Using UML to Model Relational Database Operations

Eunjee Song Shuxin Yin Indrakshi Ray
Computer Science Department
Colorado State University
Fort Collins, CO 80523, USA
Email: {song, yin, iray}@cs.colostate.edu

Abstract

The Unified Modeling Language (UML) is being used as the de-facto standard in the software industry. With the adoption of UML 2.0, the new enhancements allow this version to describe many of the elements found in today's software technology as well as Model Driven Architecture and Service-Oriented Architecture. Although OMG has released several UML Profiles to tailor the language to specific areas, relational database modeling is not fully addressed in these profiles. Many existing software applications involve complex application layer implemented in object-oriented programming languages and at the same time use relational database systems as the back-end data store. Modeling the whole system in a consistent manner will help developers and end users better understand the application. In this work we show how to model relational database operations using UML. Atomic database operations are modeled based on our framework and are used as building blocks to model more complex database operations.

1 Introduction

As software applications become more and more complicated, it is essential to capture the requirements, model the system design in different phases, and communicate frequently among end users, business analysts and developers. Object-oriented technology such as Java and .NET plays an important role in software development. A lot of these applications are designed to process huge amount of information which is saved in database system. Although object-oriented database management system (OODBMS) is gaining market share, relational database management system (RDBMS) remains the dominant database technology. Most database vendors add object-oriented features into existing RDBMS products instead of totally abandoning RDBMS. Many software professionals have to continue to design or maintain this kind of hybrid system with application layer implemented using object-oriented languages and database layer implemented using RDBMS. How to model the whole system in a consistent manner is still a challenge because there is no universal modeling standard for both object-oriented languages and RDBMS.

Relational database systems play an important role in enterprise software applications. Many vendors add their own extensions to ANSI SQL such as Oracle PL/SQL and Microsoft Transact SQL. Data warehouse is widely used in many large and medium-sized enterprises. Business information and data are processed and manipulated at database level before being used by applications. These database operations implemented in packages, stored procedures, triggers and functions can be very complex. Modeling these operations improves understandability, reusability and maintenance of the database system. It is also a way to document the dynamic aspect of database system. Entity-Relation (ER) modeling only captures the static schema and can not model dynamic operations. Current UML modeling standard does not fully address this problem as well.

Most software projects involve end users, business analysts, application development team and database team; modeling the whole system using one standard will make communication among team members much easier. Collaboration and cooperation has always been a key aspect of overall system success. The UML gives us the ability to model, in a single language, the business, application, database, and architecture of the system. By having one single language, everybody involved can communicate their thoughts, ideas, and requirements [13].

UML Data Modeling Profile [1, 9] was proposed by Rational Software from IBM. The use of Data Modeling Profile has resulted in using UML in database design [13]. UML can be used to model relational database schema and it is more expressive than ER modeling. Yet the current UML database modeling techniques mainly focus on static schema modeling. Dynamic database operations are modeled in an ad hoc manner.

Our previous work [21] identified the weakness of UML Data Modeling Profile, that is, lack of abilities to model operations. We proposed a framework on how to model operations at database level so that we could have a more comprehensive understanding of the whole system. We showed how to model atomic operations in SQL Data Manipulation Language (DML) based on relational algebra and set theory. However, we modeled the operations at a very high-level of abstraction. In this paper, we model operations at a much lower-level. This will give the application developers a better idea about the semantics of the operations. We also describe many different types of query operations since these operations are complex and are most frequently used.

SQL is a declarative language; it allows us to express what we want instead of specifying how to do it [10]. In contrast, procedural languages use statements to tell the computer exactly what to do in a step-wise fashion. SQL is more flexible than procedural languages since it hides internal implementation details. We are not trying to model the internal execution details such as parsing SQL statement, validating the statement, optimizing the statement, generating an execution plan and executing the execution plan. All these details should not be exposed to application developers. On the other hand, modeling database operations at SQL statement level makes it easier for end users who are unfamiliar with SQL key words to understand the database operations.

The remainder of this paper is organized as follows. We briefly describe ER and UML modeling in Section 2. In Section 3 we discuss related work including UML Data Modeling Profile. We propose a framework of atomic database operation modeling in Section 4. We describe how to model complex database operations using the proposed framework in Section 5. Finally we present our conclusions in Section 6.

2 Overview of ER Modeling and UML

The Entity-Relationship (ER) model was originally proposed in 1976 [5] as a way to unify the network and relational database views. Essentially the ER model is a conceptual data model that views the real world as entities and relationships. The basic constructs in ER model are the entities, attributes and relationships, all of which may be presented in an ER diagram. The ER model has been adopted as the meta-model for the ANSI Standard in Information Resource Directory System (IRDS).

The ER model focuses on conceptual and logical design phase of database but cannot capture dynamic behavior of database systems. This limitation does not prevent ER Model from being a good modeling tool since most database systems are data centric and operations are moved to business logic and application level. Later extensions to ER Model added the capability to capture more information, such as aggregation and inheritance [4, 17].

The Unified Modeling Language (UML) [19] is a general-purpose visual modeling language that is used to specify, visualize, analyze, and document the artifacts of a software system [16]. It captures decisions and understanding about systems that must be constructed. It is used to understand, design, browse, configure, maintain, and control information about systems. UML2.0 represents the biggest change that has happened yet to the UML. Within UML, there have been significant changes to the UML meta-model that defines the concepts of the language [8].

The UML 2.0 defines two major kinds of diagram types: structure diagrams and behavior diagrams [16]. The static structure of a system is expressed using various kinds of diagrams, such as class diagram, component diagram, composite structure diagram, deployment diagram, object diagram and package diagram. The dynamic behavior of a system is expressed using activity diagram, communication diagram, interaction overview diagram, state machine diagram, sequence diagram, timing diagram and use case diagram. In fact, the UML class and its structure correspond closely to the entity type and its structure in ER diagrams. In many cases, class diagrams strongly resemble ER diagrams. Differences emerge mainly in the modeling of operations and relationships. All features in ER diagram can be represented in the UML class diagram. The UML class diagram is more expressive than ER diagram for example a stereo-typed class can be used to represent a particular type of class, an OCL notation can be used to express pre and post conditions.

UML 2.0 supports the initiative for a model-driven architecture by providing the stable technical infrastructure that allows for additional automation of the software development process [7]. OMG's

Model Driven Architecture provides an open, vendor neutral approach to the challenge of business and technology [18]. Based on OMG's established standards, the MDA separates business and application logic from underlying platform. Platform Independent Model (PIM) is transformed into Platform Specific Model (PSM) using certain transformation rules. MDA is built on top of UML and modeling is the foundation for model transformation.

3 Related Work

There has been much research in the UML data modeling [2, 3, 11, 12, 14, 15]. Using various transformation techniques an ER diagram can be converted into an equivalent UML class diagram. However, almost all of the previous research focuses on static schema; none of them deals with dynamic operations at database layer. Premerlani [14] discusses the reverse engineering of RDBMS into OMT model. His approach emphasizes on the analysis of candidate keys, rather than primary keys. However, the reverse engineering process requires much user interactions. Ramanathan [15] presents an object-centered approach for doing the schema mapping. The procedure maps a 3NF relational schema into an object-oriented schema without the explicit use of inclusion dependencies and provides a greater scope for automation. Behm [3] proposes a two-phase transformation process, i.e., schema transformation followed by data migration. During the data migration process, instances of class are created, and then attributes are assigned based on a set of rules. The data mapping process can be implemented without much user intervention. Researchers [3, 11, 12, 13, 14, 15] have also proposed techniques using which a UML class diagram can be transformed into an equivalent ER diagram. Other researchers have worked on techniques that convert an ER diagram to a class diagram.

The power of the UML is not limited to object oriented software development. The UML is being applied to many other areas of software development, such as data modeling, enhancing practitioners' ability to communicate their needs and assessments to the rest of the team [9]. UML Data Modeling Profile [1, 9] was proposed to apply UML to relational database modeling.

The UML can be used to describe the complete development of relational and object relational databases from business requirements through the physical data model [1]. UML for Data Modeling Profile is implemented by Rational Rose Data Modeler; it includes descriptions and examples for each concept including database, schema, table, key, index, relationship, column, constraint, trigger and stored procedures. We will list some frequently used features of Rational Data Modeler as follows [1]:

- The stereotype <<Database>>, when used as a UML component, represents a database.
- A package using the <<Schema>> stereotype in a UML model represents a database schema.
- A class with the <<Table>> stereotype represents a relational table in a schema of a database.

- A primary key uses a PK tag in front of the column as shown, and the PK stereotyped operation is the primary key constraint.
- An FK tag represents the foreign key. It generates the foreign key constraint, which is represented by a stereotype FK on an operation.
- The Index stereotype on an operation represents a key constraint for an index.
- An identifying relationship is a relationship between two tables in which the child table must coexist with the parent table. The identifying relationship is represented with a stereotype of <<Identifying>>.
- A non-identifying association represents a relationship between two independent tables. The non-identifying relationship is created with a stereotype of <<Non-identifying>>.
- Columns are represented as attributes. Computed columns are defined by an expression.
- All constraints are defined as stereotyped operations and primary key is a special kind of constraint.
- The Trigger stereotype on an operation represents the trigger on the table.
- The Unique stereotype represents the uniqueness constraint.
- Stored procedure can be represented as a stereotyped operation using <<SP>>. Multiple stored procedures are contained inside a stereotyped template <<SP Container>>. The signature of a stored procedure can be shown besides stored procedure name.
- Activity diagram and state machine diagram can be used to model stored procedure at a high level; however, no corresponding relationship is defined between tables, SQL DML and actions in activity diagram. Tables are not directly related to states in state machine diagram.
- A view is defined in the UML as a class with the stereotype of <<View>>.

The UML Data Profile allows application software development and data modeling with one unified language. Figure 1 is an example of UML data model, taken from our previous work [21], generated using Rational Rose. A simplified Air Toxics Data Archive system is used as the underlying database. A typical air quality monitoring program involves at least monitoring sites, pollutants, sampling time and concentration values.

There are three tables in this diagram: *Pollutant*, *Site* and *Concentration*. They are represented using stereotyped class. In this example, there are two stored procedures. The first stored procedure *ReportByPollutant* uses *PollutantID* as an input parameter; this procedure will return a result set which contains all concentration information for this particular pollutant. Another stored procedure *ReportBySite* takes a *SiteID* parameter; it will return a result set which contains all concentration information at the specified site. These two stored procedures are contained in <<SP Container>> template as two stereotyped operations. Although we can use activity diagram and state machine diagram to model stored procedures, these two diagrams are used at requirement analysis and logical design level. There is no standard technique to model low-level DML such as SELECT, DELETE, UPDATE and INSERT. In many systems, the stored procedures can be very complex. Major RDBMS vendors define their own extension to SQL standard and add many language constructs to represent sequential, conditional and iterative operations. Many RDBMS also allow user-defined functions to simplify commonly used operations. Modeling both stored procedures and functions in a standard way will offer great benefits to application developers, database developers and business analysts. Note that, the UML Data Profile describes the relational database. It models the database implementation; it is at a different level of abstraction than the ER diagram.

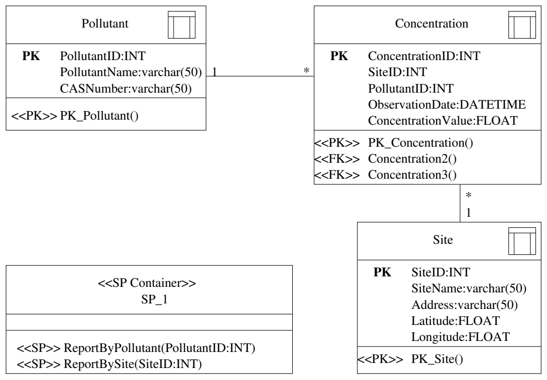


Figure 1: UML data model generated by Rational Rose

Using Rational Rose, we can only model stored procedures using activity diagram and state machine diagram in an ad hoc fashion. No standard techniques are available as to how to model dynamic operations encapsulated in stored procedures and functions at implementation level. Actions and states are represented in plain English which will create ambiguity and confusion to different people.

4 Atomic DML Operations Modeling

In this section, before showing our modeling of the relational database operations we would like to adapt the diagram, obtained from Rational Rose and used in our earlier work [21], to explain our ideas with UML 2.0 compliance. In UML, four predefined basic types are supported: *Boolean*, *Integer*, *Real*, and *String*. Therefore, *Integer* and *String* are used instead of *INT* and *varchar*. Two new data types *DateTime* and *Float* are also added to the UML class diagram. We added three other classes *Projection*, *QueryResult*, and *JoinResult*. The stereotype <<temporary>> indicates that these classes are of temporary nature - the objects of these classes get destroyed when the DML statement has completed execution. We added a new class *TableHandler* that has references to classes representing three database tables. Figure 2 shows the details. In the following subsections, we will show how the relational database operations are modeled as operations in the *TableHandler* class.

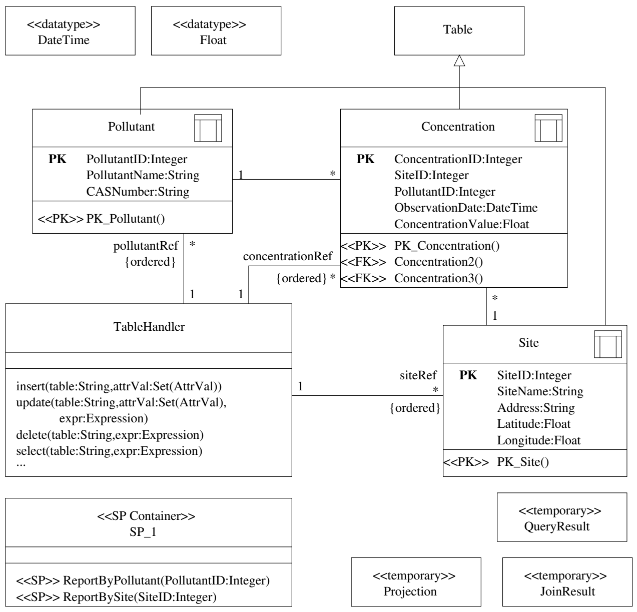


Figure 2: Air toxics data archive UML data model

4.1 Relational Databases

Relational databases are based on the relational model [6]. The relational model has a strong mathematical foundation and it is based on set theory. Each relational table in a database is considered as a set and each row corresponds to an element of the set. The eight relational operations performed on the tables are union, difference, intersection, product, projection, selection, join and division. The Structured Query Language (SQL) is the most widely used language for relational databases. SQL is broadly categorized into Data Definition Language (DDL) and Data Manipulation Language (DML). DDL is used for specifying and manipulating the structure of the database and DML is used for performing operations on the database itself. In this paper, we focus on the SQL DML constructs only.

4.2 SQL Data Manipulation Language

Database operations can be in the form of packages, stored procedures, functions, or other vendor defined language constructs. However, our modeling technique will not depend upon the format of these operations. Essentially all database operations are implemented using atomic SQL data manipulation language constructs such as INSERT, UPDATE, DELETE, and SELECT. In this section, we will model standard SQL DML constructs and use them as building blocks to model more complex operations in the next section.

We all know that there is no object in RDBMS and operations are separated from tables; while in object-oriented programming, an object encapsulates both data and operations. In our model we can think of a row as an anonymous object of a class as defined by the table. There are four predefined operations upon this object which are INSERT, UPDATE, DELETE and SELECT.

4.3 Insert Operation Modeling

Insert statement is the simplest operation in DML. Here is an example:

```
INSERT INTO Pollutant (PollutantID, PollutantName, CASNumber)
               VALUES (1, 'Ethylbenzene', 100414)
```

An insert operation can be modeled using many different diagrams, such as sequence diagram, communication diagram, activity diagram, interaction overview diagram and state machine diagram. We show a generic form of the sequence diagram in Figure 3. This represents a sequence diagram frame that can be instantiated for each insert operation. Recall that we have a class *TableHandler* that coordinates all operations involving database tables. This class consists of a single object that is shown as *:TableHandler* in Figure 3. The insert operation takes as input two parameters: *table* which represents the name of a relational table and *attrVal* which is a set of attribute-value pairs. The *:TableHandler* selects a table reference (that is, either *pollutantRef*, *concentrationRef*, or *siteRef*) from the set of all tables and forwards it to the insert operation. Using this operation, *:TableHandler* creates an anonymous object *:Table* that corresponds to a row of the table refered by the selected table reference (that is, either *Pollutant*, *Concentration* or *Site*). The insert operation does not return anything. UML 2.0 adds a useful touch to sequence diagram. We can now frame a sequence diagram by surrounding it with a border and adding a compartment in the upper left corner. The compartment contains information that identifies the diagram [20]. Each framed sequence diagram can be referred by other sequence diagrams. UML 2.0 also adds another new core construct, known as combined fragment, which expresses control structures such as choice, loop, etc. We encapsulate the whole statement into a named framed part called *Insert*. For each specific example, this generic diagram can be instantiated according to the name of table given as the first parameter. For our example SQL statement, *pollutantRef*, a table reference to the *Pollutant* table, is given as an input to the insert sequence diagram so that it may insert a new row to the *Pollutant* table as shown in Figure 4. Figure 5 shows how the *:TableHandler* gets a table reference from the table name and invokes an insert operation to the appropriate table. The box labeled *Insert* represents the details shown in Figure 4.

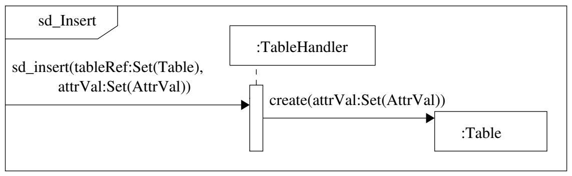


Figure 3: Insert operation sequence diagram frame

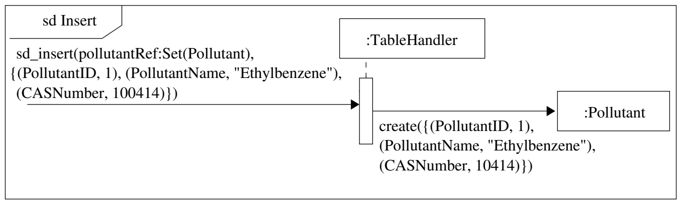

sd Insert
:TableHandler
sd_insert(pollutantRef:Set(Pollutant),
{(PollutantID, 1), (PollutantName, "Ethylbenzene"),
(CASNumber, 100414)})
:Pollutant
create({(PollutantID, 1),
(PollutantName, "Ethylbenzene"),
(CASNumber, 10414)})

Figure 4: Insert operation sequence diagram for the example

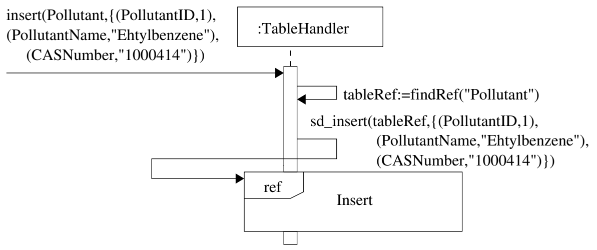

insert(Pollutant,{(PollutantID,1),
(PollutantName,"Ehtylbenzene"),
(CASNumber,"1000414")})
:TableHandler
tableRef:=findRef("Pollutant")
sd_insert(tableRef,{(PollutantID,1),
(PollutantName,"Ehtylbenzene"),
(CASNumber,"1000414")})
ref
Insert

Figure 5: Insert operation sequence diagram for the example

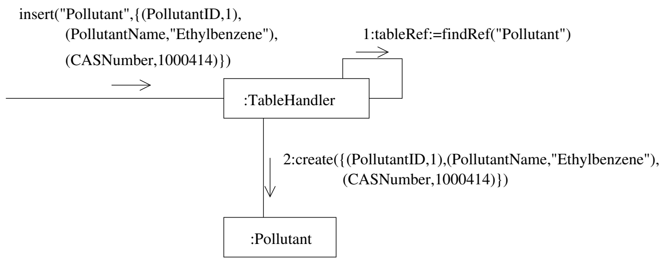

insert("Pollutant",{(PollutantID,1),
(PollutantName,"Ethylbenzene"),
(CASNumber,1000414)})
1:tableRef:=findRef("Pollutant")
:TableHandler
2:create({(PollutantID,1),(PollutantName,"Ethylbenzene"),
(CASNumber,1000414)})
:Pollutant

Figure 6: Insert operation communication diagram

The communication diagram shown in Figure 6 illustrates how our example insert operation can be represented in the form of a communication diagram. This shows that the insert operation consists of a single step that creates an anonymous object in the *Pollutant* table. The insert operation can be represented using state machine diagram and activity diagram as well. However, these pictorial representations are not very useful because a state machine diagram is good for describing behavior of objects over time in terms of state transitions triggered by events and an activity diagram is used for describing the flow of control and (optionally) data.

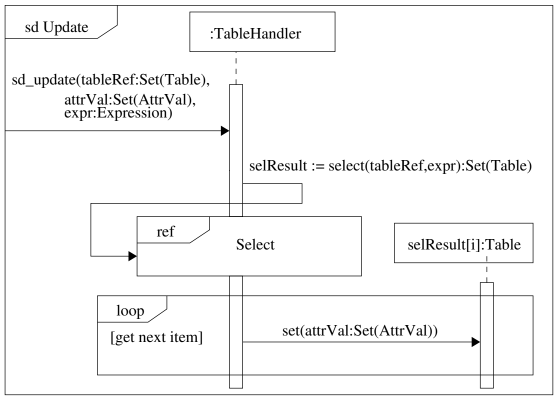
4.4 Update Operation Modeling

The update operation can be simple like the insert operation or it can be more complex. The simple UPDATE does not have a WHERE clause and it changes one or more attributes of all rows belonging to the table. Since this is a special case of the more complex UPDATE, we do not specify it using UML diagrams. An example of a simple UPDATE is given below.

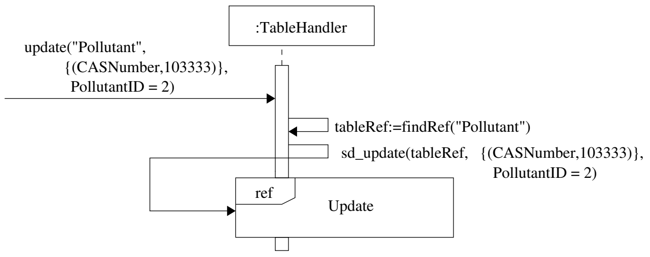
```
UPDATE Pollutant SET CASNumber = 0
```

The complex form of UPDATE includes the WHERE clause. In such case, an update operation can be decomposed into the following steps: 1) Selection: find the rows to be updated based on specific conditions first; 2) Set: update the old value with the new value in all these rows. An example of a more complex update operation appears below.

```
UPDATE Pollutant SET CASNumber = 103333
WHERE PollutantID = 2
```


sd Update
:TableHandler
sd_update(tableRef:Set(Table), attrVal:Set(AttrVal), expr:Expression)
selResult := select(tableRef,expr):Set(Table)
ref
Select
selResult[i]:Table
loop
[get next item]
set(attrVal:Set(AttrVal))

(a) A sequence diagram frame for UPDATE operation


:TableHandler
update("Pollutant", {(CASNumber,103333)}, PollutantID = 2)
tableRef:=findRef("Pollutant")
sd_update(tableRef, {(CASNumber,103333)}, PollutantID = 2)
ref
Update

(b) A sequence diagram for example UPDATE statement

Figure 7: Update operation sequence diagram

Figure 7 describes UPDATE using sequence diagrams. In the above example, we use a framed part update to represent the update statement as a whole and use select to implement the selection part of the statement. The details of how SELECT is implemented will be discussed later. Figure 7(a) represents a generic diagram of the update operation. The update operation takes in three parameters, namely, *tableRef* which is the reference to the table that must be updated, *attrVal* which is the set of attribute-value pairs (describing the attributes that must be updated and their modified values), and *expr* which is an expression needed to select the rows that needs updates. The *:TableHandler* on receiving this operation executes a select operation on it. The select operation returns *selResult* which is a set of rows of *tableRef* satisfying the condition stated in *expr*. For each row returned by *selResult*, indicated by *selResult[i]*, the attributes listed in *attrVal* must be modified to the corresponding values. Figure 7(b) shows that the *:TableHandler* selects the table *tableRef* according to the table name passed as the first parameter and invokes the update operation on the *Pollutant* table.

The communication diagram for our specific example UPDATE is shown in Figure 8. The communication diagram shows the three steps involved in the update operation. The first step, labeled with 1, involves selecting the table *Pollutant* referenced in the update operation. The next step involves finding the rows in the selected table that satisfy the expression *PollutantId = 2*. The last step is applied to several rows that were selected in the previous step. This is indicated by the label 3* shown in the figure. This step involves updating the *CASNumber* to the new value 103333.

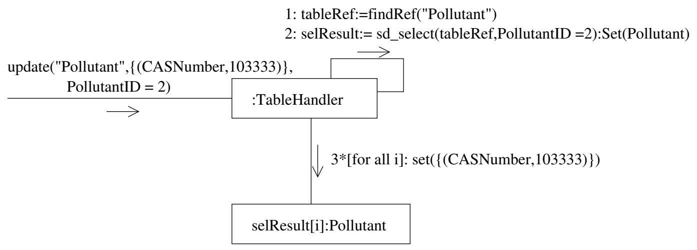
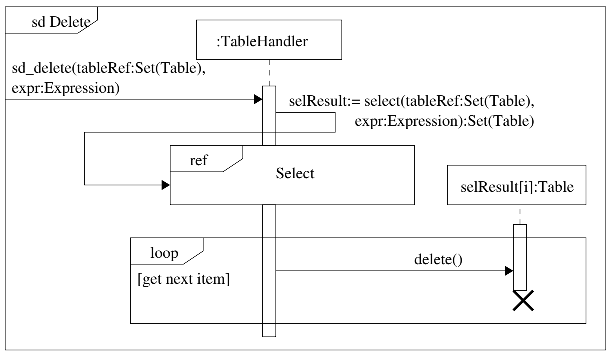

1: tableRef:=findRef("Pollutant")
2: selResult:= sd_select(tableRef,PollutantID =2):Set(Pollutant)
update("Pollutant",{(CASNumber,103333)}, PollutantID = 2)
:TableHandler
3*[for all i]: set({(CASNumber,103333)})
selResult[i]:Pollutant

Figure 8: Update operation communication diagram

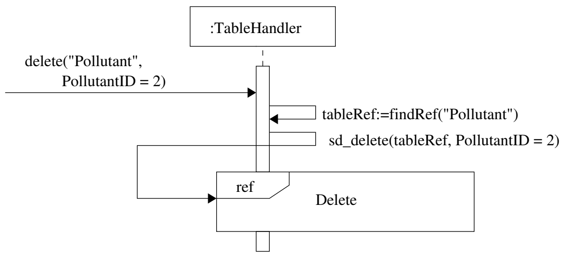
4.5 Delete Operation Modeling

Delete operation is similar to update operation except that the rows are removed from the table instead of being updated. A typical DELETE statement is listed below and its sequence diagram and communication diagram are shown in Figure 9 and Figure 10. Note that the sequence diagram is very similar to that of the update operation except that there is an X marked at the end of the lifeline which indicates that the selected row is being destroyed.

```
DELETE FROM Pollutant WHERE PollutantID = 11
```


sd Delete
:TableHandler
sd_delete(tableRef:Set(Table), expr:Expression)
selResult:= select(tableRef:Set(Table), expr:Expression):Set(Table)
ref
Select
selResult[i]:Table
loop
[get next item]
delete()

(a) A sequence diagram frame for DELETE operation


:TableHandler
delete("Pollutant", PollutantID = 2)
tableRef:=findRef("Pollutant")
sd_delete(tableRef, PollutantID = 2)
ref
Delete

(b) A sequence diagram for the example DELETE statement

Figure 9: Delete operation sequence diagram

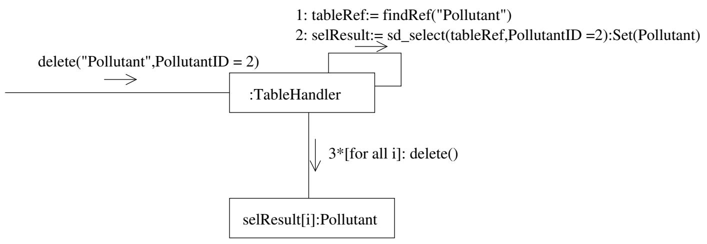

1: tableRef:= findRef("Pollutant")
2: selResult:= sd_select(tableRef,PollutantID =2):Set(Pollutant)
delete("Pollutant",PollutantID = 2)
:TableHandler
3*[for all i]: delete()
selResult[i]:Pollutant

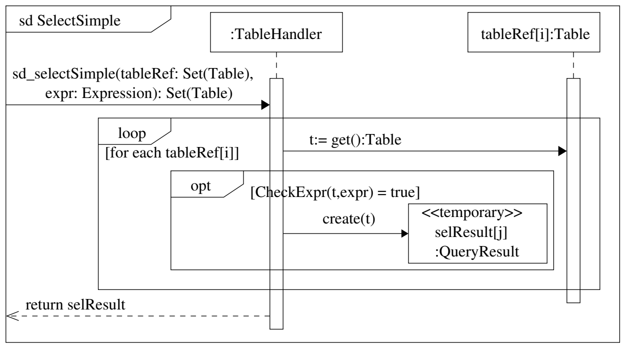
Figure 10: Delete operation communication diagram

4.6 Select Operation Modeling

Select operation is the most important operation in a database system and it can often be extremely complex. However, it does not change the database.

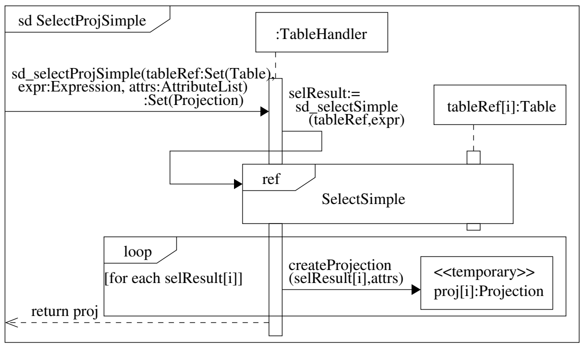
In the following we describe some of the most commonly used SELECT statements. We start with a simple select that retrieves all the attributes of a table. We call this the *SelectSimple* operation. This is shown in Figure 11(a). It takes in two parameters: *tableRef* which is the reference to the table that must be selected and *expr* which is an expression defined on the attributes of the table. The *:TableHandler* selects table *tableRef*. For each row of *tableRef*, indicated by *tableRef[i]*, it gets the values of all the attributes indicated by *t* and checks whether *expr* is satisfied. If so, a temporary table is created indicated by the stereotype <<temporary>>, and a new row called *selResult[j]* is inserted into this table. All rows of this table indicated by *selResult* is returned as the result of this query.

A slightly more complex select operation is the one that does not retrieve all but only some of the attributes of a table. We call this one *SelectProjSimple*. This one is shown in Figure 11(b). This one processes the results returned by the *SelectSimple*. For each row returned by *SelectSimple*, indicated by *selResult*, we need to select the relevant *attrs*, and a new object *proj[i]* is created and stored in a temporary table. All the rows of this temporary table, denoted by *proj*, is returned as a result of this operation.



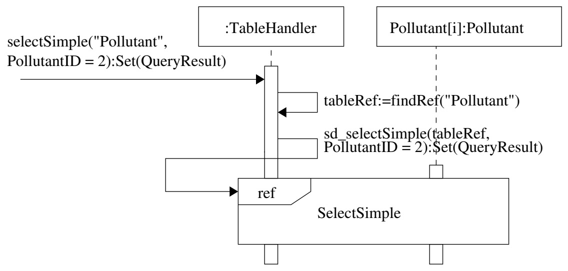
sd SelectSimple
:TableHandler
tableRef[i]:Table
sd_selectSimple(tableRef: Set(Table), expr: Expression): Set(Table)
loop
[for each tableRef[i]]
t:= get():Table
opt
[CheckExpr(t,expr) = true]
create(t)
<<temporary>> selResult[j] :QueryResult
return selResult

(a) A sequence diagram frame for a SELECT operation which retrieves all attributes



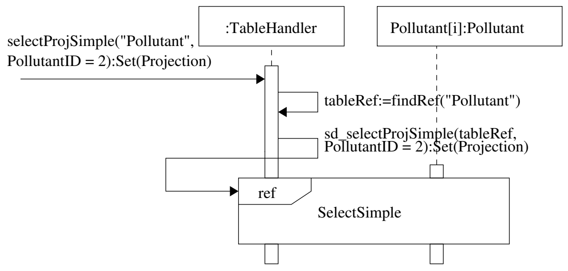
sd SelectProjSimple
:TableHandler
sd_selectProjSimple(tableRef:Set(Table), expr:Expression, attrs:AttributeList) :Set(Projection)
selResult:= sd_selectSimple (tableRef,expr)
tableRef[i]:Table
ref
SelectSimple
loop
[for each selResult[i]]
createProjection (selResult[i],attrs)
<<temporary>> proj[i]:Projection
return proj

(b) A sequence diagram frame for a SELECT operation which retrieves a list of attributes



:TableHandler
Pollutant[i]:Pollutant
selectSimple("Pollutant", PollutantID = 2):Set(QueryResult)
tableRef:=findRef("Pollutant")
sd_selectSimple(tableRef, PollutantID = 2):Set(QueryResult)
ref
SelectSimple

(c) A sequence diagram that represents a simple SELECT statement "SELECT * FROM Pollutant WHERE PollutantID = 2"



:TableHandler
Pollutant[i]:Pollutant
selectProjSimple("Pollutant", PollutantID = 2):Set(Projection)
tableRef:=findRef("Pollutant")
sd_selectProjSimple(tableRef, PollutantID = 2):Set(Projection)
ref
SelectSimple

(d) A sequence diagram that represents a simple SELECT statement "SELECT PollutantName FROM Pollutant WHERE PollutantID = 2"

Figure 11: Simple select operation sequence diagram

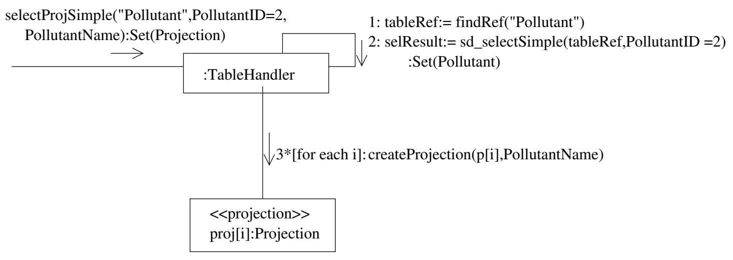
Figure 11(c) shows the sequence diagram for the example *SelectSimple* operation stated below.

```
SELECT * FROM Pollutant
WHERE PollutantID = 2
```

Similarly, Figure 11(d) shows the sequence diagram and Figure 12 shows the communication diagram for the following *SelectProjSimple* example.

```
SELECT PollutantName
FROM Pollutant
WHERE PollutantID = 2
```

Note that *SelectSimple* and *SelectProjSimple* queries involve a single table and the WHERE clause is a boolean expression over the attributes of this single table. Later on, we show how to overcome this shortcoming.



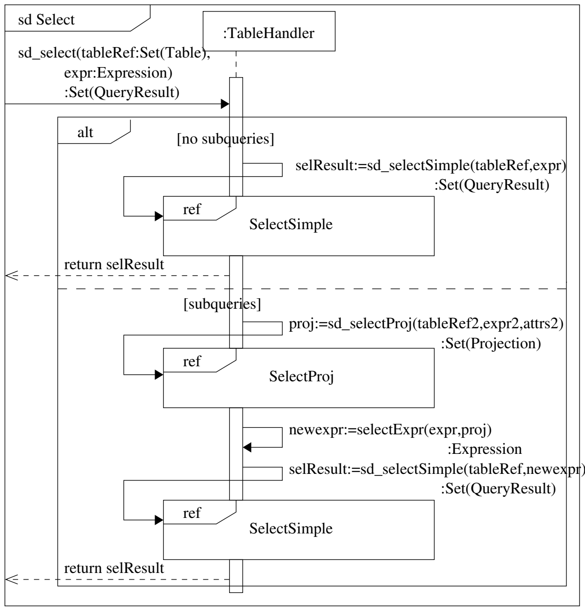
selectProjSimple("Pollutant",PollutantID=2, PollutantName):Set(Projection)
1: tableRef:= findRef("Pollutant")
2: selResult:= sd_selectSimple(tableRef,PollutantID =2) :Set(Pollutant)
:TableHandler
3*[for each i]: createProjection(p[i],PollutantName)
<<projection>> proj[i]:Projection

Figure 12: Select statement communication diagram

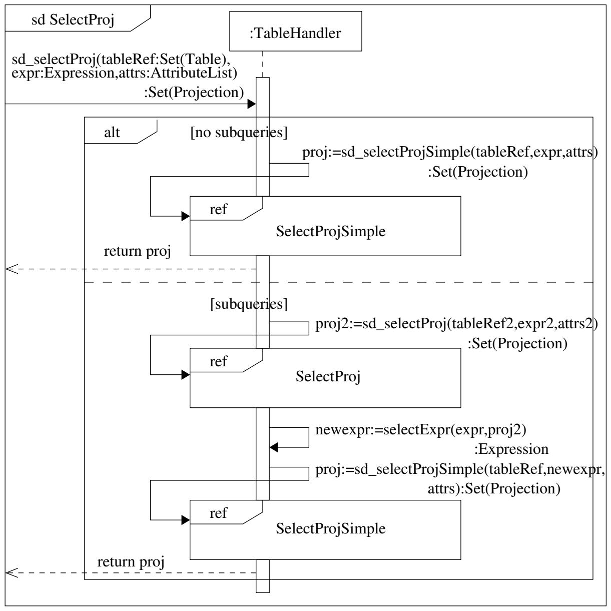
A query can itself include one or more subqueries. Any number of subqueries can be nested in a statement. An example of a query having a nested query is given below.

```
SELECT * FROM Concentration C
WHERE C.PollutantID IN
  (SELECT PollutantID FROM Pollutant P
   WHERE P.PollutantName LIKE 'A%')
```

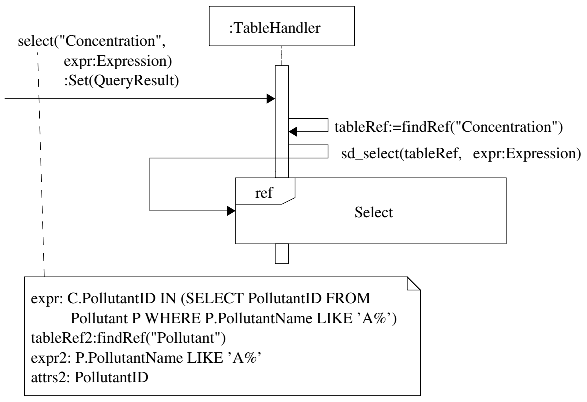
The nested queries are described in Figure 13. In the sequence diagram frames, we use *tableRef* and *expr* to denote the table and the where clause referenced in the query, respectively. We use *tableRef2*, *expr2*, and *attr2* to denote the table, where clause, and attributes referenced in the subquery. Figure 13(a) shows nested subqueries corresponding to the SELECT operation that selects all attributes from a table. We call this specification *Select*. The optional aspect of the subquery is depicted using the *alt* label. When there are no subqueries (corresponding to the top half of the *alt*), the result of *SelectSimple* is returned. Otherwise, the execution corresponds to the bottom half of the *alt* box. First, the subquery is executed. The subquery typically selects the attributes *attr2* of table *tableRef2* for rows which satisfy *expr2*. Since *expr2* can be complex involving other nested queries, the subquery is processed by *SelectProj*. The subquery returns the expression *proj*. A new expression is created involving the result returned by the subquery and the original expression. This is denoted in the figure as *newexpr:=SelectExpr(proj,expr)*. This *newexpr* then becomes an input to the query. The query is handled by *SelectSimple* which returns all the attributes of the rows satisfying *newexpr* in *tableRef*. Figure 13(b) shows the sequence diagram frame of a query that selects some attributes from a table and has a subquery. Figures 13(c) and 13(d) show specific examples of SELECT having nested queries.



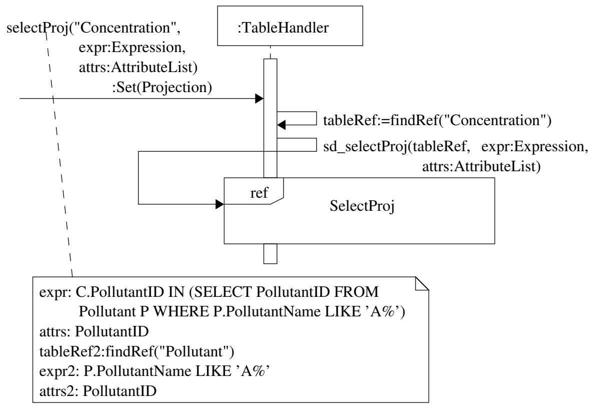
(a) A sequence diagram frame for SELECT operation that selects all attributes and has subquery


sd SelectProj
:TableHandler
sd_selectProj(tableRef:Set(Table),
expr:Expression,attrs:AttributeList)
:Set(Projection)
alt
[no subqueries]
proj:=sd_selectProjSimple(tableRef,expr,attrs)
:Set(Projection)
ref
SelectProjSimple
return proj
[subqueries]
proj2:=sd_selectProj(tableRef2,expr2,attrs2)
:Set(Projection)
ref
SelectProj
newexpr:=selectExpr(expr,proj2)
:Expression
proj:=sd_selectProjSimple(tableRef,newexpr,
attrs):Set(Projection)
ref
SelectProjSimple
return proj

(b) A sequence diagram frame for SELECT operation that selects some attributes and has subquery

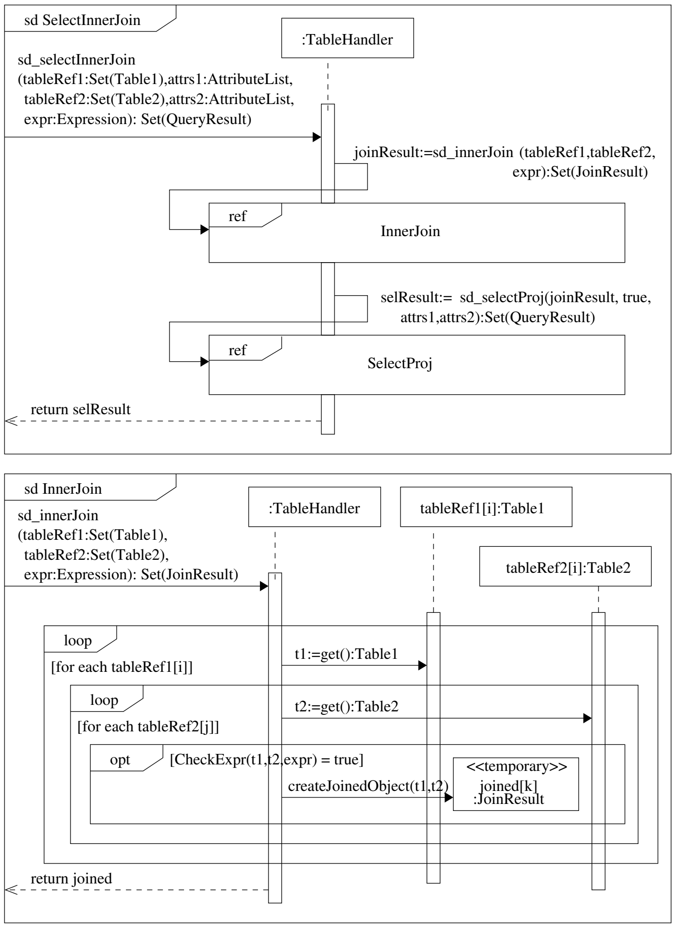

:TableHandler
select("Concentration",
expr:Expression)
:Set(QueryResult)
tableRef:=findRef("Concentration")
sd_select(tableRef, expr:Expression)
ref
Select
expr: C.PollutantID IN (SELECT PollutantID FROM
Pollutant P WHERE P.PollutantName LIKE 'A%')
tableRef2:findRef("Pollutant")
expr2: P.PollutantName LIKE 'A%'
attrs2: PollutantID

(c) A sequence diagram that represents a SELECT statement with a subquery: "SELECT * FROM Concentration C WHERE C.PollutantID IN (SELECT PollutantID FROM Pollutant P WHERE P.PollutantName LIKE 'A%')"

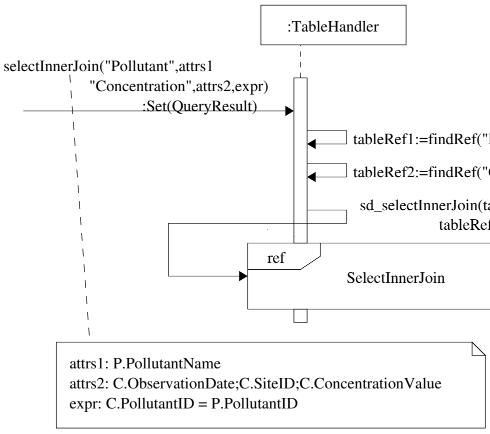

selectProj("Concentration",
expr:Expression,
attrs:AttributeList)
:Set(Projection)
:TableHandler
tableRef:=findRef("Concentration")
sd_selectProj(tableRef, expr:Expression,
attrs:AttributeList)
ref
SelectProj
expr: C.PollutantID IN (SELECT PollutantID FROM
Pollutant P WHERE P.PollutantName LIKE 'A%')
attrs: PollutantID
tableRef2:findRef("Pollutant")
expr2: P.PollutantName LIKE 'A%'
attrs2: PollutantID

(d) A sequence diagram that represents a SELECT statement with a subquery: "SELECT PollutantID FROM Concentration C WHERE C.PollutantID IN (SELECT PollutantID FROM Pollutant P WHERE P.PollutantName LIKE 'A%')"

Figure 13: Subquery operation sequence diagram



(a) A sequence diagram frame for an (INNER) JOIN operation: "SELECT Table1,attrs1, Table2.attrs2 FROM Table1, Table2 WHERE expr")



:TableHandler
selectInnerJoin("Pollutant",attrs1
"Concentration",attrs2,expr)
:Set(QueryResult)
tableRef1:=findRef("Pollutant")
tableRef2:=findRef("Concentration")
sd_selectInnerJoin(tableRef1,attr1,
tableRef2,attrs2,expr)
ref
SelectInnerJoin
attrs1: P.PollutantName
attrs2: C.ObservationDate;C.SiteID;C.ConcentrationValue
expr: C.PollutantID = P.PollutantID

(b) A sequence diagram that represents a SELECT statement with INNER JOIN: SELECT P.PollutantName, C.ObsevationDate, C.SiteID, C.ConcentrationValue FROM Concentration C, Pollutant P WHERE C.PollutantID = P.PollutantID

Figure 14: Sequence diagram for SELECT statement containing JOIN operation

Another complexity in SELECT statements arises when they have JOIN operations. Below we give an example of such a statement.

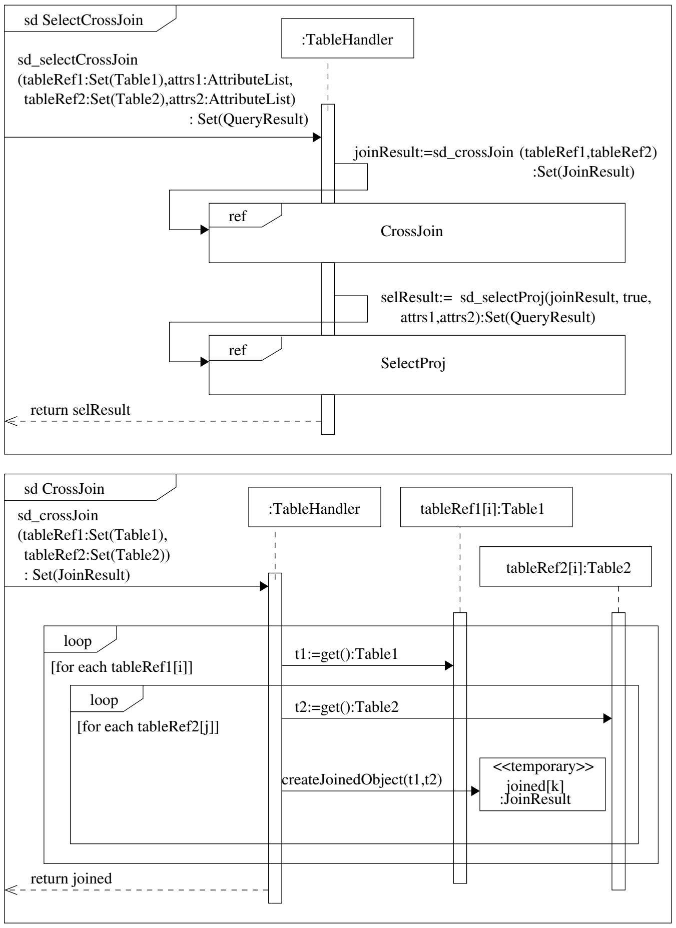
```
SELECT P.PollutantName, C.ObservationDate,
               C.SiteID, C.ConcentrationValue
FROM Concentration C, Pollutant P
WHERE C.PollutantID = P.PollutantID
```

The sequence diagram for the inner join operation is described in Figure 14(a). The SELECT statement containing a JOIN operation is processed in two steps as shown in the top part of Figure 14(a). First, an inner join operation is performed on the two tables, denoted by *tableRef1* and *tableRef2*, using the expression *expr*. The result, denoted as *joinResult*, is the input to the *SelectProj* operation. This operation selects *attrs1* and *attrs2* from *joinResult* and returns it to the user. The details of the inner join operation are given in the bottom part of Figure 14(a). The inner join operation is submitted to the *:TableHandler*. The *:TableHandler* selects *tableRef1* and *tableRef2* from the set of all tables. For each row of *tableRef1*, denoted by *tableRef1[i]*, all the attributes are fetched. Similarly, for each row in *tableRef2*, denoted by *tableRef2[j]*, all the attributes are fetched. The attributes are matched according to the conditions *expr*. If the conditions are satisfied, a temporary object called *joined[k]* is created. The process is repeated for all rows in *tableRef1* and *tableRef2* which is indicated by the *loop* boxes shown in the figure. An example inner join statement is shown in Figure 14. The outer join operations can also be modeled in a similar manner.

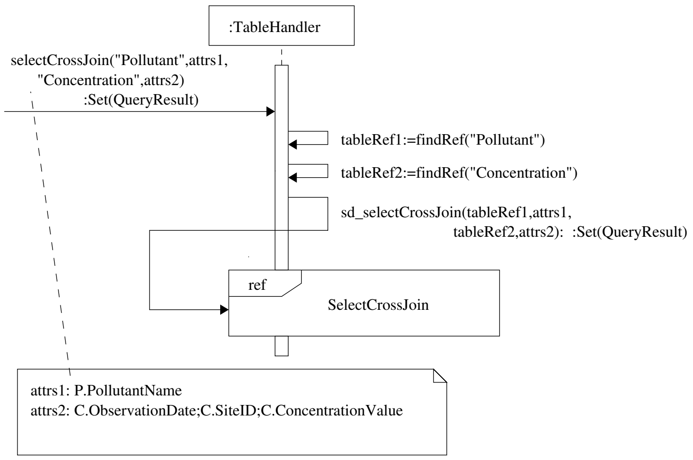
Related to the join operation is the cartesian product. Here is an example statement describing cartesian product.

```
SELECT P.PollutantName, S.SiteName
FROM Pollutant P, Site S
```

The detailed sequence diagram is given in Figure 15. The top part of Figure 15(a) is similar to that of the inner join. The main difference is in how the cartesian product is implemented which is shown in the bottom part of Figure 15(a). Here again, each row in *tableRef1* and each row in *tableRef2* are joined together to create a temporary object. However, unlike the join operation, the rows are always merged and no expression is checked before merging them.



(a) A sequence diagram frame for a CROSS JOIN operation (Cartesian Product):
"SELECT Table1.attrs1, Table2.attrs2 FROM Table1, Table2"

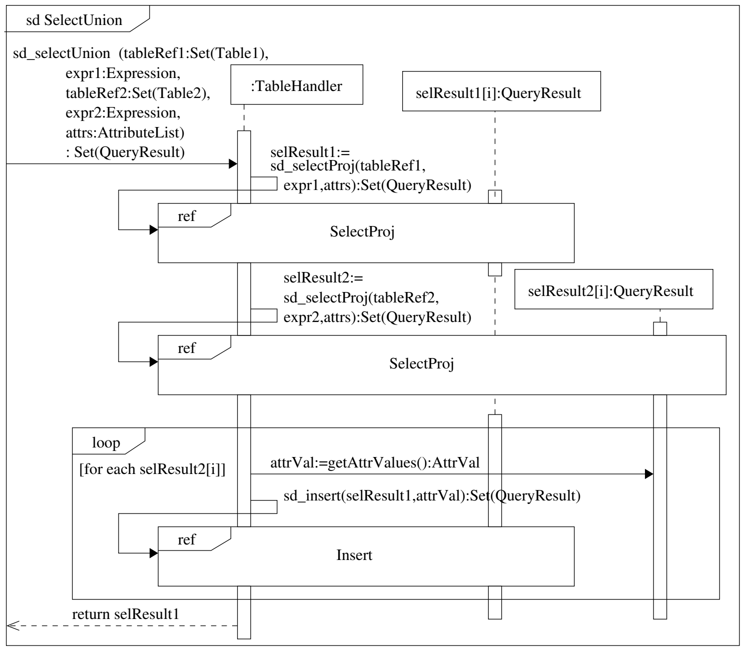

:TableHandler
selectCrossJoin("Pollutant",attrs1,
"Concentration",attrs2)
:Set(QueryResult)
tableRef1:=findRef("Pollutant")
tableRef2:=findRef("Concentration")
sd_selectCrossJoin(tableRef1,attrs1,
tableRef2,attrs2): :Set(QueryResult)
ref
SelectCrossJoin
attrs1: P.PollutantName
attrs2: C.ObservationDate;C.SiteID;C.ConcentrationValue

(b) A sequence diagram that represents a SELECT statement with CROSS JOIN:
SELECT P.PollutantName, C.ObsevationDate, C.SiteID, C.ConcentrationValue
FROM Concentration C, Pollutant P

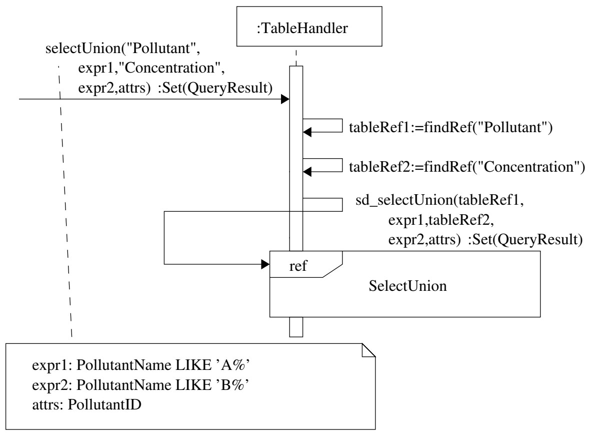
Figure 15: Product operation sequence diagram

Union operator can combine two union-compatible queries. Its semantic meaning is the same that in relational theory. We can model union operation as shown in Figure 16. The first step involves performing a select on *tableRef1* that is indicated by a *SelectProj* operation. The result that is returned is denoted by *selResult1*. The second step involves performing a select on *tableRef2* using the *SelectProj* operation. This select operation returns *selResult2*. Each row returned in *selResult2*, denoted by *selResult2[i]*, is inserted into *selResult1* using the insert operation. The resulting *selResult1* is returned to the user.

```
SELECT PollutantID FROM Pollutant
WHERE PollutantName LIKE 'A%'
UNION
SELECT PollutantID FROM Pollutant
WHERE PollutantName LIKE 'B%'
```


sd SelectUnion
sd_selectUnion (tableRef1:Set(Table1), expr1:Expression, tableRef2:Set(Table2), expr2:Expression, attrs:AttributeList) : Set(QueryResult)
:TableHandler
selResult1[i]:QueryResult
selResult1:= sd_selectProj(tableRef1, expr1,attrs):Set(QueryResult)
ref
SelectProj
selResult2:= sd_selectProj(tableRef2, expr2,attrs):Set(QueryResult)
selResult2[i]:QueryResult
ref
SelectProj
loop
[for each selResult2[i]]
attrVal:=getAttrValues():AttrVal
sd_insert(selResult1,attrVal):Set(QueryResult)
ref
Insert
return selResult1

(a) A sequence diagram frame for a UNION operation: "SELECT attrs FROM Table1 WHERE expr1 UNION SELECT attrs FROM Table2 WHERE expr2"


:TableHandler
selectUnion("Pollutant", expr1,"Concentration", expr2,attrs) :Set(QueryResult)
tableRef1:=findRef("Pollutant")
tableRef2:=findRef("Concentration")
sd_selectUnion(tableRef1, expr1,tableRef2, expr2,attrs) :Set(QueryResult)
ref
SelectUnion
expr1: PollutantName LIKE 'A%'
expr2: PollutantName LIKE 'B%'
attrs: PollutantID

(b) A sequence diagram that represents a UNION statement: SELECT PollutantID FROM Pollutant WHERE PollutantName LIKE 'A%' UNION SELECT PollutantID FROM Pollutant WHERE PollutantName LIKE 'B%'"

Figure 16: Union operation sequence diagram

The difference operation can be represented in various ways. Here is an example.

```
SELECT P.PollutantName
FROM Pollutant P
WHERE NOT EXISTS
(SELECT *
 FROM Concentration C
 WHERE P.PollutantID = C.PollutantID)
```

The semantic meaning of this query is to find all pollutant names which are not listed in *Concentration* table. This query essentially represents a "Difference" operation and its sequence diagram is shown in Figure 17. This query makes use of nested queries. Consequently, we use our previously defined constructs to model this operation.

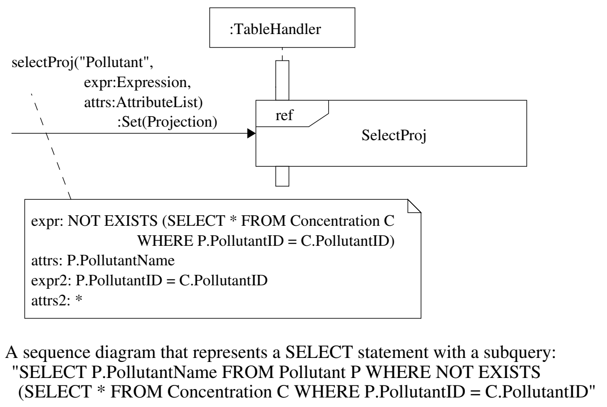

:TableHandler
selectProj("Pollutant",
expr:Expression,
attrs:AttributeList)
:Set(Projection)
ref
SelectProj
expr: NOT EXISTS (SELECT * FROM Concentration C
WHERE P.PollutantID = C.PollutantID)
attrs: P.PollutantName
expr2: P.PollutantID = C.PollutantID
attrs2: *
A sequence diagram that represents a SELECT statement with a subquery:
"SELECT P.PollutantName FROM Pollutant P WHERE NOT EXISTS
(SELECT * FROM Concentration C WHERE P.PollutantID = C.PollutantID"

Figure 17: Difference operation sequence diagram

5 Relational Database Operation Modeling

Most database products define functions, packages, and stored procedures to isolate portions of operations, hide implementation details and improve performance. These operations can be very complex and should be modeled the same as application level operations. No single diagram can model every aspect of a system. We can use a use case to represent an operation at requirement analysis level and use activity diagram, sequence diagram, communication diagram and state machine diagram at design and implementation level. On the other hand, we don't have to use all diagrams as long as we can clearly model the target system. Essentially, no matter how complex a database operation might be, it is composed of basic DML statement and SQL extension such as Transact-SQL or PL/SQL. We can use OO modeling techniques to model these language constructs. The use case template for an operation can be specified as shown in Table 1.

In this section, we will use a stored procedure as an example to demonstrate the techniques to model relatively complex database operations. Here is a system stored procedure dt_setpropertybyid used in SQL Server 7.0. It is implemented in Transact-SQL. Transact-SQL allows one to perform a series of database operations such as inserting, updatating, and/or deleting darabase records with conditions or iterations as needed. It is quite similar to a procedural programming language.

Operation Name	Give the name of the operation
Owner	Give the owner of the operation
Operation Type	Stored procedure, function, package, etc.
Involved Tables	List all tables used in this operation
Overview	A brief description of the operation
Precondition	Describe the precondition of the operation in OCL format, SQL expression or English statements.
Post Condition	Describe the post condition of this operation
Input Parameters	List all input parameters including name and data type
Output Parameters	List all output parameters including name and data type
Return Value	Give the return type

Table 1: Use case template for operations

```
CREATE PROCEDURE dbo.dt_setpropertybyid
      @id int,
      @property varchar(64),
      @value varchar(255),
      @lvalue image
  as
      set nocount on
      if exists (SELECT * FROM dbo.dtproperties
         WHERE objectid=@id AND property=@property)
      begin
      UPDATE dbo.dtproperties
          SET value=@value, lvalue=@lvalue, version=version+1
          WHERE objectid=@id AND property=@property
      end
      else
      begin
      INSERT dbo.dtproperties (property, objectid,
          value, lvalue)
          VALUES (@property, @id, @value, @lvalue)
      end
```

The above stored procedure can be modeled using use case as shown in Table 2. The use case template might vary slightly depending upon the specific RDBMS products.

We can use activity diagram, sequence diagram and other diagrams to model this stored procedure at requirement, analysis, design and implementation level. By combining all these diagrams together, we will have a much better understanding of complex database operations.

Operation Name	dt_setpropertybyid
Owner	dbo
Operation type	Stored procedure
Involved Tables	dtproperties
Overview	If the property already exists, reset the value; otherwise add property.
Precondition	dtproperties table has been created
Post Condition	A new record is added into dtproperties table if the property already exists; otherwise, reset the property.
Input Parameters	@id int, @property varchar(64), @value varchar(255), @lvalue image
Output Parameters	None
Return Value	None

Table 2: Use case of dt_setpropertybyid

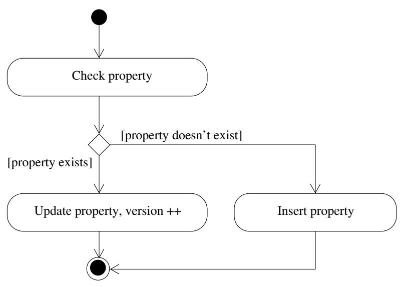


Figure 18: Stored procedure activity diagram

One advantage using activity diagram is that we can show some actions that are not directly associated with any tables, such as declaring a variable. However, activity diagram normally does not show detailed execution order. Sequence diagram and communication diagram can be used in this case. In fact, some CASE tools such as Rational Rose provide the functionality to attach activity diagram and state machine diagram to stored procedures. But forward and reverse engineering capability has not been implemented yet. Figure 19 shows the sequence diagram at implementation level. We use Select, Update and Insert statement as building blocks and place them inside predefined framed part such as alt to represent conditional execution.

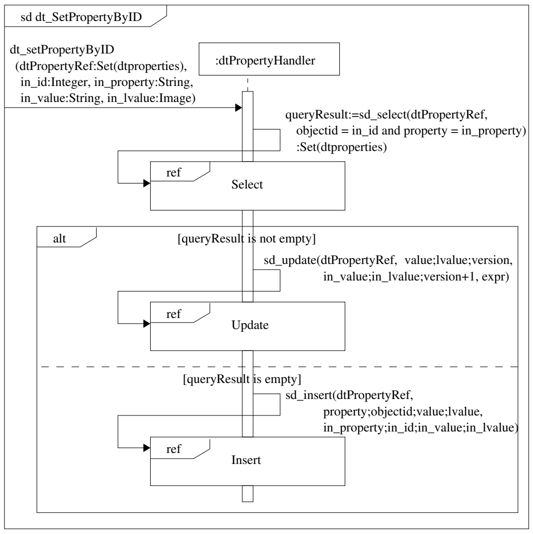

sd dt_SetPropertyByID
dt_setPropertyByID (dtPropertyRef:Set(dtproperties), in_id:Integer, in_property:String, in_value:String, in_lvalue:Image)
:dtPropertyHandler
queryResult:=sd_select(dtPropertyRef, objectid = in_id and property = in_property) :Set(dtproperties)
ref
Select
alt
[queryResult is not empty]
sd_update(dtPropertyRef, value;lvalue;version, in_value;in_lvalue;version+1, expr)
ref
Update
[queryResult is empty]
sd_insert(dtPropertyRef, property;objectid;value;lvalue, in_property;in_id;in_value;in_lvalue)
ref
Insert

Figure 19: Stored procedure sequence diagram

6 Conclusion

UML is being used as the de-facto standard in the software industry. Although OMG has released several UML Profiles for specific areas, relational database modeling is not adequately addressed in these profiles. Many existing software applications use relational database systems as back-end store. In this paper, we further expand the framework that can be used to model database-related operations using a set of UML diagrams. We believe that this approach will provide end-users and developers with a unified view of the whole system and bring the power of UML to database domain. In future, we plan to specify more complex database operations in details. Finally, we plan to develop tools that will help reverse engineer a complex database application into UML. Reverse engineering will not only help understand complex legacy database applications but will also facilitate integration of such applications.

References

[1] The UML and Data Modeling, White Paper. Technical report, Rational Software, 2003.

[2] Scott Ambler. *Agile Database Techniques*. John Wiley and Sons, Indianapolis, Indiana, USA, October 2003.

[3] Anreas Behm, Andreas Behm, Andreas Geppert, and Klaus R. Dittrich. On migration of relational schemas and data to object-oriented database systems. In *Proceedings of the 5th In-*

ternational Conference on Re-techologies for Information Systems, pages 13–33, Klagenfurt, Austria, December 1997.

[4] Tomas A. Bruce. *Designing Quality Databases with IDEF1X Information Models*. Dorset House Publishing Company Incorporated, New York, USA, October 1992.

[5] Peter Chen. The entity-relationship model- toward a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36, March 1976.

[6] Edgar F. Codd. A relational model of data for large shared data banks. *Communications of ACM*, 13(6):377–387, June 1970.

[7] Hans-Erik Erikson, Magnus Penker, Brian Lyons, and David Fado. *UML 2 Toolkit*. Wiley Publishing, Inc, Indianapolis, Indiana, USA, 2004.

[8] Martin Fowler. *UML Distilled, Third Edition: A Brief Guide to the Standard Modeling Language*. Addison-Wesley, Boston, MA, USA, September 2003.

[9] Davor Gornik. White Paper on the UML Data Modeling Profile. Technical report, Rational Software, May 2002.

[10] Martin Gruber. *Understanding SQL*. SYBEX Inc, Alameda, California, USA, 1990.

[11] Terry Halpin and Anthony Bloesch. Data modeling in UML and ORM: a comparison. *Journal of Database Management*, 10(4):4–13, October-December 1999.

[12] Robert J. Muller. *Database Design for Smarties: Using UML for Data Modeling*. Organ Kaufmann Publishers, Inc., San Francisco, California, USA, 1999.

[13] Eric J. Naiburg and Robert A. Maksimchuk. *UML for database design*. Addison-Wesley, Boston, MA, USA, 2001.

[14] William J. Premerlani and Michael R. Blaha. An approach for reverse engineering of relational databases. *Communications of the ACM*, 37(5):42–49, May 1994.

[15] Shekar Ramanathan and Julia Hodges. Reverse engineering relational schemas to object-oriented schemas. Technical Report MSU-960701, Mississippi State University, 1996.

[16] James Rumbaugh, Ivar Jasobson, and Grady Booch. *The Unified Modeling Language Reference Manual, 2nd ed.* Addison-Wesley., Reading, MA, USA, 2005.

[17] Toby J. Teorey, Dongqing Yang, and James P. Fry. A logical design methodology for relational database using the extended entity-relationship model. *Computing Survey, Vol. 18, Issue 2, June 1986, pp. 197-222.*, 18(2):197–222, June 1986.

[18] The Object Management Group. Model Driven Architecture. Technical report, http://www.omg.org/mda.

[19] The Object Management Group. Unified Modeling Language Specification. Technical report, http://www.omg.org.

[20] The Object Management Group. UML 2.0: Superstructure Specification. Version 2.0, OMG, formal/05-07-04, August 2005.

[21] Shuxin Yin and Indrakshi Ray. Relational database operations modeling with UML. In *Proceedings of the IEEE 19th International Conference on Advanced Information Networking and Applications*, Taipei, Taiwan, March 2005.