MATERIALIZED VIEW MAINTENANCE

FOR XML DOCUMENTS

FA YUAN

(B. Comp. (Hons.), NUS)

A THESIS SUBMITTED

FOR THE DEGREE OF MASTER OF SCIENCE

SCHOOL OF COMPUTING

NATIONAL UNIVERSITY OF SINGAPORE

2004

Acknowledgements

First of all, I would like to express my gratitude to my supervisor, Professor Ling Tok Wang, for his guidance and valuable advice, without which the work of this thesis would not have been possible.

I also appreciate the people in the Database Research Lab, Chen Yabing, Dong Xiaoan, Zhou Yongluan, Ji Liping, Chen Zhuo and Chen Ting, who are both very nice and helpful, and their presence has made the lab a nice place to work in.

I would also like to thank my parents for their constant support and care.

Fa Yuan

April 2004

Contents

List of Figures

Summary

Researches in the area of materialized view maintenance have gained popularity since 1990s due to its application in data warehousing. But the research on XML view maintenance is still limited. XML is rapidly emerging as a standard for publishing and exchanging data on the Web. Views over XML documents can be used to cache the interest data and to restructure it. People may be more interested in some small portion of the XML document rather than the whole set of documents. So we can specify XML views on these more interesting parts. Sometimes, we need to restructure the XML documents. Interchanging the ascendant/descendant relationships in XML data is possibly made to meet the specific needs of the database applications. Joining different XML documents is used to centralize the data. XML views are often materialized to speed up the query processing. Aggregation is often made to derive summarized information. People need only to query the materialized views rather than the whole XML source documents.

The consistency of the materialized XML view needs to be maintained against the updates of the underlying source data. Re-computing the XML materialized view from scratch each time a source XML document changes is not a feasible solution. In this thesis, we focus our work on incrementally maintaining the materialized XML view through the computation of view changes in an environment of multiple, distributed source XML documents, with a separate database for housing the XML

view.

We define the view, which can involve selection, project, join, swap and aggregation of elements on multiple source XML documents. The hierarchical structure in the view can be much different from any source. The reason we use ORA-SS to define the view is because by using ORA-SS schema diagram, we are able to define not only binary relationship type, but also n-ary relationship type, which helps define the views as we need.

Most of the existing view maintenance methods do not check whether the source update queries will make the source documents inconsistent. We will detect the invalid update query, which will make the XML document inconsistent. We defined a set of update operations with the XQuery syntax, which can be updates on both single element/attribute and subtree. The update consistency for each kind of update operation can be checked based on the ORA-SS data model. The essential constraints to validate an update query include participation constraint, key constraint, and functional dependency constraint, which can be all expressed in ORA-SS data model.

We generate view update tree which contains changes to the view and conforms to the view schema, such that we are able to merge the view update tree with the existing materialized view tree to produce the final updated view.

Aggregation attributes in the view are updated properly, when we merge the view update tree into the existing materialized view. Different strategies are taken for insertion, deletion and modification.

Beyond the normal generation of view update tree by querying all the source XML documents, we also provide view self-maintenance. By querying the XML view content, we can generate the view update tree much fast because the view resides locally while the source XML documents are remote. Information like object identifier constraint is used to achieve the view self-maintenance.

Chapter 1

Introduction

1.1 Problem Description

Database views are useful for restricting the data access rules, joining data from distributed databases, and caching commonly used data. Views can be materialized to speed up querying when the underlying data is remote, e.g., distributed, or query response time is critical [2, 5]. It is an important thing to keep the contents of the materialized view consistent with the contents of the base data as the source data are updated. Traditionally, people re-compute the sources to maintain the materialized view periodically. The current prevailing method is to compute the incremental changes to the view based on changes to the source data. In this thesis, we study the problem of incrementally maintaining materialized views for XML documents.

XML is rapidly emerging as a standard for publishing and exchanging data on the Web. Views over XML documents can be used to cache the interest data and to restructure it. People may be more interested in some small portion of the XML document rather than the whole set of documents. So we can specify XML views on these more interesting parts. Sometimes, we need to restructure the XML documents. Interchanging the ascendant/descendant relationships in XML data is possibly made to meet the specific needs of the database applications. Joining different XML documents is used to centralize the data. XML views are often materialized to speed up the query processing. People need only to query the materialized views rather than the whole XML source documents.

Incremental maintenance for materialized views in relational databases has been studied extensively [3, 11, 21] in the last few years. A survey can be found in [12]. Early work by Shmueli [18] and Blakeley [5, 6] focus on the question of incremental view maintenance in Selection-Projection-Join views and the detection of irrelevant updates. [5] and [18] use counts to annotate tuples in the view with the number of derivations. Gupta et.al. [11] extended the counting method to views with aggregates and (stratified) negation. The issue of view consistency in a concurrent warehouse environment has been studied recently. The paper [15], which incrementally maintains view using version number, is focusing to handle views over distributed source databases.

In order to maintain the materialized views for XML documents, theoretically, we can first transform all the XML documents into relations, and then use any existing relational maintenance algorithm to maintain the materialized views. The updates to the relational views are then transformed into updates to the XML views. Because each change to XML document may impact several relations, so the above maintenance method is not efficient. We will discuss it in more detail in Chapter 6. We need to find the method to directly maintain the materialized view for XML documents.

The study of materialized view maintenance for XML documents is still limited. The article [19] studies about the incremental view maintenance for semistructured data. It uses an algebraic approach to maintain the views. That is, it finds expressions that can compute delta views corresponding to the changes of base data. However, in [19], the view definition language is limited to select-project queries and only insertion update to the source document is considered. The article [20] studies the graph structured views and their incremental maintenance. However, it can only handle very simple views consisting of object collections, without edges. The article [2] studies the view maintenance for semistructured data based on the Object Exchange Model (OEM) [17] and on the Lorel query language [1] for OEM.

The above three papers have some common shortcomings. First, they do not validate the source update. Semantic constraints are not considered, such that they cannot confirm the XML document is still meaningful after the update. Second, the source updates they support are limited. For example, in insertion update, they do not support inserting an element with sub-elements. In modification update, they only support the atomic value change. Third, their view definition is too simple. They do not allow XML views that interchange the ascendant/descendant relationships in XML data, and they do not allow joining different XML documents also. Such views are natural in a tree structure data set. We will overcome the shortcomings in this thesis.

In this thesis, we introduce a set of incremental constraint checking rules to validate the source XML update based on the semantically rich Object – Relationship - Attribute model for Semi-structured data (ORA-SS) [10]. With these rules, we can make sure the source XML document is updated consistently and safely. We design the update operations consist of insertion, deletion and modification of both attributes and elements. The elements we can handle can be complex like consisting of sub-elements. We developed the incremental view maintenance to handle complex XML views, which may be resulting from interchanging ascendant/descendant relationships in source XML documents. Also joining of several XML documents are supported. The incremental maintenance algorithm is triggered to generate view update queries once an update happens to the source. Views defined in this thesis cannot generally be handled by techniques discussed in the other existing papers.

1.2 Motivating Example

In this thesis, we use an XML *Project-Supplier-Part* database as running example. The XML document 1 in Figure 1.1(a) consists of information on *suppliers*, *parts* supplied by each *supplier*, and *projects* that each *supplier* is supplying each *part* to. The XML document 2 in Figure 1.1(b) contains information on *projects* and the *department* that each *project* belongs to. We represent the document 1 and 2 as two ORA-SS instance diagrams respectively. The ORA-SS data model will be introduced in Chapter 3.

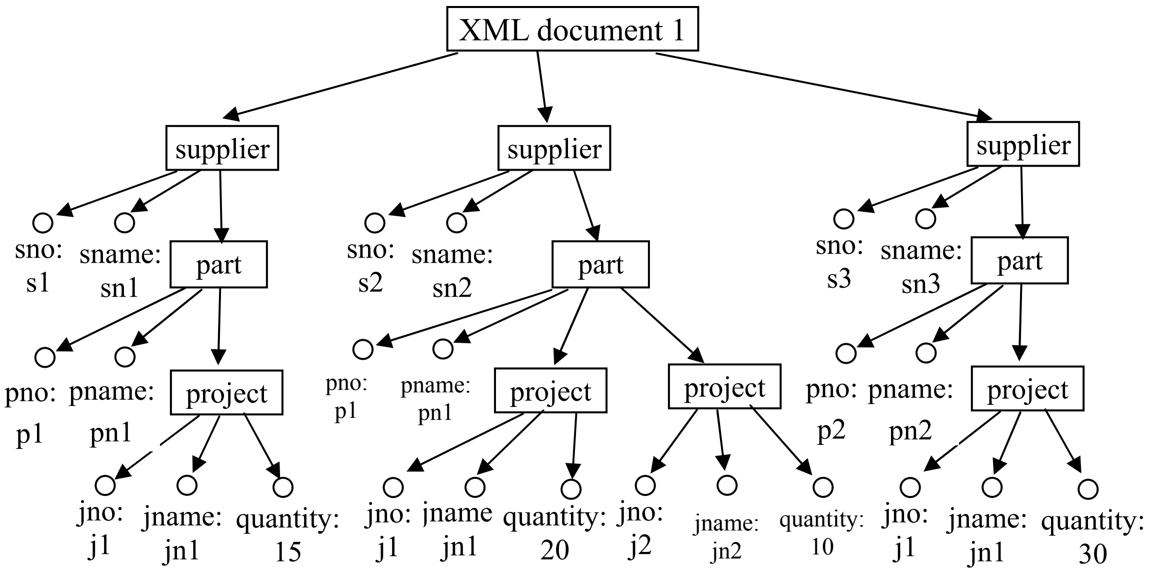


Figure 1. 1(a): ORA-SS Instance Diagram for XML Document 1 in *Project-Supplier-Part* Database

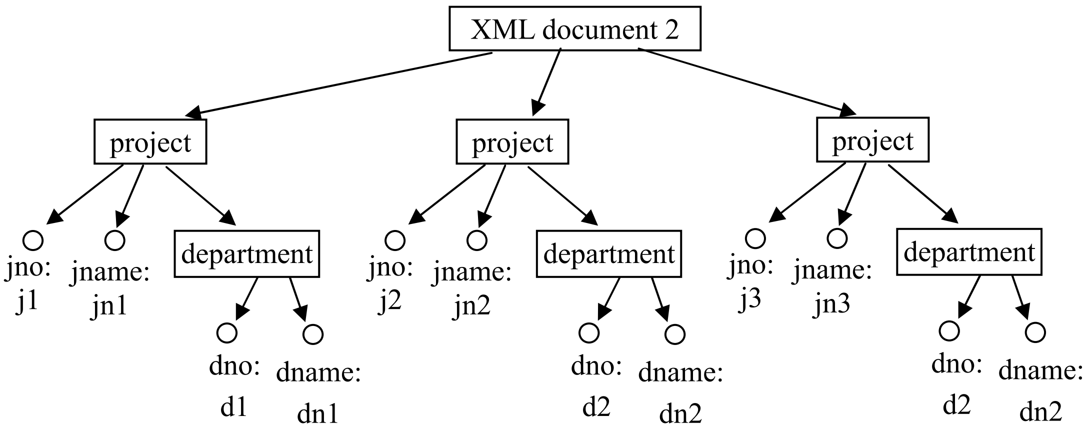


Figure 1. 2(b): ORA-SS Instance Diagram for XML document 2 in *Project-Supplier-Part* Database

We want to construct and maintain a view, which shows information of *project* of *department* dn1 and *parts* of each *project.* A new attribute called *total_quantity* is created, which is the sum of *quantity* of a specific *part* that the *suppliers* are supplying for the *project*. The initial content of the view is in Figure 1.2.

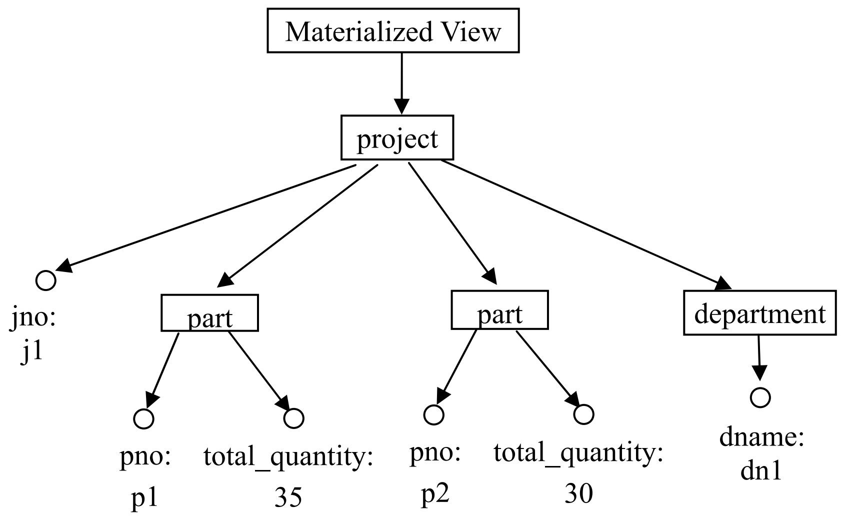


Figure 1. 3: XML View Content

Suppose *supplier* s3 is going to supply *part* p1 to *project* j1 with a quantity of 10. This will insert *part* p1 with child *project* p1 as the child element of *supplier* s3 in the source XML document 1. This source update will impact the view. The *total_quantity* of *part* p1 of *project* j1 will be increased by 10.

The updated materialized view is shown in Figure 1.3 with the updated part in the dashed circle. Compared with the whole materialized view, the update is relative small. To incrementally maintain the view is more efficient way to update the materialized view compared with the re-computation method.

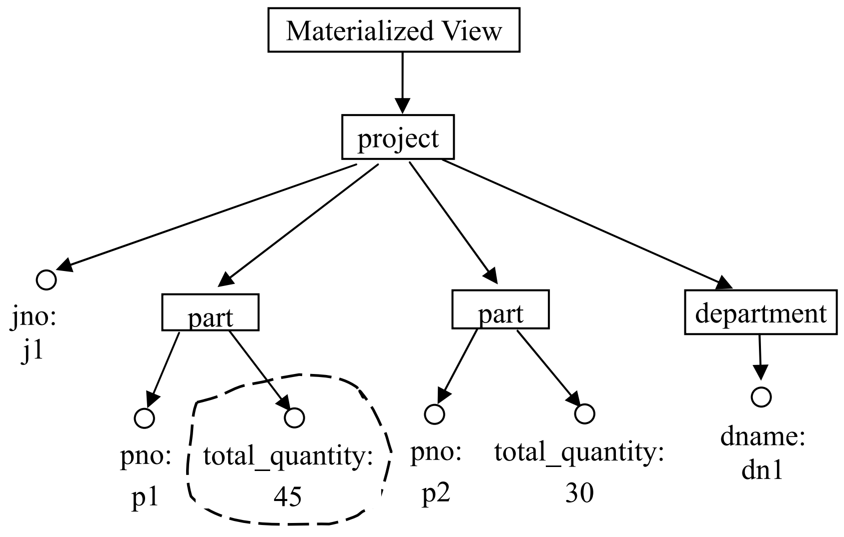


Figure 1. 4: Updated XML View Content

1.3 Research Contributions

In this thesis, we proposed an incremental view maintenance algorithm for XML documents in an environment of multiple, distributed source XML documents, with a separate database for housing the XML view.

We handle the update validation as the invalid update query will make the XML document inconsistent. We defined a set of update operations, which have the XQuery syntax. The update consistency for each kind of update operation can be checked based on the ORA-SS data model. The essential constraints to validate an update query include participation constraint, key constraint, and functional dependency constraint, which can be all expressed in ORA-SS data model.

We define the view in ORA-SS schema diagram, which can involve selection, project, join and swapping elements on multiple source XML documents. The hierarchical structure in the view can be much different from any source. Using ORA-SS schema diagram, we are able to define not only binary relationship type, but also ternary relationship type, which makes the view more meaningful.

We are able to query all the source XML documents to generate the view update tree. With ORA-SS view schema diagram, we are able to design the query plan according to the relationship types in the view schema.

Beyond the correct generation of view update tree, we also provide view self-maintenance when the update query meets the specific conditions. Information like key constraint is used to achieve the view self-maintenance for deletion and modification updates.

1.4 The Organization of this Thesis

The thesis is organized as follows.

Chapter 2 describes the ORA-SS data model and the reason why we choose ORA-SS as our data model.

Chapter 3 describes our XML update language and the validation rules to keep the XML document consistent after the update.

Chapter 4 discusses the view definition in ORA-SS schema diagram and how to make the materialized view.

Chapter 5 presents the algorithm to incrementally maintain the materialized views for XML documents.

In Chapter 6, we describe the previous works on the area of materialized view maintenance and provide a comparison between these works with that of ours. We conclude that our approach is better than the existing works because we are able to handle more complex views.

Chapter 7 discusses the conclusion and suggestions for further work.

Chapter 2

The ORA-SS Data Model

The data model we are using is ORA-SS (Object-Relationship-Attribute model for Semi-Structured data) [10]. We adopt ORA-SS because it is a semantically richer data model that has been proposed for modeling semi-structured data compared to OEM or Dataguide. Using ORA-SS, we can define flexible XML views, and develop efficient incremental view maintenance algorithm.

There are three main concepts in the ORA-SS data model, which are **object class**, **relationship type** and **attribute** (of object class or relationship type). The ORA-SS data model not only reflects the nested structured of semi-structured data, but also distinguishes object classes, relationship types and attributes. The main advantages of ORA-SS over existing data models are its abilities to specify functional dependency and referential integrity constraints. These semantics are essential for implementing an efficient XML view management system.

2.1 Object Classes

An object class in ORA-SS is like a set of entities in the real world, an entity type in an ER diagram, a class in an object-oriented diagram or an element in semi-structured data model. An object class is represented as a labeled rectangle.

Example 2.1 Consider an example where each project can have a project no, a project name, and budget. This is represented in Figure 2.1 by an object *project* with key *jno*, and attributes *sname* and *budget*.

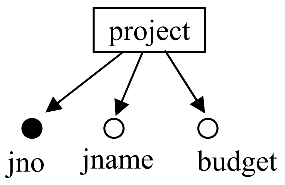


Figure 2.1: Object Project with Attributes in an ORA-SS Schema Diagram

2.2 Relationship Types

A relationship type in the ORA-SS data model represents a nesting relationship. An object class is related to another object class through a relationship type. Each relationship type has a degree and participation constraints. A relationship type of degree 2 (i.e. a binary relationship type) relates two object classes. One object class is the parent and the other is the child. A relationship type of degree 3 (i.e. a ternary relationship type) is a relationship type between three objects classes. In a ternary relationship type, there is a binary relationship type between two object classes, and a relationship type between this binary relationship type and the other object class.

A relationship type is represented by a labeled diamond in an ORA-SS schema diagram. The label, “name, n, p, c”, contains a relationship type name, an integer n indicating the degree of the relationship type (n = 2 indicates binary, n = 3 indicates ternary, etc.), the participation constraint p on the parent of the relationship type, and the participation constraint c on the child. By defining participation constraints with min:max notation, we are also able to represent numerical constraints. ?, * and + are the usual shorthand to represent the participation constraints 0:1, 0:n, and 1:n respectively. All fields in the label are optional. There is no default value for name. The default value for degree is 2. The default value for the parent participation constraint is *0:n* and the default value for the child participation constraint is *1:m*.

Example 2.2 Figure 2.2 shows a binary relationship type between *project* and *supplier*, a binary relationship type between *supplier* and *part*, and a ternary relationship type between *project*, *supplier* and *part*. The relationship type between *project* and *supplier* is annotated with “js, 2, 0:n, 0:n”, which represents a many to many relationship between *project* and *supplier*. The ternary relationship type *jsp* is a relationship type between the *project* and *supplier* relationship type and *part*. The schema in Figure 2.2 models the relationship between parts supplied by a particular supplier while supplying for a particular project, and only the parts supplied by a supplier while supplying for a project will be nested within that supplier and project.

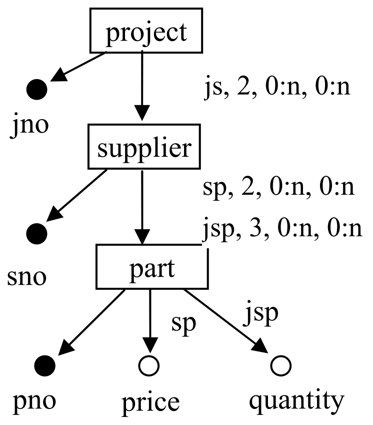


Figure 2.2: Representing ORA-SS Relationship Types

2.3 Attributes

Attributes represent properties. An attribute can be a property of an object class or a property of a relationship type.

Attributes are denoted by labeled circles, the label consists of name, [F|D: value]. The name is compulsory, and the rest of the label is optional. The letter F precedes a fix value, while D precedes a default value. The identifiers are indicated by filled circles, while other candidate keys are a double circle with the inner circle filled. An attribute's cardinality is shown inside the attribute circle, using ?, *, + to represent 0:1, 0:n, 1:n, where the default is 1:1. An attribute can be single-valued or multi-valued. A multi-valued attribute is represented using an * or + inside the attribute circle.

The special attribute name ANY denotes an attribute of unknown or heterogeneous structure.

Attributes of an object class can be distinguished from attributes of a relationship type. The former has no label on its incoming edge while the latter has the name of the relationship type to which it belongs on its incoming edge.

Example 2.3 Consider the ORA-SS schema diagram in Figure 2.2. The object *part* has a key attribute *pno*. The attribute *price* belongs to the relationship type, *sp*, between *supplier* and *part*, i.e. it is the price for a part supplied by a supplier. Attribute *quantity* belongs to the relationship type, *jsp*, between *project*, *supplier* relationship type and *part*, i.e. it is the quantity of a part supplied by a supplier for a specific project.

2.4 Functional Dependencies

Functional dependencies model real world constraints, showing how some of the attributes depend on other attributes. The functional dependencies of binary relationships can be derived from the schema diagrams. Separate functional dependency diagrams are drawn for ternary or other functional dependencies. With the separate functional dependency diagrams, we can express more information, and the information can be expressed without crowding the ORA-SS diagrams. For each functional dependency, the values of a set of objects (we call them conditional objects) determine the value of certain objects or attributes (we call them resulting objects/attributes). Sample XML functional dependency is given in Example 2.4.

Example 2.4 Consider the ORA-SS schema diagram in Figure 2.2. An instance of this schema is shown in Figure 2.3. In the schema diagram, attribute *price* is the attribute of the relationship type *sp*. One functional dependency is enforced such that one *supplier* supplies one part at the same *price* to all *projects*. The instance in Figure 2.3 satisfies this functional dependency since for different *project* j1 and j2, *supplier* s2 provide *part* p2 at the same *price* 300.

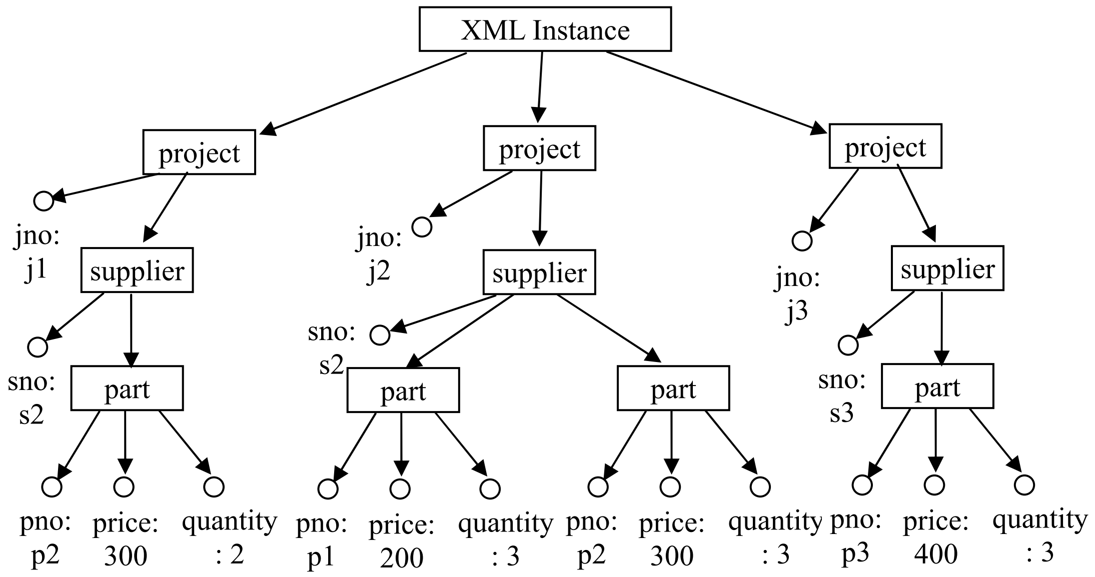


Figure 2.3: Demonstrating Functional Dependency

We provide the ORA-SS schema diagrams for our *Project-Supplier-Part* database in Figure 2.4.

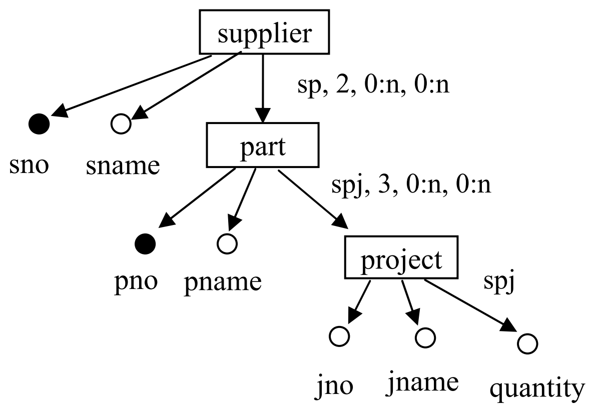


Figure 2.4 (a): ORA-SS Schema Diagram for XML Document 1 in *Project-Supplier-Part* Database

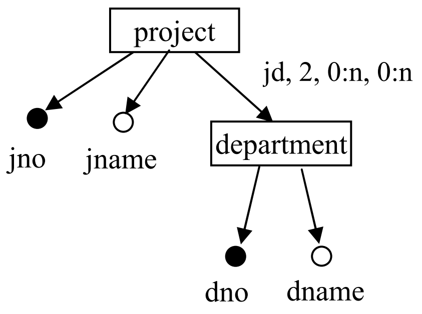


Figure 2. 4(b): ORA-SS Schema Diagram for XML Document 2 in *Project-Supplier-Part* Database

Existing semi-structured data models, like OEM, are not possible to represent the participation constraints of object classes in relationship types, whether an attribute is an attribute of an object class or an attribute of a relationship type, and the degree of n-ary relationship types for the hierarchical semi-structured data. The inadequacy of the Dataguide is its inability to express the degree of n-ary relationships for the hierarchical semi-structured data. Also Dataguide cannot express the functional dependency constraint.

An algorithm has been developed to extract the ORA-SS schema from XML documents. The algorithm has two steps. The first step is to process the XML document and generate a rough ORA-SS schema tree, which contains hierarchical information only. The second step is to ask the user necessary questions, and refine the ORA-SS schema according to the answers provided by the user. This information includes primary key and candidate keys, degrees of relationship types, participation constraints in relationship types, logic residence of attributes (whether an attribute belongs to an object class or to a relationship type), etc. Such information cannot be derived by scanning XML documents only. After answering all the questions, the ORA-SS schema will contain much more semantic information, and the user can still make changes on the properties of object classes, relationship types and attributes to refine the schema.

Chapter 3

XML Document Update

The source update can be an insertion, a deletion or a modification. The insertion operation inserts a sub-tree of object classes into a source XML document. The deletion operation deletes a sub-tree of object classes from a source XML document. The modification operation modifies the value of attribute of an object class or a relationship type in ORA-SS schema.

3.1 XML Update Language

We propose our simple XML update language in this chapter. As a good XML update language, it should be able to specify both the update point of the XML document and the update content clearly. The update point should be expressed as a path from the root of the XML document to the specific element, where the update takes place. The update content should be constructed as a XML sub-tree. It should not be represented as object ID or other internal representation as in Lorel update statement [1]. Details of it will be discussed in Chapter 6. Our XML update language is designed for clear specification of both update path and update content.

The World Wide Web Consortium has proposed an XML query language called XQuery [23]. XQuery provides flexible query facilities to extract data from real and virtual documents on the Web. The basic form of an XQuery expression consists of For, Let, Where and Return (FLWR) expressions. XQuery currently does not provide for the definition of updates.

We propose the new XQuery syntax with update language added in Figure 3.1.

```
update doc-name{
    for attr1 in XPath-expr1, attr2 in XPath-expr2, …
    let attr3 := XPath-expr3, attr4 := XPath-expr4, …
    where selection_pred1, selection_pred2, selection_pred3, …
    action1; action2; …; actionn
}
```

Figure 3.1 Syntax of Our Update Language Extending XQuery

Each $action_i$ above is an expression of the form

insert *r* **into** *e* [AT LAST]

or

delete *e*

or

replace *e* **with** *v*,

where *r* is an XML sub-tree, *e* is a simple XPath [24] expression, and *v* is text value. We use the key attribute of the objects to represent the path. For example, *e* is supplier[sname = ‘s1’]/part, which matches *part* elements that are descendants of *supplier* elements that have an attribute *sname* whose content is the string value “s1”.

In an INSERT action, the expression *e* specifies a node, *N*, immediately below which a subtree will be inserted. The subtree is specified by the expression *r*. By default, e is inserted after the last child of r. So the keyword AT LAST can always be omitted.

In a DELETE action, expression e specifies a node which will be deleted (together with its sub-tree).

In a REPLACE action, expression *e* specifies an attribute which will be modified. The new attribute value replacing *e* is specified by *v*.

Example 3.1 Consider the XML *Project-Supplier-Part* database in Figure 1.1. Suppose *supplier* s3 is going to supply *part* p1 to *project* j1 with a quantity of 10. This will insert *part* p1 as the child element of *supplier* s3 in the source XML document 1. *part* p1 has a child element *project* j1 with a quantity of value 10. In the update language, a subtree will be inserted to document 1 as follows:

```
update document1{
   let $r1 := “<part pno = ’p1’ pname = ‘pn1’>
                    <project jno=’j1’ jname = ‘jn1’>
                         <quantity>10</quantity>
                    </project>
                 </part>”

   insert $r1 into /supplier[sno = “s3”]/ AT LAST;
}
```

Example 3.2 Suppose *supplier* s2 will not supply *part* p1 to *project* j1 any longer. This will delete *project* j1 from *part* p1 of *supplier* s2. We form the following update query.

```
update document1{
   delete /supplier[sno = “s2”]/part[pno = “p1”]/project[jno = “j1”]
}
```

Example 3.3 Suppose *supplier* s2 will supply *part* p1 to *project* j1 with *quantity* 30 instead of 20. This will update the value of attribute quantity from 20 to 30.

```
update document1{
   for $a in /supplier[sno = “s2”]/part[pno = “p1”]/project[jno = “j1”]
   replace $a/quantity/text() with “30”
}
```

We have defined the XML update query language in the XQuery syntax. All the update queries can also be translated into graphical presentation in the form of ORA-SS instance diagram, which will be shown in Chapter 5. In order to keep the XML database consistent, we need to valid the update query before it is executed in the database. We discuss it in the next section.

3.2 Update Validation

There are two levels of validation for an XML document: well-formed and valid against a data model. An XML document is well formed if it follows all specifications of the World Wide Web standard. That means the XML document should satisfy two conditions. One is the ending tag matches with the beginning tag. The other is no two attributes of the same element have the same name. When a well formed XML document is associated with a schema, and it satisfies all the constraints expressed in the schema, we say the XML document is valid. The XML schema we are going to use is ORA-SS. We now present the validation rules based on ORA-SS, which should be enforced when an update operation takes place on the XML document. The constraints to be verified include functional dependency constraint, participation constraint, key constraint, and structure checking. We assume that the XML document is initially well formed and valid against the ORA-SS schema.

Rule 1: Functional Dependency Constraint Rule

This rule guarantees none of the functional dependencies in the XML document are violated. For each functional dependency, the values of a set of objects (called conditional) determine the value of certain objects or attributes (called resulting). Upon an insertion or modification update, if any functional dependency is affected, we will verify the functional dependency. Instead of verifying the functional dependency on the whole updated XML document, we will verify the functional dependency incrementally. The disadvantage of the full verification of functional dependency is time-consuming, and if the update violates the functional dependency, the time to apply the update on the XML document and verify the functional dependency on the updated XML document is wasted. So we will discover the way to incrementally verification of functional dependency.

For each affected instance F_a of any functional dependency, we just need to find another instance F_b of the same functional dependency with the same values of conditional objects in the original XML document. If there is no other instance of the same functional dependency, then the affected instance is satisfied with the functional dependency. Otherwise, we compare the values of the resulting objects and attributes of F_a and F_b. If equal, then F_a is satisfied with the functional dependency. Otherwise, Fa is not satisfied with the functional dependency.

If any of the affected instances of the functional dependency does not pass the functional dependency constraint check, we fail the source update.

Example 3.4 Consider the ORA-SS schema diagram in Figure 3.2, one functional dependency enforced is one supplier supplies one part at the same price to all projects. An instance of the schema is shown in Figure 3.3. Suppose now *supplier* s2 supplies *part* p2 at *price* 200 to *project* j3. We need to determine whether the update is valid.

We look for one relationship type *sp* from the original ORA-SS instance in Figure 3.3. If we did a depth first search, we will find *supplier* s2 is supplying *part* p2 at *price* 300 to *project* j1. Since the value of price is different from the price in the update, we conclude that the update will violate the functional dependency constraint rule. The update is invalid, and will be rejected.

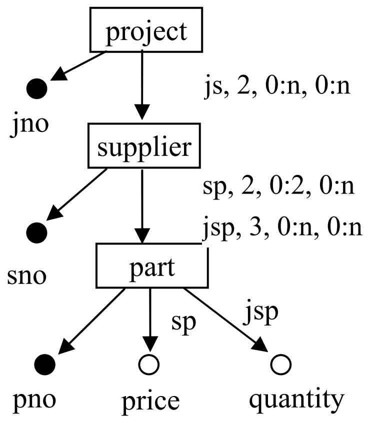


Figure 3.2: ORA-SS Schema Diagram Demonstrating Functional Dependency Constraint Rule

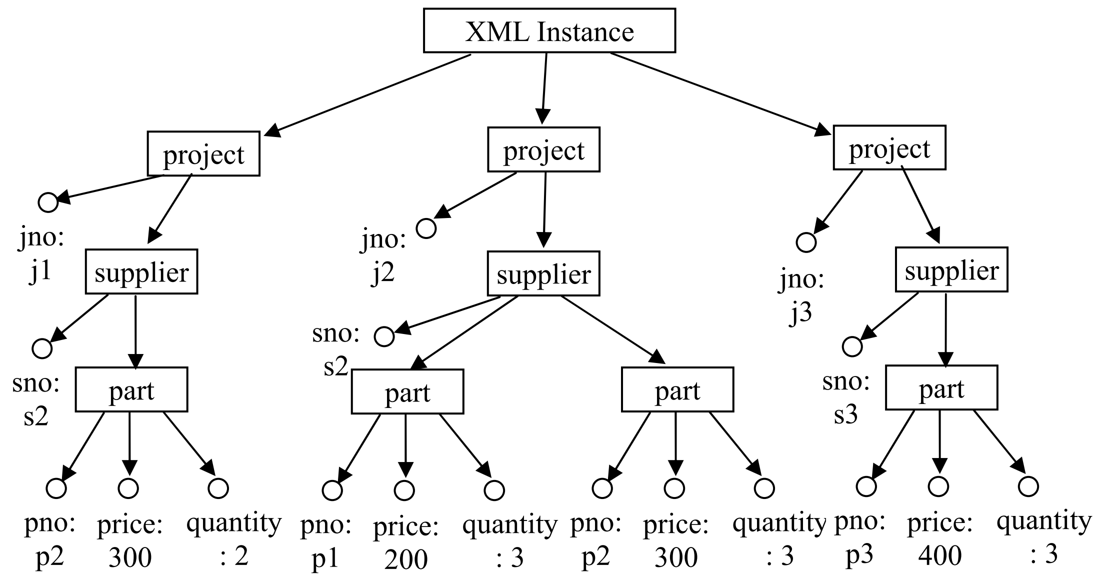


Figure 3.3: ORA-SS Instance Diagram Demonstrating Functional Dependency Constraint Rule

Rule 2: Participation Constraint Rule

This rule guarantees none of the participation constraint rules are violated. As illustrated in the Chapter 2, a relationship in the ORA-SS schema diagram has two participation constraints, one is the participation constraint on the parent of the relationship, and the other is the participation constraint on the child. The two participation constraints have the form of min:max. We say the minimum constraint is the minimum value in the participation constraint for either parent object class or child object class. The maximum constraint is the maximum value in the participation constraint. Table 3.1 shows all the participation constraint rules for insertion and deletion update. Since the modification update will only modify the value of attribute, but not object class or relationship type, so it will never violate the participation constraint in the schema.

Table 3.1 Participation Constraint Rules for Different Types of Update

Insertion	Parent Object Class	**Rule (1)** If one parent object class *P* of relationship type *R* is inserted, we need to check whether the maximum constraint of *P* is violated, also we need to check whether the minimum constraint of the child object class/relationship type of *R* is violated. For example, the relationship type course-student requires that a course has at least 6 students, so the insertion of a new course without student will violate the minimum constraint of the child object class of the relationship type, which is at least 6 students for a course.
	Child Object Class	**Rule (2)** If one child object class C of a relationship type R is inserted, we need to check whether the maximum constraint of the parent object class P of R is violated. For example, the relationship type course-student requires that a course has at most 60 students, so the insertion of a new student will violate the maximum constraint of the Course if the course has 60 students already before the insertion.
	Relationship Type	**Rule (3)** If one relationship type R is inserted, we have to check the participation constraints of each object classes of R as in Rule (1) and (2).

Deletion	Parent Object Class	**Rule (4)** If one parent object class *P* of relationship type R is deleted, we need to check whether the minimum constraint of *P* is violated. For example, the relationship type course-student requires that a student has to take at least four courses, so the deletion of an existing course may result in that the students that are taking the course take three courses after the deletion.
	Child Object Class	**Rule (5)** If one child object class C of a relationship type R is deleted, we need to check whether the minimum constraint of the parent object class P of R is violated. For example, the relationship type course-student requires that a course has at least 6 students, so the deletion of an existing student will violate the minimum constraint of the Course if the course has exactly 6 students before the insertion.
	Relationship Type	**Rule (6)** If one relationship type is deleted, we have to check the participation constraints of each object classes of R as in Rule (4) and (5).

Example 3.5 Consider the ORA-SS schema diagram in Figure 3.2 and its instance diagram in Figure 3.3. For the relationship type *sp* between *supplier* and *part*, the participation constraint for the parent object class *supplier* is set to be 0:2. That means a supplier can not supply more than two parts for a specific project. Suppose now we want to insert a new *part* p3 to *supplier* s2 for the *project* j1, it is valid since the constraint for project is not violated. But if we want to insert a new *part* p3 to *supplier* s2 for the *project* j2, it is invalid. Because the update will cause the *supplier* s2 of *project* j2 has more than two parts.

For each update, the above constraint checking rules are applied accordingly. The XML document will be kept consistent with its ORA-SS schema diagram after each update. Such that the semantic rules enforced in the XML document will remain. This property is necessary for the future processing of the XML document. In the next chapter, we are going to discuss the specification of the view using ORA-SS schema diagram and also the initialization of the materialized view.

Chapter 4

Views and Materialized Views

In this Chapter, we discuss how to define the flexible views over multiple source XML documents. There are two main approaches. One way is to define views or queries in script language like XQuery [23]. The alternative approach is to define views through source schema and view schema mappings. The latter approach alleviates user from writing complex scripts to define an XML view. Then we use the view transformation method to initialize the materialized view. The view transformation method is first proposed in [8]. Here we enrich the method to handle the complex views which can be over multiple source XML documents, have selection conditions, and have aggregation functions.

4.1 View Specification

We use ORA-SS schema diagram to specify the XML view with the following semantic meanings.

Selection: In relational database, one common feature is the selection applied to relation tuples. In the ORA-SS schema diagram of the view, we specify a selection condition via a predicate associated to an object or attribute in the ORA-SS schema diagram.

Projection: Another way to project out the interest data from source XML documents is to specify which nodes are projected, and which nodes are eliminated from the source XML documents. All objects and attributes in the ORA-SS schema diagram of the view are supposed to be projected from the source XML.

Join: Similar to relational database, we have join for a set of source XML documents. Our join is strictly more general than relational join. It could be the joining of different elements either in the same document or in the different documents. In the ORA-SS schema diagram of the view, you will see one joined object class only instead of the original two object classes.

Swap: One great feature about XML is its heterogeneity. XML documents have complex tree structures, so does the XML view. We allow the new relationship to be created in the ORA-SS schema diagram of the view. More precisely, two object classes with parent/child relationship in the ORA-SS schema diagram of the view do not necessarily have such relationship in any source XML document. They even do not have to come from the same source XML document.

Aggregation: The purpose of aggregation is to map collections of values to aggregate or summary values. Common aggregate functions are MIN, MAX, COUNT, SUM, AVG, etc. Aggregate functions can be applied to the attributes of object class or the relationship type to derive new attributes. When generating summary values, we should specify exactly where the newly computed value should be inserted. In the ORA-SS schema diagram of the view, we will specify the aggregated attribute and its aggregate function.

When the key value of certain object class is not projected in the view schema, we add a count attribute to that object class. This count attribute is to record the number of same instances of the object class in the same path of the view. So that we do not have to store the same instances multiple times. This count attribute is treated as an attribute with a COUNT aggregate function.

All in all, ORA-SS allows users to define view schema with rich semantic meanings. The following example shows the power of ORA-SS.

Example 4.1 Figure 4.1 depicts a view based on the two source schemas in Figure 2.4 by using schema mapping method. The view swaps, joins and drops the object classes in the source schemas. It shows information of *project* of *department* dn1 and *part* of each *project.* Object class *supplier* is dropped from the source schema 1. *part* and *project* are swapped. A new relationship type *jp* is created between *project* and *part*. A new attribute called *total_quantity* is created for *jp*, which is the sum of *quantity* of a specific *part* that the *suppliers* are supplying for the *project*.

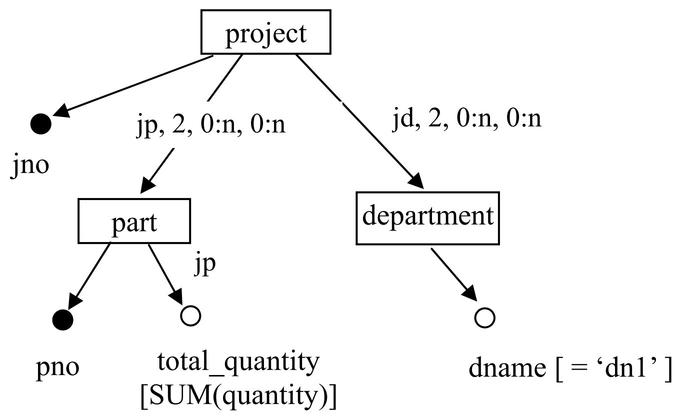


Figure 4.1: ORA-SS Schema Diagram of the View over Source Schemas in Fig. 2.4

The initial content of the materialized view specified in Figure 4.1 will be shown in Figure 4.2.

4.2 View Materialization

Having designed XML views based on ORA-SS, we can produce the views content based on some materialization strategy. In this thesis, we adopt view transformation strategy in [8] to produce materialized views. It can perform accurate and efficient view transformation based on ORA-SS. But the method is only transforming a single source ORA-SS schema to a view schema. Here we enrich the method to handle the complex views which can be over multiple source XML schemas, have selection conditions, and have aggregation functions.

Figure 4.2 depicts the materialized view for the view schema in Figure 4.1. To emphasize the importance of aggregation, we can see from this example, in the materialized view, the value of *total_quantity* for *part* p1 is 35 for project j1, which is the sum of *quantities* 15 and 20 supplied by *supplier* s1 and s2 in source XML document 1.

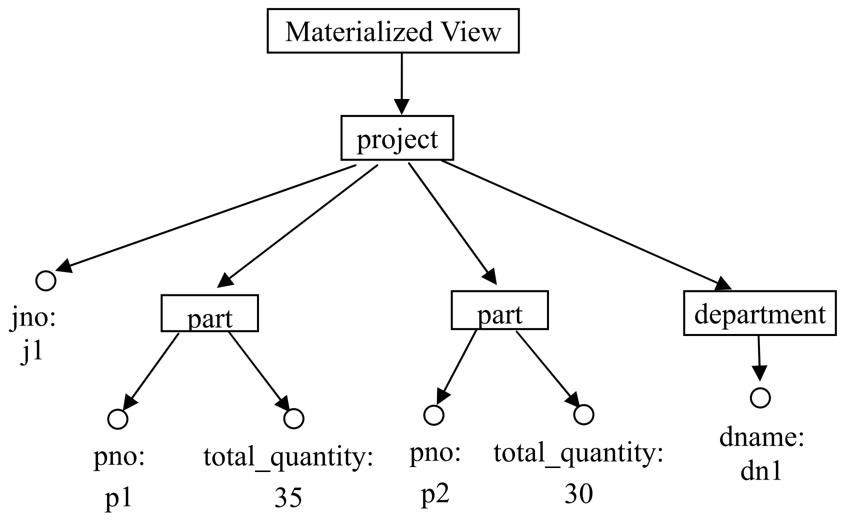


Figure 4.2: ORA-SS Instance Diagram of the View

In outline, the view materialization plan has the following four main procedures:

1) *Projection (on object type or relationship type)*
2) *Selection (on attribute of object class or relationship type)*
3) *Join (different object classes)*
4) *Aggregation (on attributes)*

In the following, let us discuss each of the procedures in more details.

The **Projection Procedure** selects instances of object classes and relationship types from the source XML documents. The paper [8] presents the strategy to retrieve the instances of object classes or relationship types by querying the source XML documents according to the object classes and relationship types in the view schema.

The **Selection Procedure** prunes the instances retrieved from Projection Procedure by checking the selection conditions in the view schema. We enforce the selection condition on the attributes of the object class or the relationship type if there is any. If the value satisfies the condition, we keep the instance, if it does not satisfy, we delete it. This procedure is not present in the paper [8] as it does not consider the selection conditions expressed in view schema.

The **Join Procedure** joins the elements with the same name and key attributes together from different source XML documents. The combined instance of the object class or relationship type has all the attributes together. This procedure is not present in paper [8] as well because the previous work does not consider the complex view which is over multiple source XML documents.

The **Aggregation Procedure** applies the aggregation function to the values of aggregate attribute if there is an aggregation function associated with the attribute. This procedure is not present in paper [8] as well because the previous work does not consider the aggregation function in the view specification. Table 4.1 shows all possible types of aggregation functions that we can handle.

Table 4.1 Cases of the Aggregation Functions in the View Specification

Type	Method
(1) COUNT	We will count the number of object classes or attributes the aggregation function is applied to and store the value in the aggregate attribute
(2) SUM	We will add all the values of the attribute that the aggregation function is applied to and store the summation in the aggregate attribute
(3) AVG	We will apply the two methods for calculating COUNT and SUM aggregate functions on the attribute, then we divide SUM by COUNT, and store the average value in the aggregate attribute
(4) MAX	We find the maximum value among all the values of the attribute the aggregate function is applied to and store it in the aggregate attribute
(5) MIN	We find the minimum value among all the values of the attribute the aggregate function is applied to and store it in the aggregate attribute

The following example shows the procedures used to initialize the materialized view based on the view materialization plan discussed above.

Example 4.2 We use the XML *Project-Supplier-Part* database in Figure 1.1. The view schema is defined in Figure 4.1. In Figure 4.3, we show the steps of creating the initial content of the materialized view. In our algorithm, we firstly retrieve the instances of project and department from source XML document 2 in Figure 1.2 (b). The result is shown in Figure 4.3 (a). Then we apply the selection condition of *department* name on the results. Only the *projects* of *department* d1 are left in Figure 4.3 (b). Then we join the *projects* in Figure 4.3 (b) with the *projects* in source XML document 1. The instances of joined *project* and *part* are retrieved as shown in Figure 4.3 (c). Lastly, aggregation function SUM is applied on attribute *quantity* of relationship type *jp* to produce the aggregate attribute *totoal_quantity*, which is shown in Figure 4.4 (d). Figure 4.4 (d) is the generated initial content of the view.

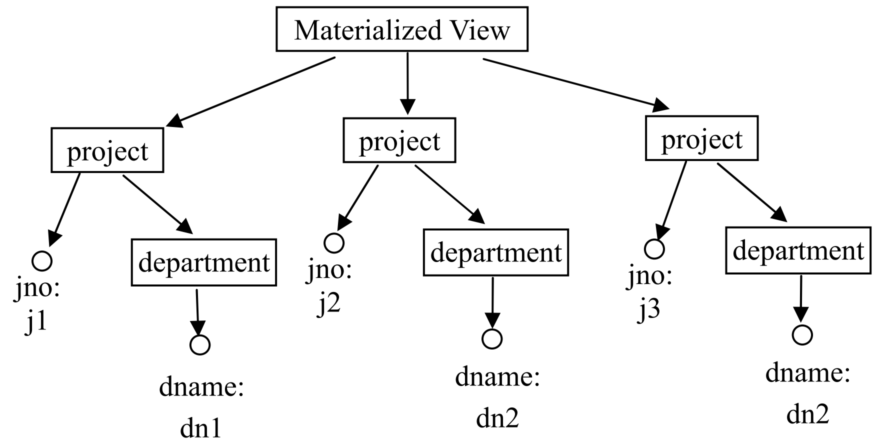


Fig 4.3 (a) Step 1: Projection of Relationship Type *project-department*

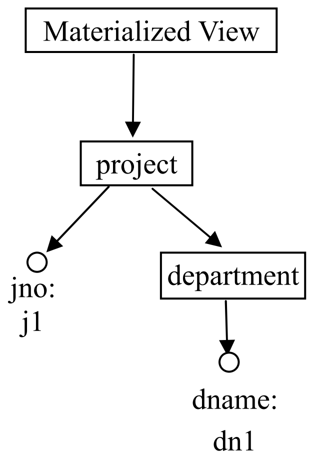


Fig 4.3 (b) Step 2: Selection on Attribute *dname* where dname = ‘dn1’

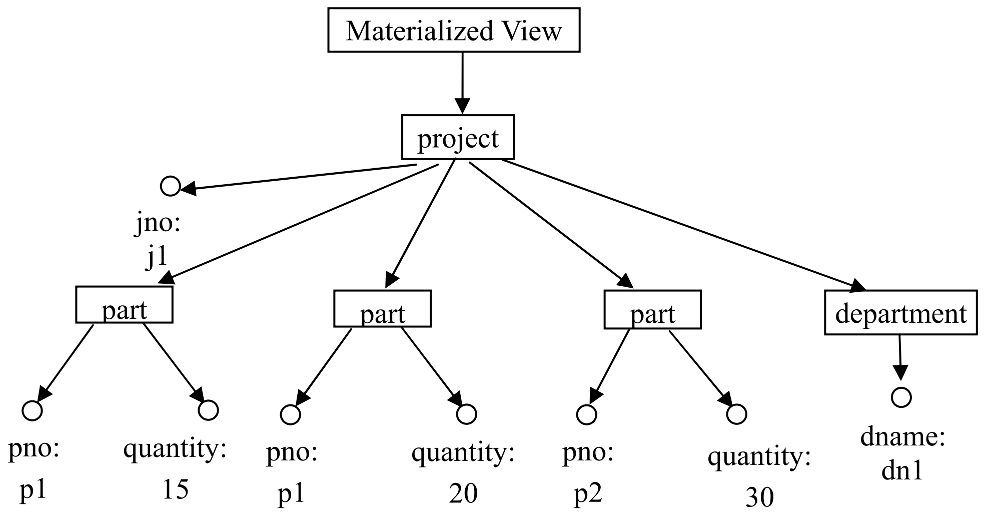


Fig 4.3 (c) Step 3: Join on *project* and Projection of Relationship Type *project-part*

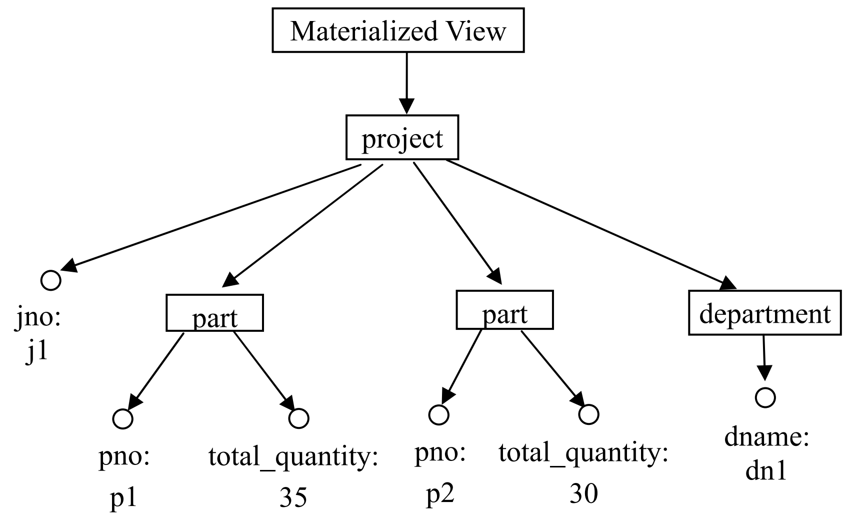


Fig 4.3 (d) Aggregation on attribute *quantity*

Figure 4.3 Generation of Initial Content of the Materialized View

In this chapter, we define the view specification using ORA-SS schema diagram, which can have very rich semantic meanings. We improve the algorithm of view transformation to perform accurate and efficient view materialization for the view defined by the ORA-SS schema diagram. In the next chapter, we will discuss our technique to incrementally maintain the materialized view upon each update to the source XML documents.

Chapter 5

Incremental XML View Maintenance

In this chapter, we discuss how the incremental maintenance for the materialized XML view is carried out. The environment that we are dealing with is that there are multiple source XML documents in one location. Multiple views defined upon the source XML documents. Views can be either in the same location as source or in a different location. If the views and source XML documents are in the different locations, we assume no data are lost in the query and query results transmission.

XML views are more complex than relational views because of their hierarchical structures. For example, elements in source XML documents may be swapped or joined in the views. Thus, it is more difficult to incrementally maintain the materialized XML views than relational views. Our technique composes of three main steps. Upon an update to a source XML document, we treat the update to the source as a list of source update trees. First, we will check whether the update is relevant, if it is an irrelevant update and will not affect the view content, we will stop here. Second, from the source update trees and other un-updated source XML documents, we compute update view trees, which contain only update part of the view. Third, we merge the update view trees with the existing materialized view tree to produce the complete updated view. In the following, we give the formal definitions of source update tree and view update tree.

Definition of Source Update Tree *A path in a source XML document is said to be in a* ***source update tree*** *iff it is from the root of the updated source XML document to the updated sub-tree in source, or to the object class or the relationship type with the modified attribute. A source update tree contains the update information and conforms to the source ORA-SS schema.*

Definition of View Update Tree *A path in an XML view is said to be in a* ***view update tree*** *iff it is from the root of view to the updated sub-tree in the view, or to the object class or the relationship type with the modified attribute. A view update tree contains the update information and conforms to the view ORA-SS schema.*

The task of our incremental maintenance is to find the update part of the view (view update tree) according to the update of the source (source update tree), and maintain the view properly. We will first give a few examples on both source update tree and view update tree, followed by the detailed algorithm of incremental view maintenance in next section.

Example 5.1 [**Insertion**] Using the XML *Project-Supplier-Part* database in Figure 1.1 and the view in Figure 4.1, suppose *supplier* s3 is going to supply *part* p1 to *project* j1 with a *quantity* of 10. This will insert *part* p1 as the child element of *supplier* s3 in the source XML document 1. *part* p1 has a child element *project* j1 with a *quantity* of value 10. The source update tree in this case is shown in Figure 5.1, which contains the path from *supplier* s3 to *project* j1. This source update will impact the view. The *total_quantity* of *part* p1 of *project* j1 will be increased by 10. The updated view is shown in Figure 5.2 with the updated *part* in the dashed circle.

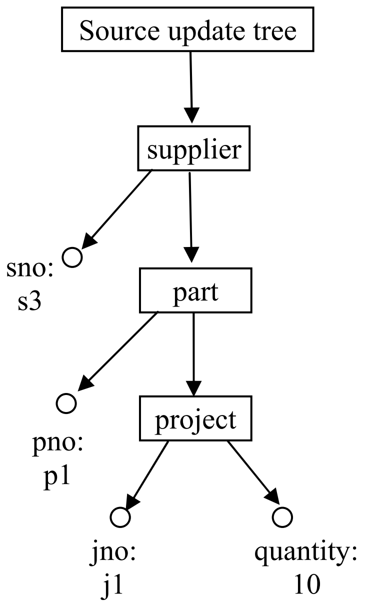


Figure 5.1: Source Update Tree in Example 5.1

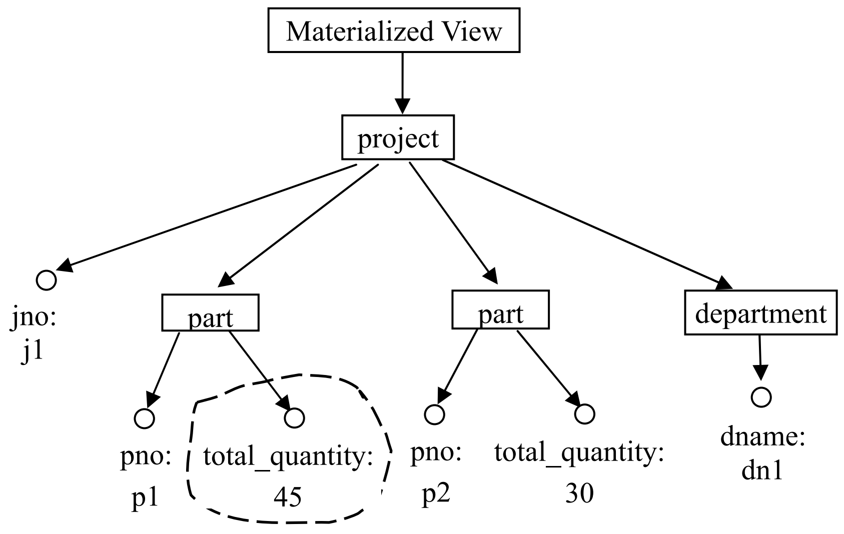


Figure 5.2 Updated Materialized View in Example 5.1

Example 5.2 [**Deletion**] Using the XML *Project-Supplier-Part* database in Figure 1.1 and the view in Figure 4.1, suppose *supplier* s2 will not supply *part* p1 to *project* j1 any longer. This will delete *project* j1 from *part* p1 of *supplier* s2. The source update tree in this case is shown in Figure 5.3, which contains the path from *supplier* s2 to *project* j1. This source update will impact the view. The *total_quantity* of part p1 of *project* j1 will be decreased by 20. The updated view is shown in Figure 5.4 with the updated part in the dashed circle.

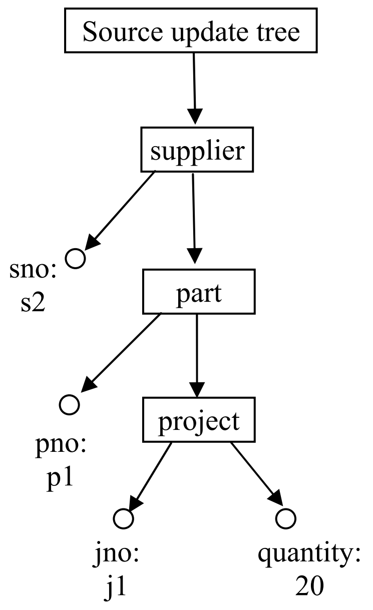


Figure 5.3: Source Update Tree in Example 5.2

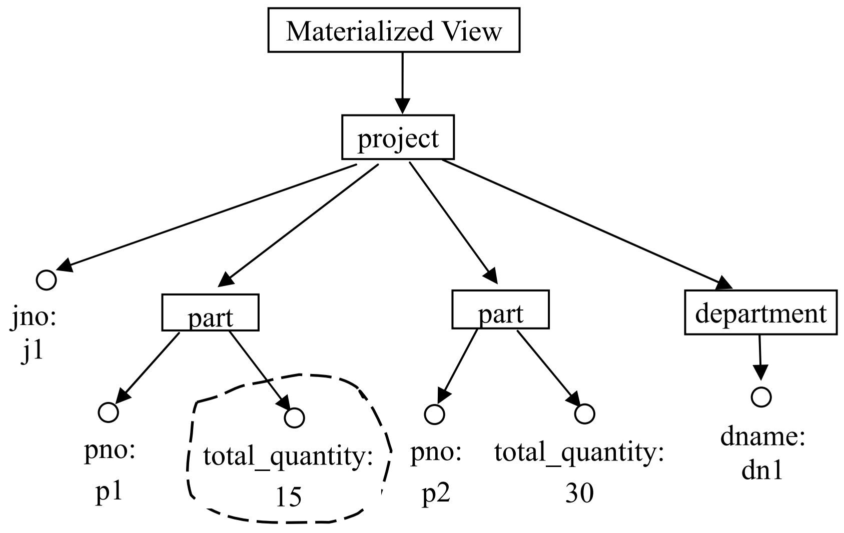


Figure 5.4 Updated Materialized View in Example 5.2

Example 5.3 [**Modification**] Using the XML *Project-Supplier-Part* database in Figure 1.1 and the view in Figure 4.1, suppose *supplier* s2 will supply *part* p1 to *project* j1 with quantity 30 instead of 20. This modification happens on the *quantity* attribute of the relationship type *pj*. 30 is the new value of the attribute, and 20 is the old value.

The source update tree in this case is shown in Figure 5.5, which contains the path from *supplier* s2 to *project* j1. The new *quantity* value is shown. The old value is also recorded in the source update tree, because later we will use it to update the aggregate attribute if there is any. This source update will impact the view. The *total_quantity* of *part* p1 of *project* j1 will be increased by 10. The updated view is shown in Figure 5.6 with the updated part in the dashed circle.

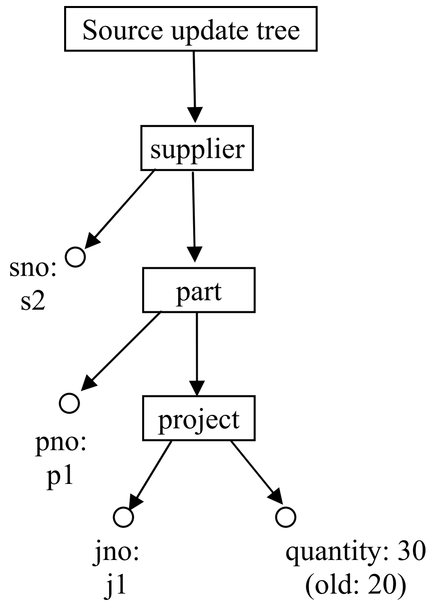


Figure 5.5: Source Update Tree in Example 5.3

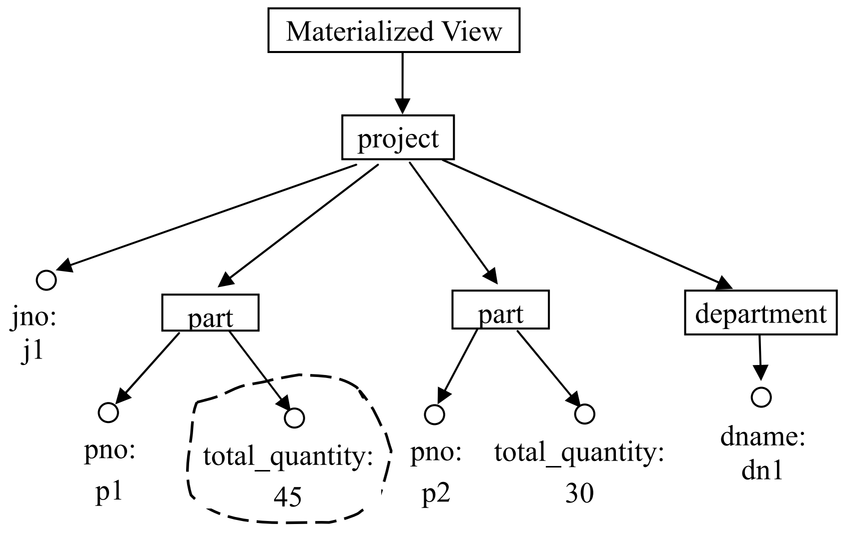


Figure 5.6 Updated Materialized View in Example 5.3

5.1 The View_Maintenance Algorithm

The View_Maintenance algorithm receives as input an update on one source XML document, the source XML documents, the ORA-SS schemas of the source, the existing materialized view, and the existing materialized view.

In outline, the main steps of the View_Maintenance algorithm are:

1. *Obtain the source update tree according to the update specification and the source document and source schema.*
2. *Check the relevance of the source update to see whether the update will affect the view. If the source update is relevant, we proceed to step 3, otherwise we stop here.*

3. *Generate the view update tree, which contains the update information to the view.*
4. *Merge the view update tree into the view to produce the completed updated materialized view.*

In the following sections we discuss the procedures used in Step 1 to 4.

5.2 The Procedure GenerateSourceUpdateTree

The procedure GenerateSourceUpdateTree receives as input an update *U* to a source XML document *D*, *D* itself, and the ORA-SS schema *S* of *D*. The procedure generates the source update tree, which contains the update information to *D* upon *U*. Table 5.1 shows all possible types of source update tree that GenerateSourceUpdateTree can produce.

Table 5.1 Cases of the GenerateSourceUpdateTree

Case	**SourceUpdateTree**
(1) insertion	We locate the points in source XML *D* where update *U* takes place. We form a source update tree by concatenating the paths of the update points and the sub-tree to be inserted into. Each path in the source update tree is not present in *D* before the insertion *U* takes place.
(2) deletion	We locate the points in *D* where the sub-tree is deleted from according to *U*. We form a source update tree by concatenating the paths of the update points and the sub-tree to be deleted. Each path in the source update tree is present in *D* before *U*, but it is no longer in *D* after the update.
(3) modification	We locate the object class or the relationship type in *D* where its attribute is modified as specified in *U*. We form a source update tree with the paths from the root of the document to the object class or the relationship type with the updated attribute. Both old value and the new value of the attribute are recorded in the tree.

Example 5.1 specifies a source update which is an insertion on the source XML document 1. According to case (1) in Table 5.1, we locate the update point first. We use object class and its key attribute value to textually represent the path. Here the path

of the update point is supplier [sno = "s3"]. The sub-tree to be inserted is made up of *part* p1 and its child element - the *project* j1 and a *quantity* attribute with the value of 10. We get the source update tree by concatenating the path and the sub-tree for this insertion, which is shown in Figure 5.1. The tree conforms to the ORA-SS schema diagram of XML document 1 in Figure 2.4 (a) as well.

5.3 The Procedure CheckSourceUpdateRelevance

The procedure CheckSourceUpdateRelevance receives as input the source update tree, the source ORA-SS schemas and the view ORA-SS schema. It checks whether the source update will have impact on the existing materialized view. We call the update which will affect the view as the relevant update. Only the relevant source update will be processed further.

5.3.1 Insertion/Deletion

Upon an insertion or deletion update, we use the following lemmas and theorem to do the source update tree relevance checking.

Lemma 5.1 *When we insert/delete a* ***subtree*** *into/from a source XML document, if all of the object classes and the relationship types in the source update tree schema* ***are not in*** *the view schema, we say the insertion/deletion is irrelevant.* (This lemma checks the schema only)

Proof We will prove Lemma 5.1 by controversy. Given all the object classes and the relationship types in the sub-tree of one insertion update are not in the view schema, suppose the update is relevant to the view and will cause the view content to be updated. Suppose object class O_1 is to be updated by having a new instance in the view due to the insertion update. There are two cases, one is O_1 is in source update tree schema, in this case, we show at least one object class in the source update tree schema is in the view schema. The other case is the newly inserted instance of O_1 is because O_1 can join with some instance of object class O_2 in the view and the instance of O_2 is in the source update tree, such that O_2 is in the source update tree schema as well. In the second case, we also show at least one object class in the source update tree schema is in the view schema. So we conclude the update tree cannot be relevant.

Lemma 5.1 checks the schema of source and view only. For example, using the source schema in Figure 5.7 and the view schema in Fig. 5.8, the deletion of any instances of the object class *department* will not affect the view as *department* is not involved in the view schema. So we conclude the insertion/deletion update on object class *department* is irrelevant.

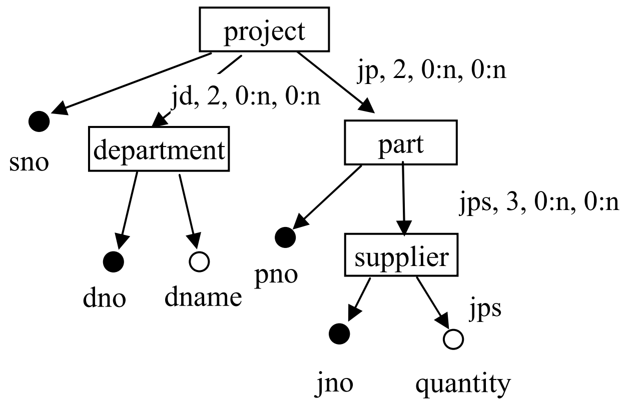


Figure 5.7 Source ORA-SS Schema Diagram for Demonstrating Lemma 5.1

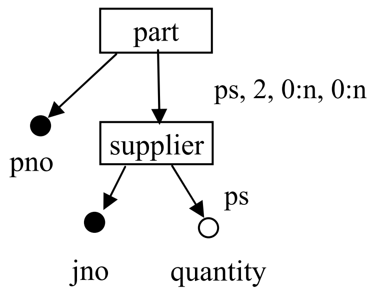


Figure 5.8 View ORA-SS Schema Diagram for Demonstrating Lemma 5.1

Lemma 5.2 When we insert/delete a subtree into/from a source XML document, for one path in the source update tree, if it does not satisfy the selection conditions of the view schema, we say the path is irrelevant.

Proof We will prove Lemma 5.2 by controversy. Given none of the path in the source update tree satisfies the selection conditions in the view schema, suppose the update is relevant to the view and will cause the view content to be updated. Suppose object class O_1 is to be updated by having a new instance in the view due to the insertion update. The attributes of O_1 must satisfy the selection conditions of the view, so that O_1

can appear in the view. If O1 is in the source update tree schema, we show that the path in the source update tree containing the instance of O_1 must satisfy the selection conditions of the view. If the newly inserted instance of O_1 is because O_1 can join with some instance of object class O_2 in the view and the instance of O_2 is in the source update tree, we say the path in the source update tree containing the instance of O_2 must satisfy the selection conditions of the view. In both two cases, we conclude at least one path in the source update tree satisfies the selection conditions of the view schema, which violates the hypothesis that none of the paths in the source update tree satisfy the selection conditions in the view schema. So we conclude the update tree cannot be relevant.

Lemma 5.2 checks the schema of the source update tree and the view schema. If the selection conditions are applied to the object class or the relationship type in the source update tree, we will check whether the source update paths satisfy the selection conditions.

Example 5.4 Using the XML *Project-Supplier-Part* database in Figure 1.1 and the view in Figure 4.1, suppose we want to add a new *project* j4 into source XML document 2 and *project* j4 belongs to *department* dn4. The source update tree is shown in Figure 5.9. For this insertion update, we will use Theorem 5.1 to check the update relevance. Firstly, Lemma 5.1 passed as object class *project* is involved in the view schema. For Lemma 5.2, we know in the view, one selection condition is that only the projects belonging to dn1 will be selected. So the selection condition is applied to the path in the source update tree, and the path does not satisfy the condition because the *project* j4 belongs to dn4 instead of dn1. So we conclude the source update tree in Figure 5.9 is irrelevant and it will not affect the view.

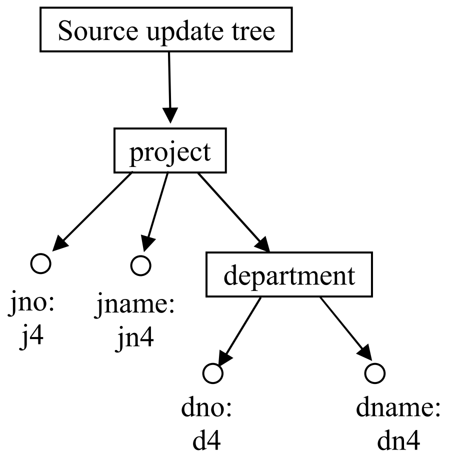


Figure 5.9: Source Update Tree in Example 5.4

Lemma 5.3 When we insert/delete a subtree into/from a source XML document, if any path in the subtree does not join with any other object classes for all the join conditions in the view schema (it has at least one join condition), we say the path is irrelevant. (It checks the view schema, the source update tree, and the source XML document)

Proof We will prove Lemma 5.3 by controversy. Given one path in the source update tree does not join with any other object classes for all the join conditions in the view schema and the path is relevant to the materialized view. Suppose object class O_1 is to be updated by having a new instance in the view due to the insertion update. The instance of O_1 can be in the path of source update tree or it can join with one object in the path of the source update tree. In both cases, it shows the path in the source update tree does join with certain object for one join condition to make O_1 be inserted into the materialized view. So we conclude the path cannot be relevant.

Lemma 5.3 checks the view schema, the source update tree and the source XML documents. Lemma 5.3 is most expensive one for evaluation and checking. Lemma 5.1 is the cheapest. The following example shows how we use the three Lemmas to determine irrelevant update tree.

Example 5.5 Using the XML *Project-Supplier-Part* database in Figure 1.1 and the view in Figure 4.1, suppose a new *supplier* s4 is going to supply *part* p1 to *project* j2 with a *quantity* of 20, and to supply *part* p2 to *project* j1 with a *quantity* of 30 as well. This update is formed into a source update as shown in Figure 5.10. Irrelevant update is not found according to Lemma 5.1 because object class project and part are present in the view schema. Again irrelevant update is not found according to Lemma 5.2 because no selection conditions in the view schema are applied to the object classes in the source update tree directly. Now let's consider Lemma 5.3, and according to the selection condition in the view, only the projects of *department* dn1 are selected. So for the projects selected from the source XML document 1 will only have *project* j1 as only j1 belongs to dn1. Here the *project* in one path of the source update tree is j2, which will not join with the *project* j1 from the source XML document 2. So that path of the source update tree with *project* j2 does not satisfy Lemma 5.3 and that path is considered as irrelevant. The irrelevant update path is removed and the relevant source update tree is shown in Figure 5.11.

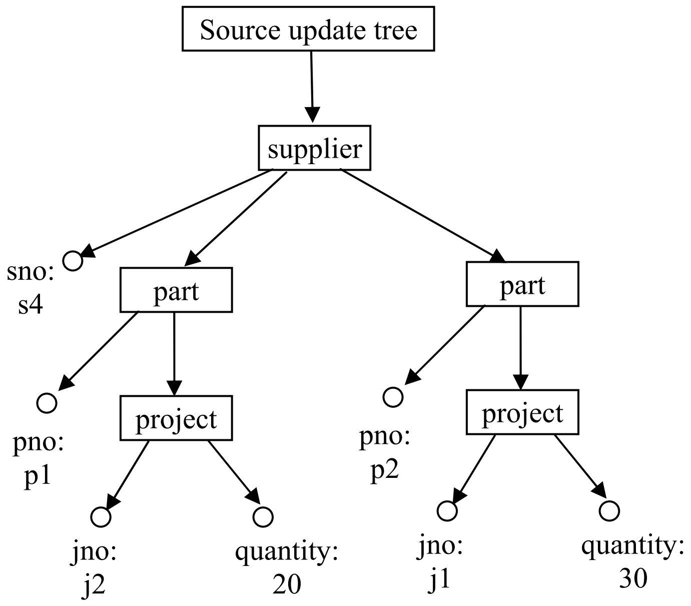


Figure 5.10: Source Update Tree in Example 5.5

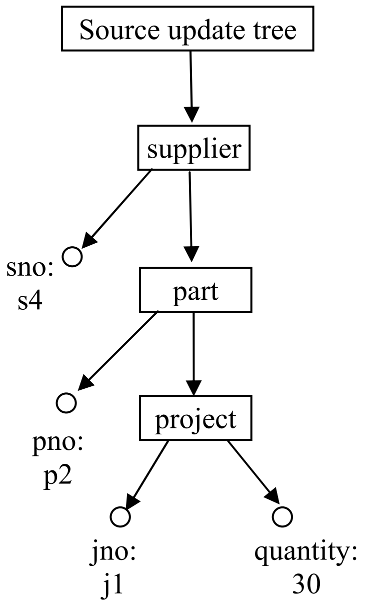


Figure 5.11: Relevant Source Update Tree in Example 5.5

Theorem 5.1 *For each source update tree, we check every path in the source update tree using Lemma 5.1, 5.2, and 5.3. If all update paths in the tree are irrelevant, we say the update tree is irrelevant, so is the source update. If some update paths in the tree are irrelevant, we say the update tree is partial relevant. The irrelevant update paths will be removed from the source update tree. So only the relevant update path will remain in the source update tree.*

Intuitively, if any of the lemmas are not satisfied, the update to source will not contribute to the view content update.

5.3.2 Modification

Upon a modification update, we use the following Checking Lemma 5.4 and 5.5 to do the relevance checking for source update tree.

Lemma 5.4 *When we modify an attribute of a source XML document, if the modification happens on the attribute, which is not involved in the view schema and it is not used as the join attribute, we say the modification is irrelevant.*

Proof We will prove Lemma 5.4 by controversy. Given the modified attribute is not in the view schema and it is not used as the join attribute, suppose the update is relevant to the view and will cause the view content to be updated. Suppose attribute A_1 is to be updated by having a new value in the view due to the modification update. There are two cases, one is A_1 is the modified attribute in source update tree schema itself, in this case, we show the modified attribute in the source update tree schema is in the view schema. The other case is A_1 is the aggregation attribute on certain attribute *A'*, which is the modified attribute in the source update tree schema. Such that *A'* is both in the source update tree schema and in the view schema. So we conclude the update tree cannot be relevant.

The following example shows how to use Lemma 5.4 to check the modification update relevance.

Example 5.6 Using the XML *Project-Supplier-Part* database in Figure 1.1 and the view in Figure 4.1, suppose we want to change the name of *project* j1 from jn1 to new_jn1 for every instance of *project* j1 in both source XML document 1 and 2. The value of the attribute *jname* of object class *project* will be affected. Since it is a modification update, we consider the Theorem 5.2 for source update relevance checking. The modified attribute *jname* is not present in the view schema, and it is not used in the join attribute as well. So we conclude the modification on *jname* is irrelevant and will not affect the view.

Lemma 5.5 *When we modify an attribute of a source XML document, when Lemma5.4* *has conclude it is a relevant update, we check whether both the new value and old value of the affected attribute satisfy the selection condition, if both do not satisfy, we say the update is irrelevant.*

Proof We will prove Lemma 5.5 by controversy. Given both the new value and the old value of the affected attribute do not satisfy the selection conditions in the view schema, and also the update is relevant to the view. Suppose attribute A_1 is the updated attribute, and Object O_1 with A_1 was inserted into the view due to the modification update on A_1. That shows the new value of A_1 satisfies the selection condition of the view. Suppose the Object O_1 with A_1 was deleted from the view due to the modification update on A_1. That shows the old value of A_1 satisfies the selection condition of the view. In both cases, it shows either the old value or the new value satisfies the selection conditions of the view, which violates the hypothesis. So we conclude the update tree cannot be relevant.

The following example shows how to use Lemma 5.5 to check the modification update relevance.

Example 5.7 Suppose in the source database, there are part with colors like yellow, blue, red, etc. Now only the red parts are selected in the view. Let’s change yellow to green for one yellow part. Since the modified attribute COLOR is involved in the view schema, the update will be treated as a relevant update by Rule 5.1. Since the old and new values of COLOR in the update do not satisfy the view selection condition, the update is treated as an irrelevant update by Rule 5.2.

All the irrelevant update paths are detected by the above lemmas. The relevant source update tree will be processed by the algorithm in next section to generate view update tree to incrementally maintain the materialized view.

5.4 The Procedure GenerateViewUpdateTree

The procedure GenerateViewUpdateTree receives as input the relevant source update tree *SUT* in source XML document D1, the un-updated source XML documents (D_2, …, D_n) and the ORA-SS schema of the view (*SV*). It generates the view update tree which contains the update information to the existing materialized view and conforms to the view schema S_v as well. It uses the view transformation technique in paper [8]. It uses the source update tree instead of the source document D1, so that the output is the update to the view instead of the initial content of the view.

As discussed in Chapter 4.2, we will follow the four steps to produce the view update tree. In the first step, we select instances of object classes and relationship types from the source update tree and the other un-updated source XML documents. The reason why we are using source update tree instead of the updated source XML document is because only the source update tree contains the update information, which is going to change the materialized view. In the second step, instances retrieved from previous step are pruned by considering the selection conditions in the view schema. In the third step, the instances from source update tree and other un-updated source XML documents, which have the same name and key attributes, are joined together. The combined instance of the object class or relationship type has all the attributes together. In the fourth step, the aggregation functions in the view schema are applied to the aggregate attributes if they are in the retrieved instances.

We use the following example to do a demonstration.

Example 5.8 Upon receiving the source update tree in Figure 5.1, we use the procedure GenerateViewUpdateTree to generate the view update tree. Firstly, we take the updated object class supplier, which has one instance j1 in the source update tree. From the view schema in Figure 4.1, there are two relationship types *jp* and *jd* associated with supplier. *jp* is a binary relationship type, in which *supplier* is the parent object class, and *part* is the child object class. *Project* j1 and *department* d1 are retrieved from source XML document 2 as they are the only instance of relationship type *jd* which satisfies the selection condition in the view schema. This *project* j1 in source XML document 2 joins with the *project* j1 in source update tree in Figure 5.1. The *part* p1 with *quantity* value 10 is returned, since *part* p1 is in the same path as *project* j1 in the source update tree. By now, all the object classes and attributes in S_v are queried. We form the view update tree from the returned instances and it is shown in Figure 5.12.

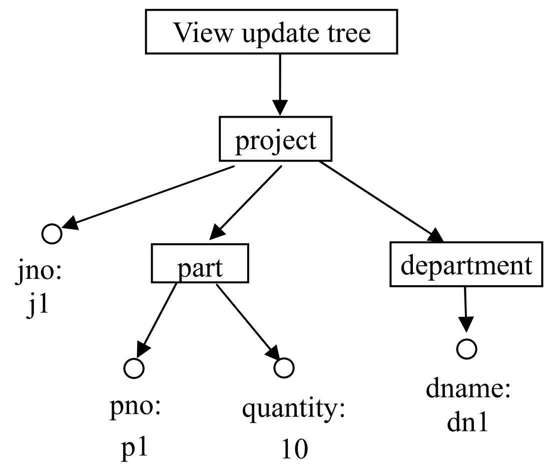


Figure 5.12 View Update Tree for Example 5.7

5.5 The Procedure MergeViewUpdateTree

The procedure MergeViewUpdateTree receives as input the view update tree, the view ORA-SS schema, and the existing materialized view MV. It merges the view update tree into the existing materialized view MV and produces the updated materialized view. Table 5.2 shows all possible situations that the view update tree could be merged into MV.

Table 5.2 Cases of the MergeViewUpdateTree

Case	Merge ViewUpdateTree Strategy
(1) insertion	For each path in the view update tree, we divide it into head_path and tail_path. The head_path is the longest sub-path from the head that already exists in the materialized view. The tail_path is the rest of the path. The tail_path will be concatenated to head_path in the materialized view. If the view has the aggregation function, it is handled as in Table 5.3.
(2) deletion	If the view does not have the aggregation function, the paths in view update tree will be deleted from the view. For each path of the view update tree, we separate it according to relationship types. We will delete the elements of each relationship type in the materialized view only if the parent object class has only one single child in the materialized view, otherwise we will drop the child object from the view only. If the view has the aggregation function, it is handled as in Table 5.3.
(3) modification	We locate each path of the view update tree in the view, and since each path is pointing to the attribute of an object class or a relationship type, we will modify the value of the attribute that the path specifies. If the view has the aggregation function, it is handled as in Table 5.3.

Table 5.3 shows how we update the value of aggregate attribute when the view update tree is merged into the materialized view. For each attribute which is associated with the aggregation function, we have both the new attribute value and the old attribute value recorded in the view update tree.

Table 5.3 Cases for Updating Aggregate Attribute for Different Types of Aggregation

Aggregation	Update	Method
(1) COUNT	Insertion	We increase the original value by 1
	Deletion	We decrease the original value by 1. If the new aggregation value drops down to 0, we will delete the object or the relationship which the aggregate attribute belongs to.
	Modification	Count will be only affected when the modified attribute is one of the conditions of the Count. We will increase the original value by 1 if the new value of the modified attribute meets the condition of the Count, otherwise, the original value will be decreased by 1.
(2) SUM	Insertion	We add the new value of the attribute to the original value
	Deletion	We deduct the value of the deleted attribute from the original value
	Modification	We update the original value by deducting the old value of the modified attribute and adding the new value of the modified attribute
(3) AVG	Insertion	We will first calculate the new value of COUNT and SUM, then we divide SUM by COUNT, and store the average value as the new value of the aggregate attribute
	Deletion	
	Modification	

(4) MAX	Insertion	We compare the original value with the new value of the attribute, and choose the bigger one.
	Deletion	We will always store the highest and the second highest value. If the current aggregate value is the same as the deleted value, the second highest is used as the new highest value. We will retrieve the second highest from source in the background.
	Modification	If the current aggregate value is modified to a larger value, we will choose the new value of the modification as the MAX value; otherwise we need to query the source to retrieve the second highest value as the new aggregate value
(5) MIN	Insertion	We compare the original value with the new value of the attribute, and choose the smaller one.
	Deletion	We will always store the smallest and the second smallest value. If the current aggregate value is the same as the deleted value, the second smallest is used as the new smallest value. We will retrieve the second smallest from source in the background.
	Modification	If the current aggregate value is modified to a smaller value, we will choose the new value of the modification as the MIN value; otherwise we need to query the source to retrieve the second smallest value as the new aggregate value

Example 5.9 Using the source update example in Example 5.1 where a source insertion happens. The view update tree is shown in Figure 5.12. In the view update tree, the two paths are Path1 = project [jno = "jn1"] / part [pno = "p1"] and Path2 = project [jno = "j1"] / department [dname = "dn1"]. The head_path of Path1 is project [jno = "j1"] / part [pno = "p1"], as it is the longest sub-path from the head of Path1, which also exists in the materialized view. The tail_path is NULL. There is an aggregate attribute *total_attribute* associated with this path. So we update the view by increasing the value of total_attribute by 10, which is the value of *quantity* in the view update tree. For the second path Path2, we will do nothing as it already exists in the view and no aggregate

attribute is associated. By now, we successfully updated the materialized view as shown in Figure 5.2. The aggregate attribute *total_quantity* in the dashed circle is modified in this case.

5.6 Strategy Analysis

In this section, we are going to analyze the complexity of the incremental maintenance strategy and compare with the view re-computation. Table 5.4 lists the four steps used in the incremental maintenance and their computation complexity analysis.

Table 5.4 Analysis for the Four Incremental Maintenance Steps

Steps	**Analysis**
GenerateSourceUpdateTree	In this step, only the objects in the affected source document are involved. Each path in the source update tree is generated by getting all the objects along the path, which is done by a linear scan, defined as *O(n)*, where *n* is the number of objects in the source XML document tree.
CheckSourceUpdateRelevance	In this step, the complexity varies according to the lemmas. Lemma 5.1 checks the schema of source and view only. Lemma 5.2 checks the schema of the source update tree and the view schema. Lemma 5.3 checks the view schema, the source update tree and the source XML documents. Lemma 5.3 is most expensive one for evaluation and checking. It does a full scan on all the objects in the source XML documents which are possibly joining with the affected object class according to the view schema. So the time complexity is *O(m)*, where *m* is the number of objects in all the source XML document trees.
GenerateViewUpdateTree	In this step, there are four sub-steps as explained early. First sub-step is the selection of objects from the source according to the view schema, the time complexity is *O(m)*, where *m* is the number of objects in all the source XML document trees. The second sub-step is pruning using the selection conditions in the view, still *O(m)*. The third step is the joining of selected object classes, still *O(m)*. The fourth sub-step is applying the aggregation function on the related attributes, the time complexity is still *O(m)* as a full scan on the related attributes will do. So combing all the four sub-steps, this step has the time complexity of *O(m)*.
MergeViewUpdateTree	In this step, we need to do a full scan on the materialized view to merge the view update tree by either concatenation or deletion on the materialized view. The time complexity is *O(m)*, where *m* is the number of object classes in the materialized view.

As discussed in Chapter 5.4, the step GenerateViewUpdateTree uses the view transformation technique in paper [8]. It uses the source update tree instead of the affected source document *D1*, so that the output is the update to the view instead of the whole updated content of the view. By using only the source update tree instead of the huge affected source XML document, we are saving a lot of time on processing the un-updated objects in the source XML document. So basically the incremental process will be much faster than the re-computation of materialized view if the update on source is relatively small compare to the source XML documents. If the source update is very huge like almost the whole source XML document, then the incremental approach has no advantage over the re-computation.

5.7 A Complete Example

To demonstrate the main strategy of incrementally maintaining the view for XML documents, we use the following complete example for deletion update. This demonstration will show the four steps to do the incremental view maintenance as sketched in Chapter 5.1.

Example 5.10 Using the XML *Project-Supplier-Part* database in Figure 1.1 and the view in Figure 4.1, suppose *supplier* s2 will not supply *part* p1 to *project* j1 and *project* j2 any longer. This will delete *project* j1 and *project* j2 from *part* p1 of *supplier* s2. Upon the source update, we do the followings, Step 1: Transform the update to source update tree, which are shown in Figure 5.13(a);

Step 2: Check the relevance of the source update tree, the irrelevant update paths are removed from the update tree and only the relevant source update paths are left. Project j2 in the source update tree will not join with any project from the source XML document 2, because only project j2 is not the project of department dn1. The path with project j2 is pruned from the source update tree and the relevant source update tree is shown in Figure 5.13(b);

Step 3: Generate the view update tree by using the method discussed in Chapter 5.4, the view update tree is shown in Figure 5.13(c);

Step 4: Delete each update path in the view update tree from the original materialized view. In this case, the view path has a SUM aggregate function on *quantity* attribute of the view update tree, according to the rule in Table 5.3, we deduct the value of the deleted attribute from the original value. The resulting view is shown in figure 5.13(d).

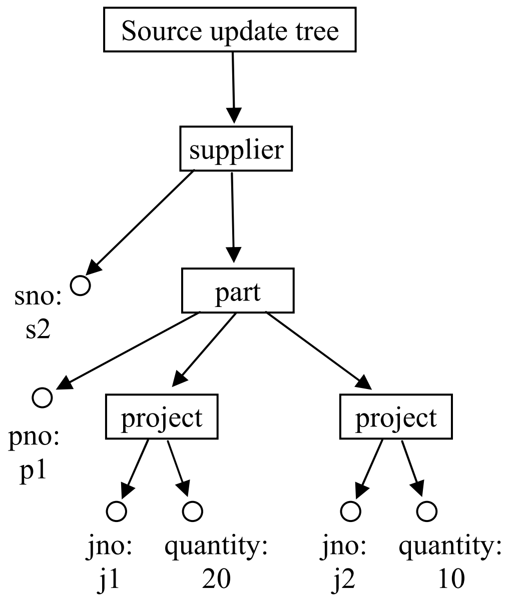


Figure 5.13 (a): Source Update Tree in Example 5.9

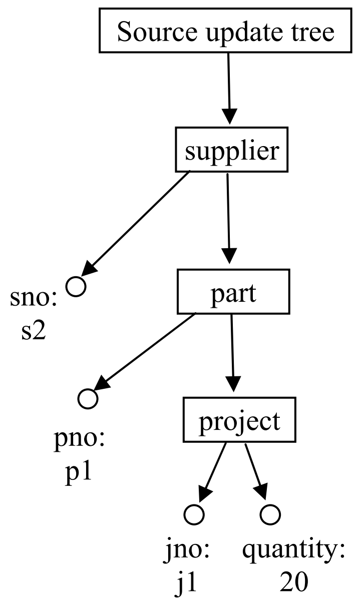


Figure 5.13 (b): Relevant Source Update Tree in Example 5.9

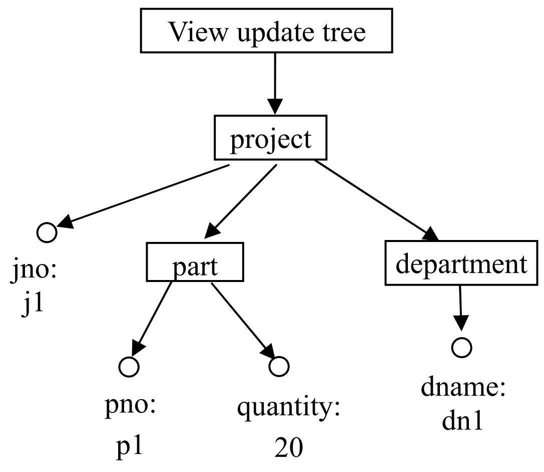


Figure 5.13 (c): View Update Tree in Example 5.9

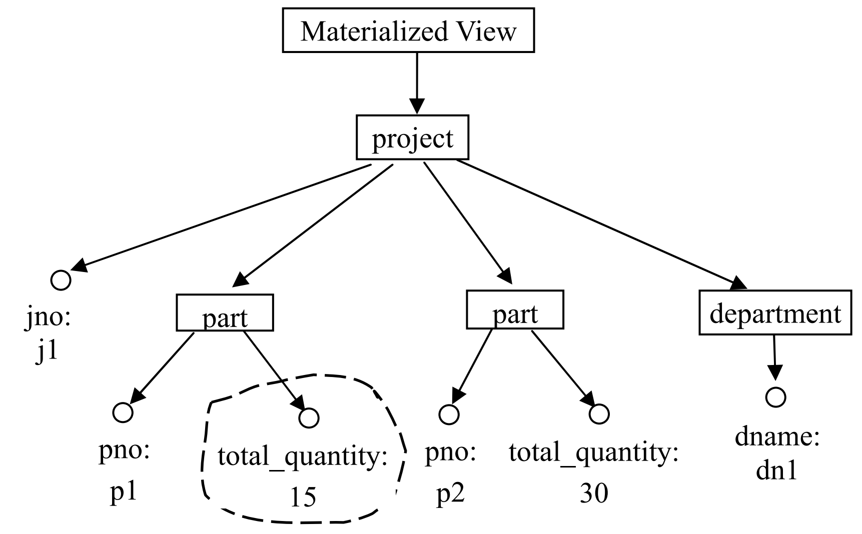


Figure 5.13 (d): Updated Materialized View in Example 5.9

We have shown the general strategy to maintain the materialized XML view so as to guarantee the consistency of the materialized views when the source XML document is updated. In next section, the general view maintenance process will be improved.

5.8 View Self-Maintenance for Deletion/Modification

In this section, we discuss one kind of situation when we can improve the general view maintenance algorithm, so that the view can be fast and efficiently maintained. This optimization is that we involve the XML view to improve the efficiency of the maintenance algorithm by cutting down the need to access the source XML documents. By querying the materialized XML view, we do not have to compute the full view update tree before we can update the materialized view. Information like object identifier constraint are used to achieve the view self maintenance.

In the cases of modification or deletion updates, we can make use of the materialized view to maintain itself by considering object identifier constraint. Lemma 5.4 states how we make use of object identifier constraint to do view self-maintenance without querying any source XML documents when the source updates are deletion or modification.

Lemma 5.4 *For a modification or deletion update to object class O, if the key of O is in the view, then maintenance can be carried out by modifying or deleting the corresponding node instances of O in the view through using the key values, without the need to compute the complete view instance.*

In example 5.11, we give an example on the view self-maintenance upon a modification update on a source XML document.

Example 5.11 We use the *Project-Supplier-Part* XML database in Figure 1.1 and the view schema is shown in Figure 5.14. The view schema is similar to the one in Figure 4.1, and the only difference is that the attribute *pname* of *part* is present in the view. The initial content of the materialized view is shown in Figure 5.15. Suppose we modify the *pname* of object *part* p2 from “pn2” to “newpn2”. Instead of generating the view update tree and merge it to the materialized view, we can use view self-maintenance. We know this is a modification on object class *part*. The key attribute of *part* is *pno*, which is in the view also. The conditions to do view self-maintenance are satisfied, so we can use Lemma 5.4 to do view self-maintenance.

Step 1: Find all instances of *part* in the materialized view with pno = ‘p2’;

Step 2: Modify the value of *pname* in each part element obtained in Step 1, and the resulting view is shown in Figure 5.16.

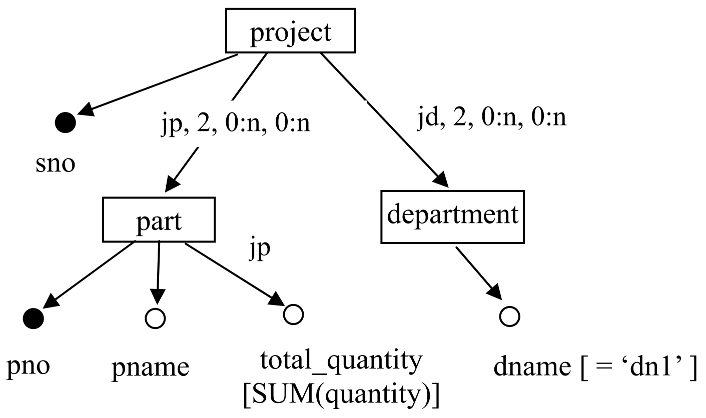


Figure 5.14: ORA-SS View Schema Diagram in Example 5.10

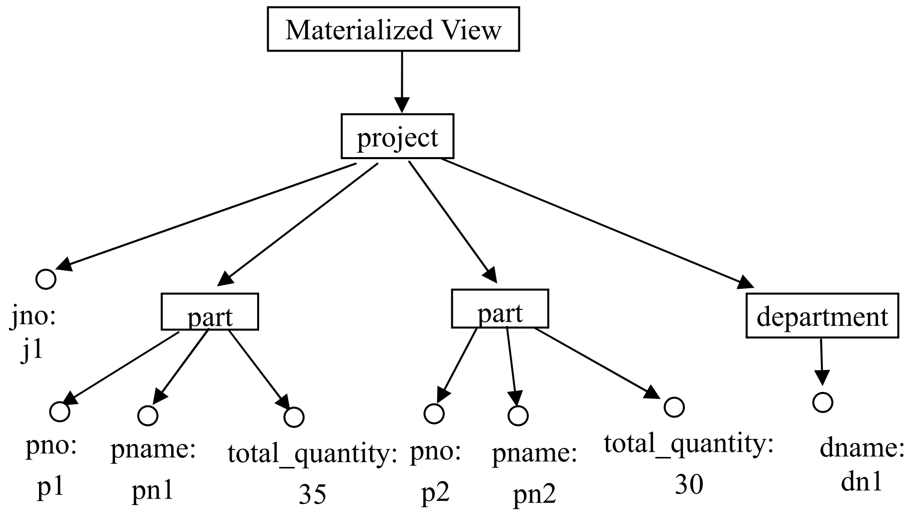


Figure 5.15: ORA-SS Instance Diagram of the View in Example 5.10

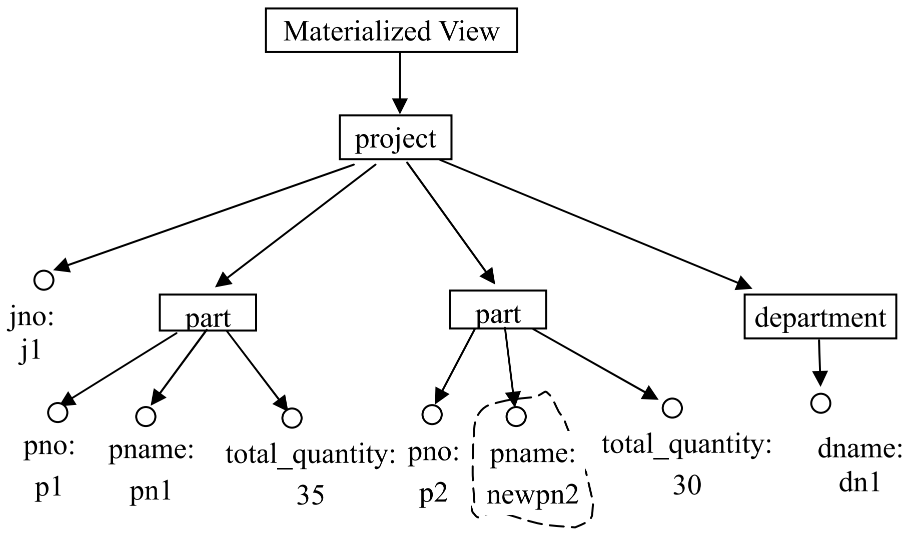


Figure 5.16: Updated ORA-SS Instance Diagram of the View in Example 5.10

In this chapter, we discuss how the materialized XML views are incrementally maintained upon an update in the source XML document. We are able to handle the XML views with complex hierarchical structures and rich semantics. Views with projection, selection, aggregation, join and swapping are able to be handled properly. Simple situation for view self-maintenance is also discussed as to show the direction on view self-maintenance for XML documents.

In the next chapter, we will survey on the previous works on the materialized view maintenance, and compare our work with them.

Chapter 6

Previous Works

In this chapter, we look at the current researches that have been done in the area of materialized view maintenance both for relational database and XML documents. We first look at in general some of the existing works in relational view maintenance, and then we examine in greater detail a few works that are closer to the work in this thesis.

6.1 Researches in View Maintenance

From 1990s, incremental view maintenance for relational database became popular. Many incremental materialized maintenance algorithms [3, 5, 6, 11, 15, 18, 21] have been developed to efficiently compute the incremental change rather than to re-compute the view from scratch in response to the updates at the data sources. A survey can be found in [12], which extensively study the problems and techniques for materialized view maintenance. Early work by Shmueli [18] and Blakeley [5, 6] focuses on the question of incremental view maintenance in Selection-Projection-Join views and the detection of irrelevant updates. [5] and [18] use counts to annotate tuples in the view with the number of derivations. Gupta et.al. [11] extended the counting method to views with aggregates and (stratified) negation.

After the complex relational views have been well handled, the issue of view consistency in centralized database systems has been studied recently. Problems of interfering updates and missing updates arise, and the main focus of the research becomes how to detect and remove the interfering updates and missing updates from the result of incremental computation. The paper [15], which incrementally maintains view using version number, is the best one on handling views over distributed source databases. [15] is designed for the environment of multiple, distributed data sources, with a separate database for housing the view relations. The view maintenance technique in [15] can handle an update transaction involving multiple source relations, and the processing of the updates for incremental computation is handled in parallel.

So far, the view maintenance techniques for relational databases have been well studied. However, the study of materialized view maintenance for XML documents is still limited. The paper [19] studies about the incremental view maintenance for semi-structured data. It uses an algebraic approach to maintain the views. That is, it finds expressions that can compute the changes to the view corresponding to the changes of source data. However, in [19], the view definition language is limited to select-project queries and only insertion update to the source document is considered. The paper [20] studies the graph structured views and their incremental maintenance.

However, it can only handle very simple views consisting of object collections, without edges. The article [2] studies the view maintenance for semi-structured data based on the Object Exchange Model (OEM) [17] and on the Lorel query language [1] for OEM. These three works will be discussed in more details in the next section, and we compare our work with them in the end.

6.2 Related Works

In this section, we examine in greater detail a few works that are closer to our research work in this thesis.

6.2.1 Abiteboul and McHugh Algorithm

The Abiteboul and McHugh algorithm [2] is defined for an environment with a single semi-structured data source, and a view in the same location. For each update to the source semi-structured data, the view maintenance algorithm is triggered to compute the changes to the view.

The view is specified using *select-from-where* query language, which intents to extract portion of the source semi-structured data. Whenever the view maintenance algorithm is triggered, it examines the view specification to search for the place where the update can be substituted. The new specification is generated according to the update. The new specification is executed upon the source semi-structured data to generate the changes to the materialized view.

[2] uses Object Exchange Model (OEM) [17] to represent both the source semi-structured data and the materialized view. The internal node ID is uniquely assigned to each object of the semi-structured data. The update query language is designed to use internal node ID also, which makes the language difficult to understand without looking at the OEM representation of semi-structured data.

Example 6.1 is taken from [2], which shows how the Abiteboul and McHugh algorithm incrementally update a materialized view upon an insertion update. *Ins* indicates it is an insertion update. &6 and &8 are two node IDs of the database in Figure 6.1. The insertion update is to add node &8 as child of &6 with the link named *Ingredient*. The view is specified in Lorel [1] as shown in Figure 6.2. The OEM representation of the materialized view is shown in Figure 6.3.

Example 6.1 Suppose the update <*Ins*, &6, *Ingredient*, &8> is performed on the database in Figure 6.1. The Baghdad Café restaurant now has two entrees with the ingredient “Mushroom”. The algorithm generates the statement to find the new entree &6, which is to be inserted into the materialized view. The updated materialized view is shown in Figure 6.5.

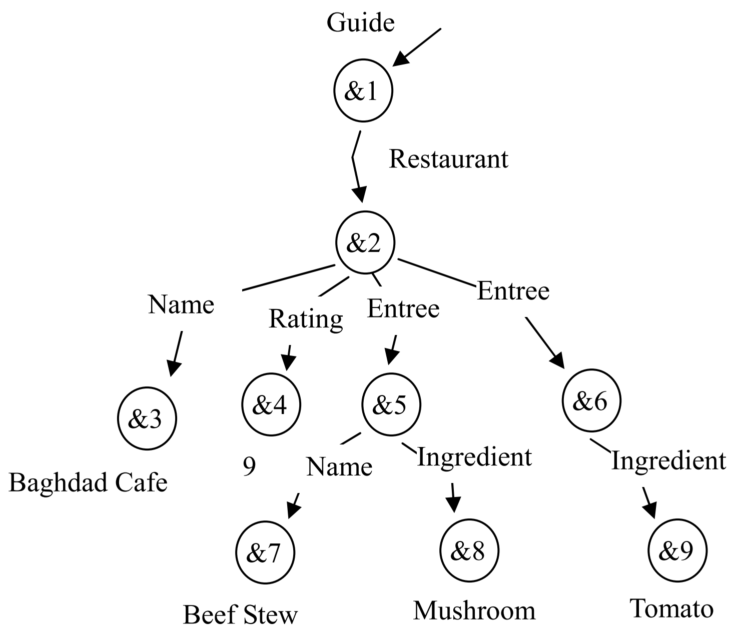


Figure 6.1 OEM Database

The following Figure 6.2 is the view specification in Lorel language [1]. It selects the specific Entrée of "Baghdad Café" restaurant and with Ingredient "Mushroom". In the language, two initials *n* and *i* in the *with* clause is defined but never used in other place of the view specification.

```
define view FavoriteEntrees as
  Entrees = select   e
            from     Guide.Restaurant r, r. Entrée e
            where    exists x in r.Name: x = "Baghdad Cafe"
            and      exists y in e.Ingredient: y = "Mushroom"
            with     e.Name n, e.Ingredient i;
```

Figure 6.2 View Specification on Lorel

The following Figure 6.3 is the materialized view content defined by the specification in Figure 6.2.

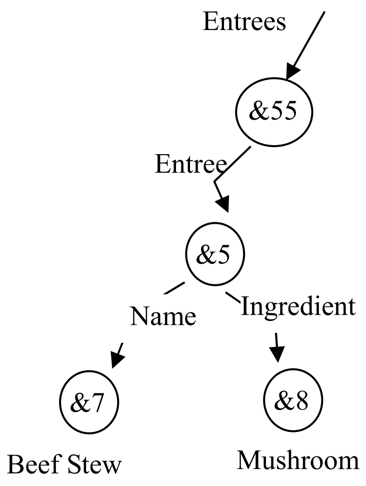


Figure 6.3 The Materialized View

As the update in Example 6.1 happens, the algorithm generates a statement to find the changes of the materialized view. The statement is shown in Figure 6.4, and the updated materialized view is shown in Figure 6.5.

ADD =	**select**	*e*
	from	*Guide.Restaurant r, r. Entrée e*
	where	**exists** *x* **in** *r.Name: x = "Baghdad Cafe"*
	and	**exists** *&8* **in** *&6.Ingredient: &8 = "Mushroom"*
	and	*e = &6;*

Figure 6.4 View Maintenance Statement

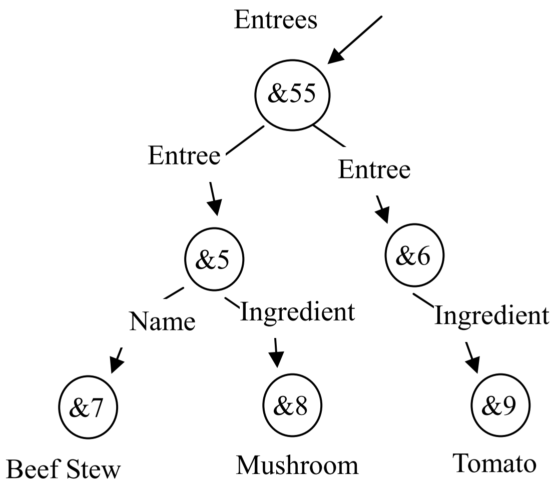


Figure 6.5 The Updated Materialized View

6.2.2 Zhuge and Garcia-Molina Algorithm

The Zhuge and Garcia-Molina algorithm [20] is defined for an environment with a single semi-structured data source, and a view in the same location. For each update to the source semi-structured data, the view maintenance algorithm is triggered to compute the changes to the view.

The paper provided a procedural algorithm for maintaining a simple type of view. This simple type of view is to retrieve a set of specific objects with their children from the source semi-structured data. That means the only hierarchical structure in the view is a binary relationship, and the view only have the set of objects and their children which are originally in the source semi-structured data and satisfying the view specification.

The view maintenance algorithm is triggered once an update takes place in the source semi-structured data. Suppose the object X in the source is updated, the algorithm first locates the ancestor object Y. After the algorithm locates Y, it tests whether the original condition in the view specification that makes Y appear or not in the view has been changed because of the recent update on X. If so, Y is inserted or deleted from the view as appropriate.

Example 6.2 is taken from [20], which shows how the Zhuge and Garcia-Molina algorithm incrementally update a materialized view upon an insertion update. *P2* is one node ID of the database in Figure 6.6. *A2* is one node ID to be inserted into the database, which represent one age element <A2, age, integer, 40>. The insertion update is to add one node *A2* as child of *P2*. The view is specified in Figure 6.7. The text representation of the materialized view is shown in Figure 6.6.

Example 6.2 Suppose the update *insert<P2, A2>* is performed on the database in Figure 6.6. The professor *P2* now has a new child element *A2*. The algorithm finds that *P2* now satisfies the view specification and could be inserted into the materialized view. The updated materialized view is shown in Figure 9.

```
<ROOT, person, set, {P1, P2}>
    <P1, professor, set, {N1, A1, S1}>
        <N1, name, string, 'John'>
        <A1, age, integer, 45>
        <S1, salary, dollar, $100,000>
    <P2, professor, set, {N2, ADD2}>
        <N2, name, string, 'Sally'>
        <ADD2, address, string, 'Palo Alto'>
```

Figure 6.6 Source Semi-Structured Data

The following Figure 6.7 is the view specification. It selects the professor element with age less than 50. The initial content of the materialized view is shown in Figure 6.8. Only professor *P1* exists in the view, since his age is 45. *P2* will not be in the view since there is no child element *age* in the source.

```
Define mview YP as:     SELECT ROOT.professor X
                        WHERE X.age < 50
```

Figure 6.7 View Specification

```
<P1, professor, set, {N1, A1, S1}>
```

Figure 6.8 The Materialized View

As the update in Example 6.2 happens, the algorithm generates the changes of the materialized view. As described in Example 6.2, the element *P2* will be added into the materialized view, and the updated materialized view is shown in Figure 6.9.

```
<P1, professor, set, {N1, A1, S1}>
<P2, professor, set, {N2, A2, ADD2}>
```

Figure 6.9 Updated Materialized View

6.2.3 Suciu Algorithm

The Suciu algorithm [19] is defined for an environment with a single semi-structured data source, and a view in a different location. For each update to the source semi-structured data, the view maintenance algorithm is triggered to compute the changes to the view. The algorithm assumes that the data transmitted in the network is not lost and misordered.

The paper uses an algebraic approach to maintain the XML views. Only views with simple selection-project feature are considered. This simple type of view is to retrieve a portion of the source semi-structured data with specific conditions in the view definition.

The database is modeled as a **rooted graph** (i.e. a graph with a distinguished node called the root), whose edges are labeled with elements with the type of strings, numbers, Booleans, etc. **Trees** form a particularly interesting subset of the rooted graphs, and they suffice to represent sets and records. In addition to the edge labels, some of the leaves of a graph are allowed to be labeled with special symbols, denoted *X*, *Y*, …, called **markers**. Unlike labels, markers are not part of the information content of the database, but are used to control (1) where updates take place, and (2) how to connect fragments of a distributed database. Markers allow us to define the concatenation operation $++_X$: given two graphs t_1, t_2 and a marker X, $t_1 ++_X t_2$ denotes the database obtained by connecting all leaves labeled *X* in t_1 to the root of t_2. All occurrences of the old marker *X* in t_1 disappear in $t_1 ++_X t_2$, as well as all markers from t_2. Figure 6.10 demonstrates the data model with $t_1 ++_X t_2$.

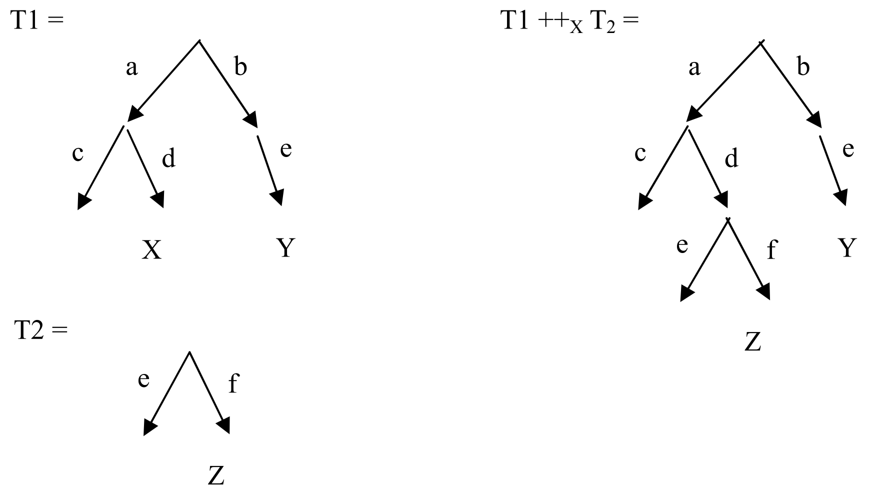


Figure 6.10 Marker Demonstration

The paper uses an algebraic approach to maintain the views. That is, it finds expressions that can compute delta views corresponding to the changes of base data. It requires a database *DB* to have all its updatable nodes explicitly marked. When the view *V* = *Q*(*DB*) is first computed, the result *V* encapsulates some (or all) markers of the updatable pages in *DB*. Suppose now that the database *DB* is updated, say at a page marked X, in that a link to a new subgraph Δ is added to that page: in notation $DB' := DB ++_X \Delta$. The server notifies the client about the update, by sending X and Δ. The client "look up" the marker X in its view, and, if present, reads the tag of the region where it occurred (R_1, R_2, or R_3), then updates the view dynamically.

The algorithm only considers the insertion and replacement update of the source semi-structured data.

6.3 Comparison

Our view maintenance algorithm is designed for the complex views which are joined from different source XML documents, and have different hierarchical structures as any of the source XML documents.

We use a user friendly data model ORA-SS data model to define both view and source XML documents. The ORA-SS schema diagram not only specifies the complex views correctly, but also ensures the unique interpretation of view definition because of its rich semantic information. The existing works are only considering the views of selection and projection of nodes of source XML documents. The views handled in the existing works are containing the binary relationship only. By using ORA-SS, we can define ternary relationships, which are necessary to retrieve valuable information from the source. The ORA-SS schema diagram of XML documents help to validate the updates of XML documents also.

We validate the source update before it is sent to trigger the view maintenance algorithm. This is to ensure the source update is valid, and the source database is consistent after the update. The source update validation process is usually ignored in the existing works.

Based on the view schema defined in ORA-SS schema diagram, we are able to compute the changes of view in the form of view update tree upon each source XML update. The way to generate the view update tree is to find the relationship object instances which are related to the update. The generated view update tree conforms to the view ORA-SS schema.

The existing works only considered one source XML document. However we maintain the view over multiple source XML documents. We involve the materialized XML view to improve the efficiency of the maintenance algorithm by cutting down the need to access the source XML documents.

The modification is usually treated as a deletion followed by an insertion update. We treat modification update as one type of update if the update is not on the joining elements. This allows us to consider the optimizing issue of view self-maintenance for a single modification update.

Chapter 7

Conclusion

7.1 Contributions

In this thesis, we proposed an incremental view maintenance algorithm for XML documents in an environment of multiple source XML documents in one database, with a separate database for housing the XML views. It supports immediate refresh of the views when source XML documents are updated.

In summary, upon a valid source update on either single element/attribute or subtree, first, we generate the source update tree, then we check the relevance of the update, thirdly, we compute the view update tree, which contain only updated part of the view. Fourthly, we merge the view update tree into the existing materialized view tree to produce the completed updated view.

Compared with the other existing works, the advantages of our work are summarized as follows.

Most of the existing methods do not validate the source update queries. We handle the update validation as the invalid update query will make the XML document inconsistent. We defined a set of update operations, which have the XQuery syntax. We define more types of updates, such as insertion and deletion of sub-tree from the source XML document. The update consistency for each kind of update operation can be checked based on the ORA-SS data model. The essential constraints to validate an update query include participation constraint, object identifier constraint, and functional dependency constraint, which can be all expressed in ORA-SS data model.

Most of the existing methods place restrictions on the view definition, such as simple views without any swapping and joining of elements in source XML documents. We do not have such requirement. We define the view in ORA-SS schema diagram, which can involve selection, project, join and swapping elements on multiple source XML documents. The hierarchical structure in the view can be very much different from any source. We even allow aggregate functions in the view definition. Using ORA-SS schema diagram, we are able to define not only binary relationship types, but also n-ary relationship types, which makes the view more meaningful.

The most advantage of our work is the use of update tree, which greatly simplifies the task of the materialized view maintenance. We traverse the source update tree and the un-updated source XML documents and combine the elements according to the view schema to generate the view update tree. Exceeding the existing works, we are able to capture all the source update information in the source update tree for different types of updates. The update for view can be refreshed into the view by merging the view update tree and the materialized view tree.

Beyond the correct generation of view update tree, we also provide view self-maintenance when the update query meets the specific conditions. By querying the materialized XML view, we do not have to compute the full view update tree before we can update the materialized view. Information like object identifier constraint is used to achieve the view self-maintenance.

7.2 Future Works

The following challenges are worth looking into:

1. We would like to trigger the view maintenance algorithm based on each update transaction, which can involve multiple updates from different source XML documents. To handle transaction, we will enable multiple changes to be specified in one single update tree. All the updates with counter effects need to be removed. Thus, the view update tree can be derived together at one time. The performance of view maintenance will certainly be improved compared to the current view maintenance triggered by each single source update.

2. We would like to develop the system which can handle order-preserving update and view maintenance. To broaden the search scope, we need an efficient

order-preserving labeling schema for XML documents. Our XML update language can be easily extended to have order information by changing the AT LAST default keyword to the specific position. Furthermore, our view maintenance algorithm needs to be enhanced by storing order information in the source update tree. When the view update tree is generated, it will have the order information as well in order to update the materialized view with order preservation.

References

[1] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The Lorel Query Language for Semistructured Data. *Journal of Digital Libraries*, 1(1), Nov. 1996.

[2] S. Abiteboul, J. McHugh, M. Rys, V. Vassalos, and J. Wiener. Incremental Maintenance for Materialized Views over Semistructured Data. In *VLDB*, pages 38-49, 1998.

[3] D. Agrawal, A. Abbadi, and T. Yurek. Efficient View Maintenance at Data Warehouses. *In proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 417-427, 1997.

[4] Shurug Al-Khalifa, H. V. Jagadish, Nick Kouda, Jignesh M. Patel, Divesh Srivastava, Yuqing Wu. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *Proceedings of ICDE*, 2002

[5] J. A. Blakeley, P. Larson, and F. W. Tompa. Efficiently Updating Materialized Views. In C. Zaniolo, editor, *ACM SIGMOD Proceedings*, page 61-71, Washington, D.C., May 1986.

[6] J.A. Blakeley, P.-A. Larson. Updating derived relations: Detecting irrelevant and autonomously computable updates. *ACM Transactions on Database Systems*, 14(3):369-400, September 1989.

[7] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A query language and optimization techniques for unstructured data. In *SIGMOD*, pages 505-516, Montreal, Quebec, Canada, June 1996.

[8] Daofeng Luo, Ting Chen, Tok Wang Ling, and Xiaofeng Meng. On View Transformation Support for a Native DBMS. *DASFAA 2004*, pages 226-231, Jeju Island, Korea, March 2004

[9] Yabing Chen, Tok Wang Ling and Mong Li Lee: Automatic Generation of XQuery View Definitions from ORA-SS views. In *22end International Conference on Conceptual Modeling (ER'2003)*, Chicago, Illinois, USA13-16 October 2003.

[10] G. Dobbie, Xiao Ying Wu, Tok Wang Ling and Mong Lee Lee. ORA-SS: An Object – Relationship - Attribute Model for Semistructured Data. Technical Report TR21/00, School of Computing, National University of Singapore, 2000.

[11] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining view incrementally. In *ACM SIGMOD Conference*, pages 157-166, Washington, DC, May 1993

[12] A. Gupta and I.S. Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Engineering Bulletin*, 18(2):3-18, June 1995.

[13] Bintou Kane, Hong Su, and Elke A. Rundensteiner. Consistently Updating XML Documents using Incremental Constraint Check Queries. In *WIDM'02*, McLean, Virginia, USA, Nov 8, 2002.

[14] Mong Li Lee, Tok Wang Ling, and W. L. Low. Designing Functional Dependencies for XML. In *EDBT*, pages, 124-141, 2002.

[15] Tok Wang Ling and Eng Koon Sze. Materialized View Maintenance Using Version Numbers. In *Proceedings of the Sixth International Conference on Database Systems for Advanced Applications*, pages 263-270, 1999.

[16] Xiaofeng Meng, Daofeng Luo, Mong Li Lee, Jing An. OrientStore: A Schema Based Native XML Storage System. In *Proceedings of the 29th VLDB Conference*, Berlin, Germany, 2003

[17] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object Exchange across Heterogeneous Information Sources. In *Proceedings of the 11th International Conference on Data Engineering*, pages 251-260, Taipei, Taiwan, Mar. 1995.

[18] O. Shmueli and A. Itai. Maintenance of views. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 240-255, Boston, June 1984.

[19] D. Suciu. Query Decomposition and View Maintenance for Query Language for Unstructured Data. In *VLDB*, pages 227-238, Bombay, India, September 1996.

[20] Y. Zhuge and H. Garcia-Molina. Graph Structured Views and Their Incremental Maintenance. In *Proceedings of the 14th International Conference on Data Engineering (DE)*, 1998.

[21] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View Maintenance in a Warehousing Environment. In *SIGMOD*, pages 316-327, San Jose, California, May 1995.

[22] World Wide Web Consortium, "XML Schema", W3C Recommendation, 2001. http://www.w3.org/XML/Schema

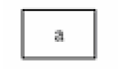
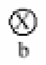
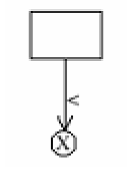
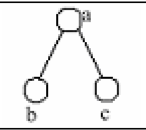
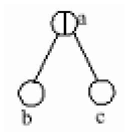
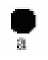
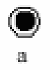
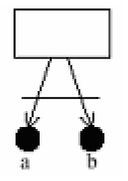
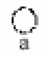
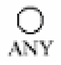
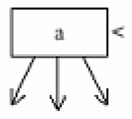
[23] World Wide Web Consortium, "XQuery: A Query Language for XML", W3C Working Draft, 2002. http://www.w3.org/XML/Query

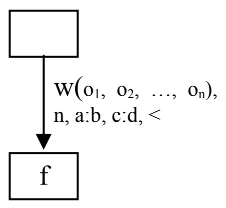
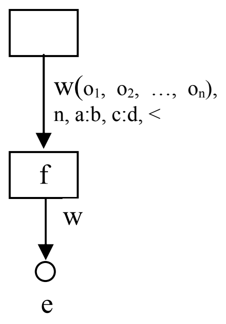
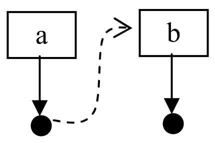
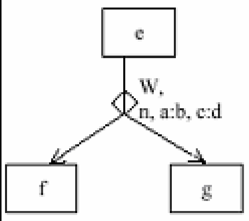
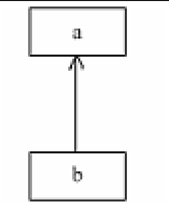
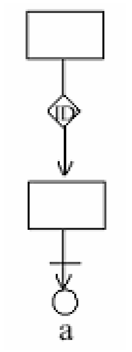
[24] World Wide Web Consortium, "XML Path Language", W3C Recommendation,

1999. http://www.w3c.org/TR/xpath

Appendix

The following table summarizes the notion of ORA-SS diagrams.

Notation	Description
	Object class with name **a**
	Attribute b, where x represent the cardinality, ? is 0 or 1, + is 1 or more, * is 0 or more, and the default for x is (0 or 1)
	Attribute **b** where the ordering of the value of the attributes is important. **x** is either + or *, and the default value is *.
	Composite attribute **a** with component attributes **b** and **c**.
	Disjunctive attributes **a** is either **b** or **c**.
	Identifier/Primary key **a**
	Candidate key **a**
	Composite candidate key
	Derived attribute
	Attribute with unknown structure or whose structure is heterogeneous
	The ordering on the attributes of object class **a** is important

	Relationship with name *W* (among object classes *o_1*, *o_2*, …, *o_n*), of degree *n*, where the participation of the parent has minimum *a* and maximum *b*, and the child has minimal *c* and maximum *d*, and the ordering of the object classes is important. The default degree is 2, default parent cardinality is 0:m, default child cardinality is 1:n, and default on ordering is no ordering.
	Attribute *e* belongs to relationship *W* (among object classes *o_1*, *o_2*, …, *o_n*). The default (without label *W* on the edge) shows that attribute *e* belongs to object class *f*.
	Reference object class *a* references object class *b*
	Disjunctive relationship: either object class *f* or object class *g*
	*b* inherits from *a* (inheritance diagram)
	Weak object class: attribute *a* is a weak identifier