Vol. 12, No. 4

The Development of Ordered SQL Packages to Support Data Warehousing

WILFRED NG
Hong Kong University of Science and Technology, Hong Kong

MARK LEVENE
University of London, UK

Data warehousing is a corporate strategy that needs to integrate information from several sources of separately developed Database Management Systems (DBMSs). A future DBMS of a data warehouse should provide adequate facilities to manage a wide range of information arising from such integration. We propose that the capabilities of database languages should be enhanced to manipulate user-defined data orderings, since business queries in an enterprise usually involve order. We extend the relational model to incorporate partial orderings into data domains and describe the ordered relational model. We have already defined and implemented a minimal extension of SQL, called OSQL, which allows querying over ordered relational databases. One of the important facilities provided by OSQL is that it allows users to capture the underlying semantics of the ordering of the data for a given application. Herein we demonstrate that OSQL aided with a package discipline can be an effective means to manage the inter-related operations and the underlying data domains of a wide range of advanced applications that are vital in data warehousing, such as temporal, incomplete and fuzzy information. We present the details of the generic operations arising from these applications in the form of three OSQL packages called: OSQL_TIME, OSQL_INCOMP and OSQL_FUZZY.

Data warehousing is a corporate strategy that addresses a broad range of decision support requirements such as querying information over its underlying databases and managing ordered data for the purpose of analysis. One of the main characteristics of data warehousing is that in order to build its foundation, it should consist of integrated data from several sources of separately developed information systems. The transmission of data relies on the network system which connects all these information systems. As a result, the integrated database has the following important features:

- **It involves huge amounts of historical data.**

  Data warehouse is described as a "subject-oriented, integrated, non-volatile, time variant" collection of data which is intended to support management decisions (Inmon, 1996). It is widely recognised that the underlying database in a data warehouse should capture transactions and snapshots in time in an efficient manner in order to carry out the activities of market forecast and strategic planning (McCabe & Grossman, 1996).

- **It is usually incomplete.**

  This is due to two main reasons. First, some sources of the databases may be incomplete in order to protect sensitive data or to improve the speed of the process of data downloading via a network. Second, it has been observed in Libkin (1995) that even if each source of the database is complete, the integrated database may still not be complete. Hence, incompleteness may show up in the integrated database or in the answer to users' queries.

- **It is mainly used for decision support in an enterprise.**

  However, many management professionals may not necessarily have good knowledge about the technical aspects of a data warehouse. As a result, their queries over the database are sometimes fuzzy in nature due to the ambiguity of natural languages. For example, they may ask to find the "best performed" shares in the Hong Kong stock market this month in order to carry out some share trading activities.

Many database researchers have recently recognised that ordering is inherent to the underlying structure of data in many database applications (Maier & Vance, 1993; Libkin, 1995; Buneman et al., 1997) including temporal information (Tansel et al., 1993), incomplete information (Codd, 1986) and fuzzy information (Buckles & Petry, 1982). However, current relational Database Management Systems (DBMSs) still confine the ordering of elements in data domains to only a few kinds of built-in orderings. SQL2 (or simply SQL) (Date, 1997), for instance, supports three kinds of orderings considered to be essential in practical utilisation: the *alphabetical ordering* over the domain of strings, the *numerical ordering* over the domain of numbers and the *chronological ordering* over the domain of dates (Date, 1990). Let us call these ordered domains *system domains* or alternatively, domains with *system ordering*.

With the advent of the *Internet* technology, there is strong evidence that the limited support for ordering provided by current relational DBMSs is inadequate for future commercial applications. For example, a large proportion of the useful business information available in global Web sites is available only in hypermedia format. Hypermedia information normally consists of a very large amount of image data and thus resolution is an effective means to manage the size of data domain element. We 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'}. This domain consists of five distinct levels of resolution and thus the users can select the appropriate level to save the transmission time for downloading a hypermedia document. However, the semantics of RESOLUTION_LEVEL cannot be captured by any one of the system orderings.

In order to alleviate the above-mentioned problems, we have extended SQL to Ordered SQL (OSQL) by providing the facility of user-defined orderings over data domains (Ng & Levene, 1997), which we refer to as *semantic orderings*. Queries in OSQL are formulated in essentially the same way as using standard SQL. We demonstrate this mode of querying with the following example, which shows how OSQL simplifies the specification of certain queries which might be useful in business decisions. We note that the following queries are not easy to formulate in SQL due to the fact that they must involve non-trivial use of aggregate functions and nesting (see Sections 25.1 and 26 in Celko (1995) and Section 9 in Pascal (2000)).

Example 1 In this example we assume that the attributes in their respective relation schemas are linearly ordered.

1. Get the third and sixth lowest share prices from a stock market.
   (Q_1) *SELECT* (SHARE_PRICE) (3,6) *FROM* STOCK_MARKET.
2. Get the names of exactly five participating banks from a syndicated loan record.
   (Q_2) *SELECT* (BANK_NAME) (1..5) *FROM* SYNDICATED_LOAN.
3. Get the names of all bosses of John.
   (Q_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_1) means that the third and sixth tuples, according to the order of SHARE_PRICE, are output and the clause (1..5) in the query (Q_2) means that the first to fifth tuples, according to the order of BANK_NAME, are output. The keyword *WITHIN* in the query (Q_3) specifies that the comparison EMPLOYEE_NAME > 'John' is interpreted according to semantic ordering of the domain EMP_RANK.

The usual way to tackle the above problems is to use a programming approach such as embedded SQL. However, as most data warehouses are built upon a *client-server* architecture, the programming approach has to pay the performance penalty in the *data extraction* process, if there are too many calls from the programming level to the relational level. In this respect, OSQL offers the advantage that it can help to relieve the burden of the bandwidth of a network system and the loads of *client processes*, if such kinds of queries can be performed in the *database server* instead of the client platform.

Herein we investigate the introduction of a package discipline into OSQL, which allows us to modularise a collection of generic operations on an ordered data domain. These operations can then be called from within OSQL whenever the package they belong to is loaded into the system. For example, the OSQL statement (Q_4) uses the function SNAPSHOT provided by the OSQL package OSQL_TIME, which returns the prices of shares of the temporal relation STOCK_MARKET in 1990.

(Q_4) *SELECT* (SHARE_PRICE) ($_*$) *FROM* SNAPSHOT(STOCK_MARKET, 1990).

The package discipline makes it easier to formulate queries relating to the underlying ordered domains of the package and allows us to extend OSQL with powerful operations, which enhance its applicability and expressiveness. 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 three advanced database applications described

Table 1: The Brief Description of Three OSQL Packages

Package Name	Brief Description
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.

in Table 1.

The use of packages is very popular and successful in many existing software systems such as *PL/SQL* in *Oracle* and most recently in *Latex2e* and *Java*. 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. For instance, a top-down design approach is adopted for the grouping of related operations in an OSQL package, within which constraints can be enforced and supported by a language construct called *enforcement*. Thus, the operations in an OSQL package can be controlled in a more coherent manner. OSQL packages can also hide the implementation details of the code of their operations. The database administrator has the flexibility to decide whether an operation should be *public* or *private*.

Related Research

A related approach is to use abstract data types to define domains and their associated operations, which can then be treated as an integral part of the data type. This approach is basically an object-oriented extension of the relational model, resulting from the strong trend of object-oriented programming in the 1980s. Examples of commercial products that conform to this approach are Illustra's DataBlades and IBM's Database Extenders. However, it is not clear that how the optimisation of programs can be carried out when using these systems if the code of the operations is introduced to the execution engine at run time. If the optimisation can be carried out at compile time, to our knowledge there has been very little research done on how these systems provide syntactic and semantic compatibilities with SQL.

The most recent version of SQL (SQL3 or SQL:1999) has the provision for a procedural extension of SQL (Melton, 1996), which allow users to define functions in abstract data types. However, the issue of ordering abstract data type instances in SQL3 is still unclear (Melton, 1996). Our work here can be employed as a useful reference point which explores the issue of incorporating order into SQL. We emphasise that our approach is novel. First, we regard partial ordering as a fundamental property of data which is captured explicitly in the ordered relational model. It results in more efficient operations than those using the programming approach to embed this property into an application program. Second, 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. Third, our approach provides maximum flexibility for users and allows the design of optimisation strategies for the execution engine of a relational DBMS.

The remainder of the paper is organised as follows. In the next section we briefly describe the ordered relational model, the query language OSQL and its package discipline. Then it follows the section which we describe in detail the contents and the uses of three OSQL packages for temporal (OSQL_TEMP), incomplete (OSQL_INCOMP) and fuzzy (OSQL_FUZZY) information. In the last section we conclude with discussion on the implementation issue of OSQL packages.

A PACKAGE DISCIPLINE FOR ORDERED DATABASES

In this section we briefly describe the ordered relational data model and its query language OSQL. Within this model, we demonstrate how OSQL packages can be applied to solve various problems that arise from many advanced applications.

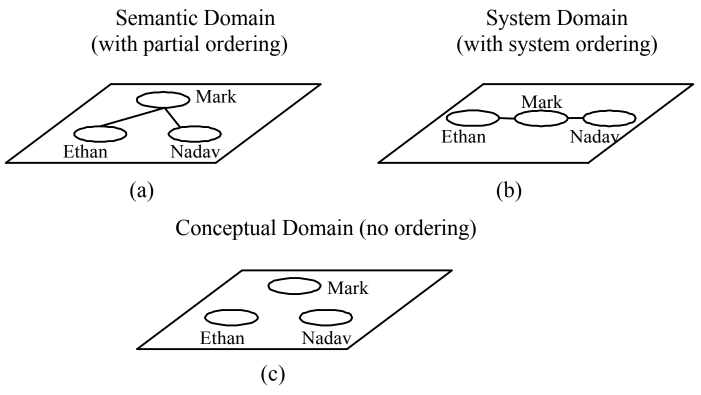
The Ordered Relational Model

We assume the reader is familiar with the relational model as presented in Ullman (1988) and Levene & Loizou (1999). A basic assumption of this model is that elements in a data domain have no explicit relations amongst them. In the ordered relational model, however, partial orderings (or simply orderings when no ambiguity arises) are included as an integral part of data domains. Without an explicit specification by the user, we assume that the domains of databases have the *system ordering* attached to them.

As an illustration we assume a domain consisting of three employee names: Ethan and Nadav being the subordinates of their boss Mark. Viewing this domain as a conceptual domain, all three elements are indistinguishable with respect to their ordering. On the other hand, viewing this domain as a system domain, the alphabetical ordering is imposed onto the conceptual domain resulting in a linear ordering of the three names. Finally, viewing this domain as a semantic domain, the boss-subordinate relationship can be explicitly captured. The three different views of this domain are depicted in the diagram shown in Figure 1.

An important notion in our model is that given a concep-

Figure 1: Domains with Different Kinds of Ordering



tual domain, apart from the system ordering assumption, we can declare one or more semantic orderings which override the default system ordering. Furthermore, the orderings of domains can be extended to tuples so that tuples in an ordered relation are ordered according to the *lexicographical ordering* of the domains associated with the attributes present in the underlying relational schema. Therefore, any change in the order of attributes in a relational schema may affect the order of tuples in an ordered relation. For the formalism of this model, readers may refer to the recent research presented in Ng (1999) and Ng, Levene & Trevor (2000).

OSQL

Ordered SQL (OSQL) is an extension of the Data Definition Language (DDL) and Data Manipulation Language (DML) of SQL for the ordered relational model. In addition to the extended DDL and DML, OSQL provides a Package Definition Language (PDL), which will be detailed later on. Herein we just describe the *SELECT* statement of the DML and the *CREATE DOMAIN* statement of the DDL; the sample of OSQL can be found in the Appendix and the detailed BNF for OSQL can be found in Appendix B in Ng (1998).

1. The DML of OSQL
SELECT < lists of attributes > [ANY | ALL] < levels of tuples > [ASC | DESC]
FROM < lists of ordered relations >
WHERE < extended predicates >

An *attribute list* above 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 semantic ordering 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 B1 in Ng (1998)). As a set of tuples in a linearly ordered relation $r = \{t_1,¼,t_n\}$ 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 $i = 1,¼,n$ and t_1 <L< t_n. In addition, a user can specify the retrieve of *ALL* the tuples or *ANY* one of the tuples in a specified level l_j when the output of a relation is partially ordered as a tree, having levels $\{l_1,¼,l_m\}$.

Following the FROM keyword is a comma separated 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:

< 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.

Example 2 Let us examine at the following OSQL statements:
(Q_5) *SELECT* (NAME, SALARY) ($_*$) *FROM* EMPLOYEE.
(Q_6) *SELECT* (SALARY, NAME) ($_*$) *FROM* EMPLOYEE.
(Q_7) *SELECT* ((NAME *WITHIN* EMP_RANK), SALARY) ($_*$) *FROM* EMPLOYEE.

Note that the ordering of tuples in an output relation depends on two factors: first on the ordering of domains of individual attributes, and second on the order of the attributes in an attribute list. The attribute list of the query (Q_5) is

Figure 2: An Employee Relation EMPLOYEE with Different Ordering

NAME	SALARY
Ethan	28K
Mark	27K
Nadav	28K

(a)

SALARY	NAME
27K	Mark
28K	Ethan
28K	Nadav

(b)

NAME	SALARY
Ethan	28K
Nadav	28K
Mark	27K

(c)

(NAME, SALARY), and thus tuples in the output answer are ordered by NAME first and only then by SALARY (see Figure 2(a)). Therefore the ordering of tuples is, in general, different to that of query (Q_6), whose list is specified as (SALARY, NAME), since the output of (Q_6) is ordered by SALARY first and then by NAME (see Figure 2(b)). It will also be different from that of (Q_7) whose list is ((NAME *WITHIN* EMP_RANK), SALARY), where the ordering of NAME is given by the semantic domain EMP_RANK shown in Figure 1 (see Figure 2(c)).

2. The DDL of OSQL

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 statement 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.

Example 3 The definition of the semantic domain shown in Figure 1(a) can be written as follows:
(Q_8) *CREATE DOMAIN* EMP_RANK *CHAR*(5) *ORDER AS* ('Ethan' < 'Mark', 'Nadav' < 'Mark').

For a large and complex domain, this syntax may be tedious. Thus OSQL provides a useful short forms {} and the keywords *OTHER* for those data items not mentioned explicitly to make the task of formulating queries easier (see Appendix B in Ng (1998) for detail). For instance, (Q_8) can be rewritten as follows:
(Q_9) *CREATE DOMAIN* EMP_RANK *CHAR*(5) *ORDER AS* ({'Nadav','Ethan'} < 'Mark').

Four major practical benefits of using OSQL in database applications can be summarised as follows: First, with few syntactical modifications to the basic form of standard SQL, OSQL provides us with new facilities that can be interfaced to existing relational DBMSs to compare attributes according to semantic orderings, in addition to the usual system orderings. Second, OSQL incorporates some of the suggestions put forward by Date (1990) to improve SQL-type query languages, mainly concerning the support of the wider use of "<" operator. Third, 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 a data warehouse. Fourth, partial ordering is a formal concept which has a simple interpretation in terms of real world entities. Due to this simplicity, OSQL can easily gain acceptance from a broad range of users.

Implementation of Ordered Domains

We now discuss two strategies in deploying an ordered domain. First, an ordered domain is implemented by using a conventional database system such as the Oracle DBMS. This strategy is attractive due to the known robustness and wide availability of conventional DBMSs. Another strategy is based on an object-oriented system such as IBM Smalltalk (Smith, 1994), which is an efficient programming language offering an *OrderedCollection* class to manipulate ordered data.

Using the first strategy the semantic domain EMP_RANK described in Figure 1 can be easily maintained by using a binary relation to represent the ordering. After executing the CREATE DOMAIN statement written as (Q_8), the OSQL system generate an internal relation called ORDERING_EMP_RANK with two attributes ORDERING_SMALL and ORDERING_LARGE to represent the semantic domain EMP_RANK.

There are still two possibilities to represent ORDERING_EMP_RANK. One possible way is to use *transitive reduction* as the representation of the semantic domain. In this method, the binary relation ORDERING_EMP_RANK, consisting of two attributes over ORDERING_SMALL and ORDERING_LARGE, implements the orderings between pairs of elements. 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, which essentially performs a closure operation.

Another way is to use the transitive closure as the representation of semantic ordering. This method has the advantage of minimising the cost of query execution time. 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 a tuple in the transitive reduction representation, then in the meantime it *implicitly* removes the ordering relationship between the two elements in the ordered pair. We also note that in this approach we have freedom to delete any tuple. In contrast, if we delete the same tuple in 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 a method.

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). We can also 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.

We now briefly discuss the implementation of a semantic domain based on an object-oriented (O-O) system; in this approach we can represent a given partial ordered domain as a set of *linear extensions* of the domain. Informally, the set of linear extensions representing a given ordered domain satisfies the criterion that it can precisely generate all the ordered pairs in the domain by imposing intersection on all linear extensions. An O-O system usually supports a rich set of linearly ordered types. For example, the programming language *Smalltalk* (Smith, 1994) provides two ordered classes called *OrderedCollection* and *SortedCollection*. When using the *OrderedCollection* class the ordering is determined by a sequence of the insertion and modification operations. When using the *SortedCollection* class users can formally state the sorting criterion by means of a *sort block*, which is a two-parameter Boolean-returning block for comparing successive ordered pair of data elements corresponding to a partially ordered set. The sort block can be specified explicitly at creation time, once the sort block is changed the entire collection is re-sorted according to a new sorted block.

Using OSQL in Advanced Applications

We now show that how OSQL can be applied to solve various problems that arise in relational DBMSs involving applications of temporal information, incomplete information and fuzzy information under the unifying framework of the ordered relational model. Let us consider the following relation EMP_DETAILS shown in Figure 3.

- **Temporal Information:**

We assume that SALARY_TIME is a time attribute whose values are timestamps of the tuples in the relation EMP_DETAILS (for simplicity in presentation, we also assume that the time stamping denotes valid time (Tansel et al., 1993)). For instance, we can see that Mark had 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. With the following query, we show how to find the salary of Mark in 1993 as follows.

(Q_{10}) *SELECT* (SALARY_TIME, SALARY) (*LAST*) *FROM* EMP_DETAILS
WHERE NAME = 'Mark' *AND* SALARY TIME <= 1993.

- **Incomplete Information:**

Suppose we have the domain INCOMPLETE_DOMAIN as in Figure 5 to capture the semantics of different null values; in this figure all known data values are more informative than the null symbol UNK (UNKnown), and UNK and DNE (Does Not Exist) are more informative than another null symbol NI (No Information) (we will address this point in detail in the next section). Let us define a semantic domain called INCOMPLETE_DOMAIN for the attribute PREVIOUS_WORK as follows:

(Q_{11}) *CREATE DOMAIN* INCOMPLETE_DOMAIN *CHAR*(10) *ORDER AS*
('NI' < 'DNE','NI' < 'UNK' < *OTHER*).

We emphasise that users have the freedom to use a semantic domain or not for comparison in an extended predicate. So it needs to specify the target semantic domain in the DML, in addition to declaring the existence of a semantic domain in the DDL. Now, we illustrate this idea by the following query, which finds the name and previous work of those employees whose previous work is more informative than NI:

(Q_{12}) *SELECT* (NAME, PREVIOUS_WORK) ($_*$) *FROM* EMP_DETAILS
WHERE (PREVIOUS_WORK > 'NI' *WITHIN* INCOMPLETE_DOMAIN).

- **Fuzzy Information:**

Suppose we have a semantic domain called QUALIFY to capture the semantic of the requirement "good science background in academic qualification" which is formulated as follows:

(Q_{13}) *CREATE DOMAIN* QUALIFY *CHAR*(10) *ORDER AS*
({'BA','MBA'}< 'MSc').

We can formulate the query of finding the names of employees with good science background in academic quali-

Figure 3: An Employee Relation EMP_DETAILS

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

fication as follows:

(Q_{14}) *SELECT* ((EDUCATION *WITHIN* QUALIFY), NAME)
($_*$) *DESC*
FROM EMP_DETAILS.

The OSQL statements in (Q_{10}) to (Q_{14}) reveal the potential of using OSQL to support the above-mentioned three advanced applications. In order to make use the capabilities of OSQL in a more systematic manner, we define a variety of generic operations with respect to these advanced applications and classify them into three OSQL packages: OSQL_TIME, OSQL_INCOMP and OSQL_FUZZY. Using these packages, we now show how the mentioned queries can be formulated in a simpler manner by embedding the operations of the OSQL packages into OSQL.

Using the package OSQL_TIME, the query (Q_{10}) can be simplified as follows:

(Q_{15}) *SELECT* (SALARY) ($_*$) *FROM* SNAPSHOT(EMP_DETAILS, 1993)
WHERE NAME = 'Mark'.

Using the package OSQL_INCOMP, the query (Q_{12}) can be simplified as follows:
(Q_{16}) *SELECT* (NAME, PREVIOUS_WORK) *FROM* EMP_DETAILS
WHERE MORE INFO(PREVIOUS_WORK, 'NI').

Using the package OSQL_FUZZY, the query (Q_{14}) can be simplified as follows:
(Q_{17}) *SELECT* (IMPOSE_FUZZY(EDUCATION, QUALIFY), NAME) (1)
FROM EMP_DETAILS.

Although we have not yet introduced the details of OSQL packages, the meaning of the operations are quite easy to understand. For instance, the operation IMPOSE_FUZZY in (Q_{17}) returns the appropriate tuples arranged in a list such that it satisfies the imposed fuzzy requirement "good science background in academic qualification".

The Structure of OSQL Packages

We now introduce the building blocks of an OSQL package; the full syntax of the PDL is given in Appendix B3 in Ng (1998). An OSQL package is defined by the following statement:

PACKAGE < package name >
< package body >
END PACKAGE.

The package body consists of the following five 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.

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 [á output variables ñ]

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 (Ng, Levene & Trevor, 2000), 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 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\time.strip*" in a function body specifies that the C program *time.strip* 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 linearly ordered. We reserve the enforcement, ENFORCE_INIT, to be used by the system for the initialization of an OSQL 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_{15}), which uses the package function SNAPSHOT, is equivalent to the query (Q_{10}), 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.

OSQL PACKAGES FOR ADVANCED APPLICATIONS

In this section we present in detail of the three OSQL packages for temporal information, incomplete information and fuzzy information, respectively. The OSQL packages can be predefined and thus made available for the database users as built-in facilities. The functions in an OSQL package can be embedded in an OSQL statement, provided that the data types of the input and output variables of a function comply with the syntax of OSQL.

OSQL_TIME: A Package for Temporal Information

The underlying semantics of time used in this OSQL package is that time is considered to be linearly ordered. 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 time data can be applied only over the time domain.

One of the many approaches (Tansel et al., 1993) 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* (Tansel et al., 1993). We assume temporal data is

Figure 4: An Employee Relation EMP_TIME Stamping with Time Intervals

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

timestamped with the time interval during which it is valid. For example, the relation EMP_TIME in Figure 4 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 £ 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 (Tansel et al., 1993). 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 5 in Tansel et al. (1993)).

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. Other standard domains available in OSQL_TIME include YEAR, MONTH, DAY, HOUR, MINUTE, SECOND. Note that these time domains support the need of using *multidimensional* databases in data warehousing (Inmon, 1996).

Note that we have not required that in OSQL_TIME contain some of the common temporal operators, such as *overlaps* and *contains* (see Chapters 4, 5 and 6 in Tansel et al. (1993)), which can be explicitly defined in order to compare time intervals, since they can be quite easily formulated in an OSQL comparison predicates. We now present the following description of the operations in OSQL_TIME in Table 2. The reader can consult Appendix for the declarations of the operations Appendix A in Ng (1998) for the full reference of the declarations pertaining to all OSQL packages.

Example 4 We use the relation EMP_TIME shown in Figure 4 whenever it is necessary.

1. IDENTIFY(YEAR) identifies the standard domain YEAR, which specifies the ordered set {1900 <L< 2000} and

Table 2: The Description of the Operations in OSQL_TIME

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 requirements for 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.

IDENTIFY(MONTH) identifies another standard domain {*JAN* <L< *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 <L< 99}.

2. Find the current salaries of all employees.
   (Q_{18}) *SELECT* (NAME, SALARY) ($_*$) *FROM* CURRENT(EMP_TIME).
3. Find the salary history of Mark.
   (Q_{19}) SELECT ($_*$) ($_*$) *FROM* HISTORY(EMP_TIME) *WHERE* NAME = 'Mark'.
4. Find the salary of Bill in 1994.
   (Q_{20}) SELECT (SALARY) ($_*$) *FROM* SNAPSHOT(EMP_TIME, 1994)
   WHERE NAME = 'Bill'.
5. Find the names of those employees who have worked for more than two years.
   (Q_{21}) *SELECT* (NAME) ($_*$) *FROM* EMP_TIME
   WHERE DURA(FROM_TIME, TO_TIME) > 2.

OSQL INCOMP: A Package for Incomplete Information

In this OSQL package, we classify the incompleteness into three unmarked *null symbols* whose semantics is given in (Codd, 1986).

1. UNK: Value exists but is UNKnown at the present time, for example some employees do not want to disclose their ages.
2. DNE: Value Does Not Exist, for example a fresh graduate does not have any previous work experience.
3. 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.

We use the notion of *more informative* values, which allows us to deduce useful information available from a relation having incomplete data (Libkin, 1995). The diagram in Figure 5 shows a partial ordering, say £, based upon the relative information content in a domain augmented with the three null values we have introduced. We can extend this partial ordering 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, $t_1[A]$ £ $t_2[A]$.

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_{11}). As we would like to make the domain INCOMP standard, we do not allow any user-defined incomplete domains in OSQL_INCOMP.

Note that the function IDENTIFY in this OSQL package is declared to be *private*, since the users are not allowed to change the meaning of various null symbols. This is to prevent the users from defining a casual notion of incompleteness, since the issue of missing information is much more

Figure 5: A Partial Ordering on a Data Domain

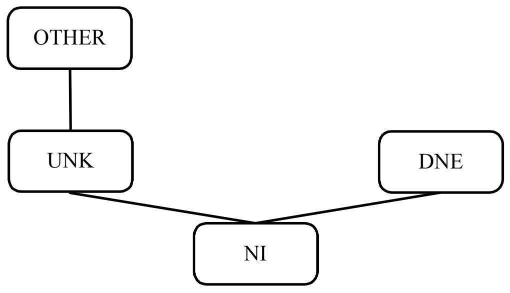


Table 3: The Description of the Operations in OSQL_INCOMP

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 requirements for an incomplete domain.
ENFORCE_INIT enforcement	To enforce the initialization which identifies the domain INCOMP as the incomplete domain used in the package.

difficult to handle than it appears (c.f. see Chapter 10 in Pascal (2000) for the problems of using SQL2 to handle null values). 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. We now present the following description of the operations in OSQL_TIME in Table 3.

Example 5 We use the relation EMP_INCOMP in Figure 6 whenever it is necessary.

Figure 6: An employee relation EMP_INCOMP

NAME	PREVIOUS_WORK
Mark	UNK
Ethan	DNE
Nadav	Administrator
Bill	Programmer
John	NI
Simon	NI

1. Find the name and previous work of those employees whose previous work is less informative than unknown (i.e. UNK).
   (Q_{22}) *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_{23}) *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_{24}) *SELECT* (NAME, PREVIOUS_WORK) ($_*$) *FROM* DNE_VAL(EMP_INCOMP, PREVIOUS_WORK).

OSQL FUZZY: A Package for 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. 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" this employee has to be promoted.

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

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, we have shown in (Q_{14}) the fuzzy requirement "good science background in 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. This strategy can be employed by an expert system to support users' decision based on fuzzy information.

We now present the description of the operations in Table 4. 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

Table 4: The Description of the Operations in OSQL_FUZZY

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 requirements for 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.

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.

Example 6 Let us consider the relation EMP_FUZZY in Figure 7 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.

Figure 7: An Employee Relation EMP_FUZZY

NAME	**EDUCATION**
Bill	MSc
Ethan	MSc
John	BSc
Mark	PhD
Nadav	MBA
Simon	A-Level

1. The fuzzy domain QUALIFY has been formulated in (Q_{13}) and the fuzzy domain CONNECT is given as the statement (Q_{25}) below.
   (Q_{25}) *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_{26}) *SELECT* (IMPOSE_FUZZY(NAME, CONNECT), IMPOSE_FUZZY (EDUCATION, QUALIFY) ($_*$) *FROM* EMP_FUZZY.

   A list of employees in which Mark appears to be the top one (the most preferred candidate) will be returned as the answer for this query.
3. We now use the functions 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. The employee Ethan appears on the top of the returned list as the answer for the query (Q_{26}) on this occasion.
4. Finally the fuzzy requirements can be listed as below by the function LIST_REQ().

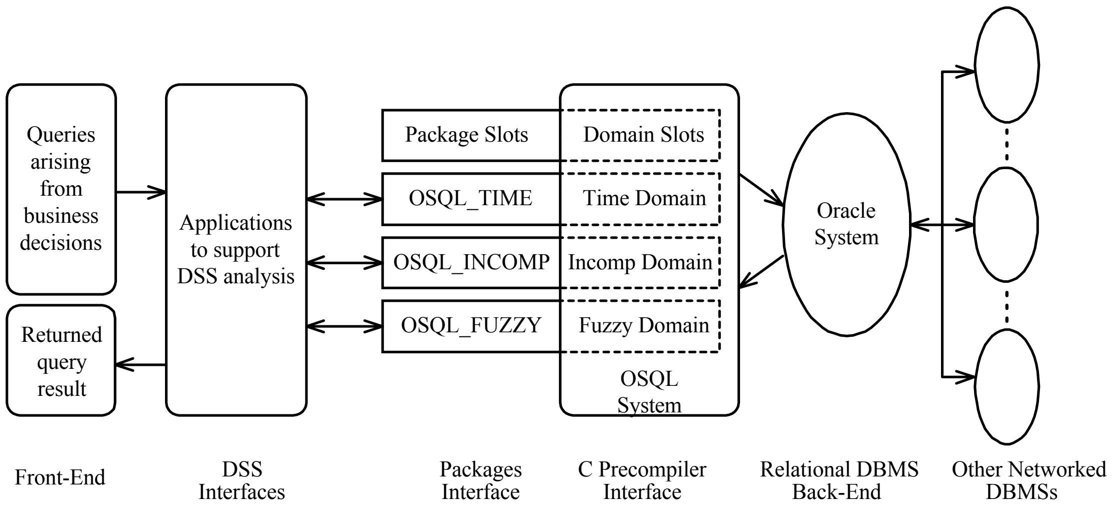
FUZZY_REQ	**PRIORITY**
CONNECT 1	
QUALIFY 2	

CONCLUSIONS

We have presented a new query language, namely OSQL, for querying ordered relational databases and a modularisation package discipline, which supports three applications of: (1) temporal information, (2) incomplete information and (3) fuzzy information. These applications are fundamental to develop data warehousing, since we have to integrate data from dynamic information sources. 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. Thus we could better adapt the data content in the data warehouse.

In Figure 8, we show our design of the system architecture, which allows OSQL statements to be entered via the front-end interface with a Decision Support System (DSS). The OSQL system can easily fit into a data warehousing strategy by offering various useful packages for deriving data required in DSS analysis. Moreover, users have the flexibility to define other customised OSQL packages tailored to the needs of an enterprise, in addition to those already mentioned, which represents as *package slots* and *domain slots* as shown in the figure. The DDL, DML and PDL of OSQL which operate over ordered relational databases have been implemented using an *Oracle8i* server for low level data management.

Figure 8: Architecture of the OSQL System in Data Warehousing



We are in the process of improving the system in order to make it possible to load more than one package into the system at the same time to support data warehousing strategy; in this case the parser is much more complex than the one that caters for a single package. 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 query "find the name and salary of the employees in 1996, the information about whose work is less informative than 'UNK'", which involves the application having temporal and incomplete information can be formulated in a unified manner by using two OSQL packages. (The relation EMP_DETAIL has been shown in Figure 3.)

(Q_{27}) *SELECT* (NAME, SALARY) ($_*$)
FROM OSQL_TIME.SNAPSHOT(EMP_DETAIL, 1996)
WHERE OSQL_INCOMP.LESS_INFO(PREV_WORK, 'UNK').

Admittedly, the current version of OSQL is not without its weaknesses when compared to those database languages which are specialised to only one particular application. In comparing to most proposed temporal extensions to define a richer set of specialised operators in handling temporal information; for instance, the six Allen's operators such as *overlaps*, *contains* and *meets* are defined in HSQL (c.f. Chapter 5 in Tansel et al. (1993)) as the primitive operations in order to compare time intervals. Such a *specialised approach* facilitates better understanding of the needed operations tailored to the specialised databases. In comparing to general fuzzy SQL extensions, we can see that, apart from those ordering information embedded in fuzzy domains, OSQL does not support users to define an algebraic function of membership (Buckles & Petry, (1982)). So the information about the numerical degree values of fuzzy quantities such as TALL, OLD and MANY cannot be directly formulated by using the basic OSQL constructs.

We emphasise that our extension to SQL is *a uniform approach*, since we provide a unified model as a basis for investigating robustness and efficiency of a set of *generic* operations and their new possible applications. Thus, an important on-going research issue is, in a formal manner, to compare OSQL with those extended SQLs specialised to handle temporal, incomplete and fuzzy information. Another limitation of using OSQL is that as a research prototype our system has not yet been developed to the standard of a fully-fledged version. In particular, we still need to study the implementational issues of how to integrate the facilities of user-defined orderings into the kernel of DBMSs at the physical level. We also need to further study those applications involving more complex types involving ordering, in such cases we have the problem of defining a large but elegant class of complex ordered types in the system. Finally, the problem of updating ordered databases has not been discussed in this paper. It can be further investigated in terms of the algorithms and formal semantics of updating ordered domains.

ACKNOWLEDGMENTS

I sincerely thank the editor and anonymous referees for their important comments and suggestions, through which this paper can be greatly improved.

REFERENCES

Buckles, B. P., & Petry, F. E. (1982). A Fuzzy Representation of Data for Relational Databases. *Fuzzy Sets and Systems* **7**, 213-226.

Buneman, P., Davidson, S., Fernandez, M., & Suciu, D.

APPENDIX SAMPLE OF OSQL GRAMMAR AND THE OSQL_TIME PACKAGE

Data Definition Language (DDL)
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 >

Data Manipulation Language (DML) 1. *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 | *})
< condition > ::= < attribute-name | value> < comparator > <{attribute-name | value}> [*WITHIN* < domain-name >] < comparator > ::= {<|>|>=|<=|<>}

Package Definition Language (PDL)
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 > :: = [{*PRI* | *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*

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

(1996). Adding Structure to Unstructured Data. *Technical Report* MS-CIS 96-21, CIS Department, University of Pennsylvania.

Casanova, M. A., Furtado, A. L., & Tucherman, L. (1991). A Software Tool for Modular Database Design. *ACM Transactions on Database Systems* **2**, 209-234.

Celko, J. (1995). *SQL For Smarties: Advanced SQL Programming*. Morgan Kaufmann Publishers.

Codd, E.F. (1986). Missing Information (Applicable and Inapplicable) in Relational Databases. *ACM SIGMOD Record* **15**(4), 53-58.

Date, C.J. (1990). *Relational Database Writings 1985-1989*. Addison-Wesley.

Date, C.J. (1997). *A Guide to the SQL Standard (4th edition)*. Addison-Wesley.

McCabe, M. C. & Grossman, D. (1996). The Role of Tools in Development of a Data Warehouse. In: *Proceedings of the 4th International Symposium on Assessment of Software Tools*, 139-145.

Inmon, W. H. (1996). *Building the Data Warehouse*. John Wiley & Sons.

Levene, M. & Loizou, G. (1999). *A Guided Tour of Relational Databases and Beyond*. Springer Verlag.

Libkin, L. (1995). A Semantics-Based Approach to Design of Query Languages for Partial Information. In: *Proceedings of the Workshop on Semantics in Databases*, 63-80.

Lu, H., Chan, H. C., & Wei, K. K. (1993). A Survey on Usage of SQL. *SIGMOD Record* **22**(4), 60-65.

Maier, D. & Vance, B. (1993). A Call to Order. In: *Proceedings of the Twelfth ACM Symposium on Principles of Databases Systems*, 1-16.

Mattos, N., & DeMichiel, L. G. (1994). Recent Design Trade-offs in SQL3. *ACM SIGMOD Record* **23**(4), 84-89.

Melton, J. (1996). An SQL3 Snapshot. In: *Proceedings of the International conference on Data Engineering*, pp. 666-672.

Ng, W., & Levene, M. (1997). An Extension of OSQL to Support Ordered Domains in Relational Databases. In: *IEEE Proceedings of the International Database Engineering and Applications Symposium*, Montreal, Canada, 358-367.

Ng, W. (1998). OSQL Grammar. *http://www.comp.polyu.edu.hk/~csshng/JDBM.html.*

Ng, W. (1999). Ordered Functional Dependencies in Relational Databases. *Information Systems* **24**(7), 535-554.

Ng, W., Levene, M., & Fenner, T. I. (2000). On the Expressive Power of the Relational Algebra with Partially Ordered Domains. *International Journal of Computer Mathematics* **74**(3-4), 53-62.

Pascal, F. (2000). *Practical Issues in Database Management*. Addison-Wesley.

Smith, D. (1994). *IBM Smalltalk: The Language*. Benjamin/Cummings.

Tansel, A. et al. (editors) (1993). *Temporal Databases: Theory, Design and Implementation*. Benjamin/Cummings.

Ullman, J. (1988). *Principles of Database and Knowledge-Base Systems, Vol I*. Rockville, MD., Computer Science Press.

Wilfred Ng obtained his B.Sc.(Hon) degree and Postgraduate Certificate of Education from the University of Hong Kong. He then received his M.Sc.(Distinction) degree in Information Technology and Ph.D. degree in Computer Science from the University College London (UCL) in the U.K. His main research interests are Databases and Information Systems, which include semantic data modeling, temporal databases, data warehousing and web data mining. Currently, Dr Ng is a professor at The Hong Kong University of Science and Technology. Dr Ng is also a professional member of the ACM and the IEEE.

Mark Levene received his PhD degree in Computer Science in 1990 from Birkbeck College, which is part of the University of London. Dr. Levene is currently a Reader in Knowledge Management in the Department of Computer Science and Information Systems at the University of London. Dr. Levene has published extensively in the area of database theory and has recently co-authored a comprehensive book on relational databases and their extensions. His main research interests are database theory and web interaction.