1-1-2005

Translating SQL queries to EJB-QL queries.

Yang Gao
University of Windsor

Translating SQL Queries to EJB-QL Queries

by

Yang Gao

A Thesis

Submitted to the Faculty of Graduate Studies and Research

through the School of Computer Science

in Partial Fulfillment of the Requirements for

the Degree of Master of Science at the

University of Windsor

Windsor, Ontario, Canada

2005

Your file Votre référence
ISBN: 0-494-09789-2
Our file Notre référence
ISBN: 0-494-09789-2

Abstract

In multi-tier applications it is common to have a relational database at the backend, a presentation tier at the front, and an object tier in between. Presentation tier will access the objects tier, which will in turn access the database. Obviously, there is a need to build a mapping between the object and the relational models. In particular, we need to study the translation techniques between object and relational queries.

Enterprise Java Beans (EJB) is a technique supporting this kind of object tier. In addition to the translation from EJB Query Language (EJB-QL) to SQL, which is needed to execute the EJB-QL, we also need to study the translation from SQL to EJB-QL when generating the object layer from legacy database application.

This thesis proposes an algorithm to translate SQL queries to equivalent EJB-QL queries. Since EJB-QL is an object query language, our work is based on translation techniques between relational and object query languages, and extends the existing works in the following aspects:

1) Existing works described the translation of a small subset of SQL queries with many restrictions. Our approach expands this subset to a larger one.

2) Our translation techniques are tailored to EJB-QL.

3) As far as we know, existing works are neither implemented nor tested. We applied and tested our system with real data from an industry product.

Key words: schema mapping, CMP field, CMR field, relational query graph, object query graph, key-based join, implicit join, explicit join, nested query, traverse

Acknowledgements

I am very happy to take this opportunity to convey my thankfulness to all people who helped me a lot during the whole process of my graduate study.

Firstly, I would like to thank my supervisor, Dr. Jianguo Lu, who gave me instructive help and valuable advice. Without his patience and valuable guidance, it could be impossible to complete this thesis fluently. Besides, he is a so nice person who can be regarded as a sincere friend in study and life.

Secondly, I would like to thank my external reader Dr. Jianwen Yang, my internal reader Dr. Alioune Ngom, Dr. Joan Morrissey and the committee Chair Dr. Arunita Jaekel, for their time and instructive suggestions and comments.

Thirdly, I would like to thank all my friends and colleagues, who discussed with me on such topic and provided useful suggestions and experience during the research.

Finally, I would like to give my special thanks to my parents, my husband, who gave me continuous support, encouragement and their endless love...

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

1.1.1 Multi-tier Applications

Multi-tier applications have become the norm for building enterprise software today. A multi-tier application usually has at least three tiers, i.e., presentation tier, object tier and database tier. The database works at the backend, the presentation tier is at the front, and the object tier stands in between. Presentation tier will not access database directly. Instead, it will access the objects, which will in turn access the database. Figure 1-1 illustrates the interactions between three tiers. In object tier, object-oriented queries are used to retrieve data from backend systems, typically a relational database. Obviously, building a mapping between the object and the relational models is needed. In particular, we need to study the translation techniques between object and relational queries.

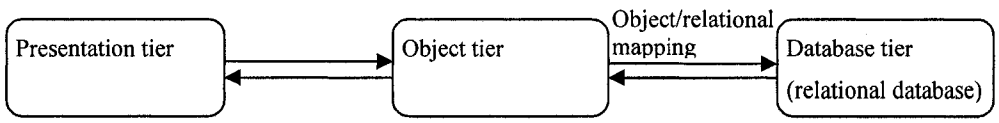


Figure 1-1 3-tier scenario

1.1.2 System Migration

However, the 2-tier model without the object layer has been widely used since early 90's. Many applications have been developed. The problem of migrating 2-tier

applications to 3-tier ones becomes more and more important. As shown in Figure 1-2, a 2-tier model intermingles presentation logic and business logic in one tier, and uses SQL queries to retrieve data from relational database directly. Consequently, the translation from SQL queries to object-oriented queries plays an important role in the migration from 2-tier to multi-tier model.

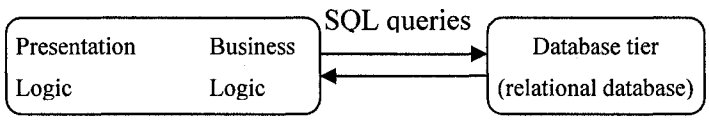


Figure 1-2 2-tier scenario

1.1.3 Enterprise Java Beans (EJBs) Upgrade

EJBs are server-side components that run in an Enterprise Java Bean (EJB) container [SUN01]. EJB is a technique supporting the object tier in a multi-tier model [LLH+01]. EJB Query Language (EJB-QL), an object query language [C04][H04], resides in the finder methods of EJBs to access the relational database. When an EJB-QL is executed, it needs to be translated into SQL statement by the EJB container.

Furthermore, when EJB1.0 was first developed in 1998, queries in those finder methods are most likely SQL-based. EJB-QL was not used until the introduction of EJB2.0 in 2001. Over the past several years many EJBs are developed and deployed, When the EJB container is upgraded to EJB2.0, SQL queries have to be translated into EJB-QL. So, in addition to the translation from EJB Query Language (EJB-QL) to SQL, we also need to study the translation from SQL to EJB-QL.

1.2 Enterprise JavaBens (EJBs)

There are three kinds of enterprise java beans: entity beans, session beans, and message driven beans.

1.2.1 Entity Bean

Entity beans are persistent objects that represent data in the database such as a relational database. For example, an instance of Customer entity bean could correspond to a row in the customer table. So changes to an entity bean result in changes to the database. The entity bean provides methods to select, add, modify, and delete underlying data. The slection of certain rows of the table corresponds to the selection of a group of beans satisfying certain condition. The selection of beans is accomplished by the finder method in the entity bean.

The EJB specification defines the Entity beans in two different categories: Container Managed Persistence (CMP) beans and Bean Managed Persistence (BMP) beans. CMP bean relies on the Container Provider's tools to generate methods that perform data access on behalf of the entity bean instances. BMP bean is more complicated than CMP because it relies on the bean developer to code the persistence logic into the bean class. In order to do this the developer must know what type of database is being used and how the bean's fields map to that database. In this thesis, we only interested in CMP entity beans.

1.2.2 Session Beans

Session beans are defined as an extension of the client application. They are the verbs in the process. Session beans drive all interaction with the database and are responsible for managing the process.

1.2.3 Message Driven Beans

Message driven beans are JMS message consumers that implement some business logic.

1.3 Thesis Overview

With EJB technology as the background, this thesis proposed an algorithm of translation from SQL queries to equivalent EJB-QL queries. Since EJB-QL is an object query language, our work is based on translation techniques between relational and object query languages. The related work was explored, and compared with our approach. A tool named "SQL2EJBQL Translator", which supports an automated translation, is designed to translate the SQL statements in finder methods in CMP1.0/1.1 entity beans to EJB-QL.

The rest of this thesis is organized as follows: In Chapter 2, the difference between SQL and EJB-QL is described. Chapter 3 describes the schema mapping between the relational and object schema, which is the first step of SQL to EJB-QL translation. Chapter 4 first introduces the approaches of the previous related papers, then explains algorithm proposed in this thesis. The overall architecture and design details of the translation tool are described in Chapter 5. The experiments and evaluation results will be given in chapter 6. Chapter 7 gives a conclusion of this thesis and discusses the work that needs to be done in the future.

Chapter 2

SQL vs. EJB-QL

Before the discussion of query translation, we introduce EJB Query Language (EJB-QL), as well as the difference between EJB-QL and SQL.

2.1 EJB Query Language (EJB-QL)

The syntax of EJB-QL is defined in the EJB 2.0 specification [SUN01]. It is an object query which has path expressions allowing navigation over the relationships defined between entity objects [AAG+01]. That is, a query can begin with one entity bean and, from there, navigate to the related beans. For example, the query can start with an Order bean and then navigate to the Order's line items. It can also navigate to the products referenced by the individual line items, and so forth.

At runtime, EJB-QL usually executes in the native language of the underlying data store. For example, a container that uses a relational database for persistence might translate EJB-QL statements into SQL statements, while an object-database container might translate the same EJB-QL statements into an object query language.

2.2 SQL vs. EJB-QL

2.2.1 General Syntax

The general syntax of the SQL and EJB-QL are very complicated. Here we give a brief description of the part of SQL and EJB-QL that can be translated by our approach.

```
<select-statement> ::= [ALL|DISTINCT] <select-clause>
                       <from-clause>
                       [<where-clause>]
```

Similar to SQL, an EJB-QL query contains a SELECT clause, a FROM clause, and an optional WHERE clause. Here is a part of the BNF syntax of EJB-QL:

```
EJB-QL::= [ALL|DISTINCT] <select_clause>
          <from_clause>
          [<where_clause>]
```

We focus on the subquery led by EXISTS, IN, and comparison predicates in where clause. We do not support the translation of groupby, order by, and having clauses.

2.2.2 Navigation vs. Join

EJB-QL allows entity objects and entity relationships to be easily navigated by path expressions. A path expression is denoted by an identification variable followed by the navigation operator (.) and a CMP field or CMR field. The type of the path expression is the type of the CMP field or CMR field to which the expression navigates.

This navigation in a path expression is composed using "inner join" semantics. This is the primary difference between EJB-QL and SQL. EJB-QL navigates to related beans, whereas SQL joins tables. In EJB-QL, when your return result is a variable that traverse a relationship by the navigation operator (`t.theAccount`), behind the scenes, a SQL JOIN condition might occur.

For example, the following SQL query bases on a banking system database schema illustrated in Table 2-1:

```
SELECT t.*
FROM TRANSRECORD T, ACCOUNT a
WHERE t.ACCID = a.ACCID and a.ACCTYPE = 'checking'
```

The equivalent EJB-QL which is against the corresponding object schema (abstract schema in EJB) (Table 2-2[1]) is:

```
SELECT object (t)
FROM Transrecord t
WHERE t.theAccount.acctype = 'checking'
```

Table Name	Column Name
ACCOUNT	ACCID,BALANCE,INTEREST,ACCTYPE
	PK = {ACCID} FK = none
CUSTOMER	CUSTOMERID,TITLE,FIRSTNAME,LASTNAME,USERID,PASSWORD
	PK = {CUSTOMERID} FK = none
TRANSRECORD	TRANSID,TRANSTYPE,TRANSAMT,ACCID
	PK = {TRANSID} FK = {ACCID(ACCOUNT:ACCID)}
CUSTACCT	CUSTOMERID,ACCID
	PK = {CUSTOMERID, ACCID} FK = {CUSTOMERID(CUSTOMER:CUSTOMERID)} {ACCID(ACCOUNT:ACCID)}

Table 2-1 The relational schema of the Banking System

The relationships between the tables are as follows:

```
ACCOUNT (many)<->(many) CUSTOMER
ACCOUNT (one) <-> (many) TRANSRECORD
```

[1] Table 2-2 explains the translated object schema in detail with entity bean's terms.

Bean Name	Abstract Schema Name	Attribute	
		CMP	CMR
Account	Account	balance	
		interest	
		accid	
		acctype	
			customers
			transrecords
Customer	Customer	customerid	
		title	
		firstname	
		lastname	
		userid	
		password	
		address	
			accounts
Transrecord	Transrecord	transid	
		transtype	
		transamt	
			theAccount

Table 2-2 The object schema (abstract schema) of the banking system example

2.2.3 Return Types

The SELECT statement of SQL allows one to retrieve records from one or more tables in the database. The return type can be the columns of tables.

In EJB-QL, the query in a finder method must return an entity bean or a collection of entity beans in which the finder method is defined. For example, a finder method for a `Customer` can only return `Customer` objects.

2.2.4 Query domain

The FROM clause of an EJB-QL query defines the domain of the query by declaring identification variables. An Identification variable designates an instance of an entity bean, it ranges over the abstract schema type of an entity bean. The FROM clause can contain multiple identification variable declarations separated by a comma (,), all identification variables must be declared in the FROM clause.

An identification variable always designates a reference to a single value. It is declared in one of the two following ways:

- A range variable is declared using the abstract schema name of an entity bean.

  ```
  Range_variable_declaration ::=abstract_schema_name [AS]
  identifier
  ```

- A collection member identification variable is declared using a collection-valued path expression. The identification variable declarations are evaluated from left to right in the FROM clause. A collection member identification variable declaration can used the result of a preceding identification variable declaration of the query string.

  ```
  Collection_member_declaration ::= IN
  (collection_valued_path_expression)[AS] identifier
  ```

For example, the following FROM clause contains two identification variable declaration clauses. The identification variable declared in the first clause is used in the second one. The range variable declaration `Customer AS c` designates the identification variable `c` as a range variable whose type is the abstract schema type, `Customer`. The identification variable `a` has the abstract schema type `Account`.

Because the clauses are evaluated from left to right, the identification variable a can use the results of the navigation on c.

```
FROM Customer AS c, IN (c.accounts) a
```

So the domain of the query may be constrained by path expressions.

In the FROM clause of SQL, all tables appearing in WHERE clause must be declared in FROM clause, while in EJB-QL it is different. Only the entity bean for which the finder method is defined, which we call it "root" bean, the bean that has many to one relationship with the root bean needs to be defined in FROM clause, or when query needs to compare multiple values ranging over the same abstract schema type, multiple range variable declarations may therefore be defined in FROM clause. Those beans, which can be navigated from the "root" bean, are not necessarily declared in FROM clause. For example, the following SQL query is based on the database schema in Table 2-1,

```
SELECT OBJECT(t)
FROM Transrecord t
WHERE t.theAccount.accttype = 'saving'
```

And EJB-QL query based on abstract schema shown in Table 2-2,

```
SELECT OBJECT(a1)
FROM Account a1, Account a2
WHERE a1.balance > a2.balance
```

2.2.5 Directionality

The directionality of a relationship specifies the direction in which you can navigate a relationship. Because foreign keys are used to join tables, relationships in a relational database are effectively bidirectional. This is why it doesn't matter in which table you

implement a one-to-one relationship, and the code to join the two tables is virtually the same. For example, suppose two tables `ORDER` and `SHIPMENT` hold a one-to-one relationship. When the foreign key is implemented in the `SHIPMENT` table, the SQL code which joins two tables would be

```
SELECT * FROM ORDER,SHIPMENT
WHERE SHIPMENT.ORDERID = ORDER.ORDERID
```

Having the foreign key implemented in the `ORDER` table, the SQL code would be

```
SELECT * FROM ORDER, SHIPMENT
WHERE SHIPMENT.SHIPMENTID = ORDER.SHIPMENTID
```

The directionality in entity beans may not correspond to the inherent directionality of the database schema. An entity bean can provide for directionality even though the database not do so easily, and vice versa [RAJ+01]. In EJB-QL, if the relationship between two entity beans is unidirectional, the relationship traversal is restricted. For example, if you declare that `Order` has a one-to-one relationship with `Shipment`, but you do not define the reverse one-to-one relationship that `Shipment` has with `Order`, you can get `Shipment` from `Order`, but not `Order` from `shipment`.

2.3 Subquery

EJB-QL 2.0 does not support subquery, but some application server vendors extend the standard EJB-QL to support subqueries. For example, WebLogic Server supports the subqueries as the operands of: the comparison operators ([NOT] IN, [NOT] EXISTS) and arithmetic operators (<, >, <=, >=, =, <> with ANY and ALL)[BEA04]. IBM WebSphere also adds extra capabilities to support such kind subqueries [IBM03].

Our system supports the translation of the following types of subqueries:

1. Subquery return type

   A subquery should only specify a single element in the SELECT clause. The return type can be one of a number of different types, such as:

   - Single CMP field type subqueries
   - Aggregate Functions
   - an entity bean

2. Nesting predicates

   - Comparison predicate
   - IN predicate
   - EXISTS predicate

2.3.1 Limitations in EJB-QL 2.0

Compared with SQL, EJB-QL has many limitations and hence not every SQL query can be translated into EJB-QL. EJB-QL is a powerful new technology that promises to improve portability of entity beans in CMP, but it covers just those aspects of SQL that is standard anyway, without enabling one to utilize advanced features provided by different databases where appropriate. Here are some restrictions in EJB-QL 2.0:

- Currently, container-managed persistence does not support inheritance. For this reason, two entity beans of different types cannot be compared in EJB-QL.
- String and Boolean comparison is restricted to = and <> in EJB-QL.
- EJB-QL does not support "outerjoin".
- EJB-QL does not support any kind of subqueries.
- EJB-QL does not support aggregation functions.

- EJB-QL does not support ORDER BY, GROUP BY and HAVING clauses.

2.3.2 EJB-QL development

The development of EJB specification started from version 1.0 and now is reaching version 3.0. There are significant changes from 1.1 to 2.0. The EJB 2.0 specification introduces EJB-QL. EJB2.1 [SUN03] adds the support of aggregation functions and ORDER BY clause in EJB-QL. EJB3.0 [SUN05] adds the support for inheritance, outerjoin, GROUP BY clause, HAVING clause and subqueries. Table 2-3 describes the development of EJB-QL from 2.0 to 3.0, it also compares the EJB-QL which supported by our translation system with different versions of EJB-QL specification.

EJB-QL / Supported Items	EJB-QL 2.0	EJB-QL 2.1	EJB-QL 3.0	Ours
ORDER BY	X	√	√	X
aggregation function	X	√	√	√
Outer join	X	X	√	X
Inheritance	X	X	√	X
subquery	X	X	√	√
GROUP BY	X	X	√	X
HAVING	X	X	√	X

Table 2-3 The development of EJB-QL from 2.0 to 3.0

Chapter 3

Schema Mapping

SQL is based on a relational model, while EJB-QL is based on entity beans, i.e., objects. To translate SQL to EJB-QL, the first step is to construct a mapping between relational and object models. This chapter describes the basics of the mapping and our approach in constructing such a mapping.

3.1 Introduction

The mapping between the relational and the object models in EJB can be viewed as the traditional object/relational mapping problem that has been extensively studied for a long time. A variety of methods have been proposed [AAK97] [RSH96] [FV95] [PTK95][CSG94][YL93][J97], many of them are in the area of database reverse engineering. Unfortunately, there is no standard way to specify the actual O/R mapping with CMP entity beans.

The EJB specification does not provide for a particular mapping method between an object model and a relational database schema. Typically programmers define the mapping in deployment descriptor, and EJB containers use the deployment descriptor to map the entities and relationships to normalized database tables. The mapping capability of the EJB container varies from implementation to implementation, which is one of the "value added" services on which vendors compete [SGT01].

An EJB container is just a specific persistence framework to leverage the power of the Enterprise JavaBeans component model, with a mapping tool put on top of it. The mapping tool generates information that maps the entity bean's container-managed fields to a data source, such as a column in a relational database table. This mapping information is stored in an XML file. Lots of sophisticated O/R mapping products are available for integration with application servers, such as TopLink[P03][ORA03], Cocobase [THO02][THO03], and BeanMaker [DB04], etc.. Most of these tools may perform the O/R mapping either automatically from existing database or based on simple specifications supplemented by the user through an easy-to-use GUI interface.

In WebSphere Application Developer, EJB object persistency mapping is aided with tools to map tables to CMP EJBs, using the top-down, bottom-up, or meet-in-the-middle methodology. For the bottom-up approach, tools are available to map existing database tables to EJB entities. For top-down, smart guides are available for creating EJBs and mapping the associated EJB entity object model to database tables. For meet-in-the-middle schemes, tooling is available for mapping EJB fields to columns and for composers (computing one CMP field value out of multiple columns), converters (type conversion and simple computation on one column), etc.

3.2 O/R mapping in EJB

O/R mappings in EJB define how EJB objects and their attributes are to be represented in the database. To simplify the procedure, here we assume that there is a one-to-one correspondence between tables and entity beans, and the relationships between entity beans are bi-directional. We use the banking system example given in section 2.2.2 to

demonstrate the general mapping rules which map the exampled relational schema to an object schema (abstract schema) of entity beans. Figure 3-1 is the ER-schema of exampled database, which models a bank with many customers. A customer may have multiple bank accounts and an account may be owned by multiple customers. A transaction record is generated for each banking transaction, such as deposit, withdrawal, or transfer of money between two accounts. A bank account may have many transaction records.

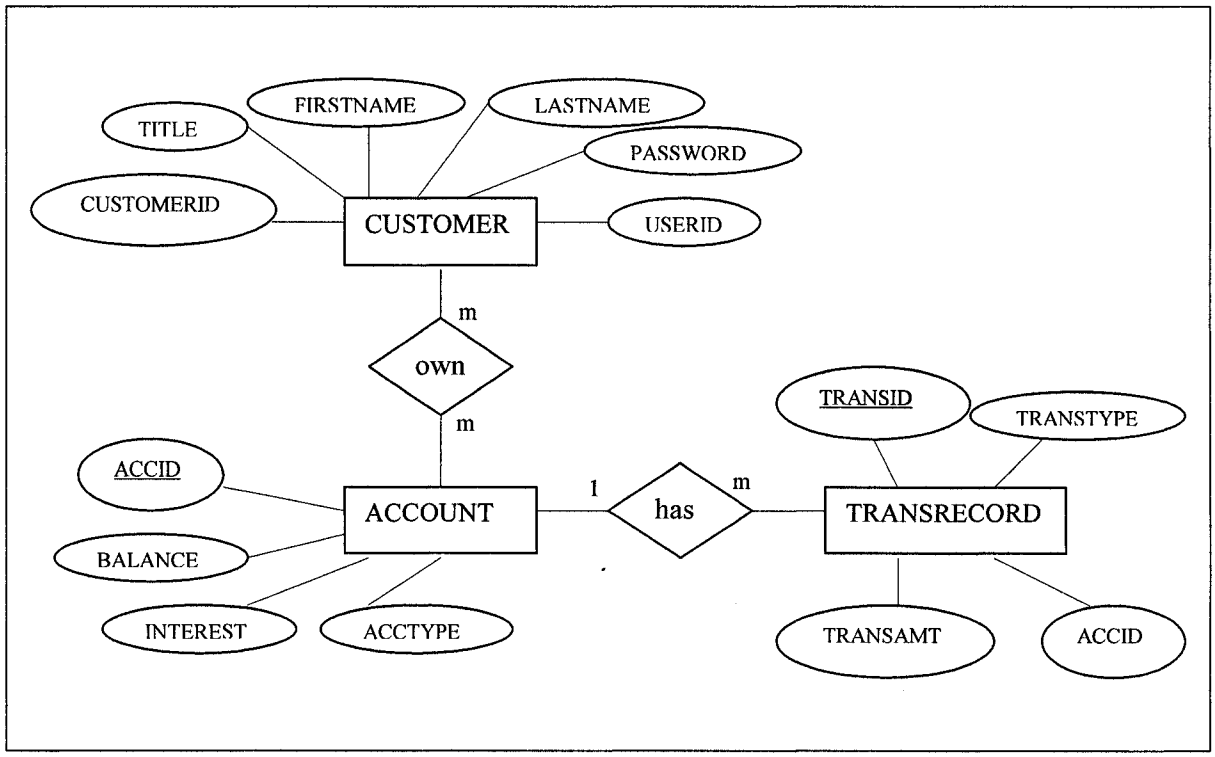


Figure 3-1 The E-R schema of the Banking System

General Mapping Rules

In the following we present the general mapping rules of schema transformation, namely, relation mapping rule and relationship mapping rule. We will demonstrate them in the example of banking system shown in Table 2-1. Each mapping rule is

described below in two parts. The first part describes the rule itself. The second part illustrates the rule with the help of an example.

Rule 1: relation mapping rule

This rule is to map each database relation (table) which is not represent a relationship to a single entity bean. Each non-foreign key column is mapped to a persistent field (CMP field) of the entity bean created for this table. SQL data types are mapped onto the corresponding Java types.

Example of banking system: Up to this step, the partially translated schema will be the following.

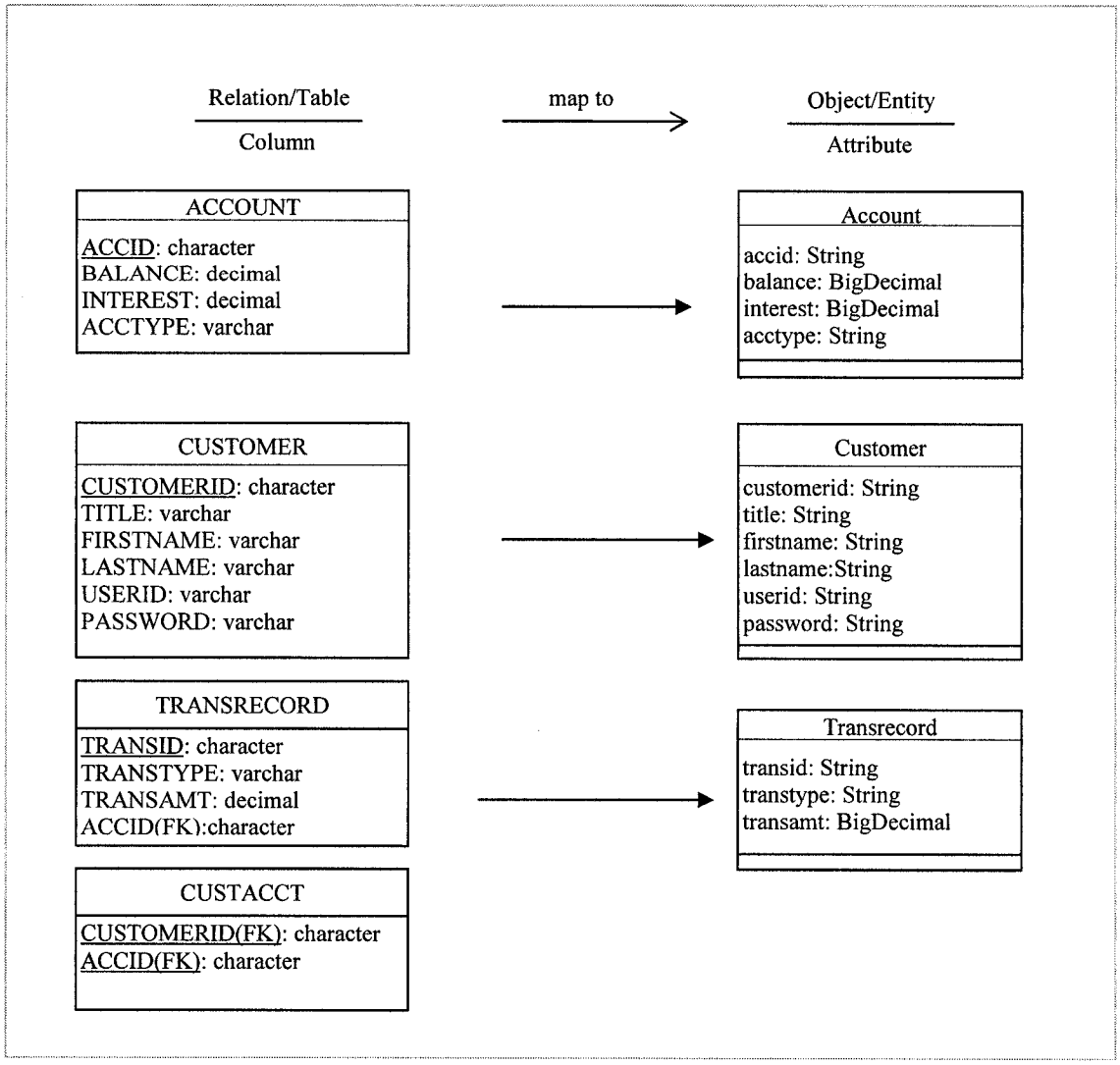


Figure 3-2 Partially translated object schema applied relation mapping rule

Rule 2: one-to-one or one-to-many relationship mapping rule

In relational database, a one-to-one or one-to-many relationship between two tables is represented by a foreign key dependency. It is mapped to object references (CMR fields) in both entity beans. Each object reference in one bean refers to the other one.

Example of banking system: There is a one-to-many relationship between two tables `ACCOUNT` and `TRANSRECORD`. The relationship is represented with the

foreign key ACCID in table TRANSRECORD, and it is mapped to a pair of references (CMR fields) – transrecords in class Account and theAccount in class Transrecord. transrecords is a set/multi-valued CMR field which refers to the objects of class Transrecord; theAccount is a single-valued CMR field referring to the object of class Account. After applying this rule, the partially translated object schema is:

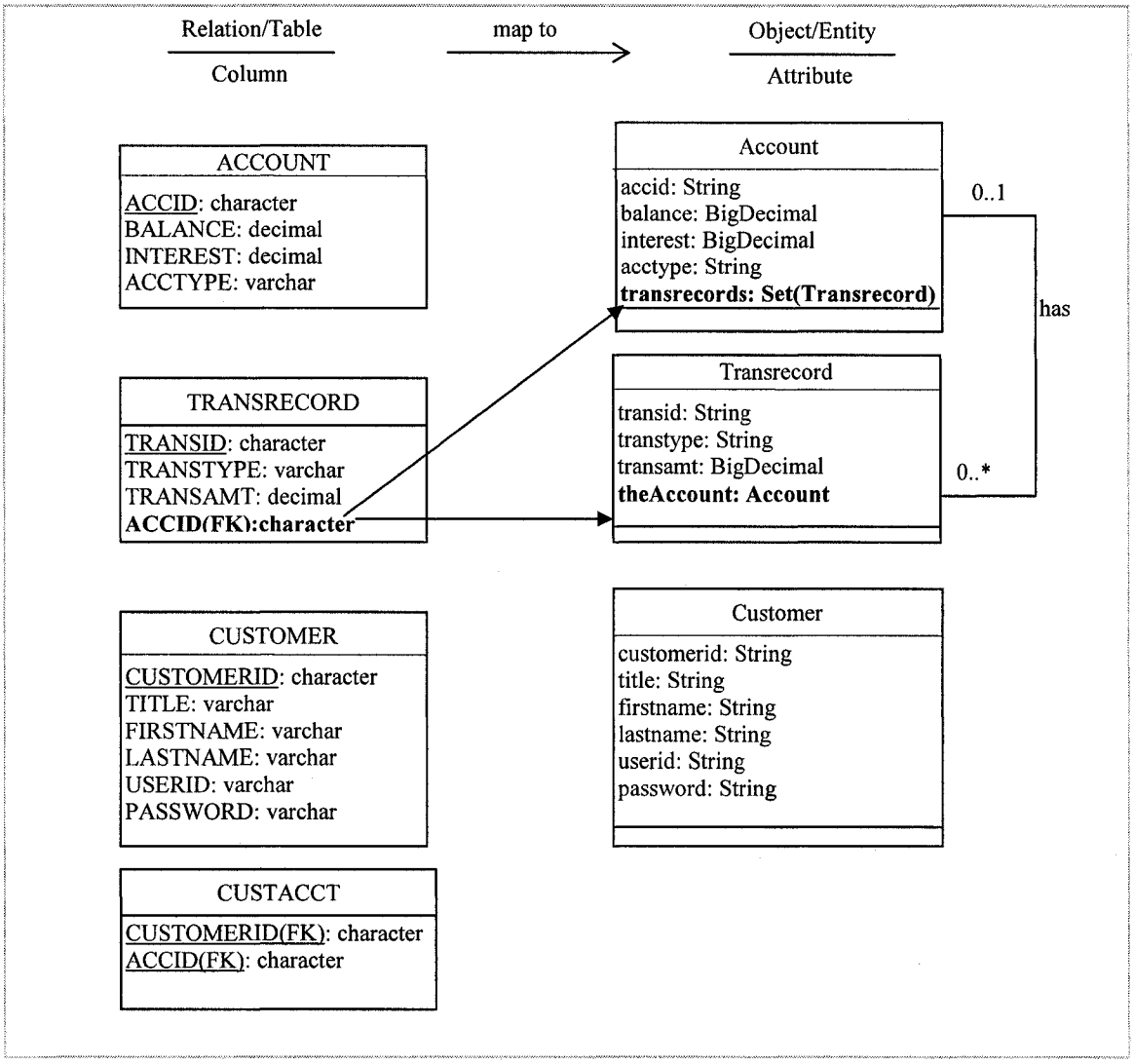


Figure 3-3 Partially translated object schema applied 1:1/1:m relationship mapping rule

Rule 3: many-to-many relationship mapping

In the relational database, the many-to-many relationship is implemented as an association table. This table only contains two foreign keys, each pointing to the table of one of the related entities. Many-to-many relationship between two tables, two set/multi-valued object references (CMR fields) are introduced in. If the association table has its own attribute, this table should be mapped to an entity.

Example of banking system: There is a one-to-many relationship between two tables `ACCOUNT` and `CUSTOMER`. The relationship is represented by the association table `CUSTACCT`, and it is mapped to a pair of set/multi-valued references (CMR fields) – `accounts` in class `Customer` and `customers` in class `Account`. `accounts` is a CMR field which refers to the objects of class `Account`, and `customers` is a CMR field referring to the object of class `Customer`.

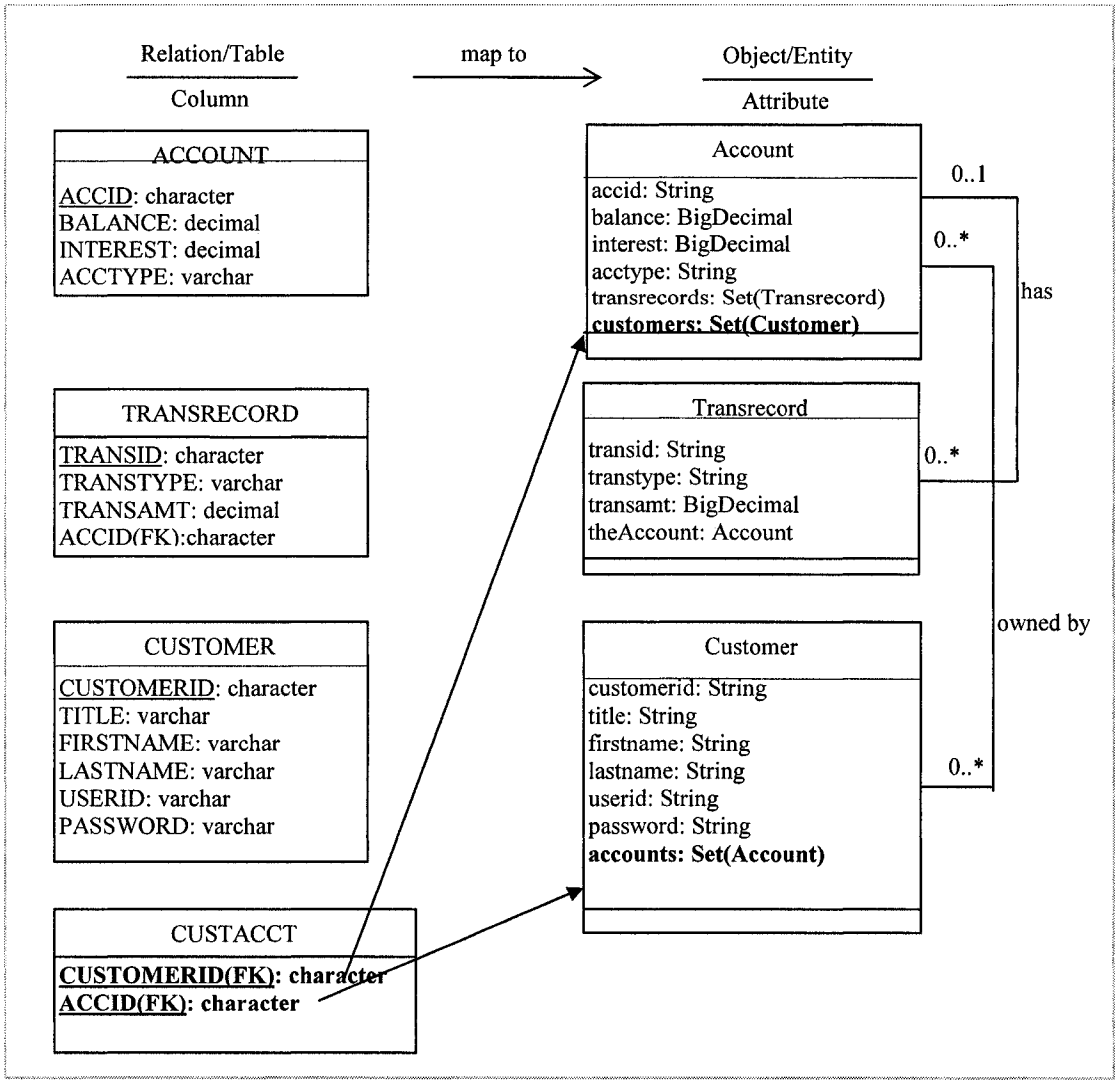


Figure 3-4 Final translated object schema applied all mapping rules

Finally, by applying the above schema mapping rules, this relational schema in Table 2-1 is transformed to the object schema (abstract schema) showed in Figure 3-4.

Inheritance is among the most powerful features of object-oriented technology. Unfortunately, it is not supported by EJB specification. Some EJB containers provide support for it as an extension to the specification. For example, in IBM's WebSphere Application Server suit, EJB inheritance is mapped to a single table, that is, the base

and all derived entity beans are mapped to the same database table or multiple tables (root/leaf).

Chapter 4

Query Translation

Since EJB-QL is nearly identical to object-oriented Query Language, the translation from SQL to EJB-QL is similar to the translation to Object Query Language (OQL).

4.1 Related Work

A few works have been done on the translation between SQL and object queries languages [MYK93] [VA95] [MK98] [YZM+95] [MYK95] [CRD94] [RC95]. [MYK93][VA95][MK98][CRD94] address the translation from SQL queries to object oriented queries. All of them use graphs to represent queries in the translation. One difference among them is that the target languages are different with each other. In [VA95], the SQL queries are translated to methods which are expressed in TM[2] [BAZ93][FKS94], a language for describing conceptual schemas of object-oriented databases, which allows the specification of data structures in terms of Classes and Sorts, and provides a computationally complete, functional data manipulation language for method and constraint specification. [CRD94] uses XSQL which is an extension to the SQL query. The target language used in [MYK93][MK98] is OQL, which is an SQL-like query language with special features dealing with complex objects, values and methods [P01].

[2] The language TM is an object-oriented data model that has been developed at the University of Twente and the Politecnico di Milano.

The goal of my thesis is to describe tactics for translating queries that contain nested blocks, aggregate functions, grouping and quantifiers. We use EJB-QL as target language to describe these tactics. However, the approach works just as well for other object oriented query languages that enjoy some or all of these features.

Since the syntax of EJB-QL is more similar to OQL, there are some strong similarities between the translation algorithms of [MYK93][MK98] and the one proposed in this thesis, but three important differences can be pointed out.

1) *Nesting queries:* [MYK93] does not discuss the translation of nested queries, in the assumption that there are unnesting algorithms [K82] that transform nested queries into an equivalent collection of simple queries. But this approach is inadequate for our purpose.

2) *Semijoin and antijoins*: as for the translation of subqueries, [MK98] proposed their solutions to translate queries with nesting predicates IN, NOT IN, EXISTS, and NOT EXISTS. However, their approach assumes these semijoins or antijoins are key based. For example, for a subquery of the form R.A (NOT) IN (SELECT S.B...), A and B should have key-foreign key relationship. Our approach removes this assumption and hence can translate more queries.

From the annotation they defined on the object schema and the definitions on the semijoin and antijoins, we understand that both of these two kinds of joins can be translated into object references between classes in the object schema. And these references in the object schema are based on foreign keys and association tables of the underlying relational schema, which means that the joins in SQL queries are key based

joins. This requires the subquery with (NOT) EXISTS predicate of a SQL statement to be correlated, which means the inner query should contain a join predicate that references the relation of the outer query, and this join is key based join. For example, in the following two queries, A and B should be a pair of key and foreign key.

Example 4.1. Queries that should have key-foreign key relationship between A and B, in [MK98]'s approach.

```
select R.* from R where R.A IN
(select S.B from S)
```

Example 4.2. A query Q2 =

```
select R.* from R where EXISTS
   (select S.* from S where R.A = S.B)
```

In my thesis, the joins and antijoins are expanded to any attributes. Meanwhile, the uncorrelated subqueries with (NOT) EXISTS predicates are also considered to be translated.

3) *Aggregate functions:* Both [MYK93] and [VA95] didn't mention the translation of SQL query which includes aggregate functions. Although EJB-QL 2.0 does not support aggregate functions, many application servers do support them. For example, IBM WebSphere allows to use aggregation functions (AVG, COUNT, MAX, MIN and SUM) in a subselect clause[3]. In my thesis, the subquery with the form of R.A = (SELECT MAX(S.B)...) can be translated.

[3] A subquery should only specify a single element in the SELECT clause.

The following table makes the comparison between my thesis and the related approaches.

Approaches	Simple query	Aggregation Function	Subquery nesting operator				
			IN, = NOT IN, <>		EXISTS / NOT EXISTS		
			Key-based join	Nonkey-based join	uncorrelated subquery	Correlated subquery (EXISTS)	
						Key-based join	Nonkey-based join
Clement Yu, et al	√	X	X	X	X	X	X
Ahmed Mostefaoui, et al	√	X	√	X	X	√	X
Ours	√	√	√	√	√	√	√

Table 4-1 The comparison of my thesis and the related approaches

4.2 Our Approach

Having compared with the related approaches of the query translation, this section describes our approach to the translation of SQL queries to EJB-QL. We focus on the translation of the subqueries with IN, EXISTS and comparison predicates.

As mentioned in 2.2.3, to satisfy the specification of EJB-QL defined in a finder method [IBM03][SUN01], the value of the query result which is specified by the SELECT clause, must be an entity bean in which the finder method is defined. So it is required that the SELECT clause of the outmost SQL statement can not have aggregation functions, and it need to be an all-column selection (for example, SELECT

CUSTOMER.*) [GW99]. In the SELECT clause of subqueries, aggregation function can be used, but only single element can be specified. WHERE clause specifies the qualification conditions of a query. We assume that the relational WHERE clause contains qualification conditions in conjunctive form.

Our translation methodology consists of three steps. In the first step, a relational query graph (RQG) of the given query is constructed. In the second step, the RQG is transformed to its corresponding object query graph (OQG). Finally, the translated EJB-QL is obtained from the transformed OQG.

The above three steps will be discussed in the following three sections. We will use the example as below.

Example 4.3. Consider the following SQL query defined on the relational schema Table *2-1*. This query finds the largest account of a particular customer.

```
select a1.*
from ACCOUNT a1, CUSTOMER c1, CUSTACCT ca1
where c1.CUSTOMERID=?
 and a1.BALANCE =
        (select MAX(a2.BALANCE)
         from ACCOUNT a2, CUSTACCT ca2
         where ca2.CUSTOMERID = c1.CUSTOMERID
            and a2.ACCID = ca2.ACCID)
 and ca1.ACCID = a1.ACCID
 and c1.CUSTOMERID = ca1.CUSTOMERID
```

4.3 Constructing a Relational Query Graph (RQG) from a relational query

We use a representation for SQL queries similar to the one used in [MK98]. A RQG is represented by a set of vertices RV, a set of undirected edges RE1, and a set of

directed edges RE2. In [MK98] a directed edge represents semijoin (with the conditions IN and EXISTS) or a special kind of antijoins (with the conditions NOT IN and NOT EXISTS), whereas in our definition, a directed edge represents a nesting operator which leads to a subquery.

Definition 1 (RQG). Given relational query Q, we define its RQG as an annotated graph: RQG(Q) = RQG(RV,RE1,RE2), where RV is a set of vertices, RE1 is a set of undirected edges, and RE2 is a set of directed edges that are defined as follows:

- A vertex v in RV corresponds to either: 1) Relation Tuple Variable (RTV)[4], denoted by RTV(v), or 2) the predicate "EXISTS". Vertex v is associated with a number which indicates the depth, or the level, of the subquery or query where the tuple variable is defined. The outermost query is of level 0.

  Each vertex can be annotated with a set of selection operations on its corresponding RTV.
- An undirected edge e_1 between vertices v_1 and v_2 in RE1 represents a join predicate. The annotation on e_1 is a join condition of the form < RTV(v_1).a op RTV(v_2).b>, where a and b are the attributes of RTVs RTV(v_1) and RTV(v_2) respectively, and op is a comparison operator such as '=' and '>'. The vertices reside in two ends can be either in the same or different query levels.
- A directed edge e_2 from vertex v_1 to vertex v_2 in RE2 indicates a nesting operator such as "IN", "NOT IN", "EXISTS", "NOT EXISTS", or a comparison operator (such as '>', '=', etc) with quantifier such as ANY and

[4] Relation Tuple Variable is a variable that "ranges over" named relation. SQL does not require the explicit introduction of a tuple variable, it allows the relation name to serve as an implicit tuple variable.

ALL. Aggregation function can appear at the right side of operator. The annotation ANNO on e_2 has the following forms:

a) RTV(v_1).a op [quantifier]*[Agg]*(RTV(v_2).b). Where op is (NOT) IN or comparison operator ("=", " >", etc.); a and b are non-key attributes of RTVs RTV(v_1) and RTV(v_2).

b) op [Agg]*(RTV(v_2).b). Where op is EXISTS; v_1 represents to EXISTS predicate; b is non-key attribute of RTV(v_2).

c) RTV(v_1).a op [quantifier]RTV(v_2).b. Where op is (NOT) IN, (NOT) EXISTS[5] or comparison operator ("="); a and b are key attributes with foreign key relationship of relation RTV(v_1) and RTV(v_2).

We obtain a RQG by parsing the SQL statement and recursively processing SELECT clause, FROM clause and WHERE clause. Different from the method proposed in 0, we obtain the RQG through not only WHERE clause, but also SELECT and FROM clause. This is because that rich information for subquery is hidden in SELECT clause and the depth of query should be associated with vertices of RQG during processing FROM clause. A set of vertices associated with a query depth are created when processing FROM clause. When dealing with WHERE and SELECT clause, a set of edges will be created.

Suppose the input SQL statement is the query in Example 4.3:

```
select a1.*
from ACCOUNT a1, CUSTOMER c1, CUSTACCT ca1
where c1.CUSTOMERID=?
 and a1.BALANCE =
```

[5] When the operator is (NOT) EXISTS, the quantifier should not appear.

```
        (select MAX(a2.BALANCE)
         from ACCOUNT a2, CUSTACCT ca2
         where ca2.CUSTOMERID = c1.CUSTOMERID
            and a2.ACCID = ca2.ACCID)
and ca1.ACCID = a1.ACCID
and c1.CUSTOMERID = ca1.CUSTOMERID
```

We start with FROM clause of the main query. For each relation we create a vertex v representing a RTV. Hence vertices a_1[6], ca_1 and c_1 are created. Since the outmost query is of both depth 0, we set depth 0 for a_1, ca_1 and c_1, denoted by $a_1(0)$, $ca_1(0)$ and $c_1(0)$. Next the WHERE clause of the main query is processed, "CUSTOMERID=?" is set as an annotation on c_1. For join predicates "a_1.ACCID=ca_1.ACCID" and "ca_1.CUSTOMERID = c_1.CUSTOMERID", two undirected edges are created to connect a_1 and ca_1, and ca_1 and c_1. Their annotations are "a_1.ACCID=ca_1.ACCID", "ca_1.CUSTOMERID = c_1.CUSTOMERID".

The comparison predicate "a_1.BALANCE =(select MAX(a_2.BALANCE)..." leads to a subquery. Again the FROM clause of the subquery will be processed first, and vertices $a_2(1)$ and $ca_2(1)$ are created. A directed edge is created from a_1 to a_2 with annotation "a_1.BALANCE =(select MAX(a_2.BALANCE)" for the sub SELECT clause.

Finally, for join predicates "c_1.CUSTOMERID = ca_2.CUSTOMERID" and "a_2.ACCID = ca_2.ACCID", we create the undirected edges between c_1 and ca_2 with

[6] To simplify notation we assume that a RTV and a vertex name are interchangeable.

annotation "c_1.CUSTOMERID=ca_2.CUSTOMERID", and a_2 and ca_2 with annotation "a_2.ACCID=ca_2.ACCID", respectively.

The generated RQG is shown in Figure 4-1.

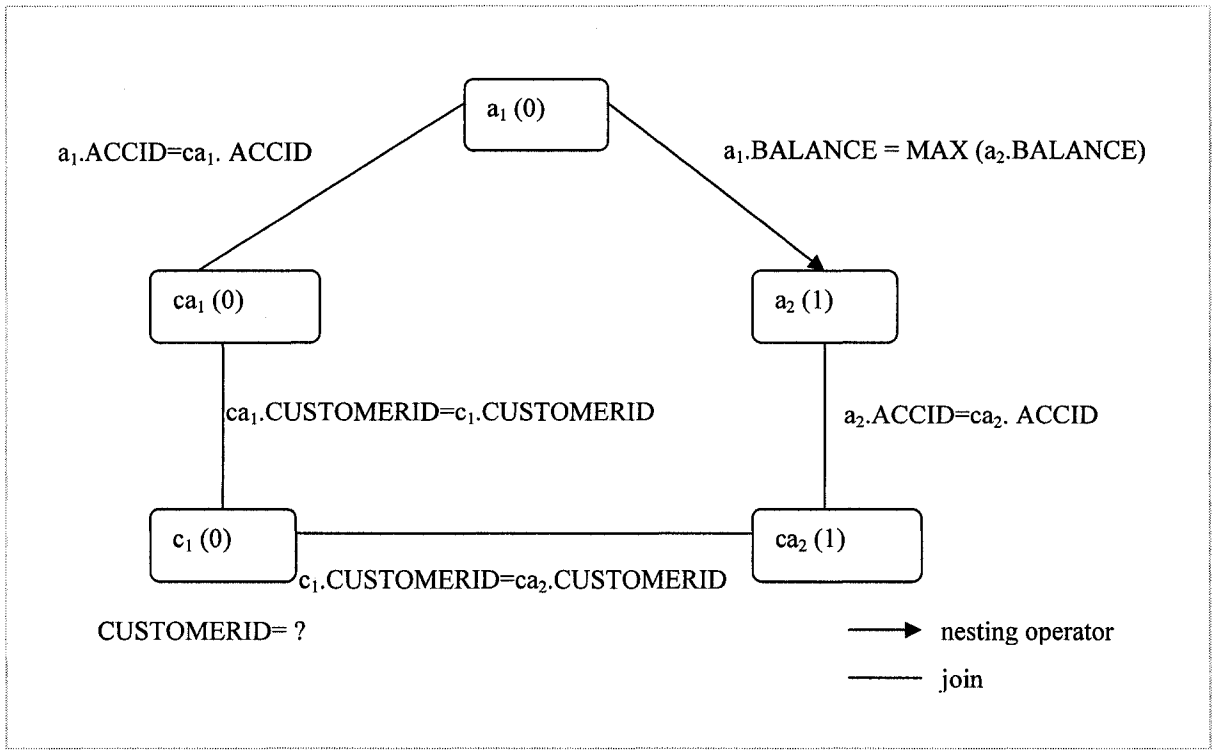


Figure 4-1 The RQG of Example 4.9

4.4 Object Query Graph (OQG)

First, we define OQG.

Definition 2.(OQG) An Object Query Graph OQG(OV,OE1,OE2) consists of a set of vertices OV, a set of undirected edges OE1, and a set of directed edges OE2.

- Each vertex v in OV represents either a Class Instance Variable (CIV) of a class or an "EXISTS" predicate. It is associated with a number indicating the depth

where the CIV is defined in a query or subquery. Each vertex v is annotated with the qualification conditions on the CIV corresponding to v.

- An undirected edge e in OE1 represents an explicit join (a join represented with a join condition). It is annotated with the explicit join condition between the two corresponding CIVs. The vertices residing in two ends can be either in the same or different query depth.
- A directed edge e in OE2 represents one of the followings:
  - a nesting operator which leads to a subquery. The arrow starts from outer query to inner query.
  - an implicit join (a join by a path expression using the object reference), between two CIVs, The vertices residing in two ends can be either in the same or different query levels. When two vertices are in different query levels, the arrow should start from outer query to inner query. The edge e from v_1 to v_2, has the annotation `CIV(`v_1`).`$\mathrm{CMRv_1}$, where CIV(v_1) is the CIV represented by v_1, $\mathrm{CMRv_1}$ is the object reference/CMR field in the class C(v_1) corresponding to v_1, with the domain class C(v_2) corresponding to v_2. The edge e indicates the need of a traversal between $\mathrm{CMRv_1}$ of class C(v_1) and the domain class C(v_2).

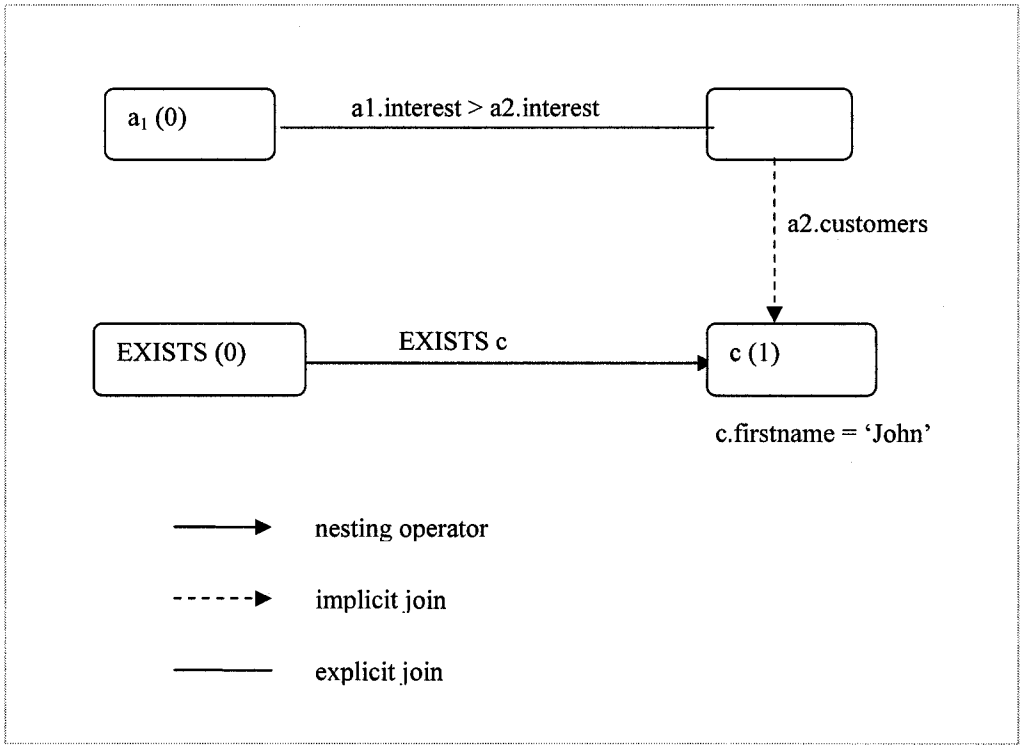


Figure 4-2 The OQG for Example4.4

Example 4.4. Figure 4-2 depicts an OQG for the EJB-QL below. The EJB-QL query selects all the Account beans whose interests are higher than the Account whose customer's first name is "John". Next section will discuss how to generate this kind of OQG from RQG.

EJB-QL=

```
select object(a1)
from Account a1, Account a2
where a1.interest > a2.interest
      and EXISTS (select object(c)
                  from IN (a2.customers) c
                  where c.firstname = 'John')
```

4.5 Translating RQG to OQG

The algorithm of transforming RQG to OQG is described as follows. Starting from this section we will use the following notational conventions:

Given a vertex v_i,

- RTV(v_i) denotes the RTV corresponding to vertex vi.
- CIV(v_i) denote the CIV corresponding to vertex vi.
- R(v_i) denotes the relation corresponding to the RTV(vi).
- C(v_i) denotes the class corresponding to the CIV(vi).

Given a relation or an attribute x,

- M(x) represents the corresponding class or attribute in a class, where M is the mapping between the relational and object schemas.

Input: A relational query graph RQG(RV, RE1, RE2), Relational Schema, Object Schema, R/O Schema mapping M.

Output: An object query graph OQG(OV,OE1,OE2) .

Algorithm:

Translate vertices in RV;

Translate the edges in OE1;

Translate the edges in OE2.

The following subsections describe these three steps.

4.5.1 Translating the vertices in RV.

For each vertex v_i, if v_i does not correspond to an association table, copy v_i to OV. Otherwise, v_i will be ignored.

Figure 4-3 shows an intermediate OQG generated from step 1 of the query graph translation algorithm. In the RQG of Figure 4-1, the vertices a_1, a_2 and c_2 are copied into OQG since they correspond to ACCOUNT and CUSTOMER relations,

which are not associate tables. Vertices ca_1 and ca_2 are ignored since they correspond to the association table CUSTACCT.

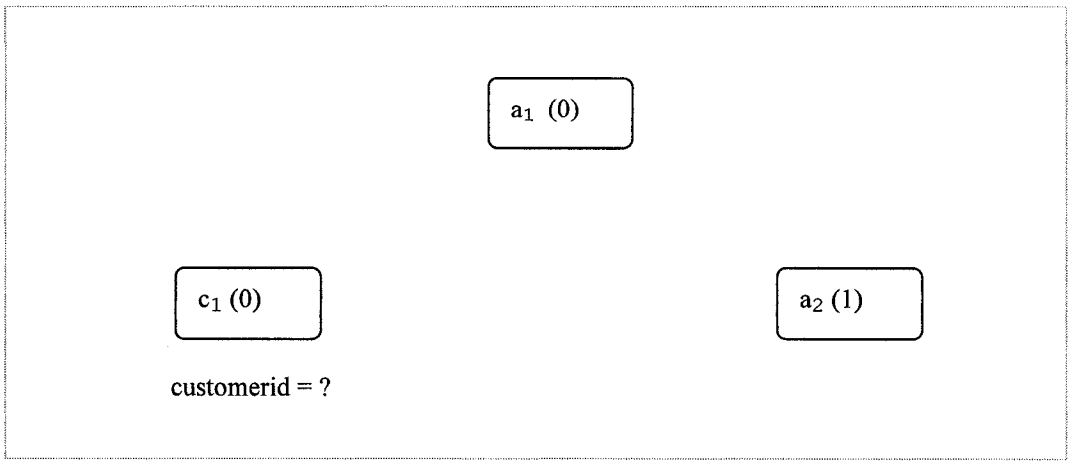


Figure 4-3 The intermediate graph after step 1 for Example 4.3

4.5.2 Translating the join edges in RE1

For each edge *e* in RE1 between vertices v_1 and v_2, supposing the annotation of *e* is <RTV(v_1).a op RTV(v_2).b>, where a and b are two attributes in relation R(v_1) and R(v_2), there are a few cases for the translation of *e,* depending on what a and b are mapped into the object schema. Three factors need to be considered in the translation:

a) Whether both R(v_1) and R(v_2) are mapped into classes;

b) Whether one of a and b corresponds to CMP or CMR fields;

c) whether RTV(v_1) and RTV(v_2) are in different level of query blocks.

According to these different cases, different translations will be produced as illustrated in the following diagram.

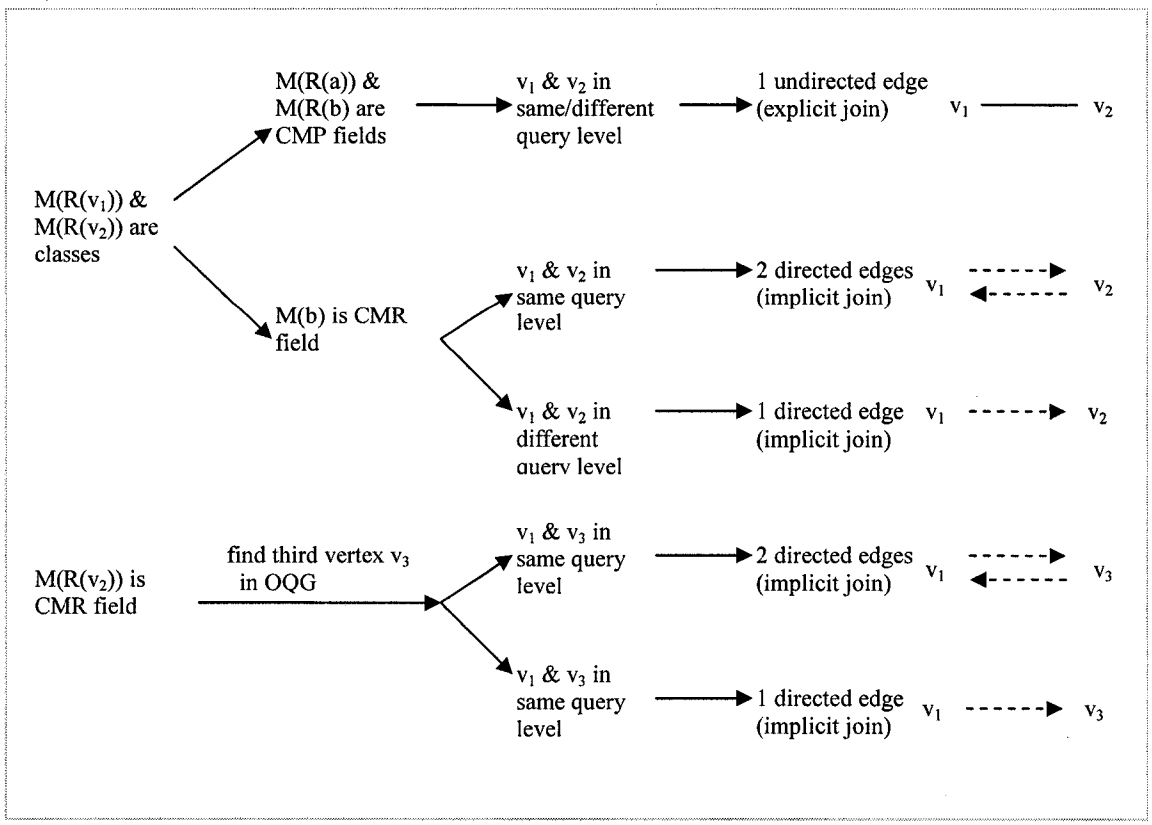


Figure 4-4 The process of translation of join edges in RE1

Case 1. Both M(R(v_1)) and M(R(v_1)) are object classes in the object schema.

Subcase1.1: Both M(a) and M(b) are CMP fields, ie., they are not object references.

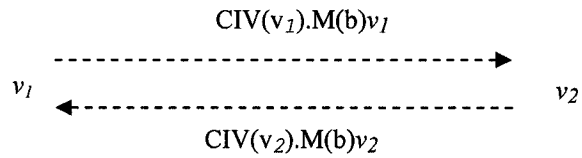
In this case the translated edge and the annotation of the edge are as follows:

$$v_1 \xrightarrow{\text{CIV}(v_1).\text{M(a) op CIV}(v_2).\text{M(b)}} v_2$$

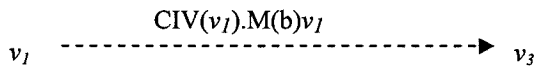
Subcase 1.2.One of M(a) and M(b), say M(a), is CMP field.

In this case, the operator op must be the equal sign, and M(b) should correspond to two CMR fields in classes M(R(v_1)) and M(R(v_2)) respectively. We call these two CMR fields M(b)v_1 and M(b)v_2, which have domain classes M(R(v_2) and M(R(v_1)), respectively.

If RTV(v_1) and RTV(v_2) are defined in the same level of query block, the translated edge will be two directed edges. The edge from v_1 to v_2, with annotation CIV(v_1).M(b)v_1, indicates the need of a traversal from the M(b)v_1 of class M(R(v_1)) corresponding to v_1 to the domain class M(R(v_2)) of M(b)v_1 which corresponds to v_2. The edge from v_2 to v_1, with annotation CIV(v_2).M(b)v_2, indicates the need of a traversal from the M(b)v_2 of class M(R(v_2)) corresponding to v_2 and the domain class M(R(v_1)) of M(b)v_2 which corresponds to v_1.



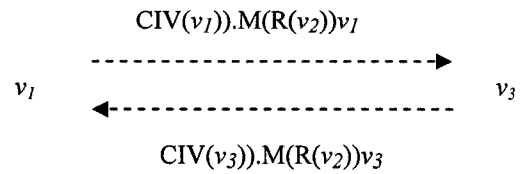
If RTV(v_1) and RTV(v_2) are defined in the different level of query blocks, supposing RTV(v_1) is in outer query block and RTV(v_2) is in inner query block, the translated edge is a directed edge from v_1 to v_2, with annotation CIV(v_1).M(b)v_1.



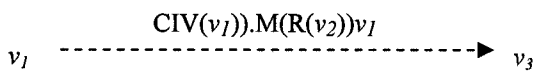
Case 2. One of the vertices, say v_2, corresponds to association table. In this case, v_2 is not copied into OQG, and M(R(v_2)) corresponds to two set-valued CMR fields in two classes represented by other vertices in OQG, say, in v_1 and v_3. Use M(R(v_2))$_{v1}$ and M(R(v_2))v_3 to denote those two CMR fields, where M(R(v_2))v_1 is in class M(R(v_1)) with domain classes M(R(v_3), and M(R(v_2))v_3 is in class M(R(v_3)) with domain class M(R(v_1)), respectively.

For each vertex v in RQG, if it corresponds to class M(R(v_3)) and is adjacent to v_2 in the RQG, remove the edge between v and v_2.

If RTV(v_1) and RTV(v_3) are defined in the same level of query block, create two directed edges in OQG between the v_1 and v_3. The edge from v_1 to v_3, with annotation CIV(v_1).M(R(v_2))$_{v1}$, and the edge from v_3 to v_1 with annotation CIV(v_3).M(R(v_2))v_3.



If RTV(v_1) and RTV(v_3) are defined in the different level of query blocks, such as RTV(v_1) in outer query block and RTV(v_3) in inner query block, create an directed edge from the v_1 to v_3, with annotation CIV(v_1)).M(R(v_2))v_1, where M(R(v_2))v_1 is in class M(R(v_1)).



4.5.3 Translating the edges in RE2

For each edge e in RE2 from vertex v_1 and v_2, where e represents a nesting operator, a directed edge is created in OQG between v_1 and v_2, and the annotation ANNO on e is converted to ANNO' on the translated edge in OE2. Different forms of ANNO will result in different ANNO'. ANNO =

a) RTV(v_1).a op [quantifier]*[Agg]*(RTV(v_2).b). ANNO' will be "CIV(v_1).M(a) op [quantifier]*[Agg]*(CIV(v_2).M(b))", where M(a) and

M(b) are CMP fields mapping to attributes a and b respectively in relational schema.

b) op [Agg]*(RTV(v_2).b). ANNO will be "op [Agg]*(CIV(v_2).M(b))", where M(b) is CMP field mapping to attribute b in relational schema.

c) RTV(v_1).a op [quantifier]* RTV(v_2).b. Assume b is a foreign key. ANNO will be [quantifier]*CIV(v_1).M(b)v_1, where M(b)v_1 is the CMR field in class M(R(v_1)) which has domain class M(R(v_2)).

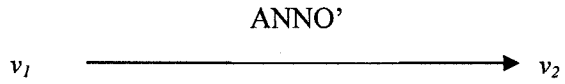


Figure 4-5 shows the OQG of the studied example after step 2 & 3 of the query graph translation. In the RQG of Figure 4-1, when translating edge e_{alcal} between a_1[7] and ca_1, ca_1 corresponds to the association table `CUSTACCT` that was transformed to two CMR fields `accounts` and `customers` in object schema. `accounts` is in class `Customer`, and `customers` is in class `Account`. We can find that vertex c_1 corresponds to class `Customer` and is adjacent to ca_1 in the RQG. The edge between ca_1 and c_1 is set as "visited". Since a_1 and c_1 are defined in the same query block, two directed implicit join edges e_{alcl} and e_{clal} in OQG between a_1 and c_1 is created, with annotations a_1.`customers` and c_1.`accounts`. For the edge e_{alca2}, ca_2 corresponds to the association table `CUSTACCT`, we find that a_2 corresponds to class `Account` and is adjacent

[7] To simplify notation we assume that a CIV and a vertex name are interchangeable

to ca_2 in the RQG, remove the edge between ca_2 and a_2. Since a_2 and c_1 are defined in the different query block, one directional implicit join edge e_{c1a2} in OQG between c_1 and a_2 is created, with annotation c_1.accounts, edge e_{a1a2} represents a nesting operator, with the annotation a_1.BALANCE =(select MAX(a_2.BALANCE) which is in the form of R_1.a op [quantifier]*[Agg]*(R_2.b). Creating an edge e_{a1a2} from a_1 and a_2 in OQG, its annotation will be "a_1.balance = (select MAX(a_2.balance), where balance is CMP field mapping to attributes BALANCE in relational schema.

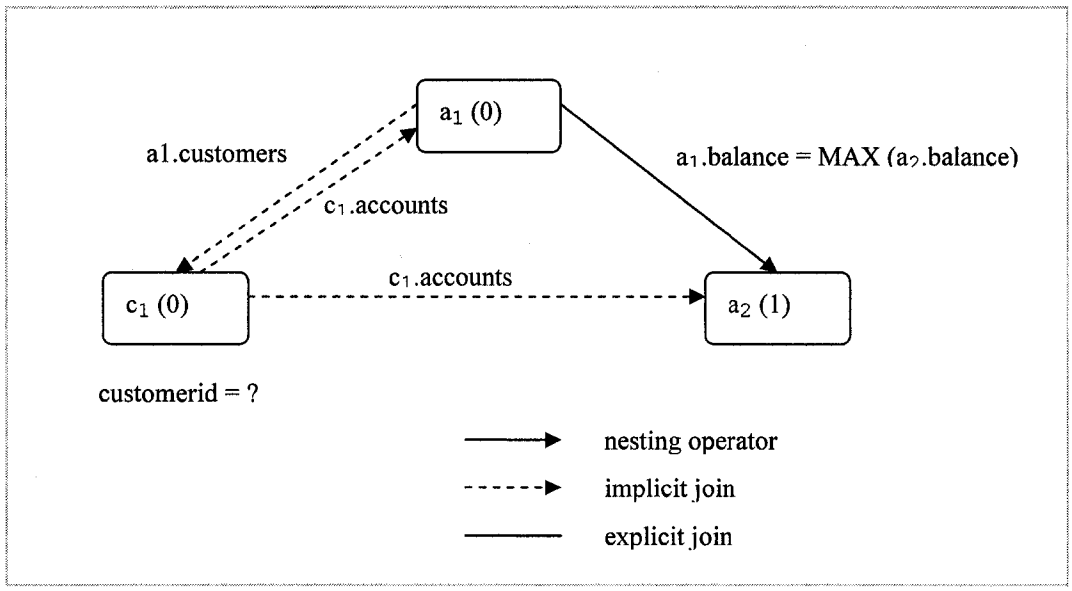


Figure 4-5 The translated Object Query Graph of Example 4.3

4.6 Generate EJB-QL query from OQG

Now we will generate the EJB-QL query from the OQG. The simple idea is to construct EJB-QL query gradually by traversing the OQG from a starting vertex v_r

selected by a user. This starting vertex should correspond to one of the CIVs defined in the outmost query. The path expressions are constructed for different types of edges. The results of traversing an edge *e* of OQG depend on the following factors:

1) The type of the object reference *cmr* represented by edge *e*. *cmr* which can be either single- or set- valued.
2) The type of the traversed edge *e*.

4.6.1 Obtaining the FROM clause

The FROM clause of EJB-QL contains the identification variables which include range variables and collection member identification variables (IDVs). The initial starting vertex is first processed and a range variable is defined in the FROM clause. While traversing the OQG, more range variables might be declared if its corresponding vertex cannot be reachable by the navigation from the initial starting vertex; or some collection member identification variables may be added when traversing a set-valued CMR field, which means the FROM clause will contain collection_valued path expression.

4.6.2 Obtaining the SELECT clause

As we have discussed in Chapter 2, in EJB-QL the SELECT clause of the outmost SQL statement needs to be an all-column selection. For R.* in the SELECT clause of main query, where R is a relation, the translated SELECT clause of the main query can be:

a) SELECT OBJECT(o), where "o" is an IDV, it is defined for a vertex in OQG which associates with a CIV(R).
b) SELECT o.r, where "o" is an IDV, "r" is a single-valued CMR field.

As we defined in 2.3, the return type of a subquery in EJB-QL can be a single CMP field, an aggregate function, a single IDV, or a path expression ended with a single-valued CMR field. Table 4-2 illustrates the possible translations of sub-select clause of a SQL statement.

Subselect clause of SQL	Subselect clause of EJB-QL	Remark
SELECT R.A	SELECT o.a	"A" is an attribute of relation "R". "o" is an IDV. It is defined for a vertex in OQG which associates with a CIV(R). "a" is a CMP field corresponding to R.A.
	SELECT o.r.a	"r" is an single-valued CMR field
SELECT AGG(R.A)	SELECT AGG(o.a)	"AGG" is one of aggregate functions (AVG, MAX, MIN, SUM)
	SELECT AGG(o.r.a)	
SELECT R.*	SELECT OBJECT(o)	
	SELECT o.r	

Table 4-2 The possible translations of sub-select clause of Example 4.3

4.6.3 Constructing WHERE clause

We now present an algorithm which constructs the WHERE clause of EJB-QL query from OQG. The difficulty of query translation from SQL queries to EJB-QL queries is the translation of the WHERE clause because the major difference between SQL and EJB-QL is showed in WHERE clause. Actually, it not only constructs the WHERE clause, but also add more contents in FROM clause (i.e. declare IDVs in FROM clause) and generate the subqueries.

The algorithm has three main steps:

- Translating initial starting vertex which is selected by a user. This step is to declare the IDV in FROM clause for the initial starting vertex.
- Eliminate all implicit join edges. This step is to set up path or declare IDV for all vertices in OQG. It will traverse the OQG through all implicit join edges in a depth-first manner, starting from the initial starting vertex. Since our OQG is a 3-dimentional graph, a layer represents a query block, the number of layers indicates the depth of the query. Depth-first is first applied to the implicit join edges connecting the vertex in different layers, then applied to the edges in the same layer. For example, Figure 4-6 is an OQG with three layers. Assume vertex `b` is the initial starting vertex, the order of traversing is: `b->d->g->h->e->f->a->c`.

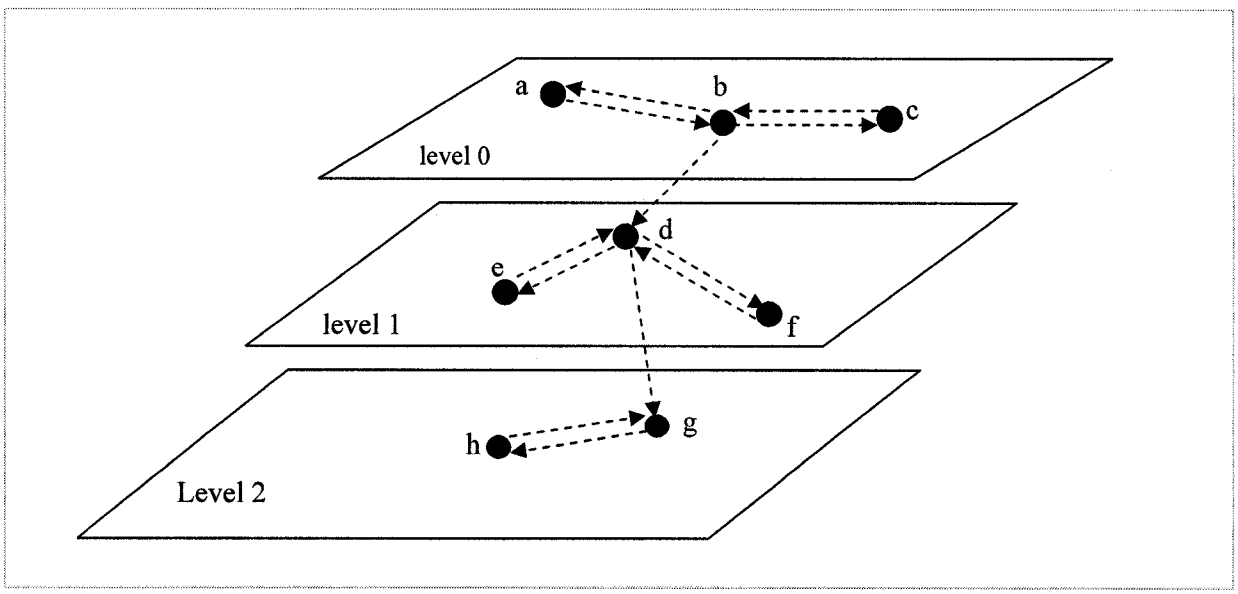


Figure 4-6 Traversing diagram for eliminating implicit join edges

The implicit join edges between two vertices in same level must be in pair, with different directions. When one of implicit join edge is traversed, both of them will

be eliminated.

- Processing vertices in OV. Now only explicit join edges and nesting operator edges are left in OQG. By process each vertex in OV, as well as its adjacent explicit join edges and nesting operator edges, the whole EJB-QL statement will be produced.

Traversing the OQG from a starting vertex v_r, different edges will be passed through. The interpretation of each edge may vary with its type. There are three types of edges in our OQG, we will analyze each type of the edges. In the following, sel-on-V represents the selections on vertex *v*, IDV(*v*) represents the IDV declared for *v*, PATH(*v*) represents the path of *v* navigated from some starting vertex. Except for the initial starting vertex, all vertices should have their own path navigating from some starting vertex. Not every vertex has an IDV, it only declared in certain circumstances. We will explain it in the following part.

Undirected edge. The undirected edge represents an explicit join. Suppose the annotation of the edge is "`CIV(`v_1`).`cmp_1 `op CIV(`v_2`).`cmp_2". The edge is translated to a join predicate of EJB-QL:

`IDV(`v_1`)|PATH(`v_1`).`cmp_1 `op IDV(`v_2`)|PATH(`v_2`).`cmp_2[8]

v_1 —— `CIV(`v_1`).`cmp_1 `op CIV(`v_2`).`cmp_2 —— v_2

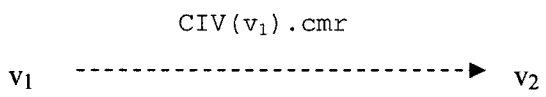
[8] IDV has the higher priority than the path to be chosen to construct a join predicate in WHERE clause.

Directed edge (implicit join). This edge represents an implicit join. The semantics of this edge is that a traversal from class $C(v_1)$ to class $C(v_2)$ is needed. Here the type of reference/CMR field *cmr* does play a role.

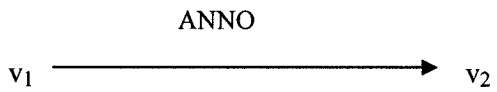
If *cmr* is single-valued, the path is generated for v_2: `IDV(`v_1`) | PATH(`v_1`).cmr`

If *cmr* is multi-valued, we should declare IDV `IDV(`v_2`)` for v_2 with the form

" `IN(IDV(`v_1`) | PATH(`v_1`).cmr) IDV(`v_2`)` " in FROM clause.



Directed edge (nesting operator). This edge represents a nesting operator. The translation result of this kind of edge mainly depends on the annotation of the edge.



a. "CIV(v_1).cmp_1 op [quantifier]*[Agg]*(CIV(v_2).cmp2)". The translation would be "`IDV(`v_1`) | PATH(`v_1`).`cmp_1 `op [quantifier]*[Agg]*(select IDV(`v_2`) | PATH(`v_2`).`cmp_2", the SELECT clause of an subquery is generated.

b. "op [Agg]*(CIV(v_2).cmp_2)". The translation would be "`EXISTS(select IDV(`v_2`) | PATH(`v_2`).`cmp_2", the SELECT clause of an subquery is generated.

c. "[quantifier]*CIV(v1).cmr_{12}". where cmr_{12} is the CMR field in class $C(v_1)$ which has domain class $C(v_2)$. The translation will depend on the nesting operator and the type of cmr_{12} (single- or set- valued).

   If cmr_{12} is set-valued, the path of v_2 is `IN(IDV(`v_1`) | PATH(`v_1`).`cmr_{12}`)`. We now declare an IDV `IDV(`v_2`)` for v_2 with the form of

IN(IDV(v_1) | PATH(v_1).cmr_{12}) IDV(v_2). If the nesting operator is "=", "EXIST", or "IN", 0 shows that there are three different forms for the same query, so the annotation will be translated in EJB-QL with one form and will not generate a subquery. If the nesting operator is "<>, NOT EXIST, or NOT IN", use "NOT EXISTS" as nesting operator to lead a subquery. A partial subquery can also be got: "NOT EXISTS (select object (IDV(v_2)) From IN(IDV(v_1) | PATH(v_1).cmr_{12}) IDV(v_2)".

If cmr_{12} is single-valued, the path for v_2 is IDV(v_1) | PATH(v_1).cmr_{12}. If the nesting operator is "=", "EXIST", or "IN", the annotation will be translated in EJB-QL with one form and has no subquery. Nothing need to be done at this stage. If the nesting operator is "<>, NOT EXIST, or NOT IN", due to the limitation of the EJB-QL syntax, the translation is failed. We will explain it in detail in next section 4.7.2

We now present an algorithm that obtains the EJB-QL from the OQG.

Input: an OQG(OV, OE1, OE2), starting vertex, selected vertex.

Output: an EJB-QL string.

A table is defined to help the translation. Table VERTEX-IDV-PATH, which stores the vertices and corresponding identification variables (IDVs) as well as their paths that will be used in path expression.

The process has the following steps:

1. Process starting vertex and all vertices which has no adjacent implicit join edges. Declare an IDV such as IDV(v_{start}) for starting vertex, no path is needed to set for the starting vertex. put them in the VERTEX-IDV-PATH. Suppose the class

associated with starting vertex is C(v_{start}), declare the IDV for starting vertex in FROM clause in the form of: C(v_{start}) IDV(v_{start}). Apply the same method on those vertices which has no adjacent implicit join edges.

In Example 4.3, assume we chose a_1 in OQG as starting vertex. After step 1, an IDV acc1 is declared for a_1, put them in table VERTEX-IDV-PATH, and put the declaration in FROM clause of the outmost query with the format: Account acc_1. We can get the partial EJB-QL query which is:

EJB-QL: "***from Account acc1***"

VERTEX	IDV	PATH
a_1	acc_1	/

Table 4-3 The table storing starting vertex, IDV and path for Example 4.3

2. Eliminate all the implicit join edges

   We use a stack ON-LINE1 to keep track of the visited vertices. Initially, the stack is empty.

   First we set starting vertex as active vertex. Repeat until active vertex is null.

   a) If there is already an active vertex, then proceed to step b). Otherwise, choose an active vertex in the following order: i) if the ON-LINE1 is not empty, pop out the top vertex from it. ii) get the first vertex from the OV.

   b) Process the active vertex v_a. Get all the adjacent outgoing implicit join edges of active vertex v_a. If v_a has no outgoing implicit join edge, inactivate the active vertex, remove v_a from OV and go back to step a). Otherwise, process them in the following order:

i) The implicit join edge *e* connecting two vertices in different level of query blocks, which starts from v_a to a vertex v_2, where v_2 is in the subquery. Translating *e* with the method described before, we can get the IDV(if have) and path for v_2. Put v_2, its IDV(if have) and path in VERTEX-IDV-PATH. Remove the edge *e*. If v_2 has no outgoing implicit join edge, inactivate the active vertex and go back to step a); otherwise, set v_2 as active vertex, process v_2.

ii) The implicit join edge *e* connecting two vertices in same query block, which starts from v_a to v_2. Remove the implicit join edge which is from v_2 to v_a. Get all adjacent outgoing implicit join edges which are in same query block of v_a, if the number of such edges >1, push the v_a into the stack ON-LINE1 . Find the CMR field from the annotation of *e*, use the method presented in i) to process e.

For Example 4.3, starting vertex a_1 is set as the active vertex. a_1 has one adjacent outgoing implicit join edge e_{a1c1}. Process e_{a1c1}. e_{a1c1} has annotation a_1.customers. The CMR field customers is set-valued. Declare an IDV $cust_1$ for c_1, and its path will be IN(acc_1.customers). Put c_1, $cust_1$ and IN(acc_1.customers) in VERTEX-IDV-PATH. Remove the edge e_{a1c1}. c1 has one outgoing implicit join edge e_{c1a2}, set c_1 as active vertex. Process e_{c1a2}. e_{c1a2} has annotation c_1.accounts. The CMR field accounts is set-valued. Declare an IDV acc_2 for a_2, and its path will be IN($cust_1$.accounts). Put a_2, acc_2 and IN($cust_1$.accounts) in VERTEX-IDV-Path. Remove the edge e_{c1a2}. Since a_2 has

no adjacent outgoing implicit join edge, inactivate the active vertex. To get an active vertex, first we check the stack ON-LINE, it is empty. Then we check OV, the vertices stored in OV is in the order of a_1, c_1 and a_2. Get them from OV and check the adjacent outgoing implicit join edges one by one, none of them has adjacent outgoing implicit join edges, remove each of them. Finally, the OV is empty, and the active vertex is null.

VERTEX	IDV	Path
a_1	acc_1	/
c_1	$cust_1$	IN(acc_1.customers)
a_2	acc_2	IN($cust_1$.accounts)

Table 4-4 The table storing all vertices, corresponding IDVs and paths for Example 4.3

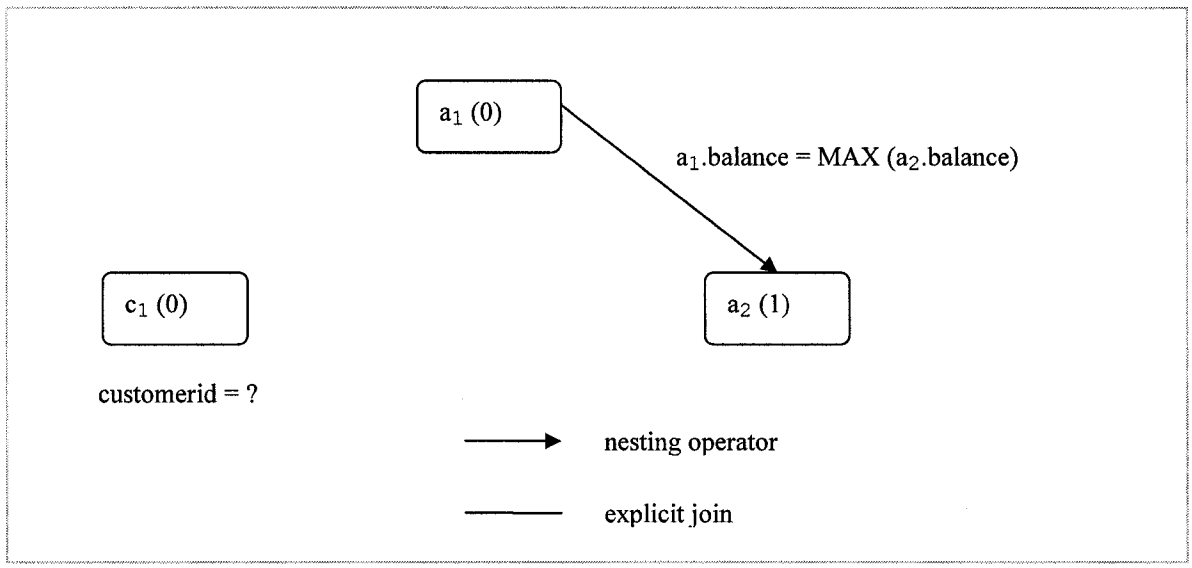


Figure 4-7 The Object Query Graph eliminated implicit join edges of Example 4.3

3. Process vertices in OV.

First we will translate the top level SELECT clause. If the selected vertex v_{sel} is the initial starting vertex, the SELECT clause will be: "`select OBJECT(IDV(v_start))`". Otherwise, assume its IDV is `IDV(v_sel)` the SELECT clause will be: "`select OBJECT(IDV(v_sel))`. If there is no IDV declared for v_{sel}, get its path from VERTEX-IDV-PATH, say, `PATH(v_sel)`. The SELECT clause will be: "`select PATH(v_sel)`".

We define a stack ON-LINE2 to keep track of vertices. It is initially empty.

Repeat until OV is empty.

Algorithm: Translate vertex.

Input: a list of vertices SUBQUEY-VERTICES, active vertex v, OV, ON-LINE2, VERTEX-IDV-PATH

Output: a String, which is part of an EJB-QL query.

0) Choose active vertex. If the active vertex is null, select an active vertex in the following order: i) get the first vertex from SUBQUERY-VERTICES; ii) pop out the top of vertex in ON-LINE2. iii) get the first vertex from OV. For an active vertex v_a, if it has been visited, go to step 5); if it is the initial starting vertex, go to step 2), otherwise process it in the following order.

1) Check v_a in table VERTEX-IDV-PATH, if it has IDV, say, `IDV(v_a)`, get the path of v_a, say, `PATH(v_a)`, declare it in the FROM clause in the form of : `PATH(v_a) IDV(v_a)`.

2) Translate the selections on v_a and put the translated selections in the WHERE clause. The translation for selections on a vertex is very straightforward. Just copy it to WHERE clause directly.

3) Translate all the adjacent undirected edge, put them in the WHERE clause. In this stage, each vertex should have a path defined for it. Some vertices may have IDVs declared for them. Use the method of translating explicit join edges presented before. The edge e is translated to an EJB-QL join predicate: "`IDV(`v_a`)|Path(`v_a`).cmp`$_{va}$ `op IDV(`v_2`)|Path(`v_2`).cmp`$_{2}$".

4) Translate all the nesting operator edges which start from the active vertex v_a. If the number of nesting operator edges starting from v_a >1, push v_a into stack ON-LINE2. Get the other end of vertex v_a for a nesting operator edge, which is v_2. Define a list TEMP-VERTICES, put v_2 in the list, get all vertices which are in the same query block with v_2, put them into TEMP-VERTICES. Translating e using the method for a nesting operator edge described before. If there are path or/and IDV generated for v_2, put them in VERTEX-IDV-PATH.

   Remove the edge e. Set the SUBQUEY-VERTICES as TEMP-VERTICES, active vertex as null.

5) Remove v from OV and SUBQUEY-VERTICES. Go to step 0).

For Example 4.3, assume the selected vertex is a_1, we get the SELECT clause for main query, which is "`select object(`acc_1`)`". Now the EJB-QL query is:

EJB-QL:

```
select object(acc1)
from Account acc1
```

The vertices in OV is $\{a_1, c_1, a_2\}$. Get the first vertex from OV, which is a_1, set it as active vertex. Using the Translate Vertex algorithm. Now SUBQUERY-VERTICES is empty, active vertex is a_1, OV is $\{a_1, c_1, a_2\}$, ON-LINE2 is empty.

Since a_1 is the initial starting vertex, so go to step 2). There is no selection on a1 and no adjacent undirected edge associated with a1. So we will skip the followed two steps and start to translate nesting operator edge e_{a1a2} in step4). The annotation on e_{a1a2} is a_1.balance = a_2.MAX (balance), which is in the form of "CIV(R_1).cmp_1 op [quantifier]*Agg(CIV(R_2).cmp_2)" . Now the EJB-QL query becomes:

EJB-QL:

```
select object(acc1)
from Account acc1
where acc1.balance = select( MAX (acc2.balance)
```

The other end of e_{a1a2} is a_2. a_2 is in level 1. Define a list TEMP-VERTICES, put a_2 in the list, get all vertices which are in same query block with a_2, put them in a list called TEMP-VERTICES, now the TEMP-VERTICES is {a_2}. Remove the edge e_{a1a2}. Set the SUBQUEY-VERTICES as TEMP-VERTICES, active vertex as null; remove a_1 from OV; go through the algorithm Translate Vertex again.

Now SUBQUERY-VERTICES is {a_2}, active vertex is null, OV is {c_1, a_2}, ON-LINE2 is empty. Get the active vertex from SUBQUERY-VERTICES, which is a_2. Process a_2. Since a2 is not initial starting vertex, declare a_2 in subquery in step 1),

EJB-QL:

```
select object(acc1)
from Account acc1
where acc1.balance = (select( MAX (acc2.balance))
                      from IN(cust1.accounts) acc2
```

Skip steps 2), 3), 4) because a_2 does not meet the conditions. In step 5), remove a_2 from OV and SUBQUERY-VERTICES. The SUBQUERY-VERTICES is empty, add the right bracket ")". Set the active vertex null; remove a_2 from OV; go through the algorithm Translate Vertex again.

EJB-QL:

```
select object(acc1)
from Account acc1
where acc1.balance = (select( MAX (acc2.balance))
                from IN(cust1.accounts) acc2)
```

Now SUBQUERY-VERTICES is empty, active vertex is null, OV is $\{c_1\}$, ON-LINE2 is empty. Get the active vertex from OV, which is c_1. Process c_1. Since c_1 is not initial starting vertex, declare c1 in the FROM clause of main query in step 1)

EJB-QL:

```
select object(acc1)
from Account acc1, in(acc1.customers) cust1
where acc1.balance = (select( MAX (acc1.balance))
                from IN(cust1.accounts) acc2)
```

In step 2), translate its selection to $cust_1$.customerid = ?.

EJB-QL:

```
select object(acc1)
from Account acc1, in(acc1.customers) cust1
where acc1.balance = (select( MAX (acc2.balance))
                from IN(cust1.accounts) acc2)
      and cust1.customerd = ?1
```

step 5), c_1 is removed from OV. OV is empty. Translation is done.

Finally, the generated EJB-QL string is:

```
select object(acc1)
from Account acc1, in(acc1.customers) cust1
where acc1.balance = (select( acc2.MAX (balance))
                      from IN(cust1.accounts) acc2)
      and cust1.customerd = ?1
```

4.7 Translation Difficulties

The translation is complicated by the fact that the selection of starting vertex is not unique during the traverse of OQG, the object reference can be either single- or set/multi-valued, the nesting structures have different types, and the syntax of EJB-QL has the restrictions. These factors influence with each other. Different selection of starting vertex may result in navigation to a different implicit join edge, of which the type of object reference/CMR field in the annotation may be different (single- or multi-valued). Because of the restrictions of EJB-QL syntax, sometimes this will cause translation fail. The annotation of a nesting operator edge decided the type of the nesting query. When it is of the form "[quantifier]CIV(v_1).cmr_{12}", the edge will be dealt with the same way as an implicit join edge, same problem will occur.

4.7.1 The selection of starting vertex

As described in 4.6, the simple idea to generate an EJB-QL query is gradually traversing an OQG from a designated starting vertex v_r. The candidates of v_r are all vertices which represent the CIVs corresponding to the RTVs defined in the outmost SQL query. Different starting vertex will result in different navigation, which may

generate different EJB-QL query or even can not produce an EJB-QL. In the following, we will explain it using examples.

As mentioned in 2.2.4 and 2.2.2, EJB-QL uses navigation to express the "inner join" operation in SQL statement. For SQL statement

```
Select R.*
From R, S
WHERE R.PK = S.FK
```

Suppose there is a 1:m relationship between R and S. Let ejbR and ejbS represent the abstract schema names of entity bean mapped from relation R and S, respectively. cmpPK is a CMP field of ejbR mapped from attribute PK. cmrS is a multi-valued CMR field of ejbR referring to ejbS; cmrR is a single-valued CMR field of ejbS referring to ejbR. For the join condition "R.PK = S.FK" in the SQL statement, there should be a path expression which indicates a navigation over a CMR field in the translated EJB-QL query.

If we chose ejbR as the start of traverse, first we declare an identification variable ejbr using the range variable declaration[9] in FROM clause:

```
from ejbR ejbr
```

According to the EJB-QL specification, an identification variable always designates a reference to a single value. For the navigation from ejbR to ejbS over the multi-valued CMR field cmrS, an identification variable must be declared in FROM clause by a collection_member_declaration in the form of:

[9] A range variable is declared using the abstract schema name of an entity bean.

```
IN (ejbr.cmrS) ejbs
```

The path expression `ejbr.cmrS` represents the join operation in the original SQL statement.

If we chose `ejbS` as the start of traverse, the identification variable `ejbs` is first declared in FROM clause:

```
from ejbS ejbs
```

For the navigation from `ejbS` to `ejbR` over the single-valued CMR field `cmrR`, if there is no condition of selection predicates applied on the `ejbR`, there would be no cause to trigger a path expression such as `"ejbs.cmrR"`. The join condition "`R.PK = S.FK`" in SQL cannot be expressed in corresponding EJB-QL. The translation can not be proceeded.

Example 4.5. For a SQL statement

```
select t.*
from TRANSRECORD t, ACCOUNT a
where a.ACCID = t.ACCID
```

`ACCOUNT.ACCID` is the primary key of the table `ACCOUNT`, and TRANSRECORD.ACCID is the foreign key of table `TRANSRECORD`. The related beans are `Account` and `Transrecord`. There is a 1:m relationship between `Account` and `Transrecord`. The CMR field in `Account` is multi-valued with name "`transrecords`" and the CMR field in `Transrecord` is single-valued with name "`theAccount`".

We have two options to select a starting vertex.

a) Using Account bean as starting vertex, the translated EJB-QL is:

```
select OBJECT(t)
from Account a, IN (a.transrecords) t
```

Navigation to transrecords results in a collection. To handle such navigation, in FROM clause, using a collection valued path expression "IN (a.transrecords) t ", the identification variable t is declared to range over the elements of the transrecords collection. "a.transrecords" represents the join operation in the original SQL statement.

b) Using Transrecord bean as root. Since the CMR field "theAccount" in Transrecord referring to Account bean is single-valued. The Account bean, which is reachable by navigation from the root bean Transrecord on a single-valued CMR field, is not necessarily declared in FROM clause. There is no place to use a path expression such as "t.theAccount" representing the join operation a.ACCID = t.ACCID in the original SQL statement. The translation will not be preceded.

4.7.2 Different types of nesting queries

The annotation of a nesting operator edge determines the type of the nesting query. When it is of the form "[quantifier]CIV(v1).cmr_{12}", the edge will be dealt with the same way as an implicit join edge.

For Example 4.1 and 4.2 illustrate two queries

```
Q1 = select R.* from R where R.A IN (select S.B from S)
Q2 = select R.* from R where EXISTS (select S.*
                  from S where R.A = S.B)
```

When A and B are not foreign key relationship, the translation is quite straight-forward. Let ejbR and ejbS denote the abstract schema names of entity bean mapped from relation R and S, respectively. cmpA and cmpBare the CMP field name mapped from attribute A and B. The translated EJB-QL queries will be:

```
EJB-QL1 = select object(r) from ejbR r where r.cmpA IN (select
s.cmpB FROM ejbS s)

EJB-QL2 = select object(r) from ejbR r where EXISTS
              (select object(s)
              from ejbS where r.cmpA = s.cmpB)
```

When there is a foreign key relationship between A and B, the translation will be similar to the method proposed by Mostefaoui in [MK98]. Mostefaoui shows that IN and EXISTS conditions can be represented by semijoins, whereas NOT IN and NOT EXISTS conditions correspond to a special kind of antijoins. In the thesis, the "=ANY" is also regarded as a semijoin, and "<>all" corresponds to anitijoin. When translating such SQL statement with semijoin, there is no need for nesting in the translated EJB-QL query. In the translation of nested query with antijoin, we should keep the nesting structure in translated EJB-QL. An m:1 relationship between the entity bean of outer query and the entity bean located in its subquery in an antijion will cause problem.

Example 4.4.

```
select a.*
from ACCOUNT a
where a.BALANCE >100 and a.ACCID IN(select t.ACCID
                                    from TRANSRECORD t
                                    where t.TRANSAMT > 20)
```

Since there is a 1:m relationship between the `Account` bean and `Transrecord` bean, we need to declare a collection member identification variable in FROM clause with a collection valued path expression "`IN (a.transrecords) t `". "`a.transrecords`" represents the join operation in the original SQL statement. The translated EJB-QL is:

```
select object (a1)
from Account a1, IN(a1.transrecords) t
where a1.balance >100 and t.transamt>20)
```

When there is a m:1 relationship between the root corresponding entity bean and the entity bean corresponded by a vertex located in its subquery, the translation is relatively simple.

Example 4.5. Consider a query Q =

```
select t.*
from TRANSRECORD t
where t.TRANSAMT > 20 and t.ACCID IN (select a.ACCID
                  from ACCOUNT a
                  where a.BALANCE >100)
```

Since there is a m:1 relationship between the `Transrecord` bean and `Account` bean, we don't need to declare an identification variable in FROM clause. "`t.theAccount`" represents the join operation in the original SQL statement. The translated EJB-QL is:

```
select object (t)
from Transrecord t
where t.transamt>20 and t.theAccount.balance>100
```

The translation of nested query with an antijion is more complicated. Here we also translate SQL subqueries with different nesting predicates in the EJB-QL with one

formulation, which uses NOT EXISTS as nesting operator to lead a subquery. [VA95] used the same way to handle NOT IN and NOT EXISTS, in which all subqueies led by NOT IN or NOT EXISTS were translated to subquery led by EXISTS, and NOT IN to NOT EXISTS.

Example 4.6 Consider two SQL queries Q1 and Q2.

Q1 =

```
select t.*
from ACCOUNT a
where a.BALANCE >100 and a.ACCID NOT IN (select t.ACCID
                  from TRANSRECORD t
                  where t.TRANSAMT > 20)
```

Q2 =

```
select t.*
from TRANSRECORD t
where t.TRANSAMT > 20 and t.ACCID NOT IN (select a.ACCID
                  from ACCOUNT a
                  where a.BALANCE >100)
```

EJB-QL1 is the translated query to Q1:

```
select object(a)
from ACCOUNT a
where a.balance >100 and NOT EXISTS (select object(t)
                  from IN(a.transrecords) t
                  where t.TRANSAMT > 20)
```

Since the relationship between the entity bean `Transrecord` and `Account` is m:1, we cannot declare an identification variable for `Transrecord` because it is reachable by the navigation from `Account` on single-valued CMR field `theAccount`, Q2 can't be translated. This is a limitation of the EJB-QL syntax. For an OQL discussed in [MK98], the NOT EXIST clause is a bloc in WHERE clause.

Assume an antijoin edge e from class instance variable C_1 to C_2 and a reference r between C_1 and C_2, the NOT EXISTS clause will be in the format of "NOT EXISTS *var* in C_1.*r* :<*list_of_conditions*>", where *var* is a C_2 variable and <*list_of_conditions*> is the selections on the variable *var*. Here the declaration of *var* does not take into account the type of the reference *r* (single- or multi-valued). So for the query Q2, if it's translated into OQL, the NOT EXISTS clause would be "NOT EXISTS *a* in t.theAccount: a.balance>100".

However, the semantics of Q2 is meaningless. Because the foreign key (ACCID) in table TRANSRECORD should be a subset of primary key (ACCID) in table ACCONT, the subquery is always false. The result of the query is therefore always an empty set. In our translation, we assume that all SQL statements are semantically meaningful.

Chapter 5

System Implementation

This chapter covers the implementation of query translation discussed in the last chapter. First, we will explain the overall design, and then we will show the design details of the main components. The query translator is implemented in programming language Java. The integrated development environment (IDE) used in developing the translator is Eclipse 3.0.

5.1 Overall Architecture

The translation system consists of three subsystems: Mapping File Generator, SQL Extractor and SQL2EJBQL Translator.

The *Mapping File Generator* takes database schema file and some extension files provided by vendor as input. These extension files describe the CMP beans and their relationships, as well as the mappings between database schema and CMP beans. The output is `ejb-jar.xml` and `cmp-mapping.xml` files. The `ejb-jar.xml` file describes the CMP beans and their relationships, the `cmp-mappings.xml` file describes the mapping between the database schema and CMP beans. This tool focuses on using extension files generated by WebSphere Application Developer (WSAD) with DB2 as backend.

The *SQL Extractor* takes as input the EJB code which contains partial queries defined in EJB1.X finder methods. The format of queries in finder methods of EJB1.X differs

from vendor to vendor, and in many cases theses queries are not complete. In IBM WebSphere Application Server, some finder methods only specifies the WHERE clause of a SQL query, the SELECT and FROM clauses are omitted. The query extraction step analyzes the EJB code and extracts the complete and standard SQL statements. This part of the tool will differ for different EJB servers. We focus on WebSphere Application Server.

.

The *SQL2EJBQL Translator* takes `ejb-jar.xml`, `cmp-mapping.xml` and SQL queries as input, and produces EJB-QL queries.

The overall architecture of the system and relationship between the three components are shown in Figure 5-1.

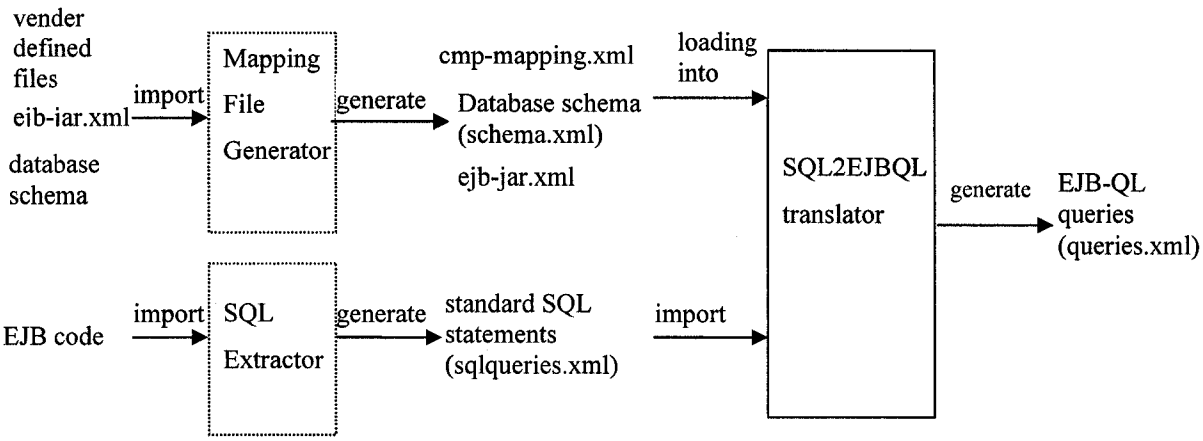


Figure 5-1 The overall architecture of SQL2EJBQL Translation System

5.2 Assumptions

Although most of the queries in finders in CMP entity beans are not complex, the automated translation of them is not a trivial one-to-one mapping, for several reasons.

- We are assuming the enterprise beans remain the same in the new and old systems. That is, although the EJB version is different, the number of beans, their logic definitions, and their relationships are not changed.

- We are assuming the mapping between database schema and EJB architecture are 1-1, i.e., map each table to a single entity bean, each column to a CMP field and relationships to CMR field.

- The database schemas in the old and new systems are not changed. In real situation, the schema will be modified more or less when a product is updated. In this case the SQL queries themselves need to be translated first.

5.3 Features

In addition to the translation of simple SPJ queries, our translator can also deal with subqueries, LIKE predicate, IN predicate, BETWEEN predicate, NULL predicate, and EXISTS predicate, etc., as well as aggregate functions.

5.4 Additional tools needed

The following additional tools are also used in our translation system:

- javacup (parser generator)
- sql4j (sql parser)
- xml4j.jar (this has been included in J2SDK1.4)
- log4j-1.2.8.jar
- gnu-regexp-1.1.4.jar

5.5 User Interface

Figure 5-2 shows an Eclipse plug-in that was made for SQL2EJBQL Translator in Eclipse3.0.

User can import vender defined files which define the CMP beans and their relationships, as well as the mappings between database schema and CMP beans, the database schema file.

There is a SQL editor which can highlight the keywords "SELECT, FROM and WHWER" is ready for user to open a sql query (.sql) from the file system, or typed in by user in GUI. The translated EJB-QL query is displayed in the "EJB-QL query" view. The user can select a CMP bean name as navigated root for EJB-QL query. For a certain SQL query, different EJB-QL query can be generated by choosing various CMP bean name. The following diagram explains how the translator works.

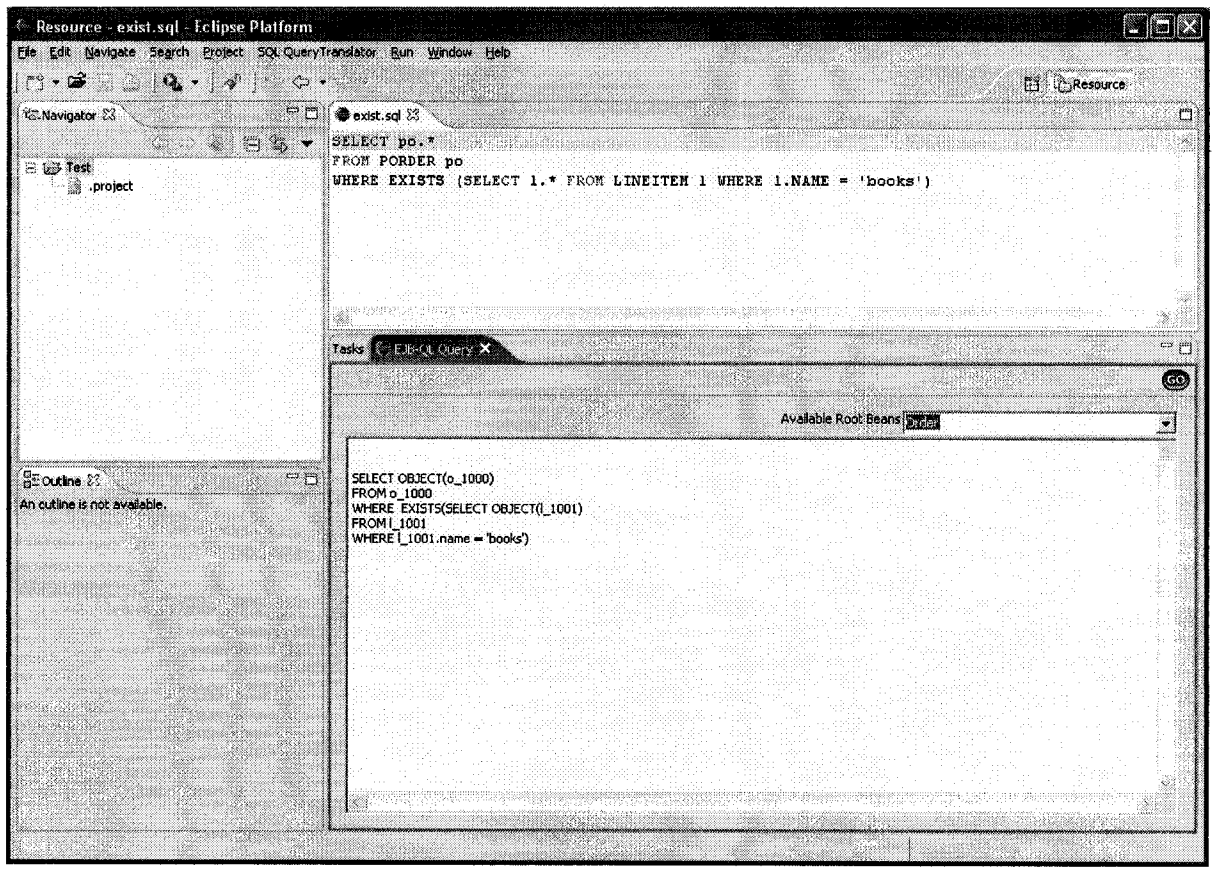


Figure 5-2 The GUI of SQL2EJBQL Translation system

Chapter 6

Experiment and Evaluation

This chapter examines the query translation implementation with testing data from actual applications. The testing focuses on the verification of the *correctness* and *completeness* of our translation. The experimental result is analyzed to evaluate the implementation of query translation.

6.1 Experimental System

Two experiments have been made in this thesis.

In section 4.1, we have made the comparison between our approach and the other two translation algorithms [MYK93][MK98] from the theoretical point of view (see Table 4-1). Both existing algorithm and our algorithm cover the translation of unnested queries and nested queries with key-based join. While queries with the following characteristics can not be translated with the existing algorithm.

- Nested queries with non-key-based join.
- Uncorrelated nested queries.
- Queries with aggregation functions.

In the first experiment, we will investigate the translations for all above kinds of queries. We will create a database system using the schema described in section 2.2.2 (Table 2-1) and the corresponding enterprise application with abstract schema (object

schema) in Table 2-2 , set up mappings between the database tables and CMP entity beans. Testing cases have been created based on the different types of queries.

The second experiment uses the data from a business application obtained from IBM. The purpose of this experiment is to test our algorithm in an enterprise application. In future, our translator is supposed to help the upgrading of EJB applications. The testing result is a reflection of completeness of our translation algorithm.

6.2 Evaluation Method

The main objectives of evaluating our algorithm are to determine how correct and complete the translation is. As we know, the application server will translate the EJB-QL queries to SQL statements when the EJBs are deployed. The application server here acts as an EJB-QL to SQL translator. To demonstrate the correctness and completeness of our algorithm, in the first experiment, testing was executed under the procedure illustrated in Figure 6-1. We construct SQL queries which cover the characteristics mentioned in previous section, the SQL2EJBQL translator takes these SQL statements as input, the output will be the translated EJB-QL queries. Again, we put these EJB-QL queries into the finder methods of CMP entity beans, and deploy these beans. Finally, we compare the regenerated SQL statements with the original ones, the correctness of the translation is verified. In our experiment, IBM WebSphere Application Developer is used.

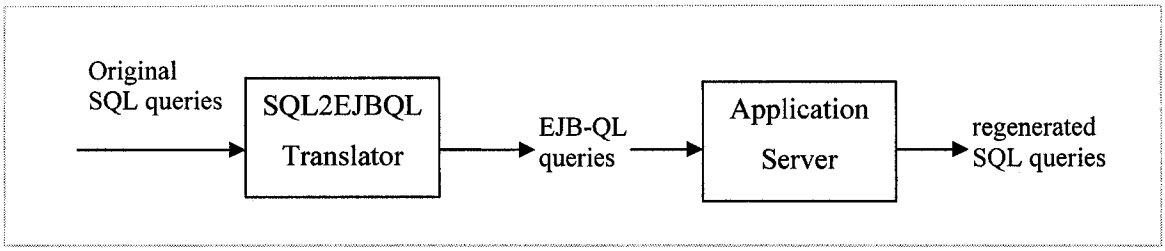


Figure 6-1 The diagram for testing procedure

In second experiment, we took a large amount of SQL statements in a real enterprise application as translation target. By analyzing the testing result, reasons for failed translation have been investigated. The result tells us the possibility of making use of our tool in an actual industry product.

6.3 Experiment 1 – BANKING database system

6.3.1 Testing procedure

This testing has four steps.

Step1: Create the following database BANKSYS using IBM DB2 7.1.

```
ACCOUNT (ACCID(PK), BALANCE, INTEREST, ACCTYPE);
CUSTOMER(CUSTOMERID(PK),TITLE,
FIRSTNAME,LASTNAME,USERID,PASSWORD,ADDRESS);
TRANSRECORD (TRANSID (PK),TRANSTYPE,TRANSAMT,ACCID(FK REFERENCES
ACCOUNT (ACCID));
CUSTACCT (CUSTOMERID(PK,FK),ACCID(PK,FK)).(CUSTOMERID) REFERENCES
CUSTOMER (CUSTOMERID), (ACCID) REFERENCES ACCOUNT (ACCID).
```

Relationships between tables:

```
ACCOUNT (many)<->(many) CUSTOMER
ACCOUNT (one) <-> (many) TRANSRECORD
```

Step 2: Create new entity beans and their relationships in WebSphere Application Developer (WSAD) 5.1, using bottom-up mapping. The mapping result is described in mapping file. We notice that bottom-up mapping does not recognize m:m relationships. It creates an intermediate entity bean and two 1:m relationships. For example: the relationship between `Customer` and `Account` is not recognized as an m:m relationship, despite the fact that the `CUSTACCT` table has only foreign keys and no other attributes. The mapping generates four entity beans: `Account`, `Customer`, `Transrecord`, `Custacct`.

Using WSAD to remove `Custacct` bean and define an m:m relationship between `Customer` and `Account` with cmr fileds `accounts` (in `Customer` bean, references Account bean) and `customers` (in `Account` bean, references `Customer` bean).

Relationships between beans:

```
Account (many)<-> (many) Customer
Account (one) <-> (many) TransRecord
```

The mappings between tables and beans are described in the Table 6-1.

Bean Name	Abstract Schema Name	Attribute		Table Name	Column
		CMP	CMR		
Account	Account	balance		ACCOUNT	BALANCE
		interest			INTEREST
		accid			ACCID
		acctype			ACCTYPE
			customers		
			transrecords		
				CUSTACCT	ACCID
					CUSTOMERID

Customer	Customer	customerid		CUSTOMER	CUSTOMERID
		title			TITLE
		firstname			FIRSTNAME
		lastname			LASTNAME
		userid			USERID
		password			PASSWORD
		address			ADDRESS
			accounts		
Transrecord	Transrecord	transid		TRANSRECORD	TRANSID
		transtype			TRANSTYPE
		transamt			TRANSAMT
			theAccount		ACCID

Table 6-1 Mappings between tables and entity beans

Step 3: Translate SQL into EJB-QL.

Input:

- SQL statement. With format (*.sql). We name it "original SQL".
- schema.xml. Database schema file.
- ejb-jar.xml. It describes the CMP beans and theirs relationships.
- cmp-mappings.xml file. It describes the mapping between the tables of database and CMP beans.

Output: EJB-QL query.

Tool: SQL2EJBQL Translator.

2) Verify the translated EJB-QL query.

Input: EJB-QL query generated in step 3.

Output: SQL statement. It is named as "regenerated SQL".

Tool: WSAD 5.1.

6.3.2 Testing Result

The mapping between SQL queries and EJB-QL queries is not a one-to-one relationship. Instead, a SQL statement can correspond to several EJB-QL queries, depending on which bean will be the start of the navigation, we call the start bean "root" in this experiment.

Note that in the test some SQL statement may not be meaningful. This is because that we need to test different syntactic structures within one simple schema.

6.3.2.1 Testing simple queries in one table

1) Original SQL:

```
select  q1.* from ACCOUNT q1
```

EJB-QL:

```
SELECT OBJECT(a_1000)FROM Account a_1000
```

Regenerated SQL:

```
select  q1.\"ACCID\",  q1.\"BALANCE\",  q1.\"INTEREST\",
q1.\"ACCTYPE\" from GAOYANG.ACCOUNT q1
```

2) Original SQL:

```
select  q1.* from ACCOUNT q1 where  q1.BALANCE > ?
```

EJB-QL:

```
SELECT OBJECT(a_1000)
FROM Account a_1000
WHERE a_1000.balance > ?1
```

Regenerated SQL:

```
"select  q1.\"ACCID\",  q1.\"BALANCE\",  q1.\"INTEREST\",
q1.\"ACCTYPE\" from GAOYANG.ACCOUNT q1 where  ( q1.\"BALANCE\"
> ?)");
```

6.3.2.2 Testing one-to-many relationship

1) Original SQL:

```
select q1.*
from ACCOUNT q1, TRANSRECORD  q2
where q1.ACCID =  q2.ACCID
```

EJB-QL:

```
Using Transrecord bean as root:
SELECT t_1000.theAccount
```

```
FROM Transrecord t_1000
```

Regenerated SQL:

```
select  q2.\"ACCID\",  q2.\"BALANCE\",  q2.\"INTEREST\",
q2.\"ACCTYPE\" from GAOYANG.TRANSRECORD q1 left outer join
GAOYANG.ACCOUNT q2 on  ( q2.\"ACCID\" =  q1.\"ACCID\")");
```

```
Using Account bean as root:
SELECT OBJECT(a_1000)
FROM Account a_1000, IN (a_1000.transrecords) t_1002
```

Regenerated SQL:

```
select  q1.\"ACCID\",  q1.\"BALANCE\",  q1.\"INTEREST\",
q1.\"ACCTYPE\" from GAOYANG.ACCOUNT q1, GAOYANG.TRANSRECORD q2
where  ( q2.\"ACCID\" =  q1.\"ACCID\")
```

2) Original SQL:

```
select q1.*
from ACCOUNT q1, TRANSRECORD  q2
where q1.ACCID =  q2.ACCID AND TRANSTYPE = ?
```

EJB-QL:

```
Using Transrecord bean as root:
SELECT t_1000.theAccount
FROM Transrecord t_1000
WHERE t_1000.transtype = ?1
```

Regenerated SQL:

```
select  q2.\"ACCID\",  q2.\"BALANCE\",  q2.\"INTEREST\",
q2.\"ACCTYPE\" from GAOYANG.TRANSRECORD q1 left outer join
GAOYANG.ACCOUNT q2 on  ( q2.\"ACCID\" =  q1.\"ACCID\")  where  (
q1.\"TRANSTYPE\" = ?)
```

```
Using Account bean as root:
SELECT OBJECT(a_1000)
FROM Account a_1000,  IN (a_1000.transrecords) t_1002
WHERE t_1002.transtype = ?1
```

Regenerated SQL:

```
select  q1.\"ACCID\",  q1.\"BALANCE\",  q1.\"INTEREST\",
q1.\"ACCTYPE\" from GAOYANG.ACCOUNT q1, GAOYANG.TRANSRECORD q2
where  ( q2.\"TRANSTYPE\" = ?)  and  ( q2.\"ACCID\" =
q1.\"ACCID\")
```

6.3.2.3 Testing many-to-many relationship

1) Original SQL:

```
select  q2.*
from CUSTOMER q1, ACCOUNT q2, CUSTACCT q3
where  q3.CUSTOMERID =  q1.CUSTOMERID and  q2.ACCID = q3.ACCID
```

EJB-QL:

```
Using Customer bean as root:
SELECT OBJECT(a_1002)
FROM Customer c_1000, IN (c_1000.accounts) a_1002
```

Regenerated SQL:

```
select  q2.\"ACCID\",  q2.\"BALANCE\",  q2.\"INTEREST\",
q2.\"ACCTYPE\" from GAOYANG.CUSTOMER q1, GAOYANG.ACCOUNT q2,
GAOYANG.CUSTACCT q3 where  ( q3.\"CUSTOMERID\" =
q1.\"CUSTOMERID\")  and  ( q2.\"ACCID\" =  q3.\"ACCID\")
```

```
Using Account bean as root:
SELECT OBJECT(a_1000)
FROM Account a_1000, IN (a_1000.customers) c_1002
```

Regenerated SQL:

```
select  q1.\"ACCID\",  q1.\"BALANCE\",  q1.\"INTEREST\",
q1.\"ACCTYPE\" from GAOYANG.ACCOUNT q1, GAOYANG.CUSTOMER q2,
GAOYANG.CUSTACCT q3 where  ( q3.\"ACCID\" =  q1.\"ACCID\")  and
( q2.\"CUSTOMERID\" =  q3.\"CUSTOMERID\")
```

2) Original SQL:

```
select  q1.*
from ACCOUNT q1, CUSTOMER q2, CUSTACCT q3
where  q2.LASTNAME = ? and  q3.ACCID =  q1.ACCID and
q2.CUSTOMERID =  q3.CUSTOMERID
```

EJB-QL:

```
Using Customer bean as root:
SELECT OBJECT(a_1002)
FROM Customer c_1000,  IN (c_1000.accounts) a_1002
WHERE c_1000.lastname = ?1
```

Regenerated SQL:

```
select  q2.\"ACCID\",  q2.\"BALANCE\",  q2.\"INTEREST\",
q2.\"ACCTYPE\" from GAOYANG.CUSTOMER q1, GAOYANG.ACCOUNT q2,
GAOYANG.CUSTACCT q3 where  ( q1.\"LASTNAME\" = ?)  and  (
q3.\"CUSTOMERID\" =  q1.\"CUSTOMERID\")  and  ( q2.\"ACCID\" =
q3.\"ACCID\")
```

```
Using Account bean as root:
SELECT OBJECT(a_1000)
FROM Account a_1000,  IN (a_1000.customers) c_1002
WHERE c_1002.lastname = ?1
```

Regenerated SQL:

```
select  q1.\"ACCID\",  q1.\"BALANCE\",  q1.\"INTEREST\",
q1.\"ACCTYPE\" from GAOYANG.ACCOUNT q1, GAOYANG.CUSTOMER q2,
GAOYANG.CUSTACCT q3 where  ( q2.\"LASTNAME\" = ?)  and  (
q3.\"ACCID\" =  q1.\"ACCID\")  and  ( q2.\"CUSTOMERID\" =
q3.\"CUSTOMERID\")
```

6.3.2.4 Testing many-to-one relationship

Original SQL :

```
select q1 . *
from TRANSRECORD q1 , ACCOUNT q2
where q2 . ACCTYPE = ? and q2 . ACCID = q1 . ACCID
```

EJB-QL:

Using Account bean as root:

```
SELECT OBJECT(t_1002)
FROM Account a_1000, IN (a_1000.transrecords) t_1002
WHERE a_1000.acctype = ?1
```

Regenerated SQL:

```
select  q2.\"TRANSID\",  q2.\"TRANSTYPE\",  q2.\"TRANSAMT\",
q2.\"ACCID\" from GAOYANG.ACCOUNT q1, GAOYANG.TRANSRECORD q2
where  ( q1.\"ACCTYPE\" = ?)  and  ( q2.\"ACCID\" =
q1.\"ACCID\")
```

Using Transrecord bean as root:

```
SELECT OBJECT(t_1000)
FROM Transrecord t_1000
WHERE t_1000.theAccount.acctype = ?1
```

Regenerated SQL:

```
select  q1.\"TRANSID\",  q1.\"TRANSTYPE\",  q1.\"TRANSAMT\",
q1.\"ACCID\" from GAOYANG.TRANSRECORD q1, GAOYANG.ACCOUNT q2
where  ( q2.\"ACCTYPE\" = ?)  and  ( q2.\"ACCID\" =
q1.\"ACCID\")
```

6.3.2.5 Testing BETWEEN predicate

1) Original SQL:

```
select  q1.* from ACCOUNT q1 WHERE q1.BALANCE BETWEEN 1500 AND
2000
```

EJB-QL:

```
SELECT OBJECT(a_1000)
FROM Account a_1000
WHERE a_1000.balance  BETWEEN  1500 AND 2000
```

Regenerated SQL:

```
select  q1.\"ACCID\",  q1.\"BALANCE\",  q1.\"INTEREST\",
q1.\"ACCTYPE\" from GAOYANG.ACCOUNT q1 where  ( q1.\"BALANCE\"
>= 1500)  and  ( q1.\"BALANCE\" <= 2000)
```

6.3.2.6 Testing IN predicate

1) Original SQL:

```
select  q1.*
from ACCOUNT q1
where q1.BALANCE IN (1500, 2000)
```

EJB-QL:

```
SELECT OBJECT(a_1000)
FROM Account a_1000
WHERE a_1000.balance IN (1500,2000)
```

Regenerated SQL:

```
select  q1.\"ACCID\",  q1.\"BALANCE\",  q1.\"INTEREST\",
q1.\"ACCTYPE\" from GAOYANG.ACCOUNT q1 where  ( q1.\"BALANCE\" =
1500 or  q1.\"BALANCE\" = 2000)
```

6.3.2.7 Testing LIKE predicate

1) Original SQL:

```
select  q1.*
from CUSTOMER q1
where q1.LASTNAME LIKE 'Ma%e'
```

EJB-QL:

```
SELECT OBJECT(c_1000)
FROM Customer c_1000
WHERE c_1000.lastname LIKE 'Ma%e'
```

Regenerated SQL:

```
"select  q1.\"CUSTOMERID\",  q1.\"TITLE\",  q1.\"FIRSTNAME\",
q1.\"LASTNAME\",  q1.\"USERID\",  q1.\"PASSWORD\",
q1.\"ADDRESS\" from GAOYANG.CUSTOMER q1 where  ( q1.\"LASTNAME\"
LIKE \'Ma%e\')
```

6.3.2.8 Testing NULL predicate

1) Original SQL:

```
select  q1.*
from CUSTOMER q1
where q1.LASTNAME IS NULL
```

EJB-QL:

```
SELECT OBJECT(c_1000)
FROM Customer c_1000
WHERE c_1000.lastname IS NULL
```

Regenerated SQL:

```
select  q1.\"CUSTOMERID\",  q1.\"TITLE\",  q1.\"FIRSTNAME\",
q1.\"LASTNAME\",  q1.\"USERID\",  q1.\"PASSWORD\",
q1.\"ADDRESS\" from GAOYANG.CUSTOMER q1 where  ( q1.\"LASTNAME\"
IS NOT NULL )
```

6.3.2.9 Testing EXISTS predicate (uncorrelated subquery)

1) Original SQL:

```
select  q1.*
from ACCOUNT q1
where EXISTS (select q2.* from TRANSRECORD q2 where q2.TRANSAMT
>1000)
```

EJB-QL:

```
SELECT OBJECT(a_1000)
FROM Account a_1000
WHERE  EXISTS(SELECT OBJECT(t_1001)
FROM Transrecord t_1001
WHERE t_1001.transamt > 1000)
```

Regenerated SQL:

```
select  q1.\"ACCID\",  q1.\"BALANCE\",  q1.\"INTEREST\",
q1.\"ACCTYPE\" from GAOYANG.ACCOUNT q1 where  ( exists (  select
1 from GAOYANG.TRANSRECORD q2 where  ( q2.\"TRANSAMT\" > 1000) ))
```

6.3.2.10 Testing subqueries (non-key-based join)

1) Original SQL: (IN predicate)

```
select  q1.*
from TRANSRECORD q1
where q1.TRANSAMT IN (select q2.TRANSAMT
                      from  TRANSRECORD q2, ACCOUNT q3
                      where q2.ACCID = q3.ACCID AND q3.ACCID =
                'a10' )
```

EJB-QL:

```
SELECT OBJECT(t_1000)
```

```
FROM Transrecord t_1000
WHERE  t_1000.transamt IN(SELECT t_1003.transamt
FROM Transrecord t_1003
WHERE t_1003.theAccount.accid = 'a10')
```

Regenerated SQL:

```
select  q1.\"TRANSID\",  q1.\"TRANSTYPE\",  q1.\"TRANSAMT\",
q1.\"ACCID\" from GAOYANG.TRANSRECORD q1 where  (
q1.\"TRANSAMT\" = ANY  ( select  q2.\"TRANSAMT\" from
GAOYANG.TRANSRECORD q2, GAOYANG.ACCOUNT q3 where  ( q3.\"ACCID\"
= \'a10\')  and  ( q3.\"ACCID\" =  q2.\"ACCID\") ) )
```

2) Original SQL : (comparison predicate)

```
select  q1.*
from TRANSRECORD q1
where q1.TRANSAMT = ANY(select q2.TRANSAMT
                        from  TRANSRECORD q2, ACCOUNT q3
                        where q2.ACCID = q3.ACCID AND q3.ACCID =
                'a10' )
```

EJB-QL:

```
SELECT OBJECT(t_1000)
FROM Transrecord t_1000
WHERE t_1000.transamt =ANY(SELECT t_1003.transamt
FROM Transrecord t_1003
WHERE t_1003.theAccount.accid = 'a10')
```

Regenerated SQL:

```
select  q1.\"TRANSID\",  q1.\"TRANSTYPE\",  q1.\"TRANSAMT\",
q1.\"ACCID\" from GAOYANG.TRANSRECORD q1 where  (
q1.\"TRANSAMT\" = ANY  ( select  q2.\"TRANSAMT\" from
GAOYANG.TRANSRECORD q2, GAOYANG.ACCOUNT q3 where  ( q3.\"ACCID\"
= \'a10\')  and  ( q3.\"ACCID\" =  q2.\"ACCID\") ) )
```

3) Original SQL : (comparison predicate with negation)

```
select  q1.*
from TRANSRECORD q1
where q1.TRANSAMT <> ALL (select q2.TRANSAMT
                        from  TRANSRECORD q2, ACCOUNT q3
                        where q2.ACCID = q3.ACCID AND q3.ACCID =
                10 )
```

EJB-QL:

```
SELECT OBJECT(t_1000)
FROM Transrecord t_1000
```

```
WHERE t_1000.transamt <>ALL(SELECT t_1003.transamt
FROM Transrecord t_1003
WHERE t_1003.theAccount.accid = 'a10')
```

Regenerated SQL:

```
select  q1.\"TRANSID\",  q1.\"TRANSTYPE\",  q1.\"TRANSAMT\",
q1.\"ACCID\" from GAOYANG.TRANSRECORD q1 where  (
q1.\"TRANSAMT\" <> ALL  ( select  q2.\"TRANSAMT\" from
GAOYANG.TRANSRECORD q2, GAOYANG.ACCOUNT q3 where  ( q3.\"ACCID\"
= \'a10\')  and  ( q3.\"ACCID\" =  q2.\"ACCID\") ) )
```

6.3.2.11 Testing subqueries (key-based join)

1) Original SQL:

```
select q1.*
from ACCOUNT q1
where q1.BALANCE >100 and q1.ACCID IN(select q2.ACCID
                                      from TRANSRECORD q2
                                      where q2.TRANSAMT > 20)
```

EJB-QL:

```
SELECT OBJECT (a_1000)
FROM Account a_1000, IN(a1.transrecords) t_1000
WHERE a_1000.balance >100 AND t_1000.transamt>20)
```

Regenerated SQL:

```
select  q1.\"BALANCE\",  q1.\"INTEREST\",  q1.\"ACCID\",
q1.\"ACCTYPE\" from GAOYANG.ACCOUNT q1, GAOYANG.TRANSRECORD q2
where  ((q1.\"BALANCE\" > 100)  and  ( q2.\"TRANSAMT\" > 20)
and  ( q2.\"ACCID\" =  q1.\"ACCID\")")
```

2) Original SQL: (EXISTS predicate, correlated subquery, key-based join)

```
select q1.*
from ACCOUNT q1
where q1.BALANCE >100 and exists (select q2.ACCID
                                  from TRANSRECORD q2
                                  where q2.TRANSAMT > 20 and
                                  q1.ACCID = q2.ACCID)
```

EJB-QL:

```
SELECT OBJECT (a_1000)
FROM Account a_1000, IN(a1.transrecords) t_1000
WHERE a_1000.balance >100 AND t_1000.transamt>20)
```

Regenerated SQL:

```
select  q1.\"BALANCE\",  q1.\"INTEREST\",  q1.\"ACCID\",
q1.\"ACCTYPE\" from GAOYANG.ACCOUNT q1, GAOYANG.TRANSRECORD q2
where  ((q1.\"BALANCE\" > 100)  and  ( q2.\"TRANSAMT\" > 20)
and  ( q2.\"ACCID\" =  q1.\"ACCID\")")
```

3) Original SQL: (IN predicate with negation)

```
select t.*
from ACCOUNT a
where a.BALANCE >100 and a.ACCID NOT IN (select t.ACCID
                    from TRANSRECORD t
where t.TRANSAMT > 20)
```

EJB-QL:

```
SELECT OBJECT(a_1000)
FROM Account a_1000
WHERE a_1000.balance >100 AND NOT EXISTS (SELECT object(t_1000)
                    FROM IN(a_1000.transrecords) t_1000
WHERE t_1000.transamt > 20)
```

Regenerated SQL:

```
select  q1.\"BALANCE\",  q1.\"INTEREST\",  q1.\"ACCID\",
q1.\"ACCTYPE\" from GAOYANG.ACCOUNT q1 where  (( q1.\"BALANCE\"
> 100)  and  ( not  exists (  select 1 from GAOYANG.TRANSRECORD
q2 where  ( q2.\"TRANSAMT\" > 20)  and  ( q2.\"ACCID\" =
q1.\"ACCID\") ) )");
```

6.3.2.12 Testing aggregation function

1) Original SQL:

```
select a1.*
from ACCOUNT a1, CUSTOMER c1, CUSTACCT ca1
where c1.CUSTOMERID=?
```

```
and a1.BALANCE =
        (select MAX(a2.BALANCE)
         from ACCOUNT a2, CUSTACCT ca2
         where ca2.CUSTOMERID = c1.CUSTOMERID
            and a2.ACCID = ca2.ACCID)
and ca1.ACCID = a1.ACCID
and c1.CUSTOMERID = ca1.CUSTOMERID
```

EJB-QL:

```
SELECT object(a_1000)
FROM Account a_1000, in(a_1000.customers) c_1000
WHERE a_1000.balance = (SELECT(MAX(a_1001.(balance))
                        FROM IN(c_1000.accounts) a_1001)
         and c_1000.customerid = ?1
```

Regenerated SQL:

```
"select  q1.\"BALANCE\",  q1.\"INTEREST\",  q1.\"ACCID\",
q1.\"ACCTYPE\" from GAOYANG.ACCOUNT q1, GAOYANG.CUSTOMER q2,
GAOYANG.CUSTACCT q3 where  (( q2.\"CUSTOMERID\" = ?)  and  (
q1.\"BALANCE\" =  ( select  max( q4.\"BALANCE\") from
GAOYANG.ACCOUNT q4, GAOYANG.CUSTACCT q5 where  ( (
q5.\"CUSTOMERID\" =  q2.\"CUSTOMERID\")  and  ( q4.\"ACCID\" =
q5.\"ACCID\") ) )  and  ( q3.\"ACCID\" =  q1.\"ACCID\")  and  (
q2.\"CUSTOMERID\" =  q3.\"CUSTOMERID\")");
```

6.3.3 Result Analysis

From the testing result, we found that:

1. All the translations for unnested queries without joins are correct.
2. The translation of aggregate functions is straightforward, the result is correct.
3. The translations for some unnested queries with joins are failed under certain conditions.
4. The translations for nested queries led by IN, EXIST and = which has key-based join are translated in one form. The translated EJB-QL is unnested.

5. The translations for nested queries led by NOT IN, NOT EXIST and <> predicates which has key-based join are translated in one form. The translated EJB-QL has nesting structure with the nesting operator "NOT EXISTS".

Table 6-2 illustrates the testing result in detail.

SQL queries	Unnested query without join	Unnested query with join	Aggregation Function	Subquery nesting operator				
				IN, = NOT IN, <>		EXISTS / NOT EXISTS		
				Key-based join	Nonkey-based join	uncorrelated subquery	Correlated subquery (EXISTS)	
							Key-based join	Nonkey-based join
Completeness	complete	incomplete	complete	incomplete	complete	complete	incomplete	complete
Correctness	correct	correct	correct	correct	correct	correct	correct	correct

Table 6-2 Testing result analysis

Remark: This result is under certain condition.

For translating a key-based join predicate between two tables T1 and T2, we assume the corresponding entity beans are E1 and E2 respectively. During the translation, E1 is selected as a start of traversal. The relationships between two tables/beans, and the selection condition on table/bean T2/E2 which is not the start of the traversal, will affect the translation result. Table 6-2 explains the translation results under different conditions.

Relationship between tables/entity beans	Translation Result	
	Selection conditions on Tables/beans	
	Yes	No
1:1	translated	failed
1:m	translated	translated
m:1	translated	failed
m:m	translated	translated

Table 6-3 Translation result for a join predicate

6.4 Experiment 2 - EJB ORDER Project

The data used in this experiment are provided by IBM WCS 5.1, which uses EJB1.0 with DB2 7.1.1 as backend. The queries are stored in an XML file. The schema of database, ejb-jar and cmp-mappings files are stored in a different XML file respectively.

6.4.1 Testing Procedure

The testing has three steps:

Step 1: Using *Mapping file generator* to generate xml files which is used to set up a translation application.

Input: All these files are generated by WSAD with DB2 as backend.

1) database schema files: *.schxmi and *.tblxmi files. For example, in this project, the input file is: `Order-OrderCaptureData_NULLID.schxmi`, `Order-OrderCaptureData_NULLID_ACCOUNT.tblxmi`, etc.)
2) ejb-jar.xml. Describe all enterprise beans information.
3) Map.mapxmi. Describe the mappings between database schema and CMP beans.

Output:

1) schema.xml. Database schema file.
2) ejb-jar.xml. Describe the CMP beans and their relationships. The file excludes the information of other beans such as BMP bean, session bean, etc.
3) cmp-mappings.xml. Describe the mapping between the database schema and CMP beans.

Step 2: Using *SQL extractor* which takes as input the EJB code containing partial queries defined in EJB1.X finder methods, to produce the standardized and complete SQL.

Input: ibm-ejb-jar-ext.xmi.

Output: queries.xml.

Step 3: Using *SQL2EJBQL Translator* to translate SQL queries to EJB-QL queries.

Input: schema.xml, ejb-jar.xml, cmp-mappings.xml, queries.xml.

Output: ejbqls.xml

6.4.2 Testing Result

170 SQL statements from finder methods in entity beans are collected and tested.

Here is the statistics:

- Total translated: 124.
- Failed: 46.

Figure 6-2 describes the statistics in details:

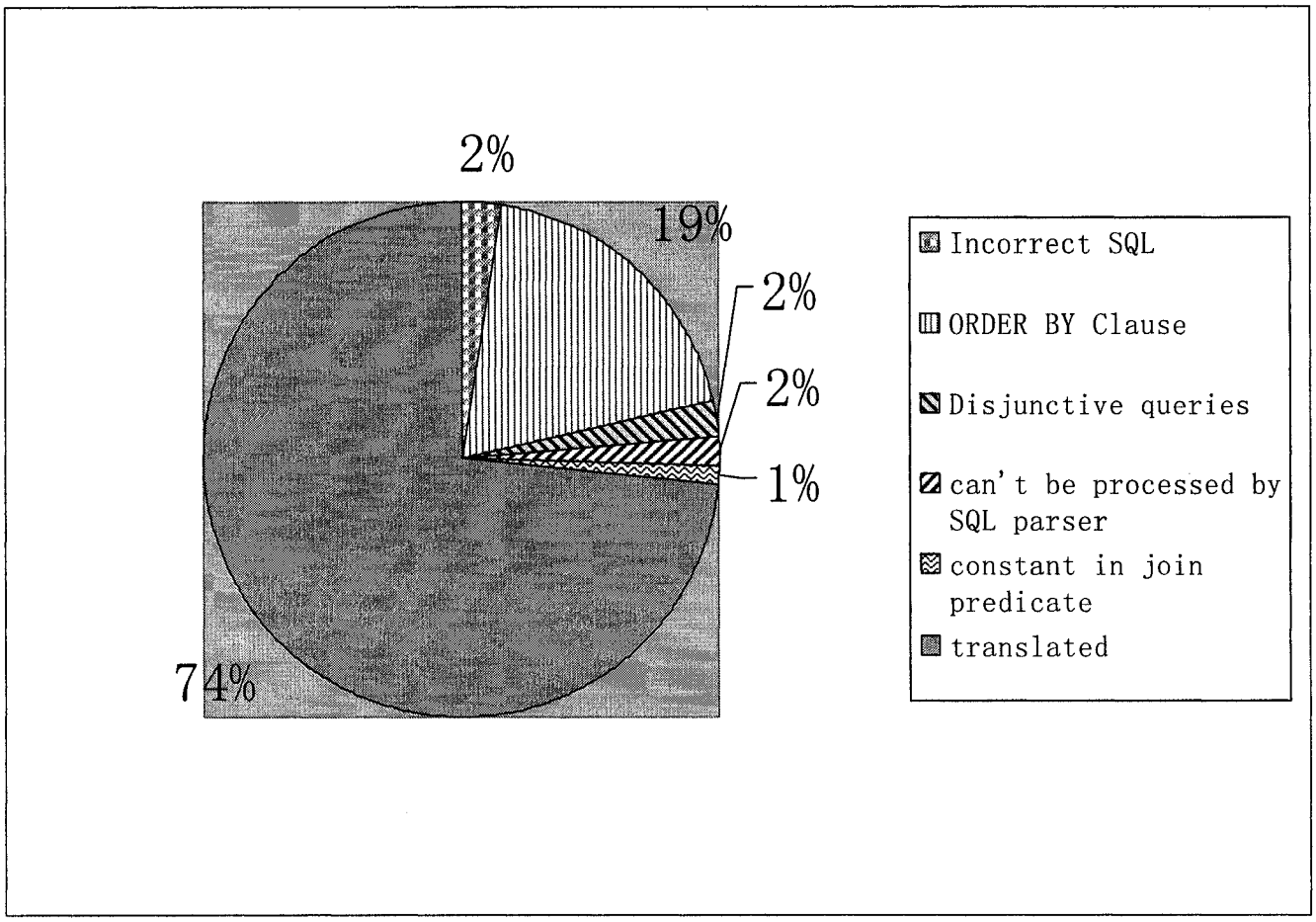


Figure 6-2 The statistics of testing results

6.4.3 Result Analysis

The reasons of failed translation are listed as below.

1. Incorrect SQL statement. Tables in SQL statement are not defined in database schema. All these queries are nested query with tables in subqueries are not defined in schema. For example:

```
SELECT SCHORDERS.*
FROM SCHORDERS
WHERE (SCHORDERS.JOB_RN = ANY (SELECT SCCJOBREFNUM FROM
SCHCONFIG WHERE MEMBER_ID = ?))
```

`SCHCONFIG` is not defined in the schema file.

Number of such queries: ***5***.

2. "ORDER BY" clause. For the time being, our translator does not support the translation of "ORDER BY" clause.

   Number of such queries: ***32***.

3. Disjunctive queries. We can't translate disjunctive queries. Number of such queries: ***4***.

4. Query can't be processed by SQL parser SQL4J.

   For example:

```
SELECT SHIPINFO.*
FROM SHIPINFO
WHERE SHIPINFO.ORDERS_ID = ? AND SHIPINFO.ADDRESS_ID = -1
```

   Number of such queries: ***3***.

   This can be fixed easily by modifying the parser.

5. Constant in join predicate.

   For "findAll" queries with the statement have "1=1" in WHERE condition. For example:

```
SELECT TRADEPOSCN.*
FROM TRADEPOSCN
WHERE 1 = 1
```

   So far, our translator does not support translating the join predicate with constant.

   Number of such queries: ***2***.

73% of SQL queries, extracted from a real industry product, were translated successfully. Being able to provide a correct and automated translation of SQL to EJB-QL, our translation tool will offer a great help in the migrations of enterprise applications.

Chapter 7

Conclusions and Future Works

7.1 Conclusions

This thesis describes an algorithm and its implementation of SQL to EJB-QL query translation, which is motivated by upgrading of EJB container. Based on the translation techniques between relational and object query languages, our work extends the existing works in the following aspects:

1) Existing works described the translation of a small subset of SQL queries with many restrictions. For example, some exclude subqueries, some require key-based semijoins and antijoins, and none deals with aggregates. Our approach takes subqueries with all types of semijoins and antijoins into consideration, furthermore, add the translation on aggregate functions. The range of target SQL queries has been expanded to a larger set.
2) Our translation techniques are tailored to EJB-QL. Although our algorithm is for the translation of SQL to EJB-QL, EJB-QL is an object query language (OQL), these techniques can be easily applied on the translation between SQL and other OQL.
3) As far as we know, existing works are neither implemented nor tested. We applied and tested our system with real data from an industry product.

The query translation system is implemented under the following assumptions:

1) Only SQL queries with SELECT-FROM-WHERE statements and also the subqueries with IN, EXISTS and comparison predicates are considered.
2) The relational WHERE clause contains qualification conditions in conjunctive form.
3) The mapping between database schema and entity beans is one to one.

In spit of all above restrictions, the implementation of a query translation system is still a challenging work. A series of experiments have been carried out implement various types of query translation. From the experimental results, we get the conclusions as follows:

1) All the translations for unnested queries without joins are correct.
2) The translation of aggregate functions is straightforward, and the result is correct.
3) The translations for some unnested queries with joins are failed under certain conditions.
4) The translations for nested queries led by IN, EXIST and = which has key-based join are translated in one form. The translated EJB-QL is unnested.
5) The translations for nested queries led by NOT IN, NOT EXIST and <> predicates which has key-based join are translated in one form. The translated EJB-QL has nesting structure with the nesting operator "NOT EXISTS".
6) 73% of SQL queries, extracted from a real industry product, were translated successfully. Being able to provide a correct and automated translation of SQL to EJB-QL, our translation tool will offer a great help in the migrations of enterprise applications.

Consequently, the algorithm proposed in this thesis performs well. The subset of target SQL queries is expanded a lot in comparison to the other existing algorithm. Because of the restrictions of the syntax of EJB-QL, as we mentioned in section 6.3.3, when two tables involved in a key-based join condition, one bean corresponding to a table is chosen as a traversing start point, if this bean has a 1:1 or m:1 relationships with the other bean, and there is no selection condition on the latter, the translation will fail. This is the main disadvantage of the algorithm.

7.2 Future Works

7.2.1 Relational Schema and EJB Mapping

The Mapping file generator to extract the mappings between database schema and enterprise beans. Defining a mapping from a set of tables in a relational database to a set of entity beans is the key issue of the tool's development. At the current stage of the development we are assuming the mapping is 1-1.

In the real world, O/R mapping between relational database and entity beans may not be a simple 1-1 mapping, instead it may be a many-many mapping in many cases. The same applies to the attribute-column mapping. A possible solution to deal with this is query rewriting, we will investigate this issue and make the implementation in future.

7.2.2 Translation upgrading with the changes of EJB specification

EJB has being developed rapidly. Now the specification of EJB is reaching version 3.0. Lot of new features have been introduced in EJB 3.0, more support was added in EJB-QL. In this thesis, our translation is mainly based on EJB-QL2.0 with some extension on aggregation functions and subqueries. We need to upgrade our translation to be able

to translate more advance SQL statement such as "left outjoin", HAVING clause, ORDER BY clause, GROUP BY etc..

7.2.3 Further integration of J2EE-compliant servers.

As we metioned before, the mapping file to bind the abstract schema to a backend database is provided by a specific EJB container, the format of the file is very vender depended. For now, our translator only provides for support of the Sun ONE Application Server and IBM WCS 5.1. The support of further J2EE- compliant servers is easy to achieve due to a modular design of SQL2EJBQL translator's software architecture.

.

Bibliography

[AAG+01] R. Adatia, F. Arni, K. Gabhart, J.Griffin, M. Juric, J. Lott, T. McAllister, A. Mulder "Professional EJB". Wrox Press, 2001.

[AAK97] Behm A, Geppert A, Dittrich KR. "On the migration of relational schemas and data to object-oriented database systems". In Proceedings of the 5th International Conference on Re-Technologies for Information Systems, Austria, 1997.

[BAI] Barry & Associates Inc. "Mapping Tables to Objects". http://www.service-architecture.com/object-relational-mapping/articles/mapping_tables_to_objects.html.

[BAZ93] H. Balsters, R. A. de By & R. Zicari, "Typed sets as a basis for object-oriented database schemas". In Proceedings Seventh European Conference on Object-Oriented Programming, July 26-30, 1993.

[BEA04] BEA Systems. EJB Query Language (EJB-QL) and WebLogic Server. Product documentation, 2004. http://e-docs.bea.com/wls/docs81/ejb/EJB-L.html#1151257.

[C04] Doug Clarke. "J2EE Persistence - Productivity with Choice". http://www.jax.de/ materialien2004 /jax/keynotes/clarke_j2ee_persistence.pdf

[CRD94] Chang, Y., Raschid, L. and Dorr, B.."Transforming queries from a relational schema to an equivalent object schema: a prototype based on F-logic". Proceedings of the International Symposium on Methodologies for Intelligent Systems, 1994.

[CSG94] M. Castellanos F. Saltor and M. Garcia-Salaco. "Semantically enrichment of relational databases into an object-oriented semantic model". In The 5th International conference on Database Applications DEXA'94, 1994.

[D87] U. Dayal. "Of nests and trees: A unified approach to processing queries that contain nested subqueries, aggregates, and quantifiers," in Proceedings of Thirteenth International Conference on Very Large Data Bases, Brighton England, September 1-4, 1987, P.M. Stocker, W. Kent & P. Hammersley, eds., Morgan Kaufmann Publishers, Los Altos, CA, 1987, 197-207.

[DB04] K. Drosten, C. Bals. "BeanMaker – a Tool for Automatic Generation of Persistent Enterprise Java Beans". Research paper, 2004.http://fhge.opus.hbz-nrw.de/volltexte/2004/21/.

[FKS94] J. Flokstra, Maurice van Keulen & J. Skowronek, "The IMPRESS DDT: A database design toolbox based on a formal specification language." In Proceedings ACM-SIGMOD 1994 International Conference on Management of Data, ACM Press, New York, NY, 1994, 506.

[FV95] C. Fahrner and G. Vossen. "Transformation of relational schemas into object oriented schemas according to odmg-93". In fourth International Conference on Deductive and Object-Oriented Database DOOD'95, pages 429-446, 1995.

[GW99] James R. Groff and Paul N. Weinberg. "SQL: The Complete Reference". MCGRAW HILL Book Company. April, 1999.

[H01] Richard Monson-Haefel. "Enterprise JavaBeans", 3rd Edition. O'Reilly, September 2001.

[H04] Jeff Hanson. "An Introduction to Java Object Persistence with EJB". November 9, 2004. Available at http://www.devx.com/Java/Article/22441.

[IBM03] IBM. "Object Finder EJB Query Language for Container Managed Persistent Entities Draft 2.0 for V5.0.1", January 2003

[J97] Fong, J.. "Converting Relational to Object-oriented Database". SIGMOD Record, Vol.26, No.1, March 1997

[K82] Won Kim, "On Optimizing an SQL-like Nested Query".ACM Trans. Database Syst. 7(3): 443-469, 1982

[K90] Won Kim. "Introduction to object-oriented databases". Massachusetts Institute of Technology, 1990.

[L02] Jianguo Lu. "Reengineering of Database Applications to EJB Based Architecture". CAISE 2002, LNCS 2348, pp. 361-376, 2002.

[LLH+01] T. Lau, J. Lu, E. Hedges and E. Xing. "Migrating E-commerce Database Applications to an Enterprise Java Environment". In Proc. of 2001 conference of the Centre for Advanced Studies on Collaborative research, Canada, 2001.

[MK98] A. Mostefaoui and J. Kouloumdjian. "Translating Relational Queries to Object-Oriented Queries According to ODMG-93". ADBIS 1998: 328-338, 1998.

[MKD+03] Vlada Matena, Sanieey Krishnan, Linda DeMichiel, Beth Stearns. "Applying Enterprise JavaBeans: Component-Based Development for the J2EE Platform". 2nd Edition, January 2003

[MYK95] W. Meng, C. Yu, W. Kim. "A Theory of Translation From Relational Queries to Hierarchical Queries". IEEE Transactions on Knowledge and Data Engineering. Volume 7, Issue 2 (April 1995) , Pages: 228 – 245, 1995.

[MYK+93] W. Meng, C. Yu, W. Kim, G. Wang, T. Pham and S. Dao. "Construction of Relational Front-end for Object-Oriented database Systems. ICDE'93

[ORA03] Oracle Corp.. "OracleAS TopLink Unit of Work Primer". White paper, 2003.

[P01] M. Petropoulos. "OQL – Object Query Language". http://www.db.ucsd.edu /People /michalis/notes/O2/OQLTutorial.htm, October 2001.

[P03] Pepperdine, K.. "Oracle9Ias TopLink by Example". http://www.oracle.com /technology/ oramag/webcolumns/2003/techariticles/pepperdine_TopLink.html. 2003.

[PTK95] J.M. Petit, F. Toumani, and J. Kouloumdjian. "Relational databases reverse engineering: a method based on query analysis". International Journal of Cooperative Information Systems, 4(2,3):2870-316, 1995.

[RAJ+01] Ed Roman, Scott W. Ambler, Tyler Jewell, Floyd Marinescu. "Mastering Enterprise JavaBeans (2nd Edition)". 2001.

[RC95] Raschid, L. and Chang, Y. "Interoperable Query Processing from Object to Relational Schemas Based on a Parameterized Canonical Representation." Inter. Journal of Intelligent and Cooperative Information Systems, 1995.

[RSH96] Ramanathan, Shekar, and Julia Hodges. "Reverse Engineering Relational Schemas to Object-Oriented Schemas". Technical Report. July, 1996.

[SUN01] SUN Microsystems Inc.. "Enterprise JavaBeans Specification, Final Version, Version2.0", August 2001.

[SUN03] Sun Microsystems, Inc. "EJB QL: EJB Query Language for Container-Managed Persistence Query Methods". Enterprise JavaBean 2.1, Final Release". September, 2003.

[SUN05] Sun Microsystems, Inc. "JSR 220: Enterprise JavaBeans, Version 3.0, Persistence API, Early Draft Review 2". February, 2005.

[SGT+01] P G Sarang, Kyle Gabhart, Andre Tost, Tim McAllister, Rahim Adatia, Matjaz Juric, Ted Osborne, Faiz Arni, Jeremiah Lott, Vaidyanathan Nagarajan, Craig A. Berry, Dan O'Connor, John Griffin, Aaron Mulder, Dave Young. "Professional EJB". Wrox Press Ltd.. 2001.

[THO] THOUGHT Inc.. "Dynamic Universal Querying with CocoBase". Technical Paper.

[THO02] THOUGHT Inc. "CocoBase ®.Enterprise O/R Version 4.5 Features and Benefits". Technical paper. 2002.

[THO03] THOUGHT Inc. "CocoBase® Enterprise O/R Optimized for IBM WebSphere Studio IDE". http://www.thoughtinc.com/ibm_wasd/, 2003.

[VA95] M. Vermeer and P. Apers. "Object-oriented views of relational databases incorporating behaviour". In the fourth International Conference on Database Systems for Advanced Applications (DASFAA'95), 1995.

[WDS+03] U. Wahli, W. Denayer, L. Schunk, D. Shaddon, M. Weiss. "EJB2.0 Development in WSAD". IMB Red book, April 2003.

[YZM+95] C.Yu, Y.Zhang, W.Meng, W.Kim, G.Wang, T.Pham, S.Dao. "Translation of Object-Oriented Queries to Relational Queries". Proc. of the 11th Intl. Conf. on Data Engineering, March 6-10, Taipei, Taiwan, 1995.

[YL93] L. Yan and T. Ling. "Translating relational schema with constraints into OODB schema". IFIP Transactions of Interoperable Database Systems (DS-5), 1993.

Vita Auctoris

Name:	Yang Gao
Place of Birth:	Jinhua, Zhejiang Province, P.R.China
Date of Birth:	1970
Education:	University of Windsor Windsor, Ontario, Canada 2002-2005 M.Sc in Computer Science
	Northwest Polytechnic University Xian, P.R. China 1988-1992 B.Sc., Materials Engineering