= YCOMP(min_vertex_1)
RETURN result_relation

PUB NORTH(ext_relation, min_vertex_1, max_vertex_1)
DEFINE
    SELECT (*) (*) FROM ext_relation
    WHERE YCOMP(MIN_VERTEX) >= YCOMP(max_vertex_1)
RETURN result_relation

PUB  PERI(min_vertex_1, max_vertex_1)
DEFINE
    SELECT 2 × (ABS(XCOMP(min_vertex_1) - XCOMP(max_vertex_1)) +
    ABS(YCOMP(min_vertex_1) - YCOMP(max_vertex_1)))
RETURN perimeter

PUB  AREA(min_vertex_1, max_vertex_1) DEFINE
    SELECT ABS(XCOMP(min_vertex_1) - XCOMP(max_vertex_1)) ×
    (YCOMP(min_vertex_1) - YCOMP(max_vertex_1)))
RETURN area

PUB  PICK_REGION()
DEFINE AS PROG space.pick
    1. Communicate with the mouse driver to get the spatial attributes of the pointed region.
    2. Return the spatial attributes of the region in the correct format.
RETURN region_parameter

PUB  SET_DISPLAY(colour, pattern, mode)
DEFINE AS PROG space.display
    1. Set the visual variables of the display environment according
       to the given parameters colour, pattern and mode.
    2. Re-activate the window to display.
RETURN

PUB  BOUNDARY(ext_relation)
DEFINE AS PROG space.boundary
    1. Draw the horizontal lines on the screen whose horizontal ranges of pixel are to be
       bounded by XCOMP(MIN_VERTEX) and XCOMP(MAX_VERTEX) and the vertical
       distances of all pixel are YCOMP(MIN_VERTEX) or YCOMP(MAX_VERTEX).
    2. Draw the vertical lines on the screen whose vertical ranges of pixel are to be
       bounded by YCOMP(MIN_VERTEX) and YCOMP(MAX_VERTEX)) and the horizontal
       distances of all pixel are XCOMP(MIN_VERTEX) or XCOMP(MAX_VERTEX).
RETURN

PUB  WHOLE(ext_relation)
DEFINE AS PROG space.whole
    1. Display all the points on the screen whose horizontal ranges of pixel are to be bounded by
       XCOMP(MIN_VERTEX) and XCOMP(MAX_VERTEX) and whose vertical ranges of
       pixel are to be bounded by YCOMP(MIN_VERTEX) and YCOMP(MAX_VERTEX).
RETURN
```

```
XCOMP(min_vertex_1)
DEFINE AS PROG space.xcomp
   1. Strip off all symbols except the X-coordinate of the spatial attribute min_vertex_1.
   2. Convert the result into an integer.
RETURN

YCOMP(min_vertex_1)
DEFINE AS PROG space.ycomp
   1. Strip off all symbols except the Y-coordinate of the spatial attribute min_vertex_1.
   2. Convert the result into an integer.
RETURN

VERIFY(space_domain)
DEFINE AS PROG space.verify
   1. Check that the space_domain is 2DSPACE.
   2. Check that the space_domain is finite and X and Y components of all elements in a
      space_domain are linearly ordered.
RETURN bool_val

ENFORCE_INIT()
DEFINE AS PROG space.enforce_init
   1. IDENTIFY(2DSPACE).
   2. ENFORCE_DISPLAY().
END

ENFORCE_DISPLAY()
DEFINE AS PROG space.enforce_display
   1. Set all the necessary default parameters for display environment such as the visual
      variables in graphical presentation, the scale of the drawing and the resolution levels.
   2. Activate the display window.
END
```

Appendix C

Survey Documents

C.1 A Mini-Manual for OSQL

C.1.1 Introduction to OSQL

Current relational database management systems (RDMSs) are based on the Codd's relational data model and their data languages are specified by the SQL standard. We extend SQL to be Ordered SQL (OSQL) in order to provide the facility of user-defined orderings over data domains in addition to the standard domain orderings such as alphabetical ordering over a domain of strings and numerical ordering over a domain of numbers. For example, the semantics of the comparison EMPLOYEE_NAME < 'Wilfred' meaning the subordinates of Wilfred, can be captured in OSQL. Queries in OSQL are formulated in essentially the same way as using standard SQL.

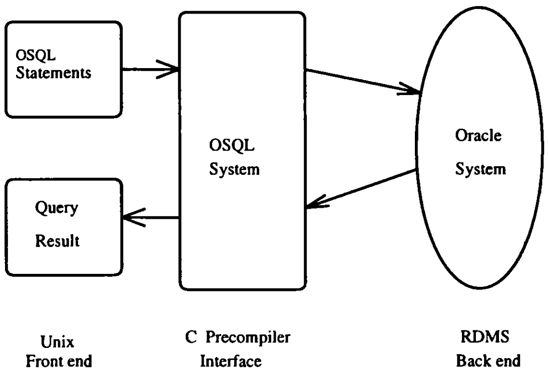


Figure C.1: Architecture of the OSQL system

In Figure C.1, we show our design of the system architecture, which allows OSQL statements to be entered via the front end unix interface, and then the OSQL precompiler generates a corresponding program consisting of a sequence of Oracle statements, which is then piped into the back end Oracle server for execution.

C.1.2 Connecting to OSQL System and Disconnecting from it

Firstly, you must make sure that you have no problem in connecting to Oracle7, i.e. you have already got an Oracle account and have successfully logged on to the system.

Secondly, you should type the following on your terminal(or add this in your .uclcs-csh-options file):

```
set path = ($path /cs/academic/phd0/violet/siuhungn/OSQL/)
```

This command set the path of the OSQL system. Now you can get into the OSQL interface by typing the command **osql**. You will be asked for your user name and then for the password of your **Oracle Account** as follows.

```
************************************************************
*       OSQL version 1.0 designed by Mr Wilfred NG         *
*                                                          *
*           Type 'quit' if you want to exit.               *
*                                                          *
*     IMPORTANT: ANY statement SHOULD be ended with ';'    *
*                                                          *
************************************************************
```

```
Enter user-name: ??????
Enter password: ??????
```

Owing to some performance problem in our Oracle Server, there may be a noticeable pause after typing your password or your OSQL statement. Please be patient. A prompt that looks like the following will appear after your account details have been verified:

```
Connected to ORACLE as user ??????
OSQL>
```

To get out of OSQL system just type **quit** as in Oracle at any time when you have this prompt.

```
OSQL>quit
Bye! Bye ! Have a good day, ??????.
```

C.1.3 Editing Queries

You must remember to terminate the query with a **semicolon**. This tells the OSQL system to start evaluating the current query and ignore the old information in the query buffer (a useful tip if you want to get rid of any mistyped query in OSQL).

A typical query and its result might look like this:

```
OSQL>select (salary) (1) from staff;
============
SALARY
============
       10000


Query returned 1 row.
```

You can use a source file to correct errors or modify your query statement. Having finished your modification, you can run the contents of your file by the OSQL command:

```
OSQL>start <some filename>
```

At the moment there are two limitations in our extension. The OSQL system does **not** accept nested queries and it assumes that all OSQL keywords are in **lower case**.

C.1.4 Using the OSQL SELECT Statement

In this section we introduce the select statement in OSQL. You are strongly encouraged to try the queries Q1 to Q15 while you are reading this section. As in Oracle SQL all data retrieval in OSQL is done with the select command. The extended form of the statement is:

```
select (attribute list) (tuple list)
from  relation list
where (comparison clause)
```

Note that in the select statement, OSQL has three extensions as follows:

1. Extension of an *attribute list*: An attribute list in OSQL is a list of attributes similar to the usual one, except that it provides us with an option that an attribute can be associated within a semantic domain by the syntax *attribute name within domain name*. The purpose of declaring a *within* clause is to override the system ordering with the semantic ordering of the semantic domain specified by the domain name. When the *within* clause is missing then the system ordering will be assumed. Note also that the attribute list should be enclosed within a pair of brackets. Let us examine at the following OSQL statements:

   ```
   Q1. select (name, salary) (*) from staff;
   Q2. select (salary, name) (*) from staff;
   Q3. select ((name within emp_exp), salary) (*) from staff;
   ```

   The attribute list of the query Q1 is (name, salary), and thus tuples in the output answer are ordered alphabetically by name first and then ordered numerically by salary. Therefore the ordering of tuples is, in general, different to that of query Q2, whose list is specified as (salary, name), since the output of Q2 is ordered by salary first and then by name. It will also be different from that of Q3 whose list is ((name within emp_exp), salary), where the ordering of name is given by the semantic domain emp_exp representing employee experience in a company. Check these queries for yourself in OSQL.

2. Extension of *tuple list*: A tuple level, which is a set of positive numbers, with the usual numerical ordering, can also be written in some short forms. Since a set of tuples in a linearly ordered relation $r = \{t_1, \dots, t_n\}$ is isomorphic to a set of linearly ordered tuples, we interpret each number i in a tuple level as an index to the position of the tuple t_i, where $i = 1, \dots, n$ and $t_1 < \cdots < t_n$. Let us consider the following example.

```
Q4. select (salary) (4) from staff;
Q5. select (salary) (2..5) desc from staff;
```

Query Q4 returns the fourth lowest salary and query Q5 returns the second to fifth highest salaries in the staff relation. The keyword *desc* is used to reverse the ordering (i.e. maximum first) of the relation You may try Q5 again without using the keyword *desc* to examine the difference fro yourself.

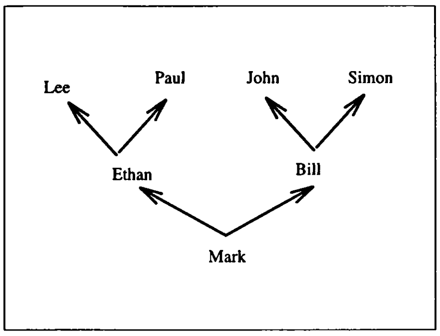
An interesting situation to consider is when the output of a relation is partially ordered as a tree, having levels $\{l_1, \ldots, l_m\}$. In such a case we choose to interpret each number j in a tuple level as an index to a corresponding tree level l_j, where $j = 1, \ldots, m$ and $l_1 < \cdots < l_m$. Hence, a user can specify the retrieve of *all* the tuples or *any* one of the tuples in a specified level l_j. We note that in the case of a linearly ordered relation, the choice of using *all* or *any* has the same effect on the output, since there is only one tuple in each level.

3. Extension of *comparison*: The meaning of the usual comparators $<, >, <=, >=$ is extended to include semantic comparison. A typical form of a semantic comparison is defined by the syntax *attribute comparator attribute within semantic domain.*

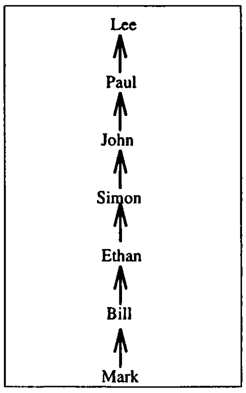
   Without the optional *within* clause, the comparison is just the conventional one and is based on the relevant system ordering. As an example of a semantic ordering, the comparison (name > 'Bill' within emp_rank) returns all names of staff members who are more senior Bill specified by the semantic domain emp_rank, and the comparison name > 'Bill' returns all names of staff members alphabetically greater than Bill.

To summarise, the ordering of tuples in an output relation depends on two factors: firstly, on the ordering of domains of individual attributes, and secondly on the order of the attributes in an attribute list.

The following is a more detailed example using two semantic domains emp_rank and emp_exp as shown in Figure C.2. The first one represents the position of members of staff, for example Mark (at level 3) is more senior than Bill (at level 1), and the second one represents the experience of the staff, for example Bill is more experienced than Lee. Note that the domain emp_rank is tree-structured whereas the domain emp_exp is linearly ordered.



(a)



(b)

Figure C.2: The semantic domains (a) emp_rank and (b) emp_exp

Q6. Find the first and the fourth lowest salaries in the staff relation.

```
        select (salary) (1,4)
        from staff;
```

Q7. Find the highest salary in the staff relation.

```
        select (salary) (last)
        from staff;
```

or equivalently,

```
        select (salary) (1) desc
        from staff;
```

Q8. Find 4 names of staff members with salary greater than 15000.

```
           select (name) (1..4)
           from staff where salary > 15000;
```

Q9. Find a staff member at the most senior level (note the brackets).

```
        select ((name within emp_rank)) (1) desc from staff;
```

or equivalently,

```
        select ((name within emp_rank)) (last) from staff;
```

Q10. Find the name of a staff member at the most junior level.

```
        select ((name within emp_rank)) (1) from staff;
```

Q11. Find all the names of staff at the most junior level (compare this with Q10).

```
        select ((name within emp_rank)) all (1) from staff;
```

Q12. List the names and salaries of all staff members in alphabetical order of names.

```
        select (name, salary) (*) from staff;
```

Q13. List the names and salaries of staff members in order of their experiences (compare this with Q12).

```
        select ((name within emp_exp), salary) (*) from staff;
```

Q14. Find the record of all the staff members who are more senior than Ethan.

```
           select (*) (*)
           from staff
           where (name > 'Ethan' within emp_rank);
```

```
Q15. Find the record of all the staff members who are more experienced than
     Ethan (compare this with Q14).
           select (*) (*)
           from staff
           where (name > 'Ethan' within emp_exp);
```

C.1.5 Language Specification

In this section we give the BNF notation for the OSQL select statement.

Conventions:

- Key words are indicated by lowercase italicized characters.
- Non-terminal symbols are enclosed with " <> ".
- Alternatives are separated by "|". If only one of the symbols is to be chosen out of several alternatives, then we enclose them with "{ }".
- Optional clauses are enclosed with "[ ]".
- Default keywords are underlined.
- A positive number begins with #.
- ... at the end if a subclause indicates that it may be repeated.

select <attribute-list> [{*any* | *all*}] <tuple-list> [{*asc* | *desc*}] *from* <relation-list> [*where* <condition>]

<attribute-list> ::= (extended-attribute [,extended-attribute]...)

<extended-attribute> ::=
{attribute-name | (attribute-name *within* domain-name | ∗)}

<tuple-list> ::= ({#n [, #n]) | *last* | #n1..#n2 | ∗})

<comparison> ::=
<attribute-name | value> <comparator> <{attribute-name | value}> [*within* domain-name]

<comparator> ::= {<|>|>=|<=|<>}

C.1.6 The Staff Table

```
========== ============ ============
NAME       NI_NO        SALARY
========== ============ ============
Bill              27891        12000
Ethan             32877        29000
John              10982        14000
Lee               34589        25000
```

```
Mark                 67001        30000
Paul                 23789        23000
Simon                32112        10000
```

End-of-Manual

C.2 Experiment Sheet on OSQL

Task I: The required database is the frame_parts and stock as shown on the last page. Answer question 3 in the questionnaire while you are doing this task.

1. List all records in the table frame_parts sorted by part_name.
2. What is the cost of the cheapest part?
3. What is the cost of the most expensive part?
4. What is the cost of the third and fifth cheapest part?
5. List all records in the table frame_parts whose cost is under 200, sorted by cost.
6. Some part construction is dependent on the availability of other parts as shown in Figure C.3. For example, the part gizmo is dependent on the parts gear and also the part cam is dependent on the parts jack. I have created a semantic domain called part_hie to represent the hierarchy of different parts.

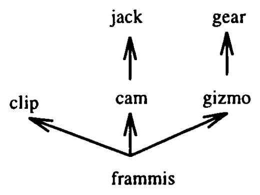
   

   Figure C.3: Relationship amongst parts in frame_parts

   List cities of the parts which are dependent on the part gear directly or indirectly (You cannot use any part name explicitly apart from gear in your query).
7. List the name of one part on which frammis is directly dependent (i.e. one level under).
8. List the names of all parts on which frammis are directly dependent.
9. Look into the stock table in which there is some missing information in records. We classify the missing information into three symbols of type UNKnown whose meaning is given as follows:

   UNK1: No information is available for this value.
   UNK2: Value does not exist.
   UNK3: Value exists but is not disclosed for some reasons.

   We use the notion of "more informative" as the following ordering:

   UNK1 < UNK2 < UNK3 < other values

   In other words, UNK3 is more informative than UNK2 and so on. I have already created a semantic domain called incomplete_domain to represent this ordering.

   List the name and country of all items in which they must more informative than or equally informative to UNK3.

Task II: Now you can quit the OSQL system. Try and answer the questions 1 to 9 of Task I in Oracle SQL (type sqlplus). If you find it difficult, you have to explain why (in principle all queries can be formulated in SQL). Answer question 4 in the questionnaire while you are doing this task.

Task III: Complete the questionnaire.

1. The table frame_parts:

```
========== ====== ==========
CITY       COST   PART_NAME
========== ====== ==========
Bath         1000 gizmo
Birmingham   3000 cam
Blackpool     100 screw C
London         20 plug B
London         50 screw A
London         50 screw B
London        100 plug A
London       5000 frammis
Manchester    500 jack
Paris         200 screw D
Sussex        100 plug C
York          150 nut1
York          150 nut2
York          150 nut3
York          500 gear
York         1000 clip
```

2. The table stock:

```
=============== ========== ========== ======
ITEM_NAME       COUNTRY    COLOUR     PRICE
=============== ========== ========== ======
TV_stand        UNK1       UNK1          200
book_case A     China      UNK1          200
book_case B     China      Black         200
buffet_unit     UNK2       UNK1         2000
coffee_table    Italy      White        1000
dining_table    Japan      Yellow       1500
dressing_table  UNK3       UNK2         1000
filing_cabinet  England    Grey         1000
folding_chair A France     UNK1          500
folding_chair B France     Brown         200
folding_chair C China      Yellow        400
```

futon	Japan	UNK3	500
soft_unit	UNK3	Black	5000
wardrobe	England	Yellow	3000

End-of-sheet

C.3 Questionnaire on Using OSQL

Pleas tick the boxes or fill in the blanks in the following questions as appropriate.

1. What is your experience of using SQL?

   ☐ I learned SQL on this course.
   ☐ Less than 2 years.
   ☐ 2 to 4 years.
   ☐ more than 4 years.

2. What programming languages (including database query languages and packages) do you know apart from SQL?

3. Number of attempts in formulating OSQL queries in order to obtain the expected query result for Task I of experiment sheet.

Questions	Less than 3 times	3 to 6 times	More than 6 times	Unsuccessful
1				
2				
3				
4				
5				
6				
7				
8				
9				

4. Number of attempts in formulating equivalent SQL statements in order to obtain the expected query result for Task II of the experiment sheet.

Questions	Less than 3 times	3 to 6 times	More than 6 times	Unsuccessful
1				
2				
3				
4				
5				
6				
7				
8				
9				

5. There are three extensions in the select statement of OSQL, namely in the attribute list, tuple list and comparison clause as given below:

```
select (attribute list) (tuple list)
from  relation list
where (comparison)
```

Please comment on the usefulness and the difficulties in using these extensions in the Tasks I and II of the experiment sheet (scale 1 means least and scale 5 means most).

Extension	Usefulness					Difficulty				
	1	2	3	4	5	1	2	3	4	5
Attribute list										
Tuple list										
Comparison										

6. Please give us your general comments on OSQL.

End-of-questionnaire
Thank you very much!

Appendix D

Sample Code from the Implementation of OSQL

D.1 Part of the Code from Dynamic.pc

```

#include <stdio.h>
#include <string.h>
#include <ctype.h>

#define MAX_ITEMS         40
#define MAX_VNAME_LEN     30
#define MAX_INAME_LEN     30
#define INPUTFILE "answer.sql"

EXEC SQL BEGIN DECLARE SECTION;
    VARCHAR sql_statement[2048];
    char *username;
    char *password;
EXEC SQL END DECLARE SECTION;
EXEC SQL INCLUDE sqlca;
EXEC SQL INCLUDE oraca;
EXEC SQL INCLUDE sqlda;
EXEC ORACLE OPTION (ORACA = YES);
EXEC ORACLE OPTION (RELEASE_CURSOR=YES);

SQLDA *select_dp;
extern char user_name[];
extern char user_password[];
extern SQLDA *sqlald();
extern void sqlprc();
extern void sqlnul();
FILE *sqlfile;

open_infile()
```

```
{

     sqlfile = fopen(INPUTFILE,"r+");
     if (!sqlfile)
     {
      printf("Warning: file error! could not find answer.sql file");
      exit(1);
     }
     return;
}

close_infile()
{

fclose(sqlfile);

}

execute_sql()
{
    EXEC SQL WHENEVER SQLERROR DO sql_error();
    EXEC SQL PREPARE S FROM :sql_statement;
    EXEC SQL DECLARE C CURSOR FOR S;
    EXEC SQL OPEN C;
    process_statement();
    EXEC SQL CLOSE C;
    EXEC SQL COMMIT WORK;
    return;
}

connect_oracle()
{
        username = user_name;
        password = user_password;
        EXEC SQL WHENEVER SQLERROR DO connect_error();
        EXEC SQL CONNECT :username IDENTIFIED BY :password;
        printf("\nConnected to ORACLE as user %s\n",username);
        return;
}

initialize_desp()
{
        EXEC SQL WHENEVER SQLERROR DO sql_error();
        select_dp =
        sqlald (MAX_ITEMS, MAX_VNAME_LEN, MAX_INAME_LEN);
        select_dp->N = MAX_ITEMS;
        return;
}
```

D.2 Part of the Code from Osql.l

```
void yyerror(char *s);
%}
%s OSQL
%%


%{
              /* control command literals
<OSQL>^sql\n { BEGIN INITIAL; osql_flag = 0; printf("SQL>");
                Initialization(); return PASS ;} ; */
%}

^[ ]*quit\n          { wrap_up();};

^help\n |
"?"\n                 {command_help();};

<OSQL>start |
<OSQL>"@"             { INFILE_MODE ;};

%{
              /* literals*/
%}


<OSQL>select     {save_sql(yytext,"SELECT");
               SELECT_MODE; ATT_MODE; return SELECT ;} ;
<OSQL>from       TOK(FROM) ;
<OSQL>where {save_sql(yytext,"WHERE"); COND_MODE; return WHERE;} ;
<OSQL>last       TOK(LAST) ;

<OSQL>create     {save_sql(yytext,"CREATE"); CREATE_MODE;
                return CREATE ;};
<OSQL>domain     {save_sql(yytext,"DOMAIN");
               DOMAIN_MODE; return DOMAIN ;};
<OSQL>order      {AGG_MODE; save_sql(yytext,"ORDER");
               ORDER_MODE; return ORDER ;};
<OSQL>table       {save_sql(yytext,"TABLE");
               TABLE_MODE; return TABLE ;};
<OSQL>drop        {save_sql(yytext,"DROP"); DROP_MODE;
                return DROP ;};
<OSQL>group        {AGG_MODE; GROUP_MODE; save_sql(yytext,"GROUP");
               return GROUP ;};
<OSQL>having        {save_sql(yytext,"HAVING"); AGG_MODE;
                return HAVING ;};
<OSQL>alter        {save_sql(yytext,"ALTER"); ALTER_MODE;
                return ALTER ;};
```

```
<OSQL>tuple        {save_sql(yytext,"TUPLE"); TUP_MODE;
               return TUPLE ;};
<OSQL>other        {if ((other_count += 2) >= 4)
               yyerror("More than one 'other'");
              save_sql(yytext,"OTHER"); return OTHER;};

<OSQL>UNO        {save_sql(yytext,"UNO"); return UNO;};

<OSQL>ABO {save_sql(yytext,"ABO"); ++other_count; return ABO;};

<OSQL>modify     {save_sql(yytext,"MODIFY"); return MODIFY;};

```

D.3 Part of the Code from Osql.y

```
%{
/* to recongnise the pattern of OSQL expression*/
#include<stdio.h>
%}
%token SELECT FROM WHERE LAST NAME DOM INTNUMBER POSINTNUM ASC DESC
       STRING ALL COMPARATOR WITHIN ANY TABLE OTHER ABO UNO
       CREATE DOMAIN ORDER AS DATANUM DATACHAR TUPLE EVERY PASS
       DROP CONJUNCTION BY GROUP HAVING ALTER ADD DELETE MODIFY

%%
pass_and_statement: pass_statement statement
                                 |
                        pass_statement
                                 |
                           statement
                                 ;

statement: select_from_where_statement
                |
           create_domain_statement
                |
           create_table_statement
                |
           drop_table_statement
                |
           drop_domain_statement
                |
           alter_table_statement
                |
           alter_domain_statement
                |
           delete_from_where_statement
                |
            error PASS { YYABORT;}
```

```
                ;


select_from_where_statement: SELECT attributes_expression
           tuples_expression_order
           FROM tables_expression
                |
           SELECT attributes_expression tuples_expression_order
           FROM tables_expression
           WHERE condition_list
                ;


attributes_expression:  '(' name_list ')'
                |       '(' EVERY ')'
                ;

tuples_expression_order: tuples_expression
                |       tuples_expression DESC
                |       tuples_expression ASC
                |       ALL tuples_expression
                |       ANY tuples_expression
                |       ALL tuples_expression DESC
                |       ANY tuples_expression DESC
                |       ALL tuples_expression ASC
                |       ANY tuples_expression ASC
                ;

tuples_expression: '(' number_list ')'
                |  '(' POSINTNUM'.''.'POSINTNUM ')'
                |  '(' POSINTNUM'.''.'LAST ')'
                |   '(' EVERY ')'
                |   '('LAST')'
                ;

name_list: name_list ',' name_dom
                |       name_dom
                ;

name_dom:       NAME
                |
                '(' NAME WITHIN NAME ')'
                ;
```