Proceedings of International Conference on Distributed
Systems, Software Engineering and Database Systems

Nested Query Processing Techniques in Object-Oriented Databases

Jorng-Tzong Horng, Jiunn-Chin Wang, Ji-Tsong Lin, Baw-Jhiune Liu

Institute of Computer Science and Information Engineering
National Central University, Chung-Li, Taiwan, R.O.C.
horng@db.csie.ncu.edu.tw

Abstract

Nested queries can simplify queries on a database for users. However, nested queries also increase the complexity of query processing. In this paper, we propose query processing techniques for nested queries in object-oriented databases. Nested query expressions are first transformed into a query graph by using query graph transformation algorithm. We adapt Access Support Relations (ASRs) technique to represent and manipulate path expressions. The query graph is then converted to an execution plan which consists of a set of operators that we defined to manipulate ASRs in the query processing. The structure of Access Support Relations is the same as relations in RDB, thus many processing techniques in relational databases can be directly applied in our query processing.

Keywords: Object-Oriented Database, Nested Query, Query Graph, Path Expression, Access Support Relation, Execution Plan.

1 Introduction

In recent years, object-oriented programming has gained a tremendous popularity in the design and implementation of complex applications, such as engineering design and manufacturing (CAD/CAM) [12], image and graphics databases, scientific databases, geographic information systems, multimedia applications [15]. These applications require more complex structure for objects, new data types for storing images or large textual items. Object-oriented databases were proposed to meet the needs of these applications.

One of the basic functionalities of database management systems is to be able to process declarative user queries. The last decade, there are some significant research in defining object-oriented query models [10] including calculus [6, 13], algebra [8], and user language [1, 2, 3, 4]. Some queries require that existing values in the database be fetched and then used in comparison condition. Such queries usually are formulated by using nested queries. Nested queries can simplify queries on a database for users. However, nested queries also increase the complexity of query processing.

Kim [11] proposed an algorithm that transforms SQL-like nested queries into equivalent flat SQL queries. He categorized nested queries into four types--type N, type J, type JX, and type JA. These queries are the set of source queries including set-membership queries using the IN operator, set-inclusion queries using the CONTAINS operator, and various aggregate functions. By transforming the nested queries into equivalent join queries, Kim enabled the query optimizer to use the most appropriate join computation method. Ganski and Wong [7] proposed an algorithm to fix the bug in Kim's algorithm. Furthermore, the algorithm they proposed extends the set of source queries that can be transformed. For example, it is able to process unnested queries containing the EXIST operator.

Baekgaard and Mark [3] proposed a two-step strategy for computing nested relational query expressions. In the first step, it transformed the nested query expressions into unnested flat query expressions. In the second step, it computed the flat query expressions. The transformation is applied to nested algebraic queries, and the transformed queries are algebraic queries as well. Their major contribution is the use of a concise and readable algebra to algebraic notation and the proof of correctness.

Yang and Liu provided techniques [16] to process nested fuzzy SQL queries. They investigated the problem of unnesting SQL-like queries in a fuzzy database environment and modified those techniques for non-fuzzy databases. They also proposed unnesting techniques for fuzzy databases. Their unnesting techniques are similar to Kim's techniques, both of them translated nest queries to equivalent join queries. Owing to large amount of join operations would exist

in flat queries after unnesting, they proposed a modified standard merge join method to make their processing more efficient.

In object-oriented database area, many proposed object query languages do not address on the processing of nested query expressions. In this paper, we propose query processing techniques for nested query expressions. Nested queries are expressed in a declarative language - Object Query Language (OQL) which is defined in ODMG-93 [5]. Nested query expressions are first transformed into a query graph by using query graph transformation algorithm. The query graph is then converted to execution plans by using transformation rules.

In this paper, we use Access Support Relations (ASRs) technique [9] to represent and manipulate path expressions in our query processing. ASRs are proposed to support associative search for objects in secondary storage, e.g., implicit join along path expression which is an arbitrary long attribute chains that may even contain set-valued attributes. Finally, we define a set of operators to be the interface of object manager. The access routines of these operators are the executable elements of query execution plans, they can manipulate internal data in processing and executing low level I/O operations by calling function calls provided by object manager. Although the processing cost of the strategy may be expensive, but it is easy to implement.

The rest of this paper is organized as follows. In Section 2, we describe three query types and give some examples. In Section 3, we present the query graph approach and explain how to translate nested query expression into the query graph. In Section 4, we describe the ASRs technique and the generation of execution plans. Conclusion and future works are given in Section 5.

2 Query Types and Examples

In this section, we briefly introduce the object query language named OQL which defines in ODMG-93 [5]. Then, we classify queries into three types and give examples.

2.1 Object Query Language (OQL)

OQL is a high-level declarative language for querying and updating database objects. It can be used as a stand-alone language or embedded in a host language program. As an embedded language, OQL query is a function which, when applied to native language input, delivers an object whose type may be inferred from the operator contributing to the query expression. There are some of design principles and assumptions about the OQL. One of these is that OQL relies on the ODMG data model. The query language supports both types of objects: mutable (i.e., having an OID) and literal (identity = their value), depending on the way those objects are constructed or selected. For more details of OQL, the reader may refer to [5].

2.2 Query Types and Examples

In this Section, we categorize queries into three types -- simple, nested, correlated. Then, examples of these three types of queries are given. The queries in this paper are defined on the class schema shown in Figure 1.

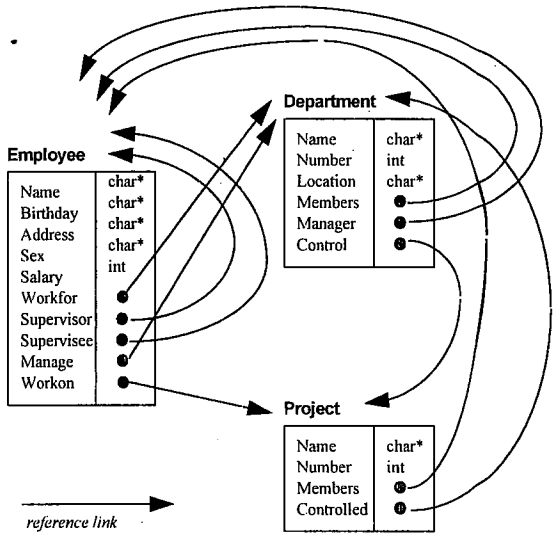

Figure 1. Class Schema of Employee Database

2.2.1 Simple Query

Simple query is an one-level query that there is no other subqueries in subclause. The format of simple query is

select TargetClause
from RangClause
where QualificationClause

The target clause is the specification of the attributes to be output; the range clause indicates the

binding variable, called object variables, to corresponding sets of instances of classes; the qualification clause specifies the qualification conditions as a Boolean combination of predicates. Q1 is an example of simple query.

Q1: Retrieve the name of employee whose supervisor is the manager of the department located in Taipei.

```
select  e.Name
from    e in Employee, d in Department
where   e.Supervisor = d.Manager and
        d.Location= "Taipei"
```

2.2.2 Nested Query

Nested queries permit the use of one SQL query within the **from** clause or **where** clause.

Q2: Q1 can be rephrased to a nested query shown in Q2.

```
select  e.Name
from    e in Employee, d in
            select   y
            from     y in Department
            where    y.Location= "Taipei"
where   e.Supervisor = d.Manager
```

2.2.3 Correlated Query

In nested queries, the variable that defined in outer queries exits in inner queries. This kind of queries is correlated queries.

Q3: Retrieve the name of employees who work on the projects are all controlled by the department located in Taipei.

```
select  e.Name
from    e in Employee
where for all x in
        e.Workon: x.Controled.Location = "Taipei"
```

Q3a: Query **Q3** can be rephrased to the following expression.

```
select  e.Name
from    e in Employee, d in
            select   y
            from     y  in  Department
            where    y.Location = "Taipei"
where for all x in e.Workon: x.Controled = d
```

Q4: Retrieve the name of employees who work on at least one of the projects are controlled by the department located in Taipei.

```
select  e.Name
from    e in Employee
where exist x in e.Workon:
            x.Controled.Location = "Taipei"
```

3 Query Processing Techniques

3.1 System Architecture

Before introducing the query graph approach, we describe the architecture of query processor of our system. The query processor architecture is simpler than the object query-processing methodology proposed by Straube and Ozsu [14]. In Figure 2, the declarative query expression is first parsed to internal form which is composed of parsing trees and several tables such as variable table, path expression table, class table,...,etc. The internal form is then translated into a query graph by Query Graph generator. Execution Plan Generator is the process of mapping query graph to a sequence of data manipulation operators. Finally, the Runtime Database Executor runs the execute plan to generate query results.

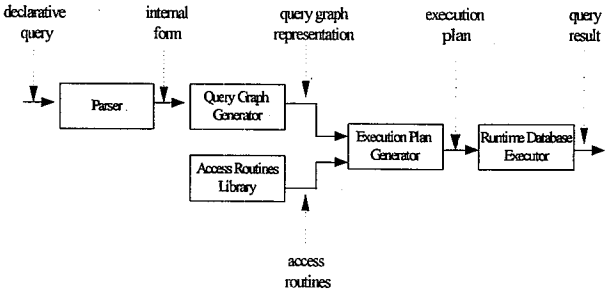

Figure 2. Query Processor Architecture

3.2 Query Graph

Query graph is a representation of query expression. The query graph is hierarchical, the hierarchical characteristic fits for the nested characteristic of queries in OQL.

3.2.1 Definitions

Definition 1. A Query Graph *G* is a connected graph

which is composed of Nodes, N, and Links, L. It can be expressed as $G = [N, L]$.

Definition 2. The nodes in N are divided into three kinds. They are *Simple-Node* N_s, *Collection-Node* Nc, and *Path-Node* N_p.

Definition 3. The *Simple-Node* N_s is the primitive node which can not be decomposed into any other nodes. It is used to represent a variable or the intermediate attribute node in path expression (attribute chain). It is composed of three elements: *Var, Type, and Filter. Var* is a variable name. *Type* is the corresponding type of the variable. *Filter* is predicate against the node. N_s can be represented as [*Var, Type, Filter*].

The type of Simple-Node can not be Literal such as int, char, float,..., etc. The Simple-Node is used to represent complex attribute rather than literal attribute. For example, **e.Name = "John"** can be represented as follows: $N_{s(e)} = [e, Employee, Name="John"]$,

Definition 4. The *Collection Node* N_c can be decomposed into another sub-query graph *Gsub* and then the project operation (project operation is similar to that in relational model) is applied to the graph. The *projection-list* is a list of attributes to be projected by *Project( )* from the subgraph *Gsub*. A collection node N_c is denoted as [*Var, Type, Filter*], where *Type is Project(Gsub, projection list)*.

The return type of this node is a bag or a set of objects. In the type hierarchy against ODMG model, bag and set is the subtype of the type Structured_Object. We implement it with persistence class in object manager to save the intermediate query result.

Definition 5. The *Path-Node* N_p is not decomposable. It is used to represent the set-attribute corresponding to a declared variable in universal or existential quantifications. A path node is denoted as $N_p = [Var, Path, Filter]$.

Definition 6. The links in L are divided into five kinds. They are *Navigation-Link* L_n, *Join-Link* L_j, *All-Link* L_{all}, *Exist-Link* L_{ext}, and *In-Link* L_{in}.

Definition 7. The *Navigation-Link* L_n is a directed primitive link that can not be decomposed. It has three elements, two nodes that are connected by a link, and a corresponding attribute. L_n can be represented as $[(N_1, N_2), Attribute]$. L_n is used to represent path expression. For example, **e.Workon** is a directed primitive link that connects two nodes, Employee and Project. The path expression can be formulated as follows:

$$N_{s(e)} = [e, Employee, null],$$
$$L_{n(workon)} = [(N_{s(e)}, N_{s(p)}), Workon],$$
$$N_{s(p)} = [p, Project, null].$$

The graphic representation of Navigation-Link is sketched in Figure 3.

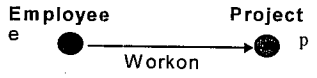

Figure 3. The graphic representation of an example of Navigation-Link

Definition 8. The *Join-Link* L_j is an undirected primitive link. It connects two nodes using the join attribute *JoinAttr*. The objects in these two nodes are joined by their oids rather than their attribute values. L_j can be represented as $[N_1, N_2, JoinAttr]$. If the *JoinAttr* is null, it means the types of both nodes are not primitive types. L_j is used to specify the join condition in a query. For example, the join condition in Q1: **e.Supervisor = d.Manager** can be formulated by using a sequence of simple nodes and navigational links and finally is joined by L_j (see below).

$$N_{s(e)} = [e, Employee, null],$$
$$L_{n(supervisor)} = [(N_{s(e)}, N_{s(e')}), Supervisor],$$
$$N_{s(e')} = [e', Employee, null],$$
$$N_{s(d)} = [d, Department, null],$$
$$L_{n(manager)} = [(N_{s(d)}, N_{s(d')}), Manager],$$
$$N_{s(d')} = [d', Employee, null],$$
$$L_{j(sv,mgr)} = [N_{s(e')}, N_{s(d')}, null]$$

The graphic representation of the above expressions is shown in Figure 4.

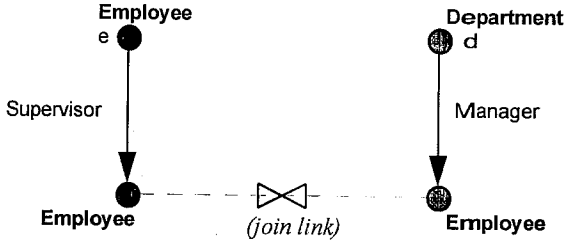

Figure 4. The graphic representation of e.Supervisor = d.Manager

Definition 9. The *All-Link* L_{all} is a directed and decomposable link. It is equivalent to a sub-query-graph *Gsub* which contains Path-Node in it. Both of nodes connected by L_{all} must be the same type. L_{all} can be represented as $[(N_1, N_2)$, *Gsub*], where *Gsub* is constructed by a Path-Node.

L_{all} is used to represent the universal quantification. For example, the predicate in Q3 in the **where**-clause is "**for all** x **in** e.Workon: x.Contorlled.Location=Taipei". We can view it as a filter of outer variable e whose filter condition is **x.Controlled.Location = "Taipei"**. The predicate in the where-clause of Q3 can be formulated as follows:

$N_{s(e)}$ = [*e*, *Employee*, *null*],
L_{all} = [($N_{s(e)}$, $N_{s(e')}$), *Gsub*],
$N_{s(e')}$ = [*e'*, *Employee*, *null*],
Gsub = [
$N_{p(x)}$ = [*x*, *e.Workon*, *null*],
$L_{n(controlled)}$=[($N_{p(x)}$,$N_{s(controlled)}$), *Controlled*],
$N_{s(d)}$ = [*d*, *Department*, *Location="Taipei"*],
]

The graphic representation of the above expressions are sketched in Figure 5.

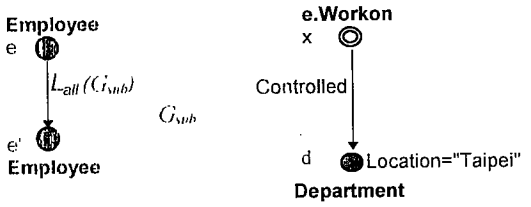

Figure 5. The graphic representation of Qualification clause in Q3

Definition 10. The *Exist-Link* L_{ext} is similar to that of All-Link except the Exist() quantification.

Definition 11. The *In-Link* L_{in} is a directed and primitive link. L_{in} can be represented as $[(N_1, N_2)]$, where N_1 and N_2 are two sub-query graphs. Let us illustrate it by using the following query, Q5.

Q5: Retrieve the name of employee whose supervisor is one of the supervisee of Taipei department manager.

select e.Name
from e.in Employee, d in Department
where e.supervisor **in** d.Manager.Supervisee **and**
d.Location = "Taipei"

The predicate in the where clause can be formulated using the following expressions.

$N_{s(e)}$ = [*e*, *Employee*, *null*],
$L_{n(supervisor)}$= [($N_{s(e)}$, $N_{s(supervisor)}$), *Supervisor*],
$N_{s(e')}$ = [*e'*, *Employee*, *null*],
$N_{s(d)}$ = [*d*, *Department*, *Location="Taipei"*],
$L_{n(manager)}$ = [($N_{s(d)}$, $N_{s(e'')}$), *Manager*],
$N_{s(e'')}$ = [*e''*, *Employee*, *null*],
$L_{n(supervisee)}$ = [($N_{s(e'')}$, $N_{s(ee)}$), *Supervisee*],
$N_{s(ee)}$ = [*ee*, *Employee*, *null*],
$L_{in(spr,spe)}$ = [($N_{s(e')}$, $N_{s(ee)}$)]

The graphic representation of the above expressions is sketched in Figure 6.

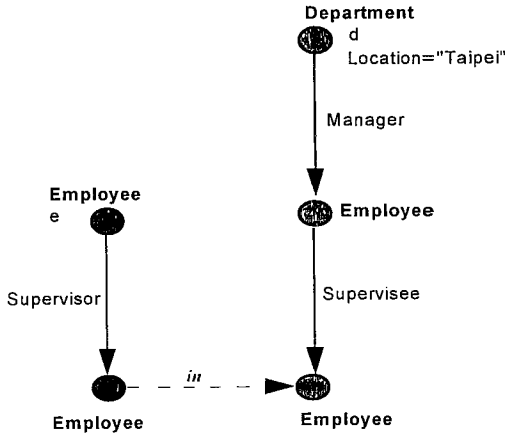

Figure 6. The graphic representation of Qualification clause of Q4

3.2.2 Query Graph Transformation

In this section, we propose an algorithm to transform a query expression into a query graph. Queries are expressed in OQL. The idea of the transformation is based on the internal storage structure, Access Support Relation. We map a path expression in a query to an ASR storage structure. A path expression maps to a combination of Simple-Nodes and Navigate-Links. A path decides the order of execution of navigation. Each

construct in query graph could map to a sequence of operators. A Simple-Node maps to an intermediate state of ASRs; A Navigate-Link maps to a navigate operator; A Join-Link maps to a join operator; A filter condition maps to a select operator; A projection list maps to a project operator. The advantage of our design is that it is easy to translate the query graph to execution plans.

Transformation Algorithm

1. A non-literal variable or non-literal attribute in a path expression is transformed into a Simple-Node N_s. The name of node is identical as that of a variable. If there is no variable name, the system will assign a unique name to this node automatically.
2. Two non-literal attributes in a path expression construct a Navigation-Link L_n.
3. The join condition in qualification clause construct a Join-Link.
4. The result of the evaluation of from-where clause is a bag or a set of objects. It is represented by a Collection-Node N_c. The **select** clause then projects required attribute values.
5. Qualification clause must be transformed into the disjunction normal form, i.e., (query **and** query **and** ...) **or** (query **and** query **and** ....) **or** .... Each subexpression in a pair of parentheses constructs a Collection-Node. These Collection-Nodes are combined by Union operation and then produce a new Collection-Node.
6. A clause containing **for all** universal quantification is transformed into to a All-Link L_{all}.
7. A clause containing **exist** universal quantification is transformed into to a Exist-Link L_{ext}.
8. A clause containing **in** quantification is transformed into a In-Link L_{in}.

Query graphs **QG1** and **QG3a** can be derived from the queries **Q1** and **Q3a** respectively by applying the above transformation algorithm. The query graphs are shown in Figure 7a and Figure 7b, respectively.

4 Access Support Relations

Access Support Relations (ASRs) techniques are proposed by Kemper and Moerkotte [17]. ASRs are introduced as a means for optimizing query processing in object-oriented database systems. The general idea is to maintain separate structures (dissociated from the object representation) to redundantly store those object references that are frequently traversed in database queries.

According to the definition of [9]. A path expression has the form : $o.A_1.....A_n$, where o is a tuple structures object containing the attribute A_1 and $o.A_1.....A_i$ refers to an object or a set of objects, all of which have an attribute A_{i+1}. The result of the path expression is the set of objects (or values) of type tn that can be reached from o via the specified attribute chain. Formally, the definition of a path expression or attribute chain can be found in [9].

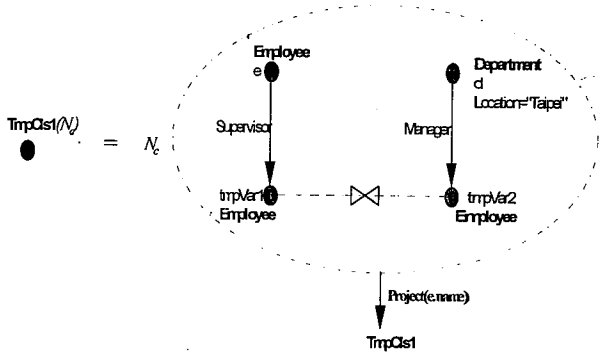

Figure 7a. Query Graph **QG1**

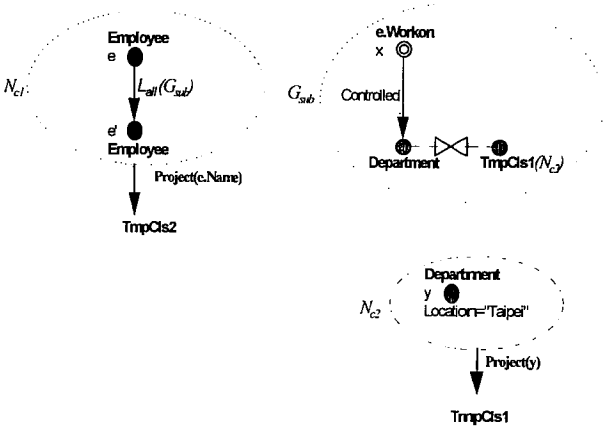

Figure 7b. Query Graph **QG3a**

4.1 Operators to Manipulate ASRs

We define a set of operators to manipulate ASRs in the query processing. These operators are described as below.

I/O Operators

- **NewAsr:** This operator is used to create a new canonical extension of ASRs (denoted ASR_{can}). This canonical extension is initialized by filling the extents of the specified class. The format of the operator is

  ASR_{can} = NewAsr(initial class).

- **SetAttrToAsr:** This operator generates a canonical extension ASR_{can1} to store the oid of $o_i.A$, where o_i is the specified object identifier in ASR_{can} $[S_0,..., S_i,...,S_n]_{can}$ and A is a set-attribute of o_i. The format of the operator is

  ASR_{can1} = SetAttrToASR(ASRcan, $o_i.A$)

  where ASR_{can1} is a temporal ASR.

- **AsrToCls:** This operator save $[S_0, \ldots, S_n]_{can}$ to a temporal class via object manager. The $[S_0, \ldots, S_n]_{can}$ forms instances of a class which attribute specification is decided by $S_0, \ldots, S_n$. The format of this operator is

  temporal class =AsrToCls(ASR_{can}).

Basic Operators

- **Select:** The select operation is used to select a subset of tuples in an ASR that satisfy a selection condition. The operator has its format as follows:

  ASR_{can} = Select(ASR_{can} , selection condition),

  or $[S_0, \ldots, S_n]_{can} = \delta_{<cond>}([S_0, \ldots, S_n]_{can})$.

- **Navigate:** If the path expression $[t_{n-1}.A_n]$ is a subpath of the path expression $[t_0.A_1....A_n]$ and there already existing a canonical extension of $ASR_{can}[S_0, \ldots, S_{n-1}]_{can}$. Also, the domain of t_{n-1} is the same as that of S_{n-1}. The operator will extend the specified $ASR_{can}[S_0, \ldots, S_{n-1}]_{can}$ to $ASR_{can}[S_0, \ldots, S_{n-1}, S_n]_{can}$, where S_n is the domain of attribute A_n. The operator has its format as follows:

  ASR_{can2} = Navigate(ASR_{can1} , $t_{n-1}.A_n$),

  or

  $[S_0, \ldots, S_n]_{can2} = \nu_{t_{n-1}.A_n}([S_0, \ldots, S_{n-1}]_{can1})$.

- **Join:** This operator is used to join two ASRs according the join condition. The format of the operator is

  ASR_{can3}=Join(ASR_{can1}, ASR_{can2}, join condition),

  or

  $[S_0, \ldots, S_{2n+1}]_{can3} = [S_0, \ldots, S_n]_{can1} \bowtie_{<join\ condition>} [S_0, \ldots, S_n]_{can2}$.

- **Union:** The operator is used to union two ASRs whose domains are compatible. The format of the operator is

  ASR_{can3}=Union(ASR_{can1}, ASR_{can2}),

  or

  $[S_0, \ldots, S_n]_{can3} = [S_0, \ldots, S_n]_{can1} \cup [S_0, \ldots, S_n]_{can2}$

- **Difference:** The operator is used to difference two ASRs whose domains are compatible. The format of the operator is

  ASR_{can3} = Difference(ASR_{can1}, ASR_{can2}),

  or

  $[S_0, \ldots, S_n]_{can3} = [S_0, \ldots, S_n]_{can1} - [S_0, \ldots, S_n]_{can2}$

- **Project:** The operator selects certain columns from the specified ASR and discards the other columns. The format of the operator is

  ASR_{can2} = Project(ASR_{can} , projection-list),

  or

  $[S_0, \ldots, S_i]_{can\,2} = \pi_{<projection\text{-}list>}([S_0, \ldots, S_n]_{can})$ $0 \le i \le n$

  where project-list is a list of attributes of the specified ASR_{can}.

Extended Operators

- **All:** The operator is used to implement All-Link.
- **Exist:** Similarly, the operator is used to implement Exist-Link.

4.2 Execution Plan Generation

Execution plan generation is the process of mapping a query graph to a set of operators defined in the previous section. In this paper, we do not address the problem of query optimization. Query optimization techniques in relational databases can be used in our nested query processing. Let us give an example to illustrate the generation of execution plan. Consider the query graph shown in Figure 7 for Query 1. The execution plan is shown as follows.

Step1: asr1= NewAsr("Employee");
Step2: asr2= Navigate(asr1, e.Supervisor);
Step3: asr3= NewAsr("Department");
Step4: Select(asr3, d.Location= "Taipei");
Step5: asr4= Navigate(asr3, d.Manager);
Step6: asr5= Join(asr2, asr4, tmpVar1= tmpvar2);

Step7: asr6= Project(asr5, e.Name);
Step8: TmpCls1= AsrToCls(asr6);

5. Conclusion and Future Works

In this paper, we present nested query processing techniques in object-oriented databases. Nested queries are expressed in a declarative language-OQL which is defined in ODMG-93. A query is transformed into a query graph by using graph transformation algorithm. We defined a set of operations to manipulate ASRs in the query processing. Queries including nested and correlated queries can be processed. We can handle large amount path expressions that exist in an object-oriented queries. Since the storage structure of ASRs is the same as relation in RDB, many processing techniques in the relational database systems can be applied in our query processing. We do not address the optimization and indexing techniques in this paper, these are our future works.

References

[1] S. Abiteboul and P.C. Kandllakis, "Object Identity as a Query Language Primitive," in *Proc. of ACM SIGMOD Conf. on Management of Data, Oregon, pp. 159-173, 1989.*

[2] M. Alashqur, S. Y. W. Su, and H. Lam, "OQL: A Query Language for Manipulating Object-Oriented Databases," in *Proc. 15th Int. Conf. Very Large Data Bases, pp. 433-442, Aug. 1989.*

[3] L. Baekgaard, L. Mark, "Incremental Computation of Nested Relational Query Expressions," in *ACM Trans. Database Syst., Vol. 20, No. 2, pp. 111-148, June 1995.*

[4] J. Banerjee, W. Kim, and K. C. Kim, "Queries in Object-Oriented Databases," in *Proc. IEEE Data Engineering Conf., pp. 31-38, Feb. 1988.*

[5] R. G. G. Cattel, T. Atwood, J. Dubl, G. Ferran, M. Loomis, and D. Wade "The Object Database Standard: ODMG-93" in *Morgan Kaufmann Publishers, 1993.*

[6] L. Fegaras, D. Maier . "Towards an Effective Calculus for Object Query Languages," in *ACM SIGMOD International Conference on Management of Data, San Jose, California, pp. 47-58, May 1995.*

[7] R. A. Ganski, H. K. T. Wong, "Optimization of nested SQL queries revisited," in *SIGMOD International Conference on Management of Data , San Francisco, California, pp. 22-33, May , 1987.*

[8] M.S. Guo, "An Association Algebra: A Mathmatical Foundation for Object-Oriented Databases," *Ph.D. dissertion, EE Depaerment, University of Florida, 1990.*

[9] A. Kemper, G. Moerkotte. "Access Support Relations: An Indexing Method for Object Bases," in *Infomation Systems Vol.17, No. 2, pp. 117-145, 1992.*

[10] W. Kim, "Model of Queries," in *Intorduction to Object-Oriented Databases, Chapter 6, 1990.*

[11] W. Kim, "On optimizing an SQL-like nested query," in *ACM Trans. Database Syst. 7, 3(Sept.), pp. 443-449, 1982.*

[12] D. Maier, "Making database systems fast enough for CAD applications," in *Object-Oriented Concepts, Applications, and Databases, W. Kim and F. Lochovsky. Eds. Reading, MA: Addison-Wesley, 1989.*

[13] G. Ozsoyoglu., Z. Ozsyoglu, and V. Matos "Extending Relational Algebra and Relational Caculus with Set-Valued Attributes and Aggregate Functions," in *ACM Transactions on Database Systems, 12(4):566-592, December 1987.*

[14] D. D. Straube, M. T. Ozsu. "Execution Plan Generation for an Object-Oriented Data Model," in *Proceedings of 2nd International Conference on Deductive and Object-Oriented Databases, C. Delobel, M. Kifer, and Y. Masunaga, Eds. SpringerVerlag, pp. 43-67, 1991.*

[15] D. Woelk and W. Kim, "Multimedia information managemant in an object-oriented database system," in *Proc. Int. Conf. Very Large Data Bases, Brighton,England, pp.319-329, Sept. 1987.*

[16] Q. Yang, C. Liu, J. Wu, C. Yu, S. Dao, H. Nakajima, "Efficient Processing of Nested Fuzzy SQL Queries, " in *Proceedings of the Eleventh International Conference on Data Engineering, pp. 131-138, 1995.*