CIFE CENTER FOR INTEGRATED FACILITY ENGINEERING

Efficiency in Instantiating Objects from Relational Databases Through Views

by

Kincho Law and
Byung Suk Lee

TECHNICAL REPORT
Number 43

February, 1991

Stanford University

If you would like to contact the authors please write to:

c/o CIFE, Civil Engineering,
Stanford University,
Terman Engineering Center
Mail Code: 4020
Stanford, CA 95305-4020

Abstract

An integration of objects and databases provides a framework in which applications take advantage of the high productivity and reusability of an object-oriented software, and at the same time the sharability and maintainability of databases. One of the approaches for achieving this integration is to instantiate objects from relational databases through views. In this approach, a view is defined by a relational query and a function for mapping between object attributes and relation attributes. The query is used to materialize the necessary data into a relation from database, and the function is used to restructure the materialized relation into objects.

The approach of instantiating objects from relational databases through views provides an effective mechanism for building object-oriented applications on top of relational databases. However, a system built in such a framework has the overhead of interfacing between two different models – an object-oriented model and the relational model – in terms of both functionality and performance. In this thesis, we address two important problems: the outer join problem and the instantiation efficiency problem.

Outer join problem: In instantiating objects, tuples that should be retrieved from databases may be lost if we allow only inner joins. Hence it becomes necessary to evaluate certain join operations of the query by outer joins, left outer joins in particular. On the other hand, we sometimes retrieve unwanted nulls from nulls stored in databases, even if there is no null inserted during query processing. In this case, it is necessary to filter some relations with selection conditions which eliminate the tuples containing null attributes in order to prevent the retrieval of unwanted nulls. We develop a mechanism for making the system generate those left outer joins and filters as needed rather than requiring that a programmer specifies it manually

as part of the query for every view definition. We also address how to reduce the number of left outer joins and filters for reducing the query processing time.

Instantiation efficiency problem: Since the advent of the relational databases, it has been universally accepted that a query result is retrieved as a single flat relation (a table). Such a relation is neither normalized nor nested if the query includes joins and has redundancies. This single table concept is not useful in our framework because a client wants to retrieve object instances. Rather, a single flat relation contains data redundantly inserted just to make the query result 'flat'. These redundant data convey no extra information but only degrade the performance of the system. This fact motivated us to look into different methods which reduce the amount of data that the system must handle to instantiate objects, without diminishing the amount of information to be retrieved. In this thesis, we present two alternative methods which retrieve a query result in less redundant structures than a single flat relation. Our result demonstrates that these two methods incur far less cost than the method of retrieving a single flat relation. We assume a computing environment that is a client-server architecture, where relational databases reside on servers and applications reside on connected workstations. Main memory database systems will benefit most from our work, although our work is useful for secondary storage database systems as well.

Acknowledgements

I would like to thank my advisor, Gio Wiederhold for his support and guidance, and for his patience and encouragement throughout this work. He always gave me his hand when I was in need, which helped me to overcome difficulties several times.

I also like to thank my reading committee, Mark Linton, Witold Litwin, and Kincho Law, for their willingness to serve on my committee and for their fruitful comments and advice.

I am grateful to my colleagues in the KSYS group. Keith Hall spent so many days to set up and maintain the computing environment for the group, even though he was busy enough with his dissertation work alone. Peter Rathmann and Tore Risch greatly helped me with their technical opinions on my questions. Some of their comments were critical to the progress of my work. Peter Rathmann also helped me out of many troubles with LaTeX'ing this draft. Ki-Joon Han, Arthur Keller, and Linda DeMichiel reviewed all or part of the draft and gave good comments. It was fortunate of me to have these competent and knowledgeable colleagues around me.

Finally, I would like to thank my wife Hye-Young for her sacrifice and endurance during all my years as a graduate student, my daughter Sonah for having been healthy and happy, and my parents for their prayers and concerns.

This research has been performed as part of the KBMS project, supported by DARPA Contract No. N039-84-C-0211. This research was also partially sponsored by the Center for Integrated Facility Engineering at the Stanford University.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

We have seen increasing effort for supporting object-oriented applications with databases. One of the approaches for this effort is to instantiate objects from relational databases through views [14, 15, 17, 8, 10, 12]. A view is defined by a relational query and a function for mapping between object attributes and relation attributes. The query is used to materialize the necessary data into a relation from databases, and the function is used to restructure the materialized relation into objects.

The approach of instantiating objects from relational databases through views provides an effective mechanism for building object-oriented applications on top of relational databases. Example applications are engineering design software such as computer-aided design (CAD) or computer-aided software engineering (CASE). These applications become more effective by utilizing the locality and information encapsulation available from an object-oriented approach. Complex objects [27, 28, 29, 42, 43, 44, 22] are typically needed in these applications. Relational databases provide sharing and flexibility, whose benefit becomes magnificent as the size of databases become larger. A system built in such a framework has the overhead of interfacing between two different models – an object-oriented model and the relational model – in terms of both functionality and performance. In this thesis, we address two important problems: outer join [35] problem and instantiation efficiency problem. The outer join problem is a functionality problem as well a performance problem, while the instantiation efficiency problem is entirely a performance problem.

1.1 Outer Join Problem

In instantiating objects, some particular conditions arise that are not so common in traditional relational database operations. First of all, as will be shown in Section 3.2.2.1, it often happens that we lose tuples that should be retrieved from databases, if we allow only inner joins. Hence, it becomes necessary to evaluate some joins of the query by *outer joins*. In particular we need *unidirectional* outer joins such as left outer joins [35]. On the other hand, we sometimes retrieve unwanted nulls from nulls stored in databases, even if there is no null inserted during query processing. In this case, it is necessary to *filter* some relations with selection conditions which eliminate the tuples containing null attributes to prevent the retrieval of unwanted nulls.

It is desirable to make the system generate those left outer joins and filters as needed rather than requiring that a programmer specifies them manually as part of the query for every view definition. We develop such a mechanism in the first part of this thesis.

Without optimization, declarative approaches such as SQL queries and views are not practical. However, optimization of queries with outer joins has rarely been treated. Since left outer joins are not symmetric, they inhibit a query optimizer from attempting to reorder joins for more efficient query processing. Furthermore, application of non-null filters is not free. It incurs the cost of evaluating the corresponding selection predicates on a base relation. We show that, for certain cases that occur frequently, these two operators can be avoided without affecting the query result.

1.2 Instantiation Efficiency Problem

The client-server architecture is becoming a standard architecture in modern computing environment. In the client-server architecture, object-oriented applications run on client workstations and access data stored in remote database servers. A view pertinent to an object type contains a relational query, which is delivered to a remote database server; The query result is retrieved from a server and is restructured into

nested relations [68, 69, 70] by a client.

Since the advent of the relational databases [24], it has been universally accepted to retrieve a query result as a *single flat relation* or a table. In fact, one of the advantages of the relational model is that it enables us to apply the same language (a relational query) uniformly on both base relations and query results. However, this concept is not useful in our work because what a client wants to retrieve is a nested relation, not a flat relation. Rather, a single flat relation contains data redundantly inserted just to make the query result 'flat'. These redundant data convey no extra information but only degrade the performance of the system. Certainly it will be more efficient to manipulate less data as long as we retrieve the same information.

In the second part of this thesis, we present two alternative methods of instantiating objects from remote relational databases through views. The two methods retrieve a query result in other structures than a single flat relation. One method retrieves a set of relation fragments and the other method retrieves a single nested relation. We will demonstrate that these two methods incur far less cost than the method of retrieving a single flat relation.

1.3 Organization of the Thesis

Following this introduction, we describe the background framework of our work in Chapter 2. Then, the outer join problem and the instantiation efficiency problem are addressed respectively in Chapter 3 and Chapter 4. We develop a rigorous system model within Chapter 3. The system model is developed basically for providing a basis for solving the outer join problem but is also used for the instantiation efficiency problem. Finally, conclusion follows in Chapter 5.

Chapter 2

Background Framework

2.1 Introduction

In this chapter, we provide the framework upon which this thesis stands. We start from a general framework for integrating objects and databases and categorize the general framework in Section 2.2 through Section 2.5. Two different dimensions are used to categorize the general framework: integration approach and binding time. Meanwhile, we narrow down our focus to the view-object framework, which is described in Section 2.6. The view-object framework is what this thesis is built upon.

2.2 Integration of Objects and Databases

We distinguish two alternative approaches to the integration of objects and databases: the *direct object storage* approach and the *indirect base relation storage* approach. In the object storage approach, an object-oriented model is used uniformly for applications and persistent storage [3, 1, 2, 5, 6, 87]; objects are retrieved and stored as objects. In the relation storage approach, an object-oriented model is used for the applications while a relational storage model is used for persistent storage [4, 8, 9, 10, 11, 12, 17, 20], and objects are retrieved by evaluating queries to databases[1].

[1]There are some systems which cannot be put strictly in either of these two categories. For Example, PCLOS [18] allows both possibilities. The storage can be relational, object-oriented, or

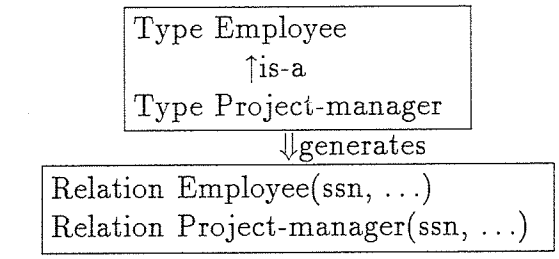
The relation storage approach incurs the overhead of mapping between different models [14, 23], but is useful for *large* databases since the relation storage approach supports *sharing* of different user views better than the object storage approach. Direct storage of objects is simple, but inhibits sharability [14]. For example, let us assume two users define `Employee` objects differently as `Employee(name, salary)` and `Employee(name, department)` respectively. In the object storage approach, the two `Employee` objects are stored separately. To provide sharing requires a separate mechanism for identifying the owners. In the relation storage approach however, this problem does not occur because the information to support the two `Employee` objects is stored in a single relation `Employee(name, salary, department)`, and their owners are distinguished by the database view mechanism.

2.3 Two Perspectives of the Relation Storage Approach

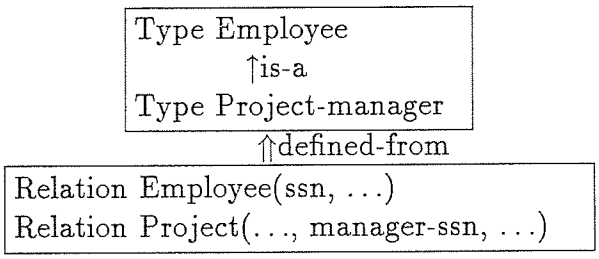
We observed two different perspectives within the relation storage approach: *object-centered* [4, 9, 11, 12] and *relation-centered* [17, 20]. In object-centered perspective, relation schemas are generated from given object schemas, i.e., types and their hierarchy. Relations are the destination for storing objects, and objects are decomposed into relations using the concept of normalization. On the other hand, in *relation-centered* perspective, object schemas are defined from given relation schemas. Relations are the source for generating objects, and objects are composed from relations. The composition of objects is useful for building object-oriented applications on top of *existing* relational databases[2]. The two perspectives may look like the two sides of the same coin, but they differ in terms of design approach. Figure 2.1 shows the two perspectives. In Figure 2.1a, the `Project-manager` type is mapped to the `Project-manager` relation. There exists a separate relation for each corresponding object type. In Figure 2.1b, there does not exist a separate `Project-manager` relation in the given

even a file system [19].

[2]We cannot throw away the relational data model in a decade. Remember that the IMS hierarchical data model implementation is still prevalent while we call the relational model 'conventional'.



(a) Object-centered perspective



(b) Relation-centered perspective

Figure 2.1: Two perspectives of relation storage approach

database. Rather, the `Project-manager` type is defined as an *abstraction* through views, such as defining a join between the `Employee` relation and `Project` relation along the `manager-ssn` foreign key. The join retrieves only the employees that are managing one or more projects. Let us consider the `Project-manager` as a *derived relation* of the `Employee` and `Project` relations. Note the derived relation is analogous to the intensional database (IDB) relation [30, 32] used in the integration of the logic-based model and relational model [32, 33, 34]. For example, the IDB relation of the `Project-manager` is written as follows using the notion of Datalog [30].

$$\texttt{Project-manager(ssn,}\cdots\texttt{)} \;:-\; \texttt{Employee(ssn,}\cdots\texttt{)}\;\&\;\texttt{Project(}\cdots\texttt{,manager-ssn,}\cdots\texttt{)}\;\&\;\texttt{ssn} = \texttt{manager-ssn}.$$

We use the relation-centered perspective throughout this thesis but the result is applicable to the object-centered perspective as well, particularly during execution (operationally).

2.4 Instantiating Objects from Relations through Views

Views provide a user-defined subset of a large database. Thus, as mentioned in Section 2.3, views are used as a tool for providing sharing and abstraction in interfacing between an object-oriented model and the relational model. We also want to use the views for instantiating objects from relations. To achieve this, views should provide mapping between heterogeneous structures of the two models. The mapping is done by linking object attributes to corresponding relation attributes. Objects have a more complex structure than relations. For instance, objects support aggregation hierarchies [86, 70] through an is-part-of relationship[3]. Hence objects have a nested structure, which is different from nested tuples because the type of an attribute can be a *reference* to another object. Therefore, given relation attributes, it is difficult to map the relation attributes to object attributes without explicitly specified mapping information. We thus need to extend the views by adding additional component for the mapping, that is, an *attribute mapping function.*

Figure 2.2 shows an example of instantiating objects through such an extended view. The object type defines the structure of objects to be retrieved from the database. The query part of the view, what we call a *view-query*, specifies how to materialize the objects from the relational database. The join between the `Employee` relation and the `Child` relation has the semantics of nesting such as "For each `Employee` tuple, retrieve the matching tuple in the `Child` relation." The outer relation is called a *source* relation and the inner relation is called a *destination* relation in our work. The attribute mapping part of the view shows the aggregation hierarchy of object attributes and their mapping to relation attributes. The mapping is one-to-one as long as there is no derived attribute among the object attributes. We use the *key* attribute of one of the relations as the source of the object identifier (oid). In Figure 2.2, the key `ssn` of the `Employee` relation is retrieved to become the oid of the `Employee`

[3]Objects also support a generalization hierarchy through is-a relationship, inheriting part of the attributes from parent objects. We regarded the inherited attributes as well as the local attributes uniformly as belonging to the objects.

Database schema: /* Underlined attributes are keys. */
Employee(ssn, e_name, sex, degree, salary, dept#)
Engineer(ssn, specialty, experience)
Department(dept#, d_name, manager_ssn, address)
Child(ssn, c_name, sex, birth_date)

Object Type Employee /* [ ] denotes a tuple. */
[name: string, dept: Department,
children: [name: string, birthDate: string]]

View:

- Query expressed in relational algebra:
  $\Pi_{\{ssn,e\_name,dept\#,c\_name,birth\_date\}}$ Employee $\underset{ssn=ssn}{\bowtie}$ Child
- Mapping between object attributes and relation attributes:

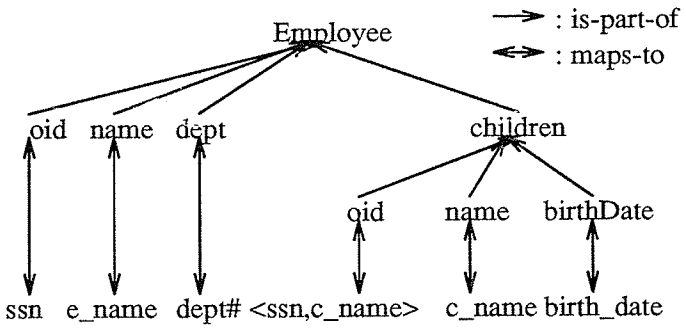


Figure 2.2: An example of instantiating an object type through views

object. Object id's are not explicitly defined in the type definition but assumed to exist implicitly. The `dept` attribute of an `Employee` object has type `Department`. We call an attribute whose type is another object type a *reference* attribute. In object-oriented paradigm, a reference is implemented with the oid of the referenced object. In our framework, the value of a reference attribute is retrieved from the key of a database relation which is mapped to the oid of the *referenced* object. Thus, in Figure 2.2, the `dept` attribute of an `Employee` object is retrieved from the `dept#` of the `Department` relation, if we assume that there exists a type `Department` whose object id is retrieved from the `dept#` of the `Department` relation. The `children` attribute defines a subobject of the `Employee` object, and each subobject has its own attributes – `name` and `birthDate`. Here a 'subobject' is defined as an object which does not have its own type definition but has its structure contained in another object which again may be a subobject of another object. Like the `Employee` object, a `children` subobject is assumed to have its object id, but the object id is not actually retrieved from a database relation. The id's of the `children` subobjects are needed for a different purpose, which will be discussed in Section 3.4.3.

2.5 Object Instantiation Time

The integration of objects and databases can be distinguished according to another dimension – the *binding time* [49, 50] of an object type. Given an object type, we define its binding time as the time when its instances are retrieved from databases into an application space.

A binding time can be distinguished into *early binding* and *late binding*. **Early binding** is a compiled approach. That is, all instances of an object type are retrieved all at once prior to the usage by an application program. In this sense, the early binding is similar to caching [57, 58] or prefetching [59]. Once all instances of an object type are retrieved, an application does not incur the cost of retrieving the instances of the same object type unless the retrieved instances are invalidated by the change of the data stored in databases. Early binding becomes a feasible idea if an application works in a canned transaction in which it is possible to preanalyze the

	Object storage	Base relation storage	
		Object-centered	Relation-centered
Early binding			View-objects
Late binding			

Table 2.1: View-object framework

set of objects that will be used by an application. On the other hand, there may be a situation in which the loading time for instantiating all instances of an object type is significant but this loading time does not pay off because the application does not use all the retrieved instances. In case only a small subset of the retrieved objects are used, **late binding** is more appropriate. Late binding is an interpreted approach. That is, instances of an object type are retrieved one at a time on demand during the execution of the application program. Late binding makes it possible for an application to retrieve only the objects that are actually needed during execution and hence takes less main memory space than early binding. However, if all the instances turn out to be used during the execution of an application, late binding strategy becomes worse than early binding by incurring as many object requests to databases as the number of used objects. Note that the early binding incurs the object request only once for a given object type as long as the retrieved instances remain valid.

From a system design point of view, we can think of a range of choice between the early binding and the late binding, i.e., between the compiled approach and the interpreted approach. This is analogous to the interpreted–compiled range (I-C range) in interfacing the Prolog with relational databases [51]. The criteria of choosing between the I-C range are the execution time and memory space. That is, ideally we want to retrieve the minimum number of objects that are needed by an application at the minimum number of object requests.

2.6 View-object Framework

2.6.1 View-objects

In [14], Wiederhold proposed database views as a tool for "connecting between *object* concepts in programming languages and *view* concepts in database systems". A view is defined by an external schema at the external level of the ANSI/SPARC architecture [25, 26]. Different groups of users can have different views on the same database. A view has been used as a mechanism for mapping between the different external schemas of different user views and the conceptual schema of the entire database in two ways. The goal of the view mechanism is twofold: *windowing* and *security*. Users access the same database through different 'windows' defined by different views. Query formulation is simplified by enabling a user to write a query as if a view were just another base relation. At the same time, users are restricted to access only a subset of a database, defined by a view[4]. The goal of windowing emphasizes using views as a tool for materializing a subset of data from relations, while the goal of security puts more emphasis on using views as a tool for managing a database system.

Wiederhold's proposal of view-objects put more emphasis on the goal of windowing, that is, using views as a tool for materializing view-objects from relations. A principal way of storing relations is to normalize them into nonredundant, unambiguously updatable form – Boyce-Codd-normal Form, for example. A materialized view is only in the first normal form and is closer to an 'object' in the sense that related attributes are brought together. For example, the view of the `Employee` object type in Figure 2.2 brings together, when materialized, the information about an employee and the information about the employee's children. Note that the attributes of an entity denoting a real world object are decomposed into the attributes of normalized relations in a database design process. We can say that a view is used to 'reassemble' the decomposed attributes into the attributes of the entity.

Objects that we are dealing with in this thesis are view-objects because the objects are instantiated by materializing a view. In our work, a view-object is a complex object which is implemented by a nested relation and supports references among objects. Table 2.1 illustrates where a view-object belongs to among the two-dimensional

[4]It is typical that a database administrator has the privilege of maintaining the security of a database system through this view mechanism by assigning views to each group of users.

categories of the framework that were discussed in Section 2.2 through Section 2.5. The view-object framework belongs to the relation-centered perspective of the relation storage approach. Early binding is assumed, that is, the results of a view-query are retrieved all at once into an application workspace and restructured into objects. The client-server architecture is appropriate for supporting the view-object framework [14]. In this architecture, a subset of the database content residing on a server is retrieved to a client workstation and used to provide objects (after necessary restructuring) during the execution of an application.

2.6.2 Related Work on View-objects

In [14], a *view-object generator* was proposed as an important component of the system implementing the view-object concept. Based on this proposal, Barsalou et al. [15] implemented a view-object generator in their Penguin project [20, 21, 22]. Besides, Cohen [16] implemented a different kind of view-object generator in his OB1 project.

2.6.2.1 Penguin

Penguin is an expert database system being built at the Stanford University for applications in the areas of biomedical engineering, civil engineering, and electrical engineering. In the Penguin project, Barsalou et al. implemented a view-object generator using a structural data model [13]. The structural data model is essentially a relational data model and is augmented with connections. The connections represent interrelational constraints such as referential integrity constraints and cardinality constraints. Barsalou et al. used an *object template* as a tool for formulating a view-query. An object template is a data structure with different attributes (or slots). Users formulate a view by designating a *pivot relation* [15] and selecting connections to follow among the connections to neighboring relations. For manipulating the *overlapping views* of multiple objects, the object templates are configured into a hierarchy. When an object needs to be instantiated, users select the corresponding object template and specify selection conditions on a set of relations defined in the object template. The system then formulates a SQL query and delivers it to the database. The query

result is restructured into view-objects using a NEST [68] procedure. At the time of this writing, a second prototyping of the Penguin project is still ongoing work at the Stanford University.

2.6.2.2 OB1

OB1 is a 'Prolog-based view-object-oriented database' designed and implemented at the David Sarnoff Research Center[5]. The goal of the OB1 project was to design and implement a Prolog-based hybrid system of relational databases and object-oriented databases. In OB1, Cohen designed a view-object manager and a direct object manager as a dual system. The purpose of the dual approach was to make it possible to move persistent data from relation storage to object storage back and forth. OB1 uses its own data definition and query language for the view-object manager. The query language is similar to SQL and can express a predicate of domain relational calculus within a query. In its implementation using Prolog, OB1 queries are translated into a Prolog goal and is executed by a standard Prolog execution mechanism. Unlike the Penguin view-object generator, no separate NEST procedure is necessary. The view-object manager materializes a nested relation directly out of relational databases.

[5] A subsidiary of SRI International

Chapter 3

Outer Joins and Filters in a View-Query

3.1 Introduction

In this chapter, we develop a mechanism for deciding on inner joins or outer joins, and prescribing non-null filters for a view-query. We first formulate our problem in a concrete manner in Section 3.2. Then, we develop a rigorous system model to facilitate the mapping between objects and relations in Section 3.3. The mechanism is developed in Section 3.4. A summary of this chapter follows in Section 3.5.

3.2 Problem Formulation

In this section we first introduce two operators: left outer join and non-null filters. Then, we formulate a problem by exaplaining the motivation, objective, and our approach to the problem.

3.2.1 The Two Operators

In Chapter 1, we mentioned the need for two operators for instantiating objects from relational databases through views: a left outer join and a non-null filter. A left outer

join is different from an inner join in that it retrieves null tuples when there is no matching tuple in the destination relation for a given source relation. A non-null filter is a selection condition for eliminating any nulls of an attribute from a base relation[1]. Formal definitions of the left outer join and the non-null filter are as follows.

Definition 3.2.1 (Left Outer Join) Given two relations R_1 and R_2, a left outer join from R_1 to R_2, denoted by $R_1 \;[\!\bowtie\; R_2$, is defined as follows.

$$R_1 \;[\!\bowtie\; R_2 = (R_1 \bowtie R_2) \cup ((R_1 - \Pi_{R_1}(R_1 \bowtie R_2)) \times \Lambda) \tag{3.1}$$

where $\bowtie$ denotes an inner join, $\pi_{R_1}(R_1 \bowtie R_2)$ denotes the projection of $R_1 \bowtie R_2$ on the attributes of R_1, and Λ denotes a null tuple consisting of nulls for all attributes of R_2. In other words, $R_1 \underset{A\theta B}{[\!\bowtie} R_2$ produces the following set of tuples.

$$\begin{aligned} &\{< t_1, t_2 > | t_1 \in R_1 \wedge t_2 \in R_2 \wedge t_1.A\theta t_2.B\} \cup \\ &\{< t_1, \Lambda > | t_1 \in R_1 \wedge \nexists t_2 (t_2 \in R_2 \wedge t_1.A\theta t_2.B)\} \end{aligned} \tag{3.2}$$

where θ denotes a comparison operator, i.e., $\theta \in \{<, \leq, >, \geq, =, \neq\}$.

For the rest of this chapter, we use a small size join symbol ($\ltimes\!\!\!\rtimes$) to denote a join which can be (has not yet been determined to be) either an inner join ($\bowtie$) or a left outer join ($[\!\bowtie$).

Definition 3.2.2 (Non-null filter) A non-null filter is a conjunction of predicates applicable to a base relation R, defined as follows.

$$R.A_1 \neq \text{null} \wedge R.A_2 \neq \text{null} \wedge \cdots \wedge R.A_i \neq \text{null} \tag{3.3}$$

where $A_1, A_2, \cdots, A_i$ are the attributes of R that are not allowed to have nulls.

3.2.2 Motivation

3.2.2.1 Why do we need left outer joins and non-null filters?

Objects are identified by their identifiers (oid's) only. In other words, an object exists even if all its attributes are nulls as long as it has an object id. Let us consider

[1]A base relation is the relation defined by the relation schema of a database, neither a view nor an intermediate relation.

the objects of typ `Employee` shown in Figure 2.2. An `Employee` object exists only if it has its oid retrieved from the `ssn` of the `Employee` relation. Assuming that the `Employee` object allows null for its `children` attribute, what will happen if the join between `Employee` relation and `Child` relation is evaluated by an inner join? Any employee tuple that has no matching tuple in the `Child` relation will be discarded. In other words, any employee without children will not be retrieved. Therefore, we must evaluate the join by an *outer join* to prevent the loss of employees that do not have children. Furthermore, what we need is not a bilateral outer join but a unilateral outer join, because we are not interested in retrieving a `Child` tuple that has no matching tuple in the `Employee` relation, that is, a child without parent. Therefore, a left outer join is adequate assuming that the source, here the `Employee`, relation is the left hand side operand of the join. We assume the source relation is always on the left hand side of a join and thus use only left outer joins for the rest of this chapter.

Now let us assume the `Employee` objects prohibit nulls for the `dept` attribute since a department affiliation is required of every employee. As mentioned in Section 2.4, the `dept` attribute is retrieved from the `dept#` of the `Employee` relation. The join between the `Employee` relation and `Child` relation is immaterial to the retrieval of `dept#` attribute. Rather, nulls of the `dept#` attribute stored in the tuples of the relation `Employee` should not be retrieved. Therefore, we must filter the `Employee` relation with a selection condition '`dept#` $\neq$ `null`'. We call this selection condition a *non-null filter*.

We see from the above examples that we frequently need left outer joins to prevent the loss of wanted objects, and non-null filters to prevent the retrieval of unwanted nulls.

3.2.2.2 Why do we want the system to do it?

Null-related semantics of object types are hard to understand and hence likely to induce errors. For example, the `Employee` type definition shown in Figure 2.2 does not distinguish between the semantics of 'employees and their zero or more children' and the semantics of 'employees with at least one child'. A left outer join is needed for the former while an inner join is needed for the latter. The distinction is entirely

the programmer's responsibility. Even if the semantics is clear, it is an effort for the programmer to determine the left outer joins and non-null filters given an object type and the corresponding view, especially if the view defines many joins. Therefore mechanization of the process is useful.

3.2.2.3 Why do we want to reduce the number of left outer joins and non-null filters?

The view-query is processed more efficiently if we can eliminate a non-null filter '$R.A \neq$ null' without affecting the query result, and thus avoid evaluating unnecessary selection conditions. Sometimes it is known at the semantic level that the column A of a relation R contains no null. An example is when A is the key of R and the entity integrity [38] is preserved.

The query also becomes more efficient if we reduce the number of left outer joins and still retrieve the same result. Sometimes left outer joins produce the same tuples as inner joins. For example in Figure 2.2, if every employee has one or more children, then the same tuples are produced by either join method. We know this fact at the semantic level, provided that the system enforces the referential integrity [38] from `Employee.ssn` to `Child.ssn`. As another example, let us consider the following directed join graph.

$$R_1 \longrightarrow R_2 \xrightarrow{\text{LO}} R_3 \longrightarrow R_4$$

where the join from R_2 to R_3 is a left outer join and the others are inner joins. If it is known that there always exists a matching tuple of R_3 for every tuple of R_2, then the result of $R_1 \bowtie R_2 \,[\bowtie\, R_3 \bowtie R_4$ is the same as $R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4$. Now, if we evaluate the join as an inner join, then the optimizer considers the three joins and will choose the most efficient order of joins. Let us assume the join order becomes $R_4 \rightarrow R_3 \rightarrow R_2 \rightarrow R_1$ in the optimal plan. On the other hand, if we evaluate the join as a left outer join, the query optimizer can not consider reversing the order of $R_2 \,[\bowtie\, R_3$ and thus can not obtain the same optimal plan. In general, converting a left outer join to an inner join allows the query optimizer to deal with a larger number of joins. This increases the number of alternative plans but will certainly

never generate less optimal plan than when left outer joins are evaluated as such and, therefore, cannot be reordered.

3.2.3 Problem Statements

Our objective is thus to develop a mechanism for the system to decide whether the joins of a query should be evaluated by inner joins or left outer joins when objects are instantiated from relational databases through views. In addition, the system decides which relations should be filtered through non-null filters. For efficiency reason, the number of left outer joins and non-null filters should be reduced whenever possible.

3.2.4 Our Approach

The heterogeneity of the object-oriented model and the relational model causes several difficulties in mapping between the two models [39]. Hence we cannot expect a simple solution to our problems without a well-defined system model. The system model should satisfy the following criteria.

- It provides the context in which we can develop a simple solution to the problem.
- It is based on a standard model and can be easily implemented in many existing systems.

Given the system model, we develop a mechanism for solving the problem. We use only one parameter that users should provide to the system. It is a *non-null option* on the object attribute as will be explained in Section 3.3.1. Users do not even have to know what a left outer joins is. To prevent losing nonmatching tuples when nulls are allowed (by default), all joins of a query are initialized to left outer joins. The semantics of the non-null options are interpreted as *non-null constraints*[2] on object attributes, and mapped to corresponding non-null constraints on the query result. Then we replace some left outer joins by inner joins and add non-null filters to some

[2]These constraints require the existence of an object attribute given the oid of an object. We would call this constraint as an *existence constraint* if this term were not already used in [30] to mean the same concept as the referential integrity.

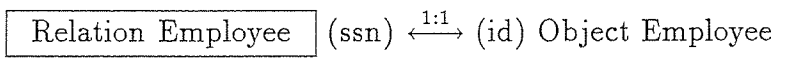
relations accordingly. Finally, the number of left outer joins and non-null filters are reduced using the integrity constraints of the data model.

The non-null options, and accordingly the non-null constraints, are used as the correctness criterion of the mechanism. Sometimes there appears to be a conflict in determing between a left outer join and an inner join. For example, let us consider two different attributes A and B that are projected from the same relation R. If A has a non-null constraint mapped from a non-null option on an object attribute but B does not have such a non-null constraint, then the join to the relation R must be an inner join for the non-null constraint on A to be satisfied while it does not have to be an inner join for B. In this case, we require the mechanism to make sure that no null value of A is retrieved, even if it also prevents null value of B from being retrieved, and hence determine the join to the relation R to be an inner join. In other words, the mechanism enforces the semantics of non-null options more strongly than the semantics of the default option, which allows nulls. We call this correctness criterion of the mechanism as a *non-null* correctness criterion.

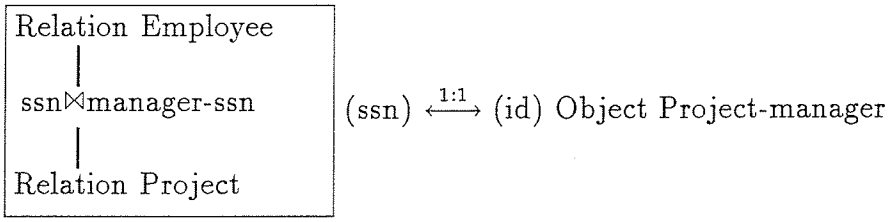
3.3 System Model

The system model has three elements: an object type model, a view model, and a data model. The object type model defines the structure of objects. No object type model has gained universal acceptance [40, 41]. Therefore we define a model which is common to many existing object-oriented models [1, 6, 8, 4, 5]. Note that we do not deal with methods, but focus only on object structures. The data model uses the relational model proposed by Codd [24]. The view model contains a relational query[3] and defines a mapping between objects and relations. We restrict the query to an *acyclic* select-project-join query with *conjunctive* join predicates.

[3]We do not assume the usage of any specific query language for our work.



(a) A pivot relation as a base relation



(b) A pivot relation as a derived relation

Figure 3.1: The concept of a pivot relation

3.3.1 Object Type Model

Many existing object-oriented models support *aggregation* through nested structures and references. For example, the `Employee` object of Figure 2.2 is an aggregation of `name`, `dept`, and `children` where `dept` is a reference to a `Department` object, and `children` is an aggregation of `name` and `birthDate`. The `children` attribute defines an embedded substructure of the `Employee` object. Thus our object type has a similar structure as the complex object [42, 43, 44].

We use value-oriented object id's [47, 48] and retrieve them from the keys of relations[4]. Those relations providing object id's are called *pivot relations* [15]. As discussed in Section 2.3, an object is mapped semantically to a derived relation rather than a base relation if no base relation provides the same semantics as the object type. Figure 3.1 illustrates these concepts. In Figure 3.1a, the `Employee` relation is the pivot relation for the `Employee` object and provides its key `ssn` as the object id. Figure 3.1*b* shows the derived relation `Project-manager` of Figure 2.1, which becomes the pivot relation for the `Project-manager` object. It is defined by $\texttt{Employee} \underset{\texttt{ssn=manager-ssn}}{\bowtie} \texttt{Project}$, and the key `ssn` of `Employee` in the join result is retrieved as the object id.

We do not consider derived attributes for our object type. Derived attributes have

[4]Tuple identifiers are usable as well. Otherwise we assume the system maintains a mapping between system-generated object id's and the keys of the corresponding relations.

no direct mapping to relation attributes and, therefore, are computed separately from relation attributes.

An object type is defined formally as a tuple of attributes, $[A_1, A_2, \cdots, X_1, X_2, \cdots]$ where each A_i is a simple attribute, and each X_i is a complex attribute. Each attribute is either local to the object or inherited from its parent, and we consider both the local and inherited attributes as 'defined' in an object type. An attribute is described in Backus-Naur Form as follows.

$$\left[\begin{array}{l} \text{attribute ::= simple attribute} \mid \text{complex attribute} \\ \text{simple attribute ::= internal attribute} \mid \text{external attribute} \\ \text{complex attribute ::= [attribute, attribute, } \cdots \text{]} \end{array}\right]$$

A *simple* attribute has an atomic value or a set of atomic values. It is either internal or external to the object. An *internal* attribute has a primitive data type such as string, integer, etc., while an *external* (or *reference*) attribute has another object type as its data type. The value of an external attribute is the oid of the referenced object. A *complex* attribute defines a subobject or a set of subobjects by embedding its type definition within the object type. In the same way as an object id is mapped from the key of a pivot relation, a subobject also has an associated oid which is mapped from the key of a base relation. However, the oid of a subobject is *not* retrieved while the oid of its (super)object is retrieved from the key of a pivot relation[5].

We need a way of telling the system whether the value of an object attribute is allowed to be null or not. This is done by attaching a *non-null* option to an object attribute. This option deliberately declares that a null value is not allowed for the attribute. It is equivalent to specifying the constraint of 'minimum cardinality > 0' on the attribute[6]. Attributes without non-null options are allowed to have null values by default.

An example is shown in Figure 3.2. The `Project` attribute defines its own attributes and becomes a subobject of the `Programmer` object. It has its object id

[5] A subobject of an object is not a stand-alone object because it has no object id.

[6] Many commercial tools for building object-oriented system applications, KEE[45, 46] for example, support this option.

```
Type Programmer
[ name: string non-null, dept: Department non-null, salary: integer,
  manager: Employee, task: string,
  Project: [ title: string non-null, sponsor: string, leader: string,
             depart: Department non-null ] ]
```

Figure 3.2: An example object type

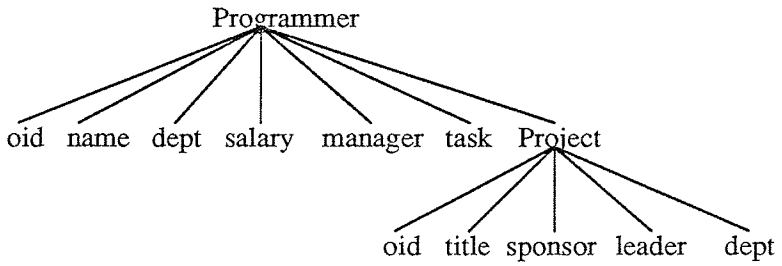


Figure 3.3: The O-tree of the Programmer object type

mapped from the key of a pivot relation in the same way the `Programmer` object does. However, only the id's of the `Programmer` objects are actually retrieved. This `Programmer` object example will be used throughout the rest of this chapter.

Given an object type, we can build a tree consisting of its object attributes. We call such a tree an *O-tree* and define it as follows.

Definition 3.3.1 (O-tree) The O-tree of an object O is a tree which has the following properties.

- Its root is labeled by 'O'.
- A leaf is labeled by a simple attribute of the object O.
- An intermediate node (non-leaf) is labeled by a complex attribute of the object O.

An example of an O-tree is shown in Figure 3.3 for the `Programmer` type.

Here we introduce two functions directly derivable from an object type: an *object set* (Oset) and an *object chain* (Ochain). These two functions are used to facilitate mapping between objects and relations.

Definition 3.3.2 (Oset) Given an object O, Oset(O) is defined as a function returning the set of the root of the O-tree and all of its *non-leaf* descendents.

For example, Oset(`Programmer`) returns {`Programmer`, `Project`}. Note that each element of an Oset has its object id mapped to the key of a pivot relation.

Definition 3.3.3 (Ochain) Given an object O and a simple attribute s_0 of the object O, Ochain(O, s_0) is defined as a function returning the chain of nodes from the root (O) of the O-tree to a descendent node labeled s_0, i.e., $O.O_1.\cdots.O_n.s_0$.

For example, Ochain(`Programmer`, `title`) returns `Programmer.Project.title` and Ochain(`Programmer`, `Project`) returns `Programmer.Project`.

3.3.2 Data Model

Integrity constraints [36, 37, 38] are a part of the data model[7]. Two kinds of integrity constraints are used in our work: referential integrity constraints and entity integrity coststraints. As mentioned in Section 3.2.2.3, these integrity constraints are useful to reduce the number of left outer joins and non-null filters.

The referential integrity constraint is defined as follows.

Definition 3.3.4 (Referential integrity constraint) A referential integrity constraint from $R.A$ to $S.B$ requires that if $R.A$ is not null then there exists a matching value of $S.B$. That is:

$$\forall a \in R.A(a = \text{null} \vee \exists b \in S.B(a = b)) \tag{3.4}$$

[7]In the Penguin project, which was introduced in Section 2.6.2.1, the connections of a structural data model provide the semantics of necessary integrity constraints, and therefore, integrity constraints need not be specified separately by a database designer.

Let us denote the referential integrity constraint by an arrow as in $R.A \mapsto S.B$.

Our definition of the entity integrity constraint is more extensive than the definition used in [38].

Definition 3.3.5 (Entity Integrity constraint) An entity integrity constraint requires one or more of the following conditions to be satisfied.

- Primary key constraint: $R.A \neq$ null if A is the primary key of R[8].
- Range constraint: If $R.A$ is not null then $a_1\theta_1\, R.A\, \theta_2 a_2$ where a_1, a_2 are non-null constants, and θ_1, θ_2 are '$<$' or '$\leq$'.
- Value constraint: $R.A = a$ or $R.A \neq a$ where a is a constant which may be null.

There can be other kinds of entity integrity constraint. For example, $R.A$ can have a *type constraint* such as 'the value of $R.A$ must be an integer'. However, those defined in Definition 3.3.5 are sufficient for our work. Figure 3.4 shows the schema, the referential integrity constraints and the entity integrity constraints of a sample database.

3.3.3 View Model

Figure 3.5 shows the components of the view model. A view consists of two parts: a query part and a mapping part. The mapping part in turn consists of an attribute mapping function (AMF) and a pivot description (PD). The AMF defines the mapping between object attributes (S_o) and relation attributes (S_r). The PD consists of a set of pivot relations (PS) and a pivot mapping function (PMF). The PMF defines the mapping between the pivot relations and the (sub)objects[9].

A high level language can be designed for defining a view. The view should be preprocessed to generate the mapping part as well as the query.

3.3.3.1 Query Part

[8] In [38], only this constraint is used as the entity integrity constraint.

[9] Or equivalently, between the keys of the pivot relations and the id's of the (sub)objects.

/* Underlined attributes are keys. */
Division(name, manager, super-division, location)
Dept(name, budget, phone#)
Emp(ssn, name, salary, dept)
Engineer(ssn, degree, specialty)
Proj-Assign(emp, proj, task)
Project(proj#, dept, leader, sponsor)
Sponsor(name, phone#, address)
Proj-Title(proj#, title)

(a) Database schema

/* $\mapsto$ denotes a referential integrity constraint. */

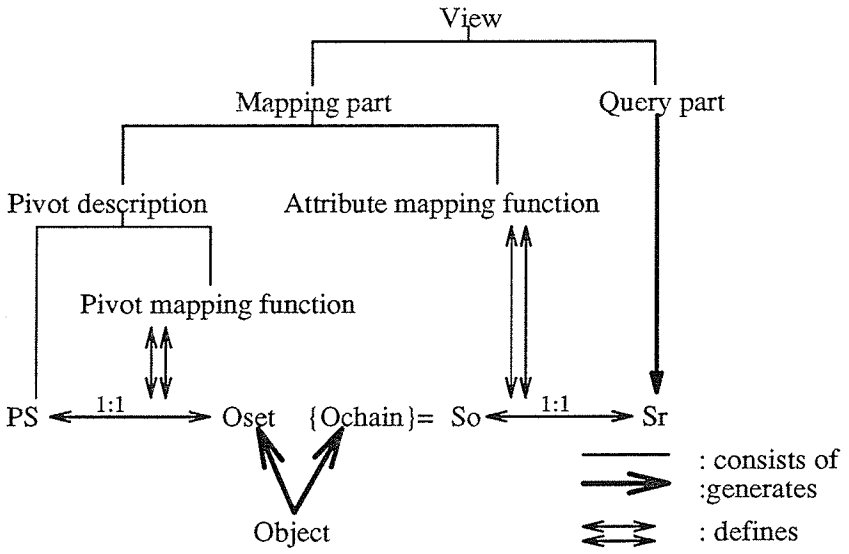
Division.manager $\mapsto$ Emp.name
Division.super-division $\mapsto$ Division.name
Dept.name $\mapsto$ Division.name
Emp.dept $\mapsto$ Dept.name
Engineer.ssn $\mapsto$ Emp.ssn
Proj-Assign.emp $\mapsto$ Engineer.ssn
Proj-Assign.proj $\mapsto$ Project.proj#
Project.dept $\mapsto$ Dept.name
Project.leader $\mapsto$ Emp.ssn
Project.sponsor $\mapsto$ Sponsor.name
Project-title.proj# $\mapsto$ Project.proj#

(b) Referential integrity constraints

The keys of all relations shown in the database schema are disallowed from having nulls. In addition, Emp.dept and Emp.name are prohibited from having nulls.

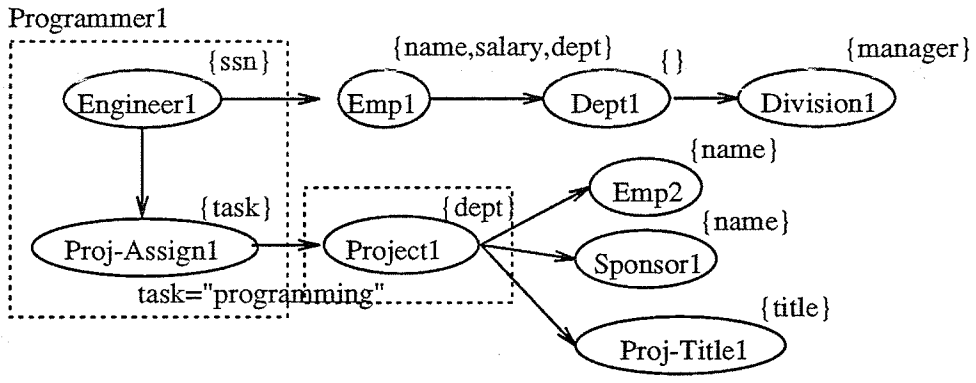
(c) Entity integrity constraints

Figure 3.4: A sample database



PS: the set of pivots Oset: object set Ochain: object chain
So: the set of Ochains of object attributes appearing in the object type
Sr: the set of relation attributes appearing in the query

Figure 3.5: Mapping between objects and relations



(The keys of Engineer1 and Project1 are mapped to the id's of the Programmer object and the Project subobject respectively. Dotted lines denote pivots.)

Figure 3.6: The query graph for the `Programmer` object

Figure 3.6 shows the query graph for the `Programmer` object. A query graph (QG) is a directed connected graph. Each vertex is represented by the node of a relation R labeled with a filter f and with the set of attributes π projected from R. Two occurrences of the same relation are distinguished by a tuple variable denoted as a subscript. Each edge represents a join specified in the query. A join is either an inner join or a left outer join. Since left outer joins are not symmetric, the edges are directed.

3.3.3.2 Mapping Part

Now we give a more rigorous description of the mapping part. The set of object attributes S_o of an object type O is represented as the set of Ochains as follows.

$$S_o = \{\text{Ochain}(O, s_0) | s_0 \in \text{Simple\_attr}(O)\}$$

where Simple_attr(O) denotes the set of *simple* attributes of an object type O. Ochain(O, s_0) was defined in Definition 3.3.3. The set of relation attributes S_r is defined as follows.

$$S_r = \{R.A | A \subseteq \text{Attr}(R)\}$$

where R is a relation occurrence in the query part of a view and Attr(R) denotes the set of attributes of R.

Since we assume no derived attribute, there exists a *one-to-one* mapping between S_o and S_r. This mapping information is contained in the attribute mapping function. The following example shows the mapping between the S_o and S_r of the `Programmer` object.

Example 3.3.1 (Attribute Mapping Function (AMF))

Programmer.name $\leftrightarrow$ $\texttt{Emp}_1$.name,
Programmer.dept $\leftrightarrow$ $\texttt{Emp}_1$.dept,
Programmer.salary $\leftrightarrow$ $\texttt{Emp}_1$.salary,
Programmer.manager $\leftrightarrow$ $\texttt{Division}_1$.manager,
Programmer.task $\leftrightarrow$ $\texttt{Proj-Assign}_1$.task,
Programmer.Project.title $\leftrightarrow$ $\texttt{Proj-Title}_1$.title,
Programmer.Project.sponsor $\leftrightarrow$ $\texttt{Sponsor}_1$.name,
Programmer.Project.leader $\leftrightarrow$ $\texttt{Emp}_2$.name,
Programmer.Project.depart $\leftrightarrow$ $\texttt{Project}_1$.dept
□

As shown in Figure 3.1, a pivot relation is either a base relation or a derived relation. If it is a base relation, its key is mapped to the object id. If it is a derived relation, the key of one of its base relations is mapped to the object id. For example, the query for the Programmer object has two pivot relations, $\texttt{Programmer}_1$ and $\texttt{Project}_1$. Here $\texttt{Project}_1$ is a base relation and $\texttt{Programmer}_1$ is a derived relation defined by $\langle\ \texttt{Engineer}_1,\ \{\texttt{Engineer}_1 \underset{\texttt{ssn=ssn}}{\bowtie} \sigma_{\texttt{task = ``programming''}}\ \texttt{Proj-Assign}_1\}\rangle$. A formal definition of a derived relation is as follows.

Definition 3.3.6 (Derived relation) A derived relation of an object type O is an ordered pair $\langle R_b, E\rangle$ where R_b is a base relation whose key is mapped to the oid of the object type O, and E is a *select-join*[10] expression such that, for arbitrary instances of the relations in E:

- $\Pi_{\text{Key}(R_b)} E \subseteq \Pi_{\text{Key}(R_b)} R_b$
- $\neg\exists E'(E' \neq E \wedge \Pi_{\text{Key}(R_b)} E' = \Pi_{\text{Key}(R_b)} E)$

That is, the result of evaluating E produces a subset of the keys available from R_b and there is no other select-join expression E' which, when evaluated, produces the same set of keys.

For every object and its subobject, there always exists one and only one relation occurrence whose key is mapped to the oid. In other words, there is a *one-to-one*

[10]Selection is not required while join is required.

mapping between the object set defined in Definition 3.3.2 and the set of pivot relations (PS). This mapping information is contained in the pivot mapping function. For example, the mapping between the Oset and PS of the `Programmer` object is as follows.

Example 3.3.2 (Pivot Mapping Function (PMF))

$\texttt{Programmer} \leftrightarrow \texttt{Programmer}_1$, $\texttt{Project} \leftrightarrow \texttt{Project}_1$

□

As mentioned in Section 3.3.1, we associate value-oriented object id's with an object and its subobjects. These oid's are invisible in the type definition and their mappings to relation attributes are not explicitly specified in the attribute mapping function. These mappings are derived from the information stored in the pivot description using the following algorithm.

Algorithm 3.3.1 (Mapping between oid's and relation attributes)
Input: Ochain, AMF without the mapping of oid's, PS, PMF.
Output: AMF with the mapping of oid's.

For each pivot relation $p \in$ PS begin
  If p is a base relation
    then append 'Ochain(O, PMF(p)).id $\leftrightarrow$ p.Key(p)' to AMF.
  else /* p is a derived relation */ begin
    Find the base relation R_b of p.
    Append 'Ochain(O, PMF(p)).id$\leftrightarrow$ R_b.Key(R_b)' to AMF.
  end.
end.

For example, given the set of pivot relations and the pivot mapping function of the `Programmer` view, Algorithm 3.3.1 derives the following mappings between the id's of the `Programmer` object and its `Project` subobject and their corresponding pivot relation keys.

Example 3.3.3 (Addition to AMF)

$\texttt{Programmer.id} \leftrightarrow \texttt{Engineer}_1\texttt{.ssn}$,

Programmer.Project.id $\leftrightarrow$ Project$_1$.proj#

□

These are appended to the AMF.

There is a constraint on the definition of the attribute mapping function. Let us consider two object attributes s_0 and s_1 which belong to the same node of an O-tree and their mapped relation attributes AMF(s_0) and AMF(s_1). Then AMF(s_0) and AMF(s_1) must either belong to the same relation or there exists a one-to-one cardinality relationship between them.

The attribute mapping function is essential for making it simple to map between objects and relations, as will be demonstrated in the following section.

3.4 Development of the Mechanism

Now we describe the mechanism for prescribing joins in a query as inner joins or left outer joins, and also for generating non-null filters for some relations in the query. We first present an overview of our mechanism, and then discuss each step in detail.

3.4.1 Overview

There are two sources of nulls retrieved from databases. One is from the nulls stored in the tuples, the other is from any outer join failure. Inner joins create nulls from the first source only, while outer joins create nulls from both sources. Objects allow nulls by default, and need only one kind of outer join, a left outer join, as explained in Section 3.2.2.1. Therefore our strategy is to initialize all joins of a query as left outer joins and then replace part of them by inner joins at each step of our mechanism.

The steps of our mechanism is as follows.

1. Compile the object type O and generate the object set (Oset) and the set of Ochain(O, s_0)'s for all the attributes defined in O.

2. Preprocess the view and generate the query and the mapping part – AMF, PMF, and PS.

3. Derive the mappings between object id's and the keys of pivot relations using Algorithm 3.3.1, and add the result to the attribute mapping function.

4. Initialize all joins of the query as left outer joins.

5. Replace all joins within derived relations by inner joins. (See Section 3.4.2.)

6. Map non-null options on object attributes to non-null constraints on the query result. Replace some joins by inner joins and add non-null filters to some relations accordingly. (See Section 3.4.3 and Section 3.4.4.)

7. Find the left outer joins which produce the same tuples as inner joins due to referential or entity integrity constraints, and replace those left outer joins by inner joins. Find also the relations whose non-null filtered attributes cannot have nulls due to entity integrity constraints, and remove the non-null filters from those relations. (See Section 3.4.5.)

3.4.2 Joins within a Derived Relation

As mentioned in Section 2.3, a derived relation is a conceptual relation defined from base relations via a select-join expression, and provides an abstraction of base relations so that the semantics of the derived relation matches the semantics of the instantiated objects.

All joins specified within a derived relation must be *inner* joins, as shown by the following theorem.

Theorem 3.4.1 Let us consider an object type O and a derived relation $\langle R_1, E\rangle$ defined according to Definition 3.3.6. If $E = R_1 \bowtie R_2 \bowtie \cdots \bowtie R_n$, then all the joins from R_1 through R_n are inner joins.

Proof: If we assume a join from R_i to R_{i+1} is a left outer join for an arbitrary $i \in [1, n]$ while the others are inner joins, then the following is true.

$$\begin{aligned} &\Pi_{\mathrm{Key}(R_1)}(R_1 \bowtie R_2 \bowtie \cdots \bowtie R_i \,[\bowtie\, R_{i+1} \bowtie \cdots \bowtie R_n) \\ &= \Pi_{\mathrm{Key}(R_1)}(R_1 \bowtie R_2 \bowtie \cdots \bowtie R_i) \end{aligned} \tag{3.5}$$

That is, there exists another select-join expression which, when evaluated, produces the same set of keys available from R_1. This violates the second condition required of E in Definition 3.3.6. Therefore, all the joins in E must be inner joins. Q.E.D.

For example, given a derived relation:

$\langle \texttt{Engineer}_1, \{\texttt{Engineer}_1 \underset{\texttt{ssn=ssn}}{\bowtie} \sigma_{\texttt{task='programming'}}\ \texttt{Proj-Assign}_1\}\rangle$

which is defined to provide the semantics of the `Programmer` object, the join between $\texttt{Engineer}_1$ and $\texttt{Proj-Assign}_1$ must be an inner join. If the join is evaluated as a left outer join, it retrieves all tuples of $\texttt{Engineer}_1$, not just those corresponding to programmers, who are defined as the engineers working on a programming task in the assigned projects.

Thus, given the set PS of pivot relations, we have the following algorithm.

Algorithm 3.4.1 (Joins within derived relations)
Input: query graph (QG) with all joins initialized as left outer joins, and the pivot set (PS).
Output: query graph (QG') modified with inner joins.

1. For each derived relation $\langle R_b, E\rangle$ in the set of pivot relations (PS),
   replace all joins in E by inner joins.

3.4.3 Mapping Non-null Options to Non-null Constraints on the Query Result

Let us consider an object O whose attribute s_0 has a non-null option. The non-null option requires that there should exist a non-null s_0 given the oid of the object. Let us denote this non-null constraint as $O.\text{id} \Rightarrow s_0$. If s_0 is a simple attribute, it is non-null if its value is not null. On the other hand if s_0 is a complex attribute, it defines a subobject. An object is non-null only if its oid is non-null. We thus interpret the semantics of non-null s_0 according to the following rule of non-null constraint.

Rule 3.4.1 (Non-null constraint) Let us consider $\text{Ochain}(O, s_o) \equiv O_0.O_1.\cdots.O_n.s_o$. If s_0 has a non-null option then, given O_n.id,

- If s_0 is a simple attribute, i.e., $O_n.\text{id} \Rightarrow s_0$, then s_0 cannot be null.

- If s_0 is a complex attribute, i.e., $O_n.\mathrm{id} \Rightarrow s_0.\mathrm{id}$, then $s_0.\mathrm{id}$ cannot be null.

For example, given the `Programmer` object of Figure 3.2, the non-null options on `name` and `dept` attributes are interpreted as `Programmer.id` ⇒ `name` and `Programmer.id` ⇒ `dept`, respectively, because `name` and `dept` are simple attributes. Besides, the non-null options on `title` and `depart` are interpreted as `Project.id` ⇒ `title` and `Project.id` ⇒ `depart`, respectively. Beware that they are *not* interpreted as `Programmer.id` ⇒ `title` and `Programmer.id` ⇒ `depart` because `title` and `depart` are the (direct) attributes of `Project` subobject instead of the `Programmer` object. On the other hand, if there were a non-null option on `Project`, it would be interpreted as `Programmer.id` ⇒ `Project.id` because `Project` is a complex attribute.

Can we map the non-null constraint defined by Rule 3.4.1 to the corresponding non-null constraint on the query result? This is possible in our model because the oid of each (sub)object always has a corresponding pivot relation key. The attribute mapping function in Example 3.3.1 showed this correspondence for the `Programmer` object. Using the correspondence, the non-null constraints on the `name` and `dept` attributes of the `Programmer` object are mapped to $\mathtt{Engineer}_1\mathtt{.ssn} \Rightarrow \mathtt{Emp}_1\mathtt{.name}$ and $\mathtt{Engineer}_1\mathtt{.ssn} \Rightarrow \mathtt{Emp}_1\mathtt{.dept}$, respectively. Likewise, if `Project` had the non-null option, its constraint would be mapped to $\mathtt{Engineer}_1\mathtt{.ssn} \Rightarrow \mathtt{Project}_1\mathtt{.proj\#}$. The non-null option on the `title` attribute is mapped not to $\mathtt{Engineer}_1\mathtt{.ssn} \Rightarrow \mathtt{Proj\text{-}Title}_1\mathtt{.title}$ but to $\mathtt{Project}_1\mathtt{.proj\#} \Rightarrow \mathtt{Proj\text{-}Title}_1\mathtt{.title}$ because `title` is defined not as an attribute of `Programmer` object but as an attribute of `Project` subobject. For the same reason, the non-null option on the `depart` attribute of `Project` is mapped to $\mathtt{Project}_1\mathtt{.proj\#} \Rightarrow \mathtt{Project}_1\mathtt{.dept}$.

More formally, a non-null option on the attribute s_0 of an object type O is translated into the non-null constraint on the query result as follows.

Algorithm 3.4.2 (Mapping non-null options)
Input: an object attribute s_0 with a non-null option, attribute mapping function (AMF), object chain (Ochain).
Output: a non-null constraint on the query result.

1. $\Omega_{0,n}.s_0 := \mathrm{Ochain}(O, s_0) \equiv O_0.O_1.\cdots.O_n.s_0$.

2. $R_p.A :=$ AMF($\Omega_{0,n}$.id). /* A is always the key of R_p. */

3. If s_0 is a simple attribute
   then $R_s.B :=$ AMF($\Omega_{0,n}.s_0$)
   else $R_s.B :=$ AMF($\Omega_{0,n}.s_0$.id). /* If s_0 is a complex attribute, B is the key of R_s. */

4. Output the constraint '$R_p.A \Rightarrow R_s.B$'.

3.4.4 Prescribing Joins and Generating Non-null Filters

With the non-null constraints on the query result, we translate them into the corresponding inner joins and non-null filters of the query. Given the constraint '$R_p.A \Rightarrow R_s.B$' obtained from Algorithm 3.4.2, it is done as follows.

Algorithm 3.4.3 (Translating non-null constraints)
Input: query graph QG′ from Algorithm 3.4.1, a non-null constraint $R_p.A \Rightarrow R_s.B$
Output: query graph QG″ modified with inner joins and a non-null filter.

1. Replace the filter f_s on R_s by $f_s \wedge (B \neq \text{null})$. /* Generate a non-null filter. */

2. /* Prescribe a join. */

   (a) Find all directed join paths from R_p to R_s.

   (b) For each path found in Step 2a,
   replace all joins on the path by inner joins.

For example, given the non-null constraints established in Section 3.4.3, the following non-null filters are generated in the query of the `Programmer` object: $\texttt{Emp}_1\texttt{.name} \neq \texttt{null}$, $\texttt{Emp}_1\texttt{.dept} \neq \texttt{null}$, $\texttt{Project}_1\texttt{.dept} \neq \texttt{null}$, $\texttt{Proj-Title}_1\texttt{.title} \neq \texttt{null}$. Besides, the following left outer joins are replaced by inner joins: $\texttt{Engineer}_1$ [⋈ $\texttt{Emp}_1$, $\texttt{Project}_1$ [⋈ $\texttt{Proj-Title}_1$.

Now we prove the correctness of Algorithm 3.4.3 with the following theorem, base on non-null the correctness criterion explained in Section 3.2.4.

Theorem 3.4.2 Given a join path $R_1 \bowtie R_2 \bowtie \cdots \bowtie R_n$ and a non-null constraint $R_1.A_1 \Rightarrow R_n.A_n$ on the join result, the materialized join result satisfies this non-null constraint if and only if all the joins are inner joins and R_n is filtered by $A_n \neq$ null.

Proof:
If part: If all joins on the join path are inner joins, any nonmatching tuples are discarded. Then, the attribute A_n in the join result can have nulls only if A_n is *not* a join attribute and some tuples of R_n have null A_n. (If it is a join attribute, any tuple of R_n with null A_n is discarded by an inner join.) However, tuples with null A_n are removed from R_n by the given non-null filter. Therefore the constraint is satisfied.
Only if part: We prove this part by contradiction. Let us first assume $R_i \bowtie R_{i+1}$ is a left outer join for some i although the constraint is satisfied and let R_{i+1} have non-matching tuples. Then a null $R_n.A_n$ is retrieved from the null tuples appended to the tuples of R_i which have no matching tuples in R_{i+1}. This contradicts the assumed constraint. Therefore all the joins must be inner joins. Next, let us assume R_n is *not* filtered by $A_n \neq$ null although the constraint is satisfied and all joins are inner joins. Then null $R_n.A_n$ is retrieved from the nulls *stored* in $R_n.A_n$ if A_n is not a join attribute. This contradicts the assumed constraint. Q.E.D.

3.4.5 Reducing the Number of Left Outer Joins and Non-null Filters

We can remove unnecessary non-null filters and further reduce the number of left outer joins by using integrity constraints.

3.4.5.1 Removing Unnecessary Non-null Filters

Considering entity integrity constraints, some non-null filters are removed if they are defined on attributes which cannot have null. A typical case is when the attribute is a key (primary key constraint) or any other non-null attribute designated in the schema definition (value constraint). For example, we can remove $\texttt{Emp}_1\texttt{.name} \neq \texttt{null}$ and $\texttt{Emp}_1\texttt{.dept} \neq \texttt{null}$ among the four non-null constraints generated in Section 3.4.4

because, as it was shown in Figure 3.4c, those two attributes are key attributes and hence prohibited from having nulls.

3.4.5.2 Further Reducing the Number of Left Outer Joins

We can also replace some left outer equijoins by inner equijoins if we consider referential integrity constraints. Since a referential integrity $R.A \mapsto S.B$ allows $R.A$ to be null, we define a stronger condition by introducing a variable `min` as follows.

Definition 3.4.1 (`min`) Given a join $R_i \bowtie R_j$, let $\texttt{min}_{ij}$ denote the minimum number of matching tuples in R_j for each tuple in R_i. Note $\texttt{min}_{ij}$ is not necessarily the same as $\texttt{min}_{ji}$.

Besides, some left outer non-equijoins can be replaced by inner non-equijoins if we consider entity intergrity constraints such as range constraints.

Using only the semantics of `min` without considering the instances of relations[11], we define the following rules for deciding whether `min` is greater than zero or not. MIN($R.A$) denotes the minimum non-null value allowed for $R.A$, and MAX($R.A$) denotes the maximum non-null value allowed for $R.A$. MIN($R.A$) and MAX($R.A$) are known from the range constraints or value constraints, if there are any, on $R.A$.

Rule 3.4.2

- Given a single join predicate $A\theta B$ for the join between two relations R_i and R_j, $\texttt{min}_{ij} > 0$ if $R_i.A$ is a non-null attribute and one or more of the following conditions are satisfied.

  $\theta \equiv$ '$=$' and $R_i.A \mapsto R_j.B$ and the filter f_j on R_j is empty, or

  $\theta \equiv$ '$>$' and MIN($R_i.A$) $>$ MAX($R_j.B$), or

  $\theta \equiv$ '$\geq$' and MIN($R_i.A$) $\geq$ MAX($R_j.B$), or

  $\theta \equiv$ '$<$' and MAX($R_i.A$) $<$ MIN($R_j.B$), or

  $\theta \equiv$ '$\leq$' and MAX($R_i.A$) $\leq$ MIN($R_j.B$), or

[11] In other words, we ignore the fact that `min` may be accidentally greater than zero at the instance level although it is judged to be equal to zero at the semantic level.

$$\theta \equiv \text{`}\neq\text{'} \text{ and } (\text{MIN}(R_i.A) > \text{MAX}(R_j.B) \text{ or } \text{MAX}(R_i.A) < \text{MIN}(R_j.B)).$$

Otherwise $\texttt{min}_{ij} = 0$[12].

- Given a *conjunctive* join predicate $A_1\theta_1B_1 \wedge A_2\theta_2B_2 \wedge \cdots \wedge A_k\theta_kB_k$ for the join between R_i and R_j, $\texttt{min}_{ij} > 0$ for the conjunction of join predicates if $\texttt{min}_{ij} > 0$ for *every* single join predicate. Otherwise $\texttt{min}_{ij} = 0$.

- Given a *disjunctive* join predicate $A_1\theta_1B_1 \vee A_2\theta_2B_2 \vee \cdots \vee A_k\theta_kB_k$ for the join between R_i and R_j, $\texttt{min}_{ij} > 0$ for the disjunction of join predicates if $\texttt{min}_{ij} > 0$ for *at least one* join predicate. Otherwise $\texttt{min}_{ij} = 0$.

- Given a join path between two relations, such as $R_i \bowtie R_{i+1} \bowtie \cdots \bowtie R_j$, $\texttt{min}_{ij} > 0$ if $\texttt{min}_{k,k+1} > 0$ for $k = i, \cdots, j-1$. Otherwise $\texttt{min}_{ij} = 0$.

If $\texttt{min}_{ij} > 0$ for a join path from R_i though R_j, we can replace all joins on the path by inner joins and still get the same query result.

Now we describe an algorithm for reducing the number of left outer joins using min. We also show the step of removing unnecessary non-null filters in the following algorithm.

Algorithm 3.4.4 (Integrity-based reduction of left outer joins and non-null filters)

Input: query graph (QG″) produced by Algorithm 3.4.3, and integrity constraints.
Output: modified query graph (QG‴).

1. Remove '$R.A \neq$ null' such that A is a non-null attribute.

2. Find all join paths between pairs of nodes, such as R_i and R_j, whose $\texttt{min}_{ij} > 0$.

3. For each join path found in Step 1,
   replace all joins on the path with inner joins.

[12] $\texttt{min}_{ij} = 0$ does not mean that $\texttt{min}_{ij}$ is *always* equal to zero. Rather, it means that it is not known at the *semantic level* whether $\texttt{min}_{ij}$ is greater than zero.

As an example, in the query of `Programmer` object, we find a join path from $\texttt{Engineer}_1$ to $\texttt{Division}_1$ for which all three joins have `min` > 0. This is because, as shown in Figure 3.4, (1) there are referential integrities $\texttt{Engineer}_1\texttt{.ssn} \mapsto \texttt{Emp}_1\texttt{.ssn}$, $\texttt{Emp}_1\texttt{.dept} \mapsto \texttt{Dept}_1\texttt{.name}$, $\texttt{Dept}_1\texttt{.name} \mapsto \texttt{Division}_1\texttt{.name}$, (2) there are integrity constraints prohibiting nulls for $\texttt{Engineer}_1\texttt{.ssn}$, $\texttt{Emp}_1\texttt{.dept}$, and $\texttt{Dept}_1\texttt{.name}$, and (3) none of the relations on the join path has a non-empty filter. We also find a join path from $\texttt{Proj-Assign}_1$ to $\texttt{Project}_1$ for which the `min` > 0. All these joins are replaced by inner joins. Note $\texttt{Project}_1 \boxed{\bowtie} \texttt{Emp}_2$ and $\texttt{Project}_1 \boxed{\bowtie} \texttt{Sponsor}_1$ cannot be replaced by inner joins because `Project.leader` and `Project.sponsor` are not non-null attributes.

3.4.6 Summary of the Mechanism

Given a query with initial left outer joins, the overall mechanism developed in Section 3.4 is as follows.

Algorithm 3.4.5 (Summary)
Input: object type O, view (query part and mapping part), relations and integrity constratins.
Output: the query part prescribed with inner joins, left outer joins, and non-null filters.

1. /* Preprocessing */

   (a) Compile the object type O and generate the object set (Oset) and the set of Ochain(O, s_0)'s for all the attributes defined in O.

   (b) Generate the query and the mapping part, AMF, PMF, and PS, from the view.

   (c) Derive the mappings between object id's and the keys of pivot relations using Algorithm 3.3.1, and add the result to the attribute mapping function.

   (d) Initialize all joins of the query as left outer joins.

2. /* Replace all joins within derived relations with inner joins. */
   For each derived relation $\langle R_b, E \rangle$ in the set of pivot relations (PS),
   replace all joins in E by inner joins.

3. For each attribute s_0 of the object O that has a non-null option,

   (a) /* Map the non-null option to a non-null constraint on the query result */

   i. $\Omega_{0,n}.s_0 := \text{Ochain}(O, s_0) \equiv O_0.O_1.\cdots.O_n.s_0$.

   ii. $R_p.A := \text{AMF}(\Omega_{0,n}.id)$. /* A is always the key of R_p. */

   iii. If s_0 is a simple attribute
   then $R_s.B := \text{AMF}(\Omega_{0,n}.s_0)$
   else $R_s.B := \text{AMF}(\Omega_{0,n}.s_0.\text{id})$. /* If s_0 is a complex attribute, B is the key of R_s. */

   iv. Output the non-null constraint '$R_p.A \Rightarrow R_s.B$'.

   (b) /* Generate a non-null filter and prescribe a join. */

   i. Replace the filter f_s on R_s by $f_s \wedge (B \neq \text{null})$. /* Generate a non-null filter. */

   ii. /* Prescribe a join. */

   A. Find all directed join paths from R_p to R_s.

   B. For each path found in Step 3(b)iiA,
   replace all joins on the path by inner joins.

4. /* Using the integrity constraint, remove all non-null filters which can be shown to be redundant, and replace left outer joins if they prove to be equivalent to inner joins. */

   (a) Remove '$R.A \neq \text{null}$' such that A is a non-null attribute.

   (b) Find all join paths between pairs of nodes, such as R_i and R_j, whose $\texttt{min}_{ij} > 0$.

   (c) For each join path found in Step 4b,
   replace all joins on the path with inner joins.

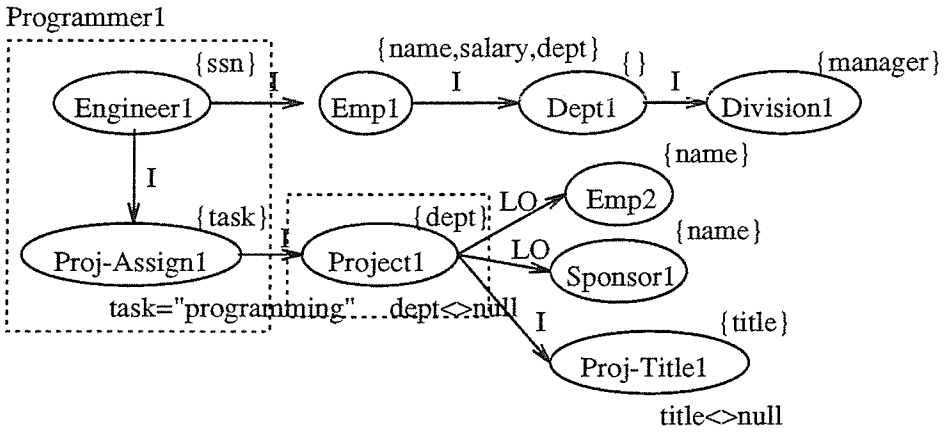


Figure 3.7: The query graph for the `Programmer` object with joins and non-null filters

The graph of the query for the `Programmer` object, labeled with joins and non-null filters, is shown in Figure 3.7. All the joins of the query except those between $\texttt{Project}_1$ and $\texttt{Emp}_2$ and between $\texttt{Project}_1$ and $\texttt{Sponsor}_1$ have been prescribed as inner joins. Two non-null filters have been attached as the selection conditions on the $\texttt{Project}_1$ and $\texttt{Proj-Title}_1$ nodes.

3.5 Summary

We developed a mechanism for automatically prescribing inner or left outer joins for the joins of a query used to instantiate objects from a relational database. It also generates non-null filters for some of the relations in the query. We developed a rigorous system model that facilitates the mapping between objects and relations. The system model consists of an object type model, a view model, and a relational data model. These models are based on a standard model or well-known models. We added a few new components to the object type model and view model. These components are easily implementable in existing systems.

The mechanism deals with an acyclic query graph. At first, all joins in the query graph are initialized to left outer joins. Then the following joins are replaced by inner joins: 1. Joins within a derived relation to provide the semantics of the object type;

2. *all* joins lying on the path from a pivot key to an attribute which has a non-null constraint on the query result. The non-null constraint is mapped from a non-null option on an object attribute and enforces the semantics of the non-null option at the relation level. Besides, non-null filters are generated for a relation attribute which has non-null constraints on it. Finally. the number of left outer joins and non-null filters is reduced whenever possible using the integrity constraints so that the query is processed more efficiently.

Our result demonstrates how simple the mechanism becomes once the system model is established. The only criterion for the mechanism to use is the non-null option on object attributes, whose semantics is mapped to the non-null constraint on the query result. The system uses the non-null correctness criterion to make sure that the semantics of a non-null option is preserved, even if it prevents other attributes without non-null options from having null values as well. The developed mechanism covers all possible cases under the non-null correctness criterion.

Chapter 4

Efficiently Instantiating Objects

4.1 Introduction

This chapter addresses the second problem of this thesis, which is to improve the performance of retrieving objects from a relational database residing on a remote server. The key idea of the performance improvement is to reduce the amount of redundant data that the system should handle in order to instantiate objects. We first formulate our concrete problem in Section 4.2. Secondly, we develop three different object instantiation methods in Section 4.3. One is the conventional method of retrieving a query result in the form of a single flat relation (table). The other two methods retrieve a query result in structures that are different from, and less redundant than, a single flat relation. Thirdly, we develop the cost models of the three different object instantiation methods in Section 4.4, and compare their costs in Section 4.5. It is followed by a summary of this chapter and a discussion of future work in Section 4.6.

4.2 Problem Formulation

4.2.1 Environment: a Remote Main Memory Database Server

The client-server architecture is becoming a standard architecture of modern computing environment by virtue of the recent development of high-speed computer network

technology. Typically multiple clients and servers work in a *request-response* mode, that is, clients make requests to servers and servers make responses to the clients. The concept of a remote database server stemmed from this concept, where requests are database queries and responses are query results.

In the earlier stage of the client-server architecture, network communication cost has been a major performance concern in accessing a remote database server. However, this concern is becoming less meaningful these days because the communication cost decreases rapidly since the advent of the high-speed computer network technology such as Ethernet local area network (LAN) [64, 63] and NSF wide area network (WAN) [65, 66]. Rather, the dominant cost of accessing a remote database server is the cost of query materialization on the server, disk access cost in particular. This statement is true as long as databases reside on a secondary storage device. Nowadays however, the number of applications running on main memory databases [76, 71, 72, 79, 73] is increasing as high density main memory chips are becoming available at a lower cost. Here, a main memory database indicates that the entire database or an actively used subset of a database fits within main memory at the same time. (According to [60], "approximately 50 – 75 % of all disk accesses occur on data stored on 2 – 3 % of the disk media".) If a main memory database is used, the disk access cost disappears or is incurred rarely, and hence the CPU cost and the network communication cost become dominant.

Considering all these facts, our work assumes the environment in which clients access remote database servers where databases are relational and stored in main memory. We assume a situation in which practically infinite main memory is available so that no disk access is necessary at all during the entire object instantiation process. Here we emphasize that we assume the availability of large main memory as the environment which can benefit most from our work. The usefulness of the result of our work is *not* restricted to main memory database systems.

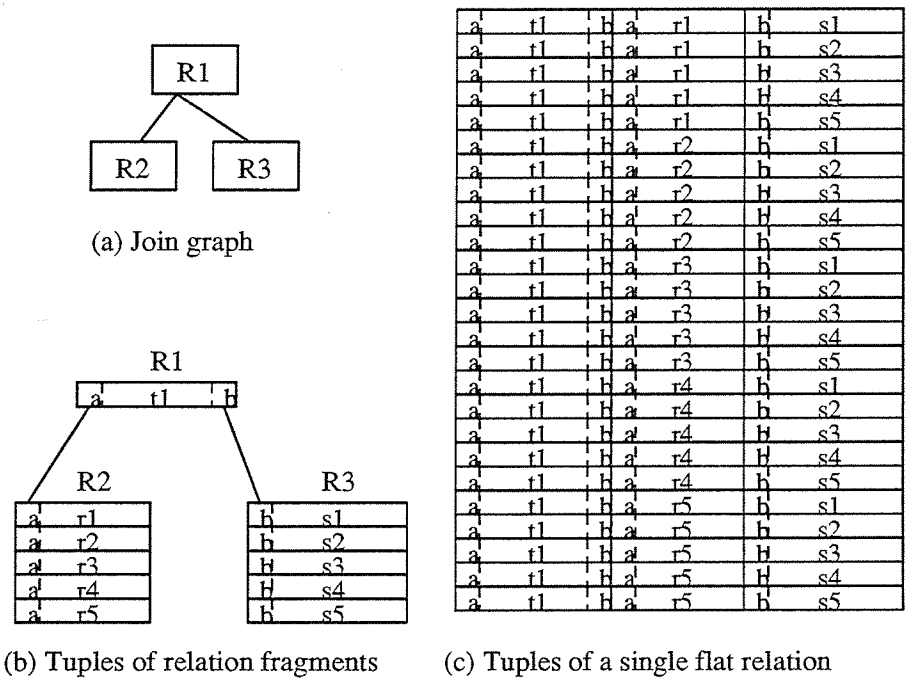


Figure 4.1: Duplicate subtuples

4.2.2 Motivation: Redundant Subtuples of a Single Flat Relation

There are two kinds of redundant subtuples in the composite tuples of a query result retrieved in the form of a single flat relation: *duplicate subtuples* and *null subtuples.*

A single flat relation contains duplicate subtuples among the composite tuples. For example, let us consider a query whose join graph is shown in Figure 4.1a. Figure 4.1b shows five matching tuples $r_i, i = 1, 2, \cdots, 5$ from R_2 and five matching tuples $s_i, i = 1, 2, \cdots, 5$ from R_3 for a tuple t_1 from R_1. Once those matching tuples are concatenated into composite tuples of Figure 4.1c, t_1 is duplicated 25 times and each of $r_i, i = 1, 2, \cdots, 5$ are duplicated 5 times, just to make the query result 'flat'.

As we discussed in Chapter 3, left outer joins are frequently needed to instantiate objects from relational databases through views [61]. If there are *outer joins* in the query, the materialized single flat relation will contain *null* subtuples for any

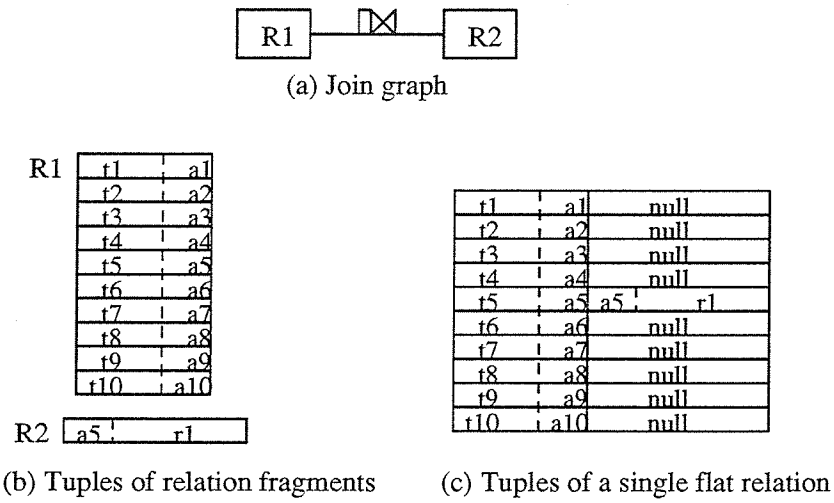


Figure 4.2: Null subtuples

nonmatching subtuples. Figure 4.2 shows an example case. Given a left outer join from R_1 to R_2, there is only one matching tuple r_1 from R_2. However, the semantics of the left outer join requires that null tuples should be inserted in place of R_2 tuples for any dangling tuples[1] of R_1. Therefore, there appear nine null subtuples in Figure 4.2c. The amount of null subtuples become significant if an outer join is followed by other (inner or outer) joins.

Those duplicate subtuples and null subtuples are inserted just to make the relation flat and do not carry any additional information. Moreover, they cause some disadvantages compared to the case they are not materialized. First of all, redundant subtuples incur the cost of materializing them. Besides, redundant subtuples increase the amount of transmitted data and thus increase the communication cost over network without conveying any more information. Furthermore, the flat relation must be restructured into a nested relation to become usable as objects. In other words, we materialize and transmit redundant subtuples which are destined to be eliminated in a restructuring process. Thus, the duplicate subtuples and null subtuples incur the additional cost of materializing them, transmitting them, and eliminating them

[1]That is, tuples which do not have any matching tuples in R_2.

without any benefit compensating for these costs. This observation motivated us to look for alternative methods which do not return a single flat relation as a query result and thus enable us to avoid retrieving duplicate or null subtuples.

4.2.3 Problem Statements

Our purpose is to develop alternative methods of instantiating objects from relational databases through views, and compare their costs to demonstrate that the alternative methods are more efficient than the conventional method of retrieving a single flat relation. Our major cost measure is the execution time. Required main memory space is another important measure of cost. Main memory space is taken into consideration in the development of object instantiation methods. However, cost comparison is carried out using time as the only cost measure.

Queries are restricted to acyclic select-project-conjunctive join queries, in the same way as in Chapter 3. For simplicity, we consider only *inner joins* in a query and do not consider any left outer joins for the rest of this chapter. This simplification indeed simplifies the developed algorithms and cost comparison tasks. Nevertheless these simplifications does not prevent us from demonstrating that the alternative methods are more efficient than the conventional method, as explained now. The semantics of a left outer join is as follows.

$$R_1 \text{ [}\bowtie\text{] } R_2 = (R_1 \bowtie R_2) \cup ((R_1 - R_1 \ltimes R_2) \times \Lambda) \tag{4.1}$$

where Λ is a null tuple of R_2, i.e., a tuple consisting of nulls for each column of R_2 and $\ltimes$ denotes a semijoin. Equation 4.1 says that tuples produced from a left outer join from R_1 to R_2 is equal to the tuples produced from an inner join plus the concatenation of the tuples of R_1 which do not have matching tuples in R_2 and a null tuple of R_2. As mentioned in Section 4.2.2, inner joins are the source of duplicate subtuples while outer joins are the source of null subtuples contained in a single flat relation query result. Therefore, a query without outer joins produces only duplicate subtuples in its single flat relation result while a query with outer joins produces null subtuples as well as the duplicate subtuples. Therefore, if the new methods that will be introduced in this thesis are more efficient than the conventional method when we

consider inner joins only, they are even more efficient for a query with outer joins as well.

4.2.4 Our Approach

First, we describe three different object instantiation methods. One is the conventional method of retrieving a query result in the form of a single flat relation and is called the SFR method. The second method retrieves a query result as a set of relation fragments, and is called the RF method. Relation fragments are materialized from base relations by reducing them with selection, projection, and join operations. Relation fragments should contain all information required for restructuring the relation fragments into a single nested relation. The third method retrieves a query result as a single nested relation and is called the SNR method. A single nested relation is a set of nested tuples, in which an attribute can define another relation.

Then, we develop the cost models of the three different object instantiation methods and compare their costs. In the client-server architecture, the object instantiation cost is the sum of local processing cost and transmission cost. The local processing cost is the total execution time spent on a server and a client. The transmission cost is the time required to send a query result to a client over communication network. Obviously, the transmission cost is more significant in the WAN environment than in the LAN environment. Since our purpose is to *compare* the costs of different methods rather than to estimate the costs precisely, the cost items common to all three methods are excluded from our cost model. Besides, we make necessary simplifications as long as the simplifications do not invalidate the cost comparison result.

4.3 Development of Object Instantiation Methods

In this section, we first give an overview of the SFR, RF, and SNR method, and then give a detailed description of each step of the three methods. Since our objective is to show that the RF method and the SNR method are more efficient than the SFR

method, we make the SFR method as efficient as possible to avoid any bias in favor of the RF method or the SNR method. As will be explained, the SNR method is essentially the same as the RF method except that nesting step is carried out by a server. Therefore, we first focus on the SFR and RF method in Section 4.3.2 through Section 4.3.3, and then discuss the SNR method separately in Section 4.3.4.

4.3.1 Overview of the Three Object Instantiation Methods

We give here a brief, abstract overview of the object instantiation steps of three different methods: the SFR method, RF method, and SNR method, focusing on the distinction among the methods. First we describe the processes of each step of the three methods and show an example of a single flat relation, relation fragments, and a single nested relation to help readers understand the distinction among them.

4.3.1.1 Overview of the Processes

Figure 4.3 illustrates the processes of the three object instantiation methods. The overall process is divided into three phases: *materialization*, *transmission*, and *translation*. The process of each phase is different for each method.

SFR method: A query is materialized into a single flat relation by a server, transmitted as such, and is translated into objects by a client. Translation is done in two steps: nesting and reference resolution. In the nesting step, a retrieved single flat relation is restructured into a nested relation by our implementation of the NEST [68] operator. The reference resolution step is needed to resolve references among objects, thus configuring the retrieved objects into a network of references.

RF method: A query is materialized into a set of relation fragments by a server, transmitted as such, and is translated into objects by a client. As in the SFR method, translation is done in two steps, nesting and reference resolution, but a different process is used for the nesting step due to the different structure of retrieved data. Since a client receives no separate information for linking tuples among relation fragments, the first thing for a client to do is to create necessary linkage information. In our work, it is done by creating indexes on join attributes. Once indexes are created, joins

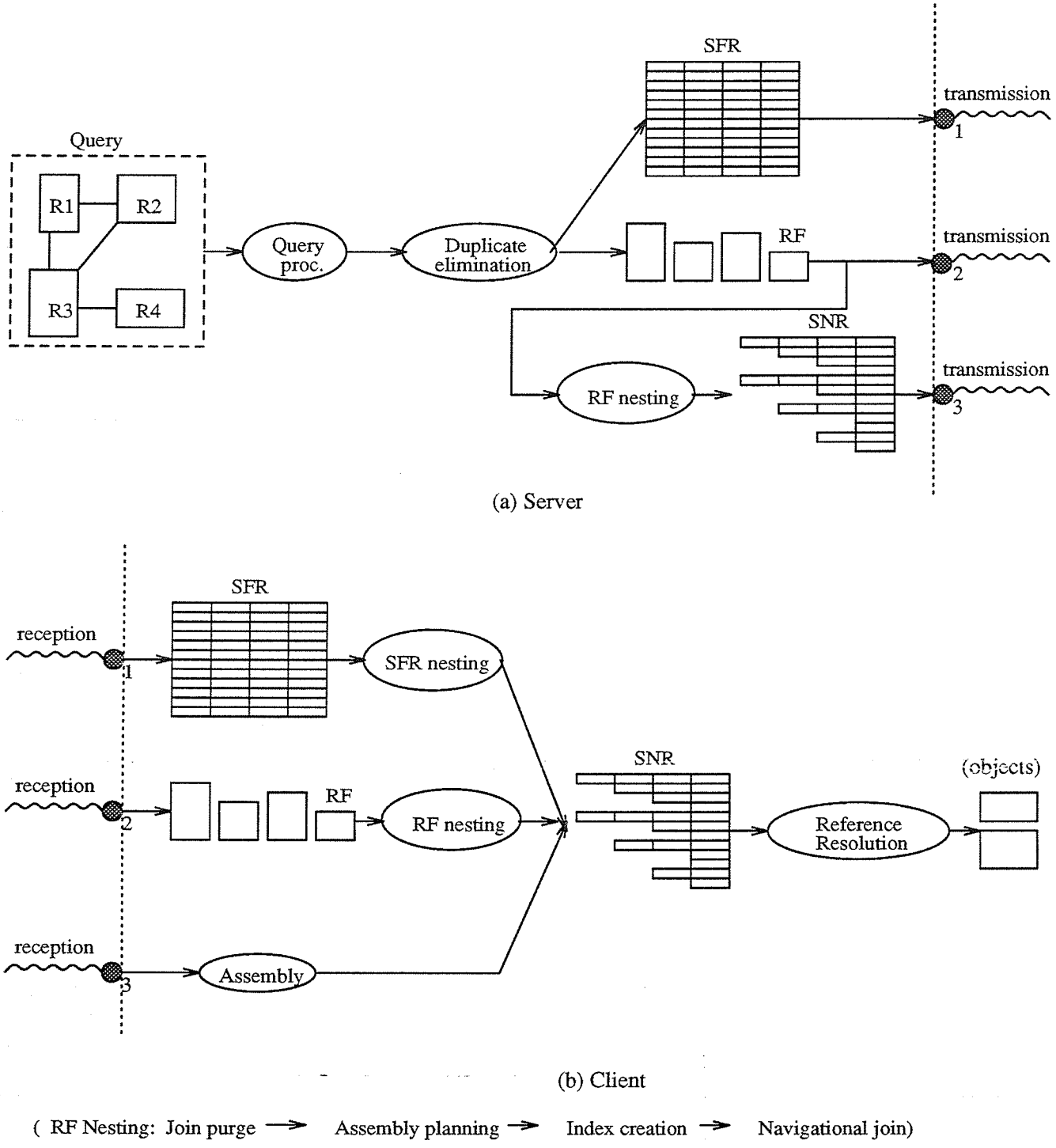


Figure 4.3: Overall processes of object instantiation

are performed starting from each tuple of the pivot relation fragment and navigating along the joins to linked relation fragments. The result of the navigational join is a single nested relation, the same one that would be produced by the nesting step of the SFR method. The reference resolution step is the same as that of the SFR method because it works on the same nested relation.

SNR method: A query is materialized into a single nested relation, transmitted as such, and is translated into objects by a client. We had once looked into materializing a query directly into a single nested relation. However, this direct approach inhibits join ordering by a query optimizer because the order of nested subrelations in a single nested relation is not necessary the same as the join order chosen by a query optimizer. In other words, reordering of joins for more efficient processing of the query can not be attempted. Hence, we decided to take an indirect approach in which a server first materializes a query result using the SFR or RF method and then nest the query result into a single nested relation.

A client does not have to do the nesting step of translation but does only the reference resolution step. The reference resolution step is the same as that of the SFR method and RF method. Therefore, the SNR method uses the same process as the SFR or RF method except that the nesting step of translation has been moved to a server. Telling in advance, the result of our work showed that the nesting of relation fragments is cheaper than the nesting of a single flat relation. Figure 4.3 shows the nesting step using relation fragments.

4.3.1.2 Examples of a SFR, RF, and SNR

Let us consider an example database containing the three relations shown in Figure 4.4a. Figure 4.5 shows the examples of a single flat relation, relation fragments, and a single nested relation, which would be materialized from the same query shown in Figure 4.4b. Each of the relation fragment is materialized from a corresponding base relation. Column values labeled with an asterisk (*) denote redundant column values for each method.

We see that the single flat relation contains duplicate subtuples. For example,

a. Relations:

DEPT(dno, dname, mgr, sec, loc, parentdiv)

EMP(eno, ename, salary, dept, esex, degree, ebdate, addr)

CHILD(parent, cname, cbdate, csex, school)

b. Query:

$$\Pi_{\{\text{dno,dname,ename,addr,cname,school}\}}(\text{DEPT} \underset{\text{dno=dept}}{\bowtie} \text{EMP} \underset{\text{eno=parent}}{\bowtie} \text{CHILD})$$

Figure 4.4: Example relations and query

three of the four "Sales" department names in the `dname` column are redundant duplicates. On the other hand, the relation fragments contain no such duplicate subtuple. However, we note that the `eno`, `dept` attributes of the relation fragment from the base relation `EMP`, and the `parent` attribute of the relation fragment from the base relation `CHILD` have never been specified in the projection set of the query. Nevertheless these attributes must be materialized to make the linkage among the tuples of the three relation fragments possible on a client. In other words, 'extra' attributes are materialized in addition to the projection set in the query and are required to perform 'joins' among the relation fragments in the nesting step. Hence we call those attributes as *extra join attributes*. As for the single nested relation, obviously it contains less number of redundant subtuples than the single flat relation, but the example shows that it still contains some redundant subtuples. In the example, `Steve` works for both the `Sales` department and the `Purchase` department and therefore, the name `Steve`, his address, and his child's name and child's school appear twice in two different nested tuples. We call this source of redundant data contained in a single nested relation as *multiple occurrences of subtuples*.

In the rest of this chapter, we shall adopt the following notations. We denote a single flat relation as T (meaning a 'Table'), a relation fragment as F_i (meaning a 'Fragment'), and a nested subrelation within a single nested relation as S_i (meaning a 'Subrelation').

dno	dname	ename	addr	cname	school
16	Sales	Steve	701A Welch Rd., Palo Alto	Tom	Bing Nursery School
16*	Sales*	Steve*	701A Welch Rd., Palo Alto*	Mike	Escondido Elementary School
16*	Sales*	Ronald	370 Hillside Drive, Redwood City	Jennifer	Stanford University
16*	Sales*	Ronald*	370 Hillside Drive, Redwood City*	Irene	McDonald High School
21	Purchase	Steve*	701A Welch Rd., Palo Alto*	Tom*	Bing Nursery School*
21*	Purchase*	Steve*	701A Welch Rd., Palo Alto*	Mike*	Escondido Elementary School*
21*	Purchase*	Andy	1090 Psyche Dr., Los Altos Hills	Kirk	U.C. Berkeley

(a) Single flat relation (SFR): Duplicate subtuples (*)

dno	dname
16	Sales
21	Purchase

from DEPT

eno	ename	addr	dept
125*	Steve	701A Welch Rd., Palo Alto	16*
124*	Ronald	370 Hillside Drive, Redwood City	16*
125*	Steve	701A Welch Rd., Palo Alto	21*
126*	Andy	1090 Psyche Dr., Los Altos Hills	21*

from EMP

parent	cname	school
124*	Jennifer	Stanford University
124*	Irene	McDonald High School
125*	Tom	Bing Nursery School
125*	Mike	Escondido Elementary School
126*	Kirk	U.C. Berkeley

from CHILD

(b) Relation fragments (RF's): Extra join attributes (*)

dno	dname	ename	addr	cname	school
16	Sales	Steve	701A Welch Rd., Palo Alto	Tom	Bing Nursery School
				Mike	Escondido Elementary School
		Ronald	370 Hillside Drive, Redwood City	Jennifer	Stanford University
				Irene	McDonald High School
21	Purchase	Steve*	701A Welch Rd., Palo Alto*	Tom*	Bing Nursery School*
				Mike*	Escondido Elementary School*
		Andy	1090 Psyche Dr., Los Altos Hills	Kirk	U.C. Berkeley

(c) Single nested relation (SNR): Multiple occurrence of subtuples (*) $\leq$ duplicate subtuples

Figure 4.5: Examples of a SFR, RF, and SNR

4.3.2 Materialization in the SFR Method and RF Method

In the materialization phase of the SFR method and the RF method, a query is materialized into a single flat relation or a set of relation fragments depending on the method. The materialization phase consists of two steps: *query processing* and *duplicate elimination*. In main memory databases, the choice of query processing strategies [84, 85, 75, 77, 78, 80, 83] is based on the criteria of the number of CPU cycles and memory space efficiency rather than the number of disk accesses and disk space efficiency. The results of comparing different query processing strategies obtained by some researchers [84, 85, 75] showed that hash-based query processing strategies are faster than others when large main memory is available. On the other hand, a main memory database system used in OBE [73, 83, 78] implemented a *pipelined nested loop join* [83, 78] with array indexes and obtained good performance in both time and memory space. One advantage of using this join algorithm is that it does not create intermediate relations during query processing.

4.3.2.1 Query Processing for a Single Flat Relation (SFR)

Whichever join algorithm may be used for query processing, a join between two relations, $R_1 \underset{\eta_1 \theta \eta_2}{\bowtie} R_2$, produces the following set of tuples.

$$\{\langle t_1' t_2' \rangle | t_1 \in R_1, t_2 \in R_2, t_1.\eta_1 \theta t_2.\eta_2, t_1' = t_1.(\pi_1 \cup \zeta_1), t_2' = t_2.(\pi_2 \cup \zeta_2)\} \qquad (4.2)$$

where $\pi_i, i = 1, 2$, denotes the set of attributes of R_i that are specified in the projection set of the query, and $\zeta_i, i = 1, 2$, denotes the set of attributes that are needed for subsequent join computations. Note that join attributes (η_i) are discarded unless they are elements of $\pi_i \cup \zeta_i$.

If we use the pipelined nested loop join strategy which showed successful performance in OBE, the join processing algorithm becomes as follows.

Algorithm 4.3.1 (SFR Query processing)
Input: base relations $R_i, i = 1, 2, \cdots, n$, and a query
Output: a set of composite tuples of the query result.

Procedure:

Let Φ_i denote a conjunction of join predicates between R_i and $R_1, \cdots, R_{i-1}$, respectively. Each R_i is assumed to have already been filtered by applicable selection conditions.

For each $t_1 \in R_1$
  For each $t_2 \in R_2$ satisfying Φ_2
  $\ddots$
    For each $t_n \in R_n$ satisfying Φ_n
      Output $t_1.\pi_1 \parallel t_2.\pi_2 \parallel \cdots \parallel t_n.\pi_n$. /* $\parallel$ denotes 'concatenation'. */

4.3.2.2 Query Processing for Relation Fragments (RF)

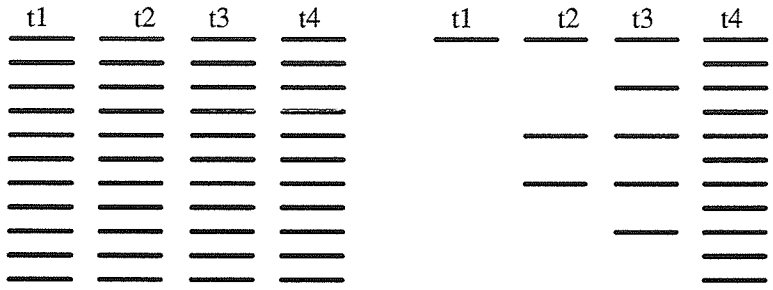
We focus on how Algorithm 4.3.1 should be modified to materialize a set of relation fragments instead of a single flat relation, rather than inventing a different algorithm.

First of all, the single Output statement in Algorithm 4.3.1 must be decomposed into multiple Output statements, i.e., one Output for each relation fragment. Secondly, join attributes (η_i's) should be materialized in addition to $\pi_i \cup \zeta_i$ so that a client can build indexes on the join attributes. Thus, a join between two relations, $R_1 \underset{\eta_1 \theta \eta_2}{\bowtie} R_2$, should produce the following set of tuples.

$$\{\langle t_1' t_2' \rangle | t_1 \in R_1, t_2 \in R_2, t_1.\eta_1 \theta t_2.\eta_2, t_1' = t_1.(\pi_1 \cup \zeta_1 \cup \eta_1), t_2' = t_2.(\pi_2 \cup \zeta_2 \cup \eta_2)\} \quad (4.3)$$

where η_i denotes the set of join attributes. Accordingly, the Output statement of Algorithm 4.3.1 is modified to Output $t_1.(\pi_1 \cup \eta_1); t_2.(\pi_2 \cup \eta_2); \cdots; t_n.(\pi_n \cup \eta_n)$. Thirdly, a tuple from an outer nested loop need not be emitted unless it is a new tuple. For example, $t_1 \in R_1$ in the outermost loop need be emitted only once for each completion of all of its inner loops. We can use switches which denote whether a new tuple has been obtained from the outer loop, to avoid these unnecessary emissions.

The following algorithm shows a pipelined nested loop join algorithm modified from Algorithm 4.3.1 to the above discussion, i.e., using multiple output statements, emitting necessary join attributes, and using switches (SW's) to avoid unnecessary emission of tuples.



(a) SFR query processing (b) RF query processing

Figure 4.6: Tuples emitted from base relations

Algorithm 4.3.2 (RF Query processing)

Input: base relations $R_i, i = 1, 2, \cdots, n$, and a query

Output: a set of relation fragments $F_i, i = 1, 2, \cdots, n$.

Procedure:

Let Φ_i denote a conjunction of join predicates between R_i and $R_1, \cdots, R_{i-1}$, respectively. Each R_i is assumed to have already been filtered by applicable selection conditions.

For each $t_1 \in R_1$,
  Set SW_1.
  For each $t_2 \in R_2$ satisfying Φ_2,
    Set SW_2.
    $\ddots$
      For each $t_n \in R_n$ satisfying Φ_n,
        Set SW_n.
        For each $\mathrm{SW}_i, i = 1, 2, \cdots, n$,
          If SW_i is set then begin
            Output $t_i.(\pi_i \cup \eta_i)$.
            Reset SW_i.
          end

Note that some of the attributes emitted for a relation fragment are extra join

attributes, that is, not specified in the projection set of the query but are still needed to build the linkage among the relation fragments in the translation phase. Here comes a formal definition of extra join attributes.

Definition 4.3.1 (Extra join attributes) Given the set of attributes $\mathcal{F}_i$ of a relation fragment F_i and a projection set $\mathcal{P}$ specified in the query, $\mathcal{F}_i - \mathcal{P}$ is the set of extra join attributes of F_i.

4.3.2.3 Tuples Emitted from Query Processing

We see, by comparing Algorithm 4.3.1 and Algorithm 4.3.2, that there is little difference in the cost of the query processing itself. The query processings of both the SFR method and the RF method execute the same nested loops. That is, if the cardinalities of the relations $R_i, i = 1, 2, \cdots, n_f$ are $N_i, i = 1, 2, \cdots, n_f$, both algorithms take $O(N_1 N_2 \cdots N_{n_f})$ time. However, the numbers of tuples that are actually emitted from each base relation by the output statement are different in each algorithm. Figure 4.6 illustrates this difference. In Algorithm 4.3.1, the tuples from each base relation that satisfy all join conditions are emitted as a composite tuple once for every innermost loop execution. Therefore, $N_1 N_2 \cdots N_{n_f}$ composite tuples are emitted for the entire loops. On the other hand, in Algorithm 4.3.2, a tuple from a base relation is emitted only if the execution of its inner loops has been completed. Therefore, Algorithm 4.3.2 never emits more tuples than Algorithm 4.3.1.

4.3.2.4 Duplicate Elimination

The query result has duplicates if there are duplicate tuples in the base relations specified in the query. Besides, projections performed in a query processing can produce duplicate tuples in the query result. These duplicate tuples result in duplicate objects when they are translated into objects. We disallow any duplicate objects to be instantiated from a database because duplicate objects are regarded as separate objects in an object-oriented model. Therefore, duplicate tuples are removed from the final query result. Duplicate elimination can be done either using sorting or hashing. Given a relation of N tuples, a sorting costs $O(N \log_2 N)$ and a hashing costs $O(N/B)$

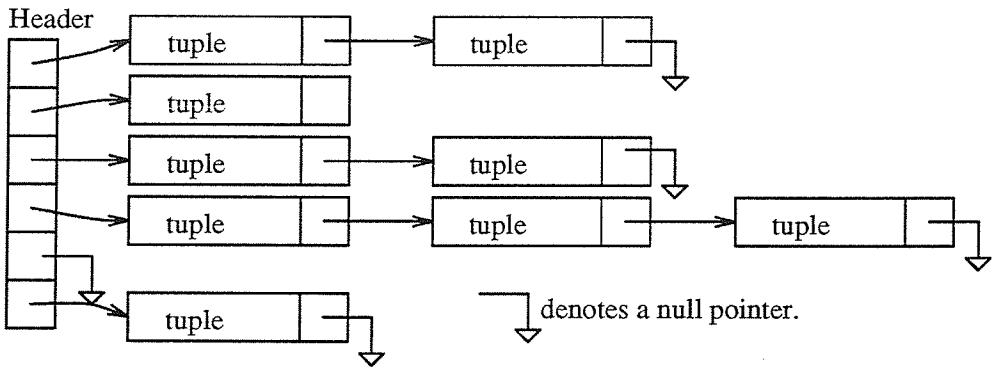


Figure 4.7: The structure of a chained bucket hashing for duplicate elimination

where B is the size of a bucket header. It is anticipated that hashing be faster than sorting. Another advantage of hashing is that hashing can be pipelined between query processing and transmission because hashing is processed tuple by tuple. That is, each tuple of the final query result can be hashed immediately as soon as it is available and a tuple which has no duplicate in the already hashed set of tuples can be transmitted immediately. On the other hand, if sorting is used, all tuples of the final query result must be collected before sorting can start.

We use a simple chained bucket hashing [92] (alias, an open hashing [93]), whose structure is shown in Figure 4.7. The bucket header is an array of pointers to chains of buckets. Each bucket in a chain is a record of two entries – a tuple and a pointer to the next bucket. Given this structure, the algorithm for eliminating duplicates in pipelining with transmission becomes as follows.

Algorithm 4.3.3 (Duplicate elimination)

1. Allocate a hashing bucket header.

2. For each tuple t_o output from the query processor,

   (a) Compute a hashed address $h(t_o)$ using the entire tuple as the input where h is a hashing function.

(b) i. Traverse the chain of buckets starting from the header located at the address $h(t_o)$.
For each bucket visited,
A. Compare the tuple t_o with the tuple t_b contained in the bucket.
B. If $t_o = t_b$ then go to Step 2c.
ii. /* No same tuple as t_o was found in the chain. */
Insert a new bucket containing t_o into the chain and transmit t_o.

(c) Continue.

In Algorithm 4.3.3, actual tuples are stored in hashing buckets. We can reduce the memory space allocated for hashing buckets if we store *pointers* to the tuples of base relations instead of the tuples emitted from a query processor. This method certainly reduces the memory space allocated for buckets, but requires the reading of base tuples and projection on them every time a query result tuple is to be compared with the bucket entry. We will assume the storage of actual tuples for our cost modeling.

4.3.3 Translation in the SFR Method and RF Method

In the translation phase, a received query result is restructured into objects that can be used by the application. The translation process depends on the structure of the objects defined by the object model. In our work, objects realize the aggregation hierarchy [86, 87] through nested structure and references among objects. It motivated us to design the complete process of translation in two steps, *nesting* and *reference resolution*, as mentioned in Section 4.3.1. In the nesting step, we restructure the retrieved relation or relation fragments into a nested relation. In the reference resolution step, references among objects are resolved by making pointer linkages among the nested relations.

Figure 4.8 and Figure 4.9 illustrate the nesting processes of the SFR method and the RF method, respectively. The nesting step is carried out differently for the SFR method and the RF method. In the SFR method, it is done by decomposing received tuples into subtuples corresponding to different nested subrelations, and assembling the decomposed subtuples into nested tuples. On the other hand in the

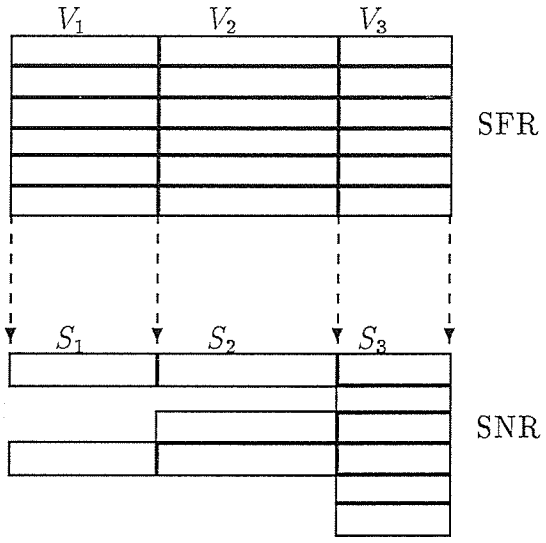


Figure 4.8: SFR nesting process

RF method, it is done by *creating indexes* on the join attributes of the relation fragments and performing *navigational join*. Navigation starts from the pivot relation fragment and follows the joins of the query to find matching tuples in the joined relation fragments. At least one matching tuple always exists in each relation fragment because the relation fragments have already been fully reduced by the same join operations on a server. The matching tuples thus found are assembled into nested tuples according to an *assembly plan* generated by comparing the join tree and the nesting format tree. The *join purge* step chooses only one of the conjunctive join predicates from each join in the join tree.

The reference resolution step is out of our scope because its process is specific to the object schema defined by the application. Besides, omitting this step does not affect the cost comparison result because the reference resolution processes of the RF method and SFR method are identical.

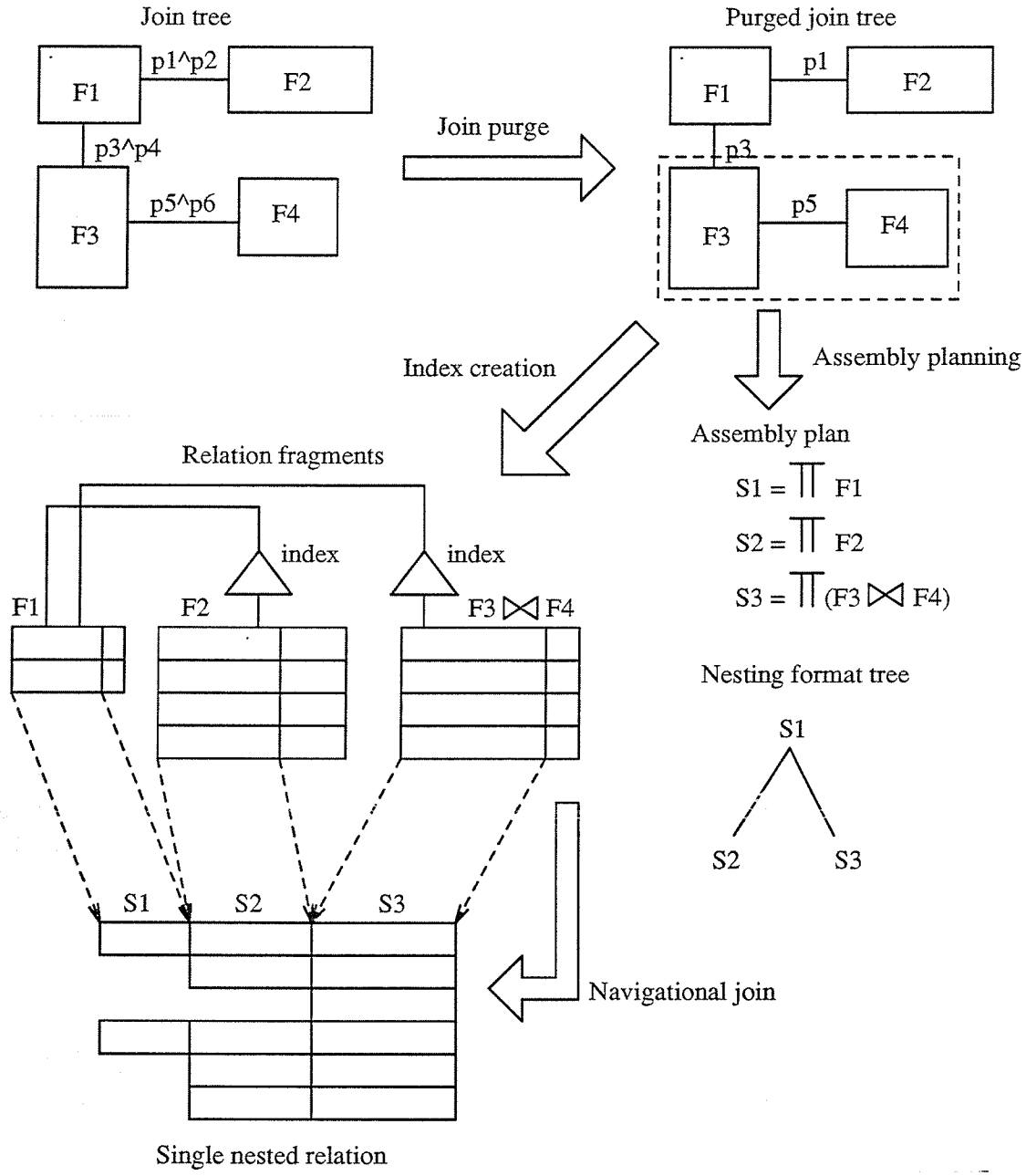


Figure 4.9: RF nesting process

4.3.3.1 Generation of a Nesting Format

A nesting format [70] is a string interpreted as the schema of a nested relation. It provides the information necessary to restructure a flat relation or a set of relation fragments into a nested relation. We denote the nesting of subrelations with parentheses such as A(BC)(D(E))[2].

We can build an O-tree, given the type definition of an object O. An example is shown in Figure 4.10c for the object type shown in Figure 4.10a. Given this O-tree and the attribute mapping function whose mapping is shown in Figure 4.10c, the nesting format for restructuring the result of the query shown in Figure 4.10b is generated in the following procedure. (Object type, O-tree, and attribute mapping function have been defined in Part I.)

Algorithm 4.3.4 (Generation of a nesting format)
Input: O-tree, and attribute mapping function (AMF)
Output: a nesting format
Procedure:

1. Starting from the root of the O-tree, recursively replace each node by the list of its children.

2. Replace each object attribute in the list produced by Step 1 by the corresponding relation attribute mapped by the AMF.

3. Strip off the outermost parentheses from the list produced in Step 2.

Figure 4.10c shows an example of the mapping between object attributes and relation attributes appearing in a query. Let us assume the schema of the retrieved relation is KADEHGIJ. Step 1 of Algorithm 4.3.4 generates (oidA_o(D_oE_o(H_oG_o))(I_oJ_o)), which is replaced by (KA(DE(HG))(IJ)) in Step 2 and becomes KA(DE(HG))IJ in Step 3. This format becomes the schema of the nested relation. For example, HG is mapped to H_oG_o which is nested within a complex attribute F_o. F_o is nested within a complex attribute B_o, which is an attribute of O. These two levels of nesting is reflected by the two levels of parentheses in the generated nesting format.

[2] In [70], the same format is denoted as A(BC) * (D(E)*)*.

a. Object type: ('simple' denotes a literal such as an integer, string, etc., or an object id (oid). Its value can be an atomic literal or a set of atomic literals.)

Type O
[A_o : simple,
B_o: [D_o: simple, E_o: simple,
F_o: [H_o: simple, G_o: simple]],
C_o: [I_o: simple, J_o: simple]]

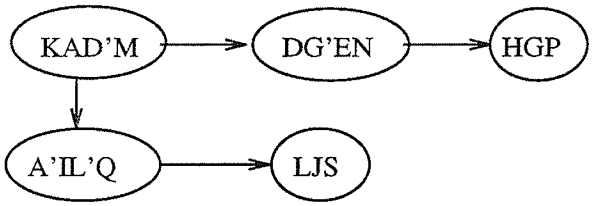
b. Query:

- Select-project-join expression:

$$\Pi_{\mathtt{KADEHGIJ}}(\sigma_1\mathtt{KAD'M} \underset{\mathtt{D'}\theta\mathtt{D}}{\bowtie} \sigma_2\mathtt{DG'EN} \underset{\mathtt{G'}\theta\mathtt{G}}{\bowtie} \sigma_3\mathtt{HGP} \underset{\mathtt{A}\theta\mathtt{A'}}{\bowtie} \sigma_4\mathtt{A'IL'Q} \underset{\mathtt{L'}\theta L}{\bowtie} \sigma_5\mathtt{LJS})$$

where $\theta \in \{=, \neq, <, \leq, >, \geq\}$.

- Join graph of the query:



c. O-tree, and mapping of its attributes to relation attributes: (O is the object type. oid denotes an object id, and the other capital letters subscripted with o denote object attributes.)

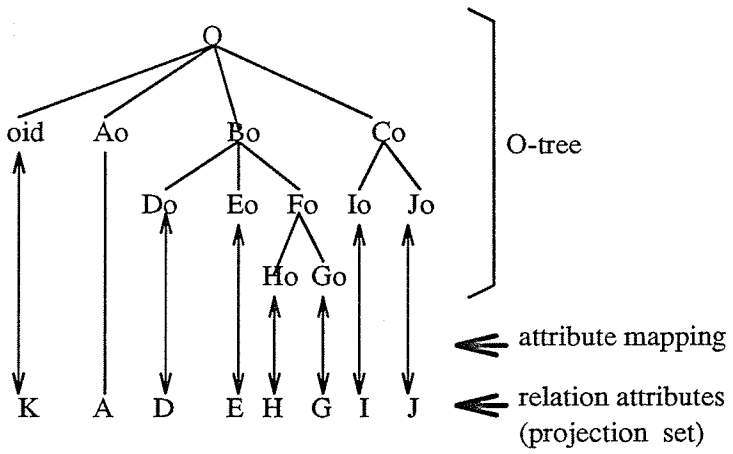


Figure 4.10: An example of object type, view, and O-tree

a. Flat relation schema: `KADEHGIJ`

b. Nesting format: `KA(DE(HG))(IJ)`

c. Nesting format tree:

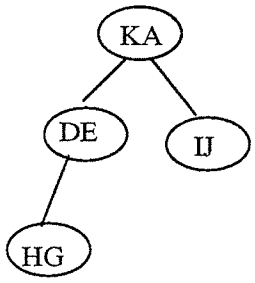


Figure 4.11: An example of a nesting format and its nesting format tree

We can draw out the hierarchy of nested subrelations from a nesting format. An example is shown in Figure 4.11. The root of the tree represents a subrelation which is not nested within any other subrelation, and its descendents represents subrelations nested within their parents. We call such a tree as a *nesting format tree.* In particular, the subrelation represented by the root is called a *pivot subrelation* because the root always contains an attribute which is mapped to an oid.

4.3.3.2 The Structure of a Single Nested Relation

For both the SFR method and the RF method, searching is required every time a tuple is to be inserted into an output single nested relation. The tuple is inserted only if there does not exist the same tuple in the single nested relation. Hence, the number of searchings performed is always greater than the number of insertions performed. In particular, a large portion of tuples that are attempted for insertion are discarded for the SFR method if the number of duplicate subtuples in a single flat relation is large. Considering these facts, the structure of a single nested relation was determined to show good *searching* performance.

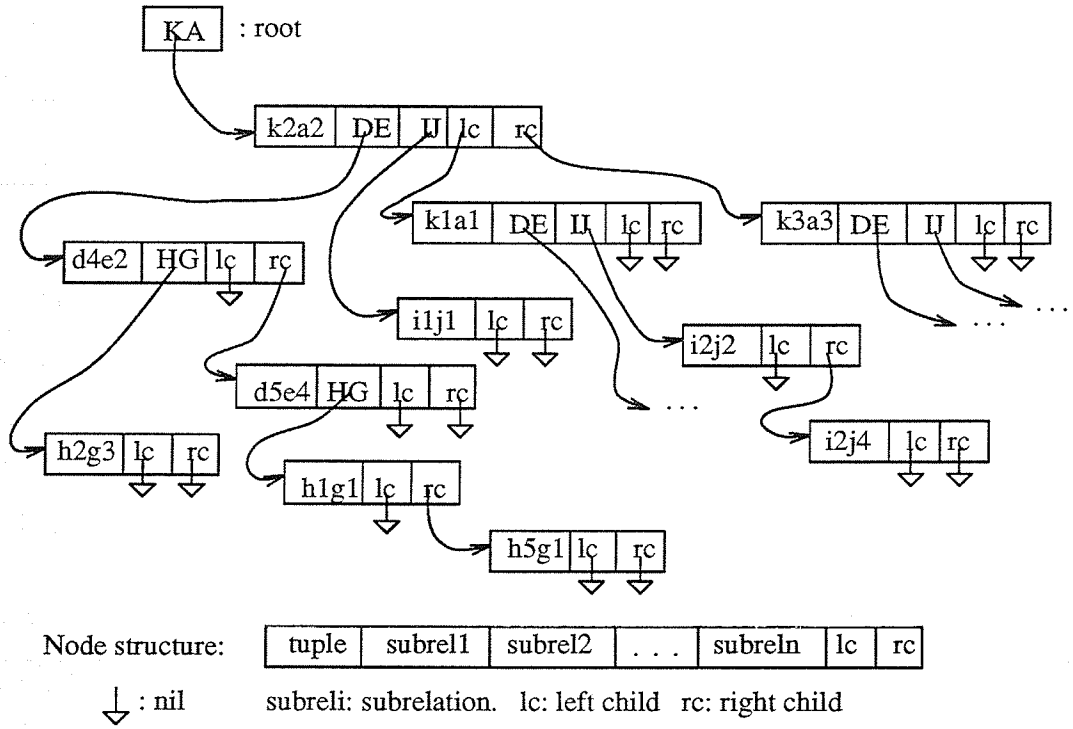


Figure 4.12: The structure of a single nested relation

Figure 4.12 shows the structure of a single nested relation used for our work. Each nested subrelation is implemented as a binary search tree. The root SNR node contains only the pointer to the root of the binary search tree representing the pivot subrelation. Each node of a binary search tree contains a tuple value, pointers to the roots of the nested binary search trees, and pointers to its left child and right child. Both searching and insertion of a tuple within a nested subrelation take $O(\log_2 N)$ time where N is the number of tuples currently inserted in a binary search tree.

4.3.3.3 Nesting of a Single Flat Relation

In [68], NEST was introduced as an operator for restructuring a flat relation into a nested relation. Similar concepts were also described in [69, 70]. Our nesting process described here is an instance of implementing the NEST operator.

Figure 4.13 shows an example of the relation instance before and after the nesting step. The single flat relation of Figure 4.13a was obtained by evaluating the query in Figure 4.10b on a set of relation instances. In Figure 4.13b, `IJ` is independent of `DEHG`, but is dependent only on `KA`. SFR nesting can be performed pipelined with the reception of the tuples from a server. Each received tuple is decomposed into subtuples where each subtuple is an instance of each node of the nesting format tree. Each subtuple is then inserted into an output single nested relation. Since there may exist duplicate subtuples in a single flat relation, it must be checked before insertion if there already exists the same subtuple in the single nested relation. Figure 4.12 shows the result of inserting the first three tuples of the single flat relation of Figure 4.13a into an empty single nested relation. Figure 4.13b shows the nested tuples of the final single nested relation schematically.

An algorithm for the SFR nesting step is as follows.

Algorithm 4.3.5 (SFR Nesting)
Input: received tuples of a single flat relation, and a nesting format tree (NFT).
Output: A single nested relation (SNR).
Procedure:

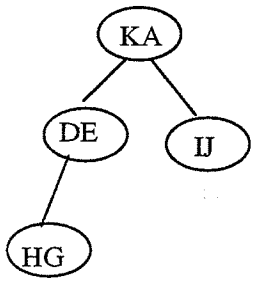
1. Allocate an empty (root only) single nested relation SNR.

K	A	D	E	H	G	I	J
k_2	a_2	d_4	e_2	h_2	g_3	i_1	j_1
k_1	a_1	d_2	e_2	h_4	g_2	i_2	j_4
k_3	a_3	d_5	e_4	h_1	g_1	i_2	j_2
k_2	a_2	d_5	e_4	h_5	g_1	i_1	j_1
k_1	a_1	d_2	e_2	h_4	g_2	i_2	j_2
k_3	a_3	d_5	e_4	h_5	g_1	i_2	j_2
k_1	a_1	d_2	e_2	h_2	g_3	i_2	j_4
k_3	a_3	d_5	e_4	h_1	g_1	i_2	j_4
k_1	a_1	d_2	e_2	h_2	g_3	i_2	j_2
k_3	a_3	d_5	e_4	h_5	g_1	i_2	j_4
k_2	a_2	d_5	e_4	h_1	g_1	i_1	j_1

(a) Retrieved single flat relation (before nesting starts)

K	A	$(D$	E	$(H$	$G))$	$(I$	$J)$
k_2	a_2	d_4	e_2	h_2	g_3	i_1	j_1
		d_5	e_4	h_1	g_1		
				h_5	g_1		
k_1	a_1	d_2	e_2	h_2	g_3	i_2	j_2
				h_4	g_2	i_2	j_4
k_3	a_3	d_5	e_4	h_1	g_1	i_2	j_2
				h_5	g_1	i_2	j_4

(b) Single nested relation (after nesting completes)



(c) Nesting format tree

Figure 4.13: An example of nesting a single flat relation

2. w_p := the root of the empty SNR.

3. u_p := the root of NFT.

4. For each tuple t_r received from a server,
   Assemble(w_p,u_p,$\Pi_{u_p} t_r$). /* Project t_r on u_p. */

In Algorithm 4.3.5, Assemble(w_i, u_i, t_i) inserts a tuple t_i into a binary search tree whose root is the node pointed by $w_i.u_i$ – the u_i field of an insertion entry node w_i. This binary search tree belongs to a nested subrelation S_i corresponding to the node u_i of the nesting format tree.

Algorithm 4.3.6 (Assemble_SFR)
Input: a node (w_i) of SNR, a node (u_i) of NFT, and a tuple t_i to be inserted.
Output: SNR with t_i inserted if t_i is new.
Procedure:

1. w_r := the node pointed by $w_i.u_i$.
   /* w_r is the root of a binary search tree to be searched. */

2. If (w_c := Search(w_r, t_i)) = NOT_FOUND
   then Insert-tuples(w_i, u_i, t_i)
   else /* There exists t_i already. */

   (a) Ψ := the set of u_i's children (u_c) in NFT.

   (b) If $\Psi = \{\ \}$ then Return
   else
   For each $u_c \in \Psi$,
   Assemble($w_c, u_c, \Pi_{u_c} t_r$). /* Project t_r on u_c. */

In Algorithm 4.3.6, Search(w_r, t_i) finds a SNR node whose tuple value = t_i among the nodes of the binary search tree rooted by w_r, and returns NOT_FOUND if no tuple t_i is found or returns the SNR node containing the tuple t_i if one is found.

Algorithm 4.3.7 (Search)
Input: A node (w_i) of a binary search tree and a tuple t_i to be searched for.

Output: Return NOT_FOUND or a found node.
Procedure:

If w_i = nil then return NOT_FOUND
else if w_i.tuple = t_i then return w_i
else if (w_i.tuple < t_i) then return(Search(w_i.LChild, t_i))
else return(Search(w_i.RChild, t_i)).

Insert-tuples(w_i, u_i, t_i) inserts tuple t_i into the binary search tree whose root is the node pointed by $w_i.u_i$, and also inserts all of t_i's nested subtuples corresponding to u_i's descendents obtained from the nesting format tree.

Algorithm 4.3.8 (Insert-tuples)
Input: A node (w_i) of SNR, a node (u_i) of NFT, and a tuple t_i to be inserted.
Output: t_i is inserted into the nested subrelation whose root is pointed by $w_i.u_i$, and all of t_i's nested tuples are inserted into u_i's nested subrelations.
Procedure:

1. w_e := Insert(w_i, u_i, t_i). /* Insert the tuple t_i. */
2. /* Insert t_i's nested subtuples. */
   $\Psi := \{u_c | u_c$ is a child of u_i in the NFT.$\}$
   If $\Psi = \{\ \}$ then Return
   else
   For each $u_c \in \Psi$,
   Insert-tuples($w_e, u_c, t_r.u_c$).

Insert(w_i, u_i, t_i) inserts t_i into the binary search tree whose root is pointed by $w_i.u_i$ and returns the inserted node.

Algorithm 4.3.9 (Insert)
Input: A node (w_i) of SNR, a node (u_i) of NFT, and a tuple t_i to be inserted.
Output: Returns the inserted node.
Procedure:

Allocate an empty node w_e.
Copy t_i to w_e.tuple.
w := the node pointed by $w_i.u_i$
/* w is the root of the binary search tree into which w_i is to be inserted. */
If w = nil /* An empty nested subrelation. */
    then $w_i.u_i$:= the address of w_e. /* Insert w_e. */
else while w_e is not inserted begin
    if $t_i < w$.tuple then
        if (w := the node pointed by w.lc) = nil
            then insert w_e as the left child of w
    else
        if (w := the node pointed by w.rc) = nil
            then insert w_e as the right child of w
end
Return w_e.

4.3.3.4 Index Structure for Relation Fragments: Chained Bucket Hashing

Since no linkage information among the relation fragments is retrieved from a server in the RF method, a client has to build necessary linkage information using the received relation fragments. Our method is to create indexes on the join attributes of the relation fragments. In the main-memory resident environment, the choice of an appropriate index structure is based on the criteria of the number of CPU cycles and memory space efficiency. In [74, 75], Lehman et al. showed the performance comparison of different index structures of the following category.

- Order-preserving indexes: Array [72], AVL Tree [88], B Tree [89], T Tree [74, 75].
- Randomizing (hashing) indexes: Linear Hashing [91], Modified Linear Hashing [74, 75], Extended Hashing [90], Chained Bucket Hashing.

The performance was compared for the index insertion (or equivalently, creation), random search, a query of mixed operations, range query, scan, deletion, and for the index memory utilization – the ratio between the memory allocated and the memory

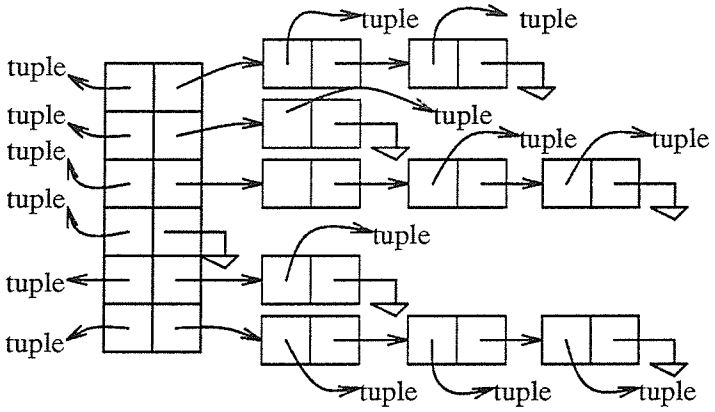


Figure 4.14: The structure of a chained bucket hashing index

actually containing data. According to their result, the Chained Bucket Hashing shows the best performance in all the above operations except the range query, for which any kind of hashing index is inappropriate. Since we need only creation and random search of indexes, the inability of supporting range queries causes no problem. The Chained Bucket Hashing requires approximately 1.2 to 1.5 times more memory than the other indexes. We assume this is not significant and do not worry about memory overflow because indexes are built on relation fragments which have already been fully reduced before being transmitted. We thus choose to use the Chained Bucket Hashing index.

Figure 4.14 shows the structure of a chained bucket hashing index used in our work. It is configured of a bucket header table and chained buckets linked to each header. Note that, unlike the chained bucket hashing of Figure 4.7, each bucket header and chained bucket contains a *pointer* to a tuple instead of an actual tuple [74, 75]. Storing pointers reduces the main memory space allocated for the hashing table. Those pointers are used to extract attribute values when needed. We pay the price of additional pointer followings rather than duplicating the tuples of relation fragments in the buckets. It was observed in [74] that the Chained Bucket Hashing index organized in this structure shows the best storage cost/performance ratio when the size of the bucket header table is approximately a *half* of the number of tuples to

be indexed.

4.3.3.5 Nesting of Relation Fragments

Nesting of retrieved relation fragments is performed in four steps: *join purge*, *assembly planning*, *index creation*, and *navigational join*. Figure 4.15 shows an example of the data structure in each step of nesting relation fragments. The relation fragments were obtained from the same query (Figure 4.10b) that was used to obtain the single flat relation of Figure 4.13a.

4.3.3.5.1 Join purge In the join purge step, we remove any redundant join predicates from the joins specified in the query, leaving only the minimal number of joins. A conjunction of join predicates in a query can be reduced to a single join predicate by choosing one of them arbitrarily. This reduction does not affect the result of the nesting step. The following theorem shows it.

Theorem 4.3.1 Let us consider a conjunctive join predicate $A_1\theta_1B_1 \wedge A_2\theta_2B_2 \wedge \cdots \wedge A_n\theta_nB_n$ between two relation fragments F_1 and F_2 that have been retrieved from a server in the RF method. Then, for an arbitrary pair of tuples $\langle t_1, t_2\rangle$ where $t_1 \in F_1, t_2 \in F_2$,

$$(t_1.A_1\theta_1t_2.B_1) \wedge (t_1.A_2\theta_2t_2.B_2) \wedge \cdots \wedge (t_1.A_n\theta_nt_2.B_n) \tag{4.4}$$

if and only if

$$t_1.A_i\theta_it_2.B_i \text{ for some } i \in [1, n] \tag{4.5}$$

Proof: Since the 'only if' part is obvious, we prove only the 'if' part.
If part: Let us assume Equation 4.4 is not satisfied although Equation 4.5 is satisfied. Then, there exists at least one $j \in [1, n]$ such that $j \neq i$ and $\neg(t_1.A_j\theta_jt_2.B_j)$. However, if $t_1.A_j\theta_jt_2.B_j$ is false, $t_1 \notin F_1$ if $t_2 \in F_2$ and $t_2 \notin F_2$ if $t_1 \in F_1$ by the definition of join. It contradicts with the given assumption that $t_1 \in F_1$ and $t_2 \in F_2$. Q.E.D.

It will be good in practice to select the join predicate which takes the minimum computation time, such as an equijoin on integer attributes.

$K\ A\ D'$
$k_2\ a_2\ d_4$
$k_1\ a_1\ d_2$
$k_3\ a_3\ d_5$
$k_2\ a_2\ d_5$

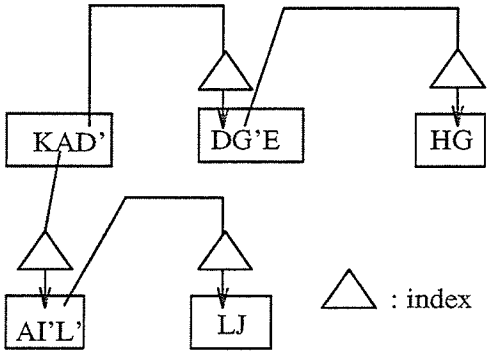
$D\ G'\ E$
$d_5\ g_1\ e_4$
$d_2\ g_2\ e_2$
$d_2\ g_3\ e_2$
$d_4\ g_3\ e_2$

$H\ G$
$h_1\ g_1$
$h_4\ g_2$
$h_5\ g_1$
$h_2\ g_3$

$A'\ I\ L'$
$a_3\ i_2\ l_3$
$a_1\ i_2\ l_3$
$a_2\ i_1\ l_1$

$L\ J$
$l_1\ j_1$
$l_3\ j_2$
$l_3\ j_4$

(a) Relation fragments (before nesting starts)



(b) After index creation

$K\ A$	$(D\ E\ (H\ G))$	$(I\ J)$
$k_2\ a_2$	$d_4\ e_2$ $\boxed{h_2\ g_3}$	$i_1\ j_1$
	$d_5\ e_4$ $\boxed{h_1\ g_1 \\ h_5\ g_1}$	
$k_1\ a_1$	$d_2\ e_2$ $\boxed{h_2\ g_3 \\ h_4\ g_2}$	$i_2\ j_2 \\ i_2\ j_4$
$k_3\ a_3$	$d_5\ e_4$ $\boxed{h_1\ g_1 \\ h_5\ g_1}$	$i_2\ j_2 \\ i_2\ j_4$

(c) After navigational join (nesting completes)

Figure 4.15: An example of nesting a set of relation fragments

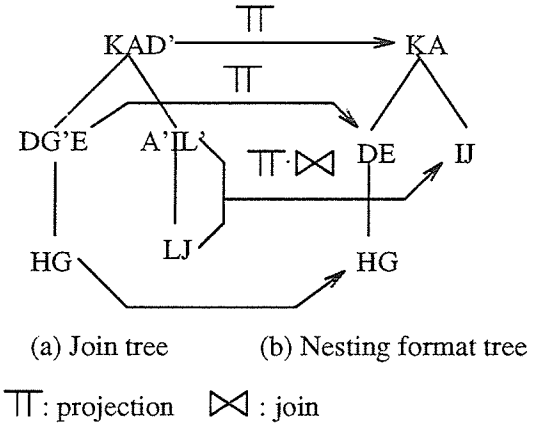


Figure 4.16: An example of an assembly plan

Since we are dealing with an acyclic query, the join graph of a query is always a tree rooted by the pivot relation fragment. From now on, we use the term *join tree* interchangeably with join graph.

4.3.3.5.2 Assembly planning In this step, a plan of how to assemble the tuples which will be collected from the navigational joins is set up. An assembly plan is a transformation from the nodes of a join tree to the nodes of a nesting format tree. Figure 4.16 contrasts the join tree and the nesting format tree for the object and view shown in Figure 4.10. As illustrated in Figure 4.16, a node of a nesting format tree is obtained from one or more nodes of a join tree via relational projections and joins. A node of a join tree represents a relation fragment while a node of a nesting format tree represents a nested subrelation of a single nested relation. Joins are needed only if a node of the nesting format tree has a schema which is not a subset of the schema of any relation fragment but is split into the schemas of two or more relation fragments. For example, the `IJ` node is split into two relation fragments `A'IL'` and `LJ`. A join is needed to merge the relation fragments `A'IL'` and `LJ` into the nested subrelation `IJ`. Projections are used to remove the extra join attributes. Note that there must exist one and only one matching tuple of `LJ` for each tuple of `A'IL'`. Otherwise `I` and `J` cannot belong to the same node of the nesting format tree.

In the following discussion, we denote a node of a join tree as v_i and a node of nesting format tree as u_i. Note there is a one-to-one mapping between the set of relation fragments $\{F_i\}$ and the set of the nodes $\{v_i\}$ of a join tree, and between the set of nested subrelations $\{S_i\}$ and the set of the nodes $\{u_i\}$ of a nesting format tree. Let us introduce here two functions defining these one-to-one mappings – **RFJT** from $\{F_i\}$ to $\{v_i\}$ and **NSRNFT** from $\{S_i\}$ to $\{u_i\}$ – for later use.

An *assembly plan* is defined as a set of expressions of the following form.

$$u := \Pi_u(v_1 \bowtie v_2 \cdots \bowtie v_k)$$

where Π_u denotes the projection on the schema of u. For example, we obtain the following assembly plan from the join tree and nesting format tree of Figure 4.16.

Example 4.3.1 (Assembly plan)

$$\begin{aligned} \mathtt{KA} &:= \Pi_{\mathtt{KA}}\mathtt{KAD'} \\ \mathtt{DE} &:= \Pi_{\mathtt{DE}}\mathtt{DG'E} \\ \mathtt{HG} &:= \mathtt{HG} \\ \mathtt{IJ} &:= \Pi_{\mathtt{IJ}}(\mathtt{A'IL'} \underset{\mathtt{L'}\theta\mathtt{L}}{\bowtie} \mathtt{LJ}) \end{aligned}$$

□

In a more abstract form, we consider an assembly plan **AP** as a function of the nodes of a join tree (or equivalently, relation fragments), i.e., $u = \mathrm{AP}(v_1, v_2, \cdots, v_k)$. We use the same function AP for both the schema of the nodes of the join tree and nesting format tree and their instance tuples. For example, AP(A'IL', LJ) returns IJ and $\mathrm{AP}(a_3i_2l_3, l_3j_2)$ returns i_2j_2.

The algorithm for generating an assembly plan is as follows.

Algorithm 4.3.10 (Assembly planning)
Input: a join tree (JT), a nesting format tree (NFT).
Output: an assembly plan (AP).
Procedure:

1. For each node v newly visited (not marked as 'visited') while traversing JT starting from the root,

   (a) Find a node u in NFT whose schema is a subset of the schema of v.

   (b) If a node u is found then

      i. If $u = v$ then add $u := v$ to AP
      else add $u := \Pi_u v$ to AP

      ii. Mark v as 'visited'.

   else

      i. Find the set of nodes $\{v \equiv v_1, v_2, \cdots, v_k\}$ of a minimal subtree of JT rooted by v such that the union of the schema of $v_1, v_2, \cdots, v_k$ contains the schema of u.

      ii. Add $u := \Pi_u(v_1 \bowtie v_2 \cdots \bowtie v_k)$ to the AP.

      iii. Mark $v_1, v_2, \cdots, v_k$ as 'visited'.

4.3.3.5.3 Index creation Once redundant joins are removed, indexes are created on the join attribute of each relation fragment except the pivot relation fragment. For example, given the relation fragments of Figure 4.15, indexes are created on DG'E.D, HG.G, A'IL'.A, and LJ.L. KAD' is the pivot relation fragment. Index creation can start only when the entire tuples of a relation fragment are available because a hashing index requires the number of indexed tuples to be known before an index is created. Since the tuples of relation fragments are transmitted in row-wise order, i.e., different tuples from different relation fragments are intermixed, the index creation on relation fragments can start only after all relation fragments are received. Given the structure of the chained bucket hashing index described in Section 4.3.3.4, the algorithm of creating an index is as follows.

Algorithm 4.3.11 (Index creation using hashing)
Input: a relation fragment F_i, and a join attribute A_i of F_i.
Output: a chained bucket hashing index on the attribute A_i of F_i.
Procedure:

1. Allocate a bucket header table.
2. Scan the column A_i of F_i linearly.
   For each scanned value of $F_i.A_i$,
   (a) Compute the hashed address $h(F_i.A_i)$ where h is a hashing function.
   (b) Insert the value of $F_i.A_i$ into the hashing index at the address $h(F_i.A_i)$.
   /* No duplicate checking is done. */

4.3.3.5.4 Navigational join Once indexes are created and an assembly plan is generated, we perform navigational joins on the relation fragments. The navigational join starts from each tuple of the pivot relation fragment and follows the joins of the join tree to find matching tuples from all relation fragments. For example in Figure 4.15, we perform joins starting from each tuple of KAD′ and find matching tuples from DG′E and A′IL′ respectively. Then, for each matching tuple of DG′E and A′IL′ found in previous joins, matching tuples are found from HG and LJ respectively. Note that there always exist one or more matching tuples because non-matching tuples have already been discarded in the materialization phase.

The set of matching tuples thus found are assembled into a nested tuple according to the assembly plan generated by Algorithm 4.3.10. For example, starting from the third tuple of KAD′, $[k_3a_3d_5]$, the following set of matching tuples are found from each relation fragment as the result of navigation.

Example 4.3.2 (Matching tuples $[k_3a_3d_5]$)

- $[k_3a_3d_5]$ from KAD′.
- $[d_5g_1e_4]$ from DG′E.
- $[h_1g_1], [h_5g_1]$ from HG.
- $[a_3i_2l_3]$ from A′IL′.
- $[l_3j_2], [l_3j_4]$ from LJ.

□

These tuples are assembled into one nested tuple in the last row of Figure 4.15c. Given the assembly plan shown in Example 4.3.1, $[k_3a_3d_5]$ is projected on the projection set KA, $[d_5g_1e_4]$ is projected on DE. $[h_1g_1]$ and $[h_5g_1]$ are not projected because their projection set is the same as the schema of the relation fragment HG. $[a_3i_2l_3]$ from A'IL' is merged with $[l_3j_2]$ and $[l_3j_4]$ from LJ respectively, and projected on IJ to produce $[i_2j_2]$ and $[i_2j_4]$. Duplicate checking is required when the tuples are assembled into a nested tuple because projections may produce duplicate subtuples. Carrying out navigational joins in this way for all tuples of the pivot relation fragment, we obtain the nested relation shown in Figure 4.15c.

The following algorithm describes the procedure of a navigational join more rigorously.

Algorithm 4.3.12 (Navigational join)
Input: relation fragments $F_i, i = 1, 2, \cdots, n$(F_1 is the pivot relation fragment.); indexes on the join attributes of $F_i, i = 2, 3, \cdots, n$; a join tree (JT), a nesting format tree (NFT), and an assembly plan (AP).
Output: a single nested relation (SNR).
Procedure:

1. Allocate an empty single nested relation SNR.
2. $w_p :=$ the root of the empty SNR.
3. $u_p :=$ the root of NFT.
4. For each $t_p \in F_1$, /* F_1 is the pivot relation fragment */.
   Assemble(w_p, u_p, t_p).

Assemble(w_p, u_p, t_p) starts navigation from t_p and collects the set of tuples $\{t_i | t_i \in F_i, i = 2, 3, \cdots, n\}$, that satisfy the join conditions among $F_1, F_2, \cdots, F_n$. Then, it inserts t_p and the collected set of matching tuples into a single nested relation. For each insertion, it first finds, from the assembly plan, the set of relation fragments that are to be merged to produce the tuple to be inserted and their associated assembly plan expression. Secondly, the assembly plan expression is executed on those tuples to

be merged. Only projection is performed if no merging is prescribed in the assembly plan expression. Thirdly, the resulting tuples are inserted to corresponding nested subrelations. Every insertion is preceded by a searching for checking if there already exists a duplicate tuple.

Algorithm 4.3.13 (Assemble_RF)
Input: A node (w_i) of the SNR, a node (u_i) of the NFT, a join tree (JT), an assembly plan (AP), and a tuple t_0 from which we start navigation.
Output: SNR with newly inserted tuples.
Procedure:

1. w_r := the node pointed by $w_i.u_i$.
   /* w_r is the root of the binary search tree of the nested subrelation to be searched. */

2. Find $\mathcal{V} \equiv \{v_1, v2, \cdots, v_k\}$ from AP such that $u_i = \mathrm{AP}(v_1, v_2, \cdots, v_k)$.
   /* $k > 1$ if and only if a merging is required. */

3. /* Let F_i be $\mathrm{RFJT}^{-1}(v_i)$ for $v_i \in \mathcal{V}$, and let Φ_i be the join predicate between F_i and F_j where $\mathrm{RFJT}(F_j)$ is the parent of $\mathrm{RFJT}(F_i)$ in JT. */

   /* Find the tuples from $F_1, F_2, \cdots, F_k$ that match t_0. */
   For each $t_1 \in \mathrm{Match}(t_0, F_1, \Phi_1)$,
   For each $t_2 \in \mathrm{Match}(t_1, F_2, \Phi_2)$,
   $\ddots$
   For each $t_k \in \mathrm{Match}(t_{k-1}, F_k, \Phi_k)$,
   (a) $t_c := \mathrm{AP}(t_1, t_2, \cdots, t_k)$. /* Execute the assembly plan. */
   (b) If $(w_c := \mathrm{Search}(w_r, t_c)) = \mathrm{NOT\_FOUND}$
   then $w_c := \mathrm{Insert}(w_i, u_i, t_c)$.
   (c) Ψ := the set of u_i's children in NFT.
   (d) If $\Psi = \{\}$ then return
   else For each $u_c \in \Psi$,
   $\mathrm{Assemble}(w_c, u_c, t_c)$.

where Search and Insert are the same algorithms as Algorithm 4.3.7 and Algorithm 4.3.9, respectively. No duplicate checking is performed if no projection is required in the assembly plan, although it is not explicitly shown in Algorithm 4.3.13.

Given a tuple $t_i \in F_i$, the matching tuples in another relation fragment F_j, which is connected to F_i through a join condition $A_i \theta A_j$, are collected as follows.

Algorithm 4.3.14 (Match)
Input: a tuple $t_i \in F_i$, a relation fragment F_j, and a join condition $t_i.A\theta t_j.B$ where $t_j \in F_j$.
Output: $\{t_j | t_j \in F_j, t_i.A\theta t_j.B\}$.
Procedure:

1. Compute the address of a bucket header using $t_i.A$ as the hashing key.
2. For each bucket from the bucket header through the end of the chain,
   (a) If $t_i.A\theta t_j.B$ is satisfied then collect the pointer to t_j, where $t_j \in F_j$ is a tuple pointed by the bucket entry. (Remember that each bucket entry contains a pointer to a tuple.)

If the Match process and its subsequent process (the execution of AP, searching for checking duplicates, and insertion to SNR) are pipelined, the pointer to t_j is not collected but passed to projection operator to compute $t_j.\pi_j$.

4.3.3.5.5 Summary In summary, the nesting of relation fragments into a single nested relation is performed as follows.

Algorithm 4.3.15 (RF nesting)
Input: a set of relation fragments, a nesting format tree, and a join tree.
Output: a single nested relation.
Procedure:

1. Purge the joins of the join tree by removing all join predicates from conjunctive join predicates except one arbitrarily selected join.

2. Generate an assembly plan by comparing the join tree and the nesting format tree.

3. Create indexes on the join attribute of each relation fragment except the pivot relation fragment.

4. For each tuple of the pivot relation fragment,

   Perform navigation along the joins of the join tree and find the set of matching tuples from each relation fragment, and assemble the set of matching tuples into a nested tuple according to the assembly plan.

4.3.4 The SNR Method

As mentioned in Section 4.3.1, the materialization of a single nested relation is performed as the materialization of relation fragments followed by the nesting of relation fragments into a single nested relation. Hence it is sufficient to focus only on the modifications needed to adapt the RF method to the SNR method, that is, to move the nesting step to a server and transmit a single nested relation.

Query processing is exactly the same as that of the RF method and therefore, Algorithm 4.3.2 can be used without modification if we use the pipelined nested loop join algorithm. The process of eliminating duplicate tuples from materialized relation fragments is also the same as the one shown in Algorithm 4.3.3 except that tuples are written to an output buffer instead of being transmitted to a client.

Once the tuples of the relation fragments are collected, they are restructured into a single nested relation on a server. The same steps as those of the RF nesting described in Section 4.3.3.5 are used except that the navigational join step is modified so that matching tuples are not only assembled into nested tuples but also transmitted to a client. According to Algorithm 4.3.13, the tuples of nested subrelations are transmitted in a depth-first search order of the nesting format tree. Delimiters are needed to distinguish between the tuples of different nested subrelations. For example, the stream of data transmitted for the single nested relation of Figure 4.15c is as follows. '⟨' and '⟩' are delimiters.

Example 4.3.3

Header: $\langle KA\langle DE\langle HG\rangle\rangle\langle IJ\rangle\rangle$

Data:

$\langle k_2a_2\langle d_4e_2\langle h_2g_3\rangle\rangle\langle d_5e_4\langle h_1g_1h_5g_1\rangle\rangle\langle i_1j_1\rangle\rangle\langle k_1a_1\langle d_2e_2\langle h_2g_3h_4g_2\rangle\rangle\langle i_2j_2i_2j_4\rangle\rangle$

$\langle k_3a_3\langle d_5e_4\langle h_1g_1h_5g_1\rangle\rangle\langle i_2j_2i_2j_4\rangle\rangle$

□

where $\langle KA\langle DE\langle HG\rangle\rangle\langle IJ\rangle\rangle$ is a header describing the format of the data stream following the header. A data stream consists of segments. A segment contains the tuples that will belong to the same nested subrelation when they are assembled into a single nested relation by a client. Example 4.3.3 shows three segments starting with k_1a_1, k_2a_2, and k_3a_3, respectively.

What remains for a client to complete the nesting process is to parse the received data stream and assemble the extracted tuples into a single nested relation. Algorithm 4.3.16 describes the assembly process. For each tuple t_i read from the data stream, t_i is inserted as a nested subtuple of the previous tuple if t_i is preceded by '$\langle$'. Otherwise, t_i is inserted in the same nested subrelation as the previous tuple. In the following algorithm, w_o is the current insertion entry node and w_o is the latest inserted node. The current insertion entry node is moved one level up for each '$\rangle$'. We assume the availability of a function named 'Super' which returns the (super)node for which the node w_p is a nested subtuple. For example, if w_p is the node containing the tuple d_4e_2 in the single nested relation of Figure 4.12, then Super(w_p) returns the node containing the tuple k_2a_2.

Algorithm 4.3.16 (Assemble_SNR)

Input: formatted stream of tuples of a single nested relation, nesting format tree (NFT).

Output: an assembled single nested relation.

Procedure:

1. Allocate an empty single nested relation (SNR).

2. w_o := the root of the empty SNR.

3. For each data item d read from the data stream,

   - If $d =$ '$\langle$' then $w_p := w_o$.
   - If $d = t_i$ (a tuple) then
     Find the schema S_i of t_i from the header.
     $u_p :=$ NSRNFT(S_i).
     $w_o :=$ Insert(w_p, u_p, t_i).
   - If $d =$ '$\rangle$' then $w_o := w_p$; $w_p :=$ Super(w_p).

where the process of the Insert is shown in Algorithm 4.3.9. Note we do not need searching preceding an insertion because duplicates have already been eliminated on a server.

Summarizing, the object instantiation process of the SNR method is executed in the following steps.

- Materialization: Query processing, duplicate elimination, join purge, assembly planning, index creation, and navigational join (and transmission).
- Translation: assembly and reference resolution.

4.3.5 Data Transmitted in Different Methods

In the transmission phase, data prepared by a server is transmitted to the client which sent a query to the server. As mentioned in Section 4.3.2, transmission occurs pipelined with the materialization process. That is, tuples of the materialized query result are transmitted as soon as they become available. As discussed in Section 4.3.1.2, the structure of transmitted data differs for each of the three object instantiation methods, and have different set of redundant data. The RF method and the SNR method obviously remove redundant data transmitted in the SFR method. However, they still have their own source of redundant data.

4.3.5.1 Sources of Redundant Data in the RF and SNR Methods

In the RF method, some relation fragments contain *extra join attributes*. For example, the relation fragment A'IL' shown in Figure 4.15a contains two join attributes A' and L'.

As illustrated in Figure 4.6, the number of tuples emitted for each relation fragment by the query processor is never larger than that for a single flat relation. Besides, if a relation fragment contains no extra join attribute, it is guaranteed that the duplicate elimination step eliminates more tuples from the relation fragment than from the single flat relation that would be retrieved by the SFR method for the same query. In other words, the relation fragment do not contain more tuples than the corresponding single flat relation. An exception may occur if a relation fragment does have extra join attributes whose combined domain cardinality (the number of distinct values) is higher than the combined domain cardinality of the other attributes. In that case, it may happen that less tuples are eliminated from the relation fragment. We anticipate that this situation happens rarely.

In the SNR method, a server transmits a linearized stream of nested tuples. The stream of data contains no duplicate subtuples unlike the SFR method[3], and no extra join attribute unlike the RF method. However, a subtuple is transmitted *multiple* times if it belongs to multiple tuples of the transmitted nested relation. This phenomenon occurs when there is a many-to-many cardinality relationships between two joined relations. For example in the formatted stream of Example 4.3.3, the tuple h_2g_3 of the subrelation `HG` appears as the subtuples of two different tuples of `DE`. Likewise, h_1g_1 and h_5g_1 also appear twice in different tuples of the nested relation.

4.3.5.2 Trade-offs between Different Methods

It is certain that the method which incurs the minimum transmission cost is the one that produces the least amount of redundant data. We observe that the RF method has a trade-off with the SFR method depending on which is larger between the amount of redundant data eliminated by the fragmented materialization of query result and the amount of redundant data introduced due to the extra join attributes. Besides, there is a trade-off between the RF method and the SNR method depending on which is larger between the overhead of the extra join attributes in the RF method and the overhead of the multiple subtuple occurrences in the SNR method. On the other hand, the SNR method always transmits less amount of data than the SFR method

[3]and no null subtuples if we were considering outer joins

because a single nested relation cannot have more tuples than its corresponding single flat relation.

We anticipate that, in practical cases, the transmission costs of the RF method and the SNR method are comparable, and either method can be more efficient depending on the query. Besides, we anticipate that the redundancy due to the extra join attributes in the RF method is insignificant compared to the redundancy of the duplicate subtuples in the SFR method.

4.4 Development of a Cost Model

4.4.1 A Platform for Cost Modeling

In this section, we develop cost formulas for each step of the three object instantiation methods. It is a too complicated task to obtain a cost model of main memory-resident operations [83] if it is ever possible. As for the cost model of disk-based operations, it is sufficient to count the number of page reads/writes in accessing disks, or including the cost of buffer management together if higher precision is needed. However, the cost of main memory-resident operations depends on so many factors such as the hardware used, programming language, programming style, and system load. Our purpose is to compare the costs of different object instantiation methods, rather than to estimate the costs. In other words, our concern is to find out which method among the SFR, RF, and SNR methods is the winner given a couple of different situations within our range of interest.

Thus, we make necessary approximations and simplifications in the forthcoming cost modeling and cost comparison. First, as mentioned in Section 4.2.4, the cost items that are common to all three methods are excluded from consideration. More specifically, we omit the query processing cost from the materialization phase and the reference resolution cost from the translation phase. Secondly, we exclude the cost of accessing schema information from our cost models and consider only the cost of operations on data tuples. Schema access cost becomes negligible when the number of manipulated tuples becomes large enough. Thirdly, we ignore the effect of the

difference in the speed of a server and a client. Even if their speeds are noticeably different, its effect on the cost comparison result is minimal in the environment where the network communication cost is significant.

We consider only complex queries – queries with one or more joins – to develop our cost model. In other words, we consider only the case of $n_r > 1$ in our cost model. The SFR, RF, and SNR methods become identical if a query is a simple query, i.e., has no join. That is, the base relation specified in a simple query is reduced to a selected and projected fragment, transmitted to a client, and linked to other objects through reference resolution step. Nesting step is not needed for the single fragment.

4.4.1.1 Cost Parameters

Table 4.1 and Table 4.2 show the values of cost parameters for elementary main memory operations and network communications, respectively, that are used in our cost formulas. We have experimented with both the CPU time and elapsed time for measuring main memory cost parameters. Our experiment showed that the elapsed time varied significantly at every run depending on the system load. On the other hand, the CPU time was measured to be stable. Therefore, we chose the *CPU time* as an appropriate measure of the main memory execution time. As for the network communication time, CPU time did not show any noticeable difference between LAN and WAN while elapsed time did show a big (20 times) difference. Our experiment showed that most of the elapsed time for a WAN was spent on the communication network which carries the data. Local processes were blocked until data arrives. On the other hand, we verified that most of the elapsed time for a LAN was spent on local hosts for sending and receiving data. These facts lead us to the conclusion that the *elapsed time* was more appropriate for measuring communication cost parameters. Thus, we used different measures for the main memory cost parameters and communication cost parameters. Appendix A explains how the values of the cost parameters were obtained. Precisely speaking, the values of the cost parameters of main memory operations are different on a server and a client. Nevertheless we use the same cost values for both the server and the client as an approximation.

Parameter	Description	Value
C_{bs}	The cost of elementary binary search operation (compare and move left or right).	19 μsec
C_{cm}	The cost of comparing two tuples.	9.2 μsec
C_{ci}	The initial cost of copying a tuple.	11 μsec
C_{cb}	The per-byte cost of copying a tuple.	0.17 μsec/byte
C_e	The cost of evaluating a join predicate (equijoin on attributes of type integer).	16 μsec
C_{fl}	The per-byte cost of folding a tuple into an integer.	0.92 μsec/byte
C_{hc}	The cost of hashed address computation using an integer hashing key.	9.5 μsec
C_{ma}	The cost of allocating memory within workspace.	1.2 μsec
C_{mp}	The cost of moving (reading or writing) a pointer.	0.88 μsec
C_{pi}	The initial cost of performing a projection on a tuple.	4.3¡ μsec
C_{pb}	The per-byte cost of performing a projection on a tuple.	1.1 μsec/byte
C_{si}	The initial cost of computing an integer hashing key from a scanned relation column.	17 μsec
C_{sn}	The per-tuple cost of computing an integer hashing key from a scanned relation column.	14 μsec/tuple

Table 4.1: Main memory cost parameters (CPU time)

Parameter	Description	Value	
		LAN	WAN
C_l	The latency of sending a message.	2.5 msec	53 msec
C_b	The per-byte data transmission cost.	3.4 μsec/byte	60 μsec/byte

Table 4.2: Communication cost parameters (elapsed time)

SFR (T)	
Parameter	Description
N_t	The cardinality after duplicate elimination.
d_t	The ratio between the cardinality after duplicate elimination and the cardinality before duplicate elimination. ($d_t \leq 1.$)
T_t	Tuple size.
RF ($F_i, i = 1, 2, \cdots, n_f$ where F_1 is the pivot relation fragment)	
n_f	The number of relation fragments.
N_{f_i}	The cardinality of F_i after duplicate elimination.
d_{f_i}	The ratio between the cardinality after duplicate elimination and the cardinality before duplicate elimination. ($d_{f_i} \leq 1.$)
$D_{f_{ij}}$	The domain cardinality – the number of distinct values – of the join column of F_j for the join between F_i and F_j.
T_{f_i}	The tuple size of F_i.
ρ_{f_i}	The extra join attribute (EJA) ratio, i.e., the ratio between the size of extra join attributes in F_i and the tuple size of F_i. ($\rho_{f_i} \leq 1.$)
SNR ($S_i, i = 1, 2, \cdots, n_s$ where S_1 is the pivot nested subrelation)	
n_s	The number of nested subrelations in a single nested relation.
N_{s_i}	The cardinality of S_i.
T_{s_i}	The tuple size of S_i.

Table 4.3: Data Parameters

We use the following short-hand notations in our cost formulas.

$$C_{colscan}(N) = C_{si} + C_{sn}N \text{ for scanning } N \text{ tuples.} \quad (4.6)$$

$$C_{copy}(T) = C_{ci} + C_{cb}T \text{ for copying a tuple of size } T \text{ bytes.} \quad (4.7)$$

$$C_{project}(T) = C_{pi} + C_{pb}T \text{ for projecting a subtuple of size } T \text{ bytes out of a tuple.} \quad (4.8)$$

4.4.1.2 Data Parameters

The paramaters of the data transmitted in different object instantiation methods are shown in Table 4.3.

4.4.1.2.1 Alternative Parameters: α_{ij} and β_{ij} We define α_{ij} as the domain selectivity of the join column of a relation fragment F_j, i.e., the average number of tuples with the same value of a join column (after duplicate elimination), for the join between F_i and F_j. Thus, the value of α_{ij} is related to N_{f_j} and $D_{f_{ij}}$ as follows.

$$\alpha_{ij} = \frac{N_{f_j}}{D_{f_{ij}}} \tag{4.9}$$

Since relation fragments are fully reduced before they are joined again on a client, two joined relation fragments F_i and F_j have the same domain cardinality of their join columns, i.e., $D_{f_{ji}} = D_{f_{ij}}$. Hence, α_{ij} can be interpreted to denote the average number of matching tuples in F_j for each tuple of F_i. We call α_{ij} as *selectivity* from F_i to F_j. Since $D_{f_{ji}} = D_{f_{ij}}$, the following is always true.

$$N_{f_j} \leq N_{f_i}\alpha_{ij} \tag{4.10}$$

where the equality holds if and only if $D_{f_{ji}} = N_{f_i}$, that is, all values in the join column of F_i for the join to F_i are unique.

β_{ij} is defined as the *average degree of nesting*, i.e., the average number of tuples in S_j for each tuple of S_i where S_j is a direct nested subrelation of S_i. Given S_i and its nested subrelation S_j, β_{ij} can be interpreted as the ratio between N_{s_i} and N_{s_j}.

$$\beta_{ij} = \frac{N_{s_j}}{N_{s_i}} \tag{4.11}$$

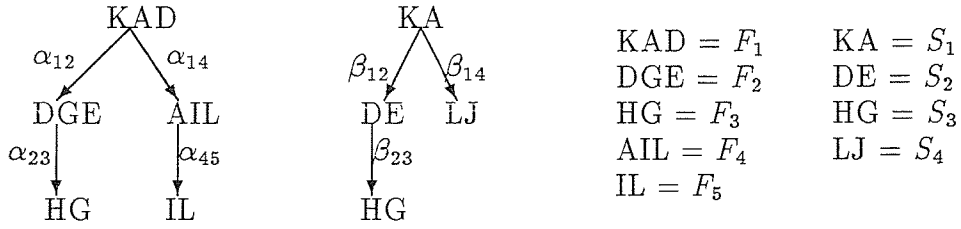
$\beta_{ij} \geq 1$ since we are considering only inner joins.

Figure 4.17 contrasts α_{ij}'s and β_{ij}'s in accordance with the join tree and the nesting format tree of Figure 4.16. $\{\alpha_{ij}\}$ maps onto $\{\beta_{ij}\}$. Note some α_{ij}'s map to the same β_{ij} if two or more relation fragments are merged to become a single nested subrelation. For those α_{ij}'s and β_{ij}'s that are mapping counterparts, $\alpha_{ij} \neq \beta_{ij}$ in general.

From Equation 4.11, we can derive the value of N_{s_i} as follows.

$$N_{s_i} = N_{s_1} \prod_{(\text{NSRNFT}(S_p),\text{NSRNFT}(S_q)) \in P_{1i}} \beta_{pq} \text{ for } i = 2, 3, \cdots, n_s \tag{4.12}$$

where P_{1i} denotes the path from NSRNFT(S_1) to NSRNFT(S_i) in the nesting format tree. That is, N_{s_i} is computed as N_{s_1} multiplied by all β_{pq}'s between S_1 and S_i.



(a) Join tree (b) Nesting format tree

Figure 4.17: α_{ij} vs. β_{ij}

4.4.1.2.2 Relationships between Data Parameters Provided with the same query, the data parameters of different methods shown in Table 4.3 are not independent of one another but are related by some quantitative relationships. Let us now discuss the relationships between different data parameters.

SFR vs. RF: It was indicated in Section 4.2.2 that the cardinality (N_t) of a single flat relation increases monotonically with respect to the cardinality (N_{f_1}) of the pivot relation fragment multiplied by the α_{ij}'s of all joins ($F_i \bowtie F_j$'s) in the query, independent of the query shape. There are $n_f - 1$ joins among the relation fragments after the purging step of Algorithm 4.3.15. Hence,

$$N_t \leq N_{f_1} \prod_{\langle \mathrm{RFJT}(F_i), \mathrm{RFJT}(F_j) \rangle \in E(\mathrm{JT})} \alpha_{ij} \tag{4.13}$$

where $E(\mathrm{JT})$ denotes the set of the edges of the join tree JT. $\langle \mathrm{RFJT}(F_i), \mathrm{RFJT}(F_j) \rangle \in E(\mathrm{JT})$ means that there is a join between F_i and F_j. The equality holds true if (not only if) there is no extra join attribute in $F_i, i = 1, 2, \cdots, n_f$.

Since relation fragments may have extra join attributes while a single flat relation has no such extra attribute, the following relationship exists between T_t and T_{f_i}'s.

$$T_t = \sum_{i=1}^{n_f} T_{f_i}(1 - \rho_{f_i}) \tag{4.14}$$

SFR vs. SNR: We can think of N_t as the number of tuples generated when we 'flatten' the nested tuples of a corresponding single flat relation. The cardinality of the pivot subrelation S_1 is N_{s_1}, and a single tuple of a nested subrelation S_i is

replicated β_{ij} times when it is flattened with its nested subrelation S_j. Hence,

$$N_t = N_{s_1} \prod_{\langle \text{NSRNFT}(S_i), \text{NSRNFT}(S_j) \rangle \in E(\text{NFT})} \beta_{ij} \tag{4.15}$$

where $E(\text{NFT})$ denotes the set of edges in the nesting format tree NFT. $\langle \text{NSRNFT}(S_i), \text{NSRNFT}(S_j) \rangle \in E(\text{NFT})$ means that S_j is a (direct) nested subrelation of S_j.

The attributes of the flattened relation are composed of the *atomic* attributes of the original nested relation. Since flattening does neither add nor remove any of the atomic attributes, we obtain the following relationship between the tuple sizes of the flat relation and the original nested relation.

$$T_t = \sum_{k=1}^{n_s} T_{s_k} \tag{4.16}$$

RF vs. SNR: As mentioned in Section 4.3.3.5.2, the number of the nodes of join tree is no more than the number of the nodes of a nesting format tree because two or more relation fragments can be merged to a single nested subrelation of the single nested relation. Hence,

$$n_s \leq n_f \tag{4.17}$$

The cardinality of the pivot relation fragment and the cardinality of the pivot nested subrelation are always equal because both contain the key of the pivot relation.

$$N_{f_1} = N_{s_1} \tag{4.18}$$

4.4.2 Derivation of Cost Formulas

In this section, we develop cost formulas of each step of object instantiation. Remember the query processing cost and the reference resolution cost are not included in our partial cost model.

4.4.2.1 Duplicate Elimination Cost

The duplicate elimination process (Algorithm 4.3.3) is the same for all three methods except that it is applied to a single flat relation for the SFR method, and to each relation fragment for the RF or SNR method. Therefore, it is clear that the cost

of the duplicate elimination is proportional to $(N_t/d_t)T_t$ for the SFR method and to $\sum_{i=1}^{n_f}(N_{f_i}/d_{f_i})T_{f_i}$ for the RF or SNR method.

We make the following assumptions for the hashing of tuples which was described in Algorithm 4.3.3.

- We allocate as many bucket headers as half of the cardinality of a hashed relation (query processing output), and the cardinality of the hashed relation is estimated by a query optimizer.
- The shift folding technique [81, 82] is used for the hashing of tuples. In this technique, a tuple is partitioned into several parts of an integer size. All but the last parts have the same length. The parts are then added together to obtain an integer hashing key.

Given these assumptions, the cost of eliminating duplicate tuples from a hashed relation is derived as follows. Let N be the cardinality of the relation after duplicate elimination and T be the tuple size of the relation, and d be the ratio of the cardinality after duplicate elimination over the cardinality before duplicate elimination ($d \leq 1$).

The allocation of a bucket header costs C_{ma}. Step 2 of Algorithm 4.3.3 is repeated N/d times. The cost of computing a hashed address using the shifted folding technique is computed as a function of the tuple size T as follows.

$$C_{tuphash}(T) = C_{fl}T + C_{hc} \tag{4.19}$$

Among the N/d hashed tuples, N tuples are actually inserted and the other $N/d - N$ tuples are discared. Therefore, the probability of a tuple being inserted is d and the probability of being discarded is $1 - d$. If the same tuple already exists, it takes the cost of traversing average half of a bucket chain $C_{mp} + (N_b/2)(C_{cm} + C_{mp})$ where N_b is the number of buckets inserted in the chain so far. Otherwise, it cost the traversal of the entire bucket chain $(C_{mp} + N_b(C_{cm} + C_{mp}))$, and the insertion of a new bucket in the chain $(C_{ma} + C_{copy}(T) + 2C_{mp})$.

N_b is obtained as follows. It was assumed that the size of a bucket header table we allocate is 50% of the cardinality of the hashed relation. That is, $N/2d$ headers are allocated, and N hashed entries are eventually inserted into these headers. If

$N > N/2d$, i.e., $d > 1/2$, all buckets headers are eventually filled, assuming the hash function distributes a hashing key uniformly over the bucket header table. In this case, the ultimate value of N_b becomes $N/(N/2d) = 2d$. Otherwise, if $d \leq 1/2$, only N bucket headers out of $N/2d$ headers are filled and the ultimate value of N_b becomes 1. As for the intermediate value of N_b in the middle of insertions, we use half of the ultimate value as an expected value. Thus,

$$N_b = \text{MAX}\left(d, \frac{1}{2}\right) \tag{4.20}$$

The cost of transmitting the inserted tuple is part of the transmission cost and is not included here. Thus, the cost of inserting a hashed tuple into a chain of hashing buckets is computed as a function of T and d as follows.

$$\begin{aligned} C_{tupinsert}(d,T) &= d(C_{mp} + N_b(C_{cm} + C_{mp}) + C_{ma} + C_{copy}(T) + 2C_{mp}) + \\ &\quad (1-d)(C_{mp} + \frac{N_b}{2}(C_{cm} + C_{mp})) \end{aligned} \tag{4.21}$$

where the value of N_b is computed as follows.

Using Equation 4.19 and Equation 4.21, the SFR duplicate elimination cost is computed as follows.

$$C_{sfrde} = C_{ma} + \frac{N_t}{d_t}(C_{tuphash}(T_t) + C_{tupinsert}(d_t, T_t)) \tag{4.22}$$

The cost of eliminating duplicate tuples from relation fragments $F_i, i = 1, 2, \cdots, n_f$, is computed as follows.

$$C_{rfde} = \sum_{i=1}^{n_f}(C_{ma} + \frac{N_{f_i}}{d_{f_i}}(C_{tuphash}(T_{f_i}) + C_{tupinsert}(d_{f_i}, T_{f_i}))) \tag{4.23}$$

Since the query result of the SNR method is also a set of relation fragments, its duplicate elimination cost is the same as that of the RF method except that it incurs the additional cost of writing non-duplicate tuples to an output buffer instead of transmitting them to a client as in the RF method. The cost of writing non-duplicate tuples from a relation fragment F_i to an output buffer is $C_{copy}(T_{f_i})N_{f_i}$. Thus, the cost of eliminating duplicate tuples from $F_i, i = 1, 2, \cdots, n_f$, to be used in the SNR method is computed as follows.

$$C_{snrde} = C_{rfde} + \sum_{i=1}^{n_f} C_{copy}(T_{f_i})N_{f_i} \tag{4.24}$$

4.4.2.2 Nesting Cost

4.4.2.2.1 Binary Search Tree Searching and Insertion Costs The searching (Algorithm 4.3.7) and insertion of one tuple (Algorithm 4.3.9) are used commonly for all three object instantiation methods. Hence, we deal with their cost formulas separately here. We assume all binary search trees implementing nested subrelations are well-balanced. In fact, well-balanced trees are common and degenerate trees are very rare [92]. Even if a binary tree should be balanced sometimes, a tree balancing involves only pointer movements and incurs negligible cost.

Let M be the number of tuples that are attempted to be inserted into a binary search tree. Every attempt of insertion requires one searching to check if the same tuple has already been inserted into the binary search tree. Let N denote the number of tuples that are actually inserted into a binary search tree. According to Knuth [92], a single searching requires about $1.386 \log_2 k$ comparisons (k is the number of nodes currently in the binary search tree) for a well-balanced binary search tree, considering both a successful search and an unsuccessful search. If we assume the insertion of the N tuples out of M tuples occurs at a regular interval, the value of k is incremented at every M/N insertion attempts. Then, the total searching cost for inserting N tuples out of the attempted M tuples is computed as follows.

$$C_{binsearch}(M, N) = \sum_{k=1}^{N} (\frac{M}{N} 1.386 C_{bs} \log_2 k) \tag{4.25}$$

Insertion cost is the sum of the cost of an unsuccessful searching and the cost of inserting a node as a leaf of the binary search tree. An unsuccessful searching of a binary search tree requires $log_2(k+1)$ comparisons. Node insertion at the leaf requires the allocation of an empty node (C_{ma}), copying tuple into the node ($C_{copy}(T)$), and writing a pointer to the node in the parent node (C_{mp}). Thus, the total cost of inserting N tuples to a binary search tree is computed as follows.

$$C_{bininsert}(N, T) = \sum_{k=1}^{N} (C_{bs} \log_2(k+1) + C_{ma} + C_{copy}(T) + C_{mp}) \tag{4.26}$$

There will be N_{s_i} tuples inserted into a nested subrelation S_i of the final output single nested relation. Let $S_{par(i)}$ denote the nested subrelation such that NSRNFT($S_{par(i)}$)

is the parent of NSRNFT(S_i). Then, there exist $N_{s_{par(i)}}$ binary search trees implementing the nested subrelation S_i, i.e., one binary search tree for each tuple of $S_{par(i)}$. Let M_{s_i} denote the number of tuples that are attempted to be inserted into S_i. If we assume tuples are uniformly distributed into every binary search tree of S_i, $M_{s_i}/N_{s_{par(i)}}$ tuples are attempted for insertion and $N_{s_i}/N_{s_{par(i)}}$ tuples are actually inserted into each binary search tree of S_i. Thus, the total cost of inserting N_{s_i} tuples into S_i out of the attempted M_{s_i} tuples is computed as follows.

$$C_{sisearch}(M_{s_i}, N_{s_i}, N_{s_{par(i)}}) = N_{s_{par(i)}} C_{binsearch}(\frac{M_{s_i}}{N_{s_{par(i)}}}, \frac{N_{s_i}}{N_{s_{par(i)}}}) \tag{4.27}$$

$$C_{siinsert}(N_{s_i}, T_{s_i}, N_{s_{par(i)}}) = N_{s_{par(i)}} C_{bininsert}(\frac{N_{s_i}}{N_{s_{par(i)}}}, T_{s_i}) \tag{4.28}$$

4.4.2.2.2 SFR Nesting Cost We consider only the costs of projecting tuples, searching tuples (Algorithm 4.3.7), and inserting tuples (Algorithm 4.3.8), which are the operations on data tuples and whose costs are dominant.

According to Algorithm 4.3.5, N_t tuples are assembled to a single nested relation. Each one of the N_t tuples is decomposed into subtuples belonging to different nested subrelations $S_1, S_2, \cdots, S_{n_s}$ by projections. For each of the N_t tuples, the projection of the tuple on the schema of S_i costs $C_{project}(T_{s_i})$. The searching of S_i for the projected subtuple costs $C_{sisearch}(N_t, N_{s_i}, N_{s_{par(i)}})$, and the insertion of the projected subtuple into S_i costs $C_{siinsert}(N_{s_i}, T_{s_i}, N_{s_{par(i)}})$. Hence, the total cost of assembling N_t tuples of a single flat relation into a single nested relation is computed as follows.

$$C_{sfrnest} = \sum_{i=1}^{n_s} (C_{project}(T_{s_i}) N_t + C_{sisearch}(N_t, N_{s_i}, N_{s_{par(i)}}) + C_{siinsert}(N_{s_i}, T_{s_i}, N_{s_{par(i)}})) \tag{4.29}$$

4.4.2.2.3 RF Nesting Cost We ignore the costs of the join purge step and the assembly planning step because they are not operations on data tuples. Accordingly, we approximate the RF nesting cost as the sum of the index creation cost and the navigational join cost.

$$C_{rfnest} = C_{ixcrt} + C_{navjn} \tag{4.30}$$

The number of joins among the relation fragments is always one less than the number of the relation fragments ($n_f - 1$) after the join purge step.

Index creation (Algorithm 4.3.11): For a relation fragment F_i, the cost of bucket header allocation is C_{ma}. The linear scan costs $C_{colscan}(N_{f_i})$. As for the hashing of join column, we assume all join attributes are integers so that no folding is required. For each of the N_f scanned join column values, hashing computation costs C_{hc} and insertion to a hashing bucket chain takes the cost of allocating a bucket (C_{ma}), writing a pointer (C_{mp}) to the tuple containing the hashed attribute, and two pointer writings ($2C_{mp}$) to make connections to other buckets. Note we do not need to scan the entire chain of buckets because no duplicate checking is required. Hence, the cost of creating $n_f - 1$ indexes on $F_i.A_i$'s for $i = 2, 3, \cdots, n_f$, where F_1 is the pivot relation fragment, is computed as follows.

$$C_{ixcrt} = \sum_{i=2}^{n_f}(C_{ma} + C_{colscan}(N_{f_i}) + (C_{hc} + C_{ma} + 3C_{mp})N_{f_i}) \tag{4.31}$$

Navigational join (Algorithm 4.3.12): The allocation of an empty single nested relation costs C_{ma}. As for the assembly cost (Algorithm 4.3.13), we consider only the costs of following operations on data tuples: the cost of finding matching tuples (Algorithm 4.3.14), the cost of executing assembly plans (AP) on the found tuples, and the cost of inserting (Algorithm 4.3.9) the resulting tuples into the single nested relation after checking for duplicate tuples (Algorithm 4.3.7).

Matching (Algorithm 4.3.14): The cost of Match$(t_i, F_j, t_i.A\theta t_j.B)$, denoted by $C_{match_{ij}}$, is computed as follows. First, hashing of a join column costs C_{hc}. Let N_b denote the expected length of the chain of buckets including the header bucket. Then, in Step 2, it costs $N_b(2C_{mp} + C_e)$ to follow the chain of buckets – one C_{mp} for reading a pointer to a tuple $t_j \in F_j$, the other C_{mp} for reading the pointer to next bucket, and C_e for evaluating the join predicate $t_i.A\theta t_j.B$. α_{ij} tuples of F_j are collected from Match$(t_i, F_j, t_i.A\theta t_j.B)$. The collection of matching tuples incurs only the cost of writing α_{ij} pointers ($C_{mp}\alpha_{ij}$). Thus, the cost of finding matching tuples of F_j for each tuple t_i of F_i is computed as a function of α_{ij} as follows.

$$C_{match_{ij}}(\alpha_{ij}) = C_{hc} + N_b(2C_{mp} + C_e) + C_{mp}\alpha_{ij} \tag{4.32}$$

where the value of N_b is obtained as:

$$N_b = \mathrm{MAX}\left(N_{f_j}/D_{f_{ij}}, 2\right) \quad (4.33)$$

$$= \mathrm{MAX}(\alpha_{ij}, 2) \text{ by Equation 4.9} \quad (4.34)$$

in the same way we obtained the ultimate value of N_b for Equation 4.20. As mentioned in Section 4.3.3.4, we assume the allocate bucket header size is 50% of the cardinality of a hashed relation fragment.

The cost of the entire matching process is the sum of the cost of linear scan on the pivot relation fragment ($C_{colscan}(N_{f_1})$) and the cost of finding matching tuples from the other relation fragments.

$$C_{match} = C_{colscan}(N_{f_1}) + \sum_{i \notin \mathrm{Leaf(JT)}} L_{f_i} C_{match_{ij}}(\alpha_{ij}) \quad (4.35)$$

where Leaf(JT) denotes the set of the leaves of the join tree JT and L_{f_i} is obtained as follows.

$$L_{f_i} = N_{f_1} \prod_{\langle \mathrm{RFJT}(F_p), \mathrm{RFJT}(F_q) \rangle \in P_{1i}} \alpha_{pq} \quad (4.36)$$

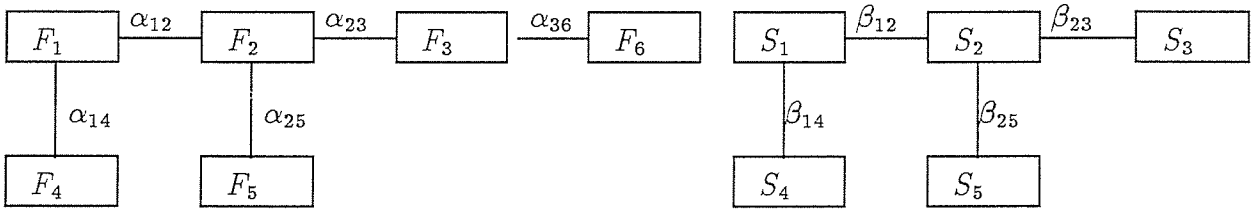
where P_{1i} is a path from RFJT(F_1) to RFJT(F_i).

Execution of assembly plans (Step 3a of Algorithm 4.3.13): The tuples of relation fragments that are found by the matching process are merged according to the prescription of the assembly plan. Let m_i be the number of relation fragments whose tuples are merged to produce tuples to be inserted into a nested subrelation S_i, and let $T'_{s_j}, j = 1, 2, \cdots, m_i$, denote the size of the attributes projected from each one of the to-be-merged relation fragments. Then, the following equation holds true.

$$T_{s_i} = \sum_{j=1}^{m_i} T'_{s_j} \quad (4.37)$$

The case of merging two tuples from two relation fragments requires two projections on the tuples. Extending from this case, the cost of merging m_i tuples from m_i relation fragments into the tuple of a nested subrelation S_i is obtained as $\sum_{j=1}^{m_i} C_{project}(T'_{s_j})$. This formula can be rewritten as a function of T_{s_i} and m_i using Equation 4.8 and Equation 4.37.

$$C_{apexec_i}(T_{s_i}, m_i) = (m_i - 1)C_{pi} + C_{project}(T_{s_i}) \quad (4.38)$$



(a) Join tree (b) Nesting format tree

$S_1 = \Pi F_1$	$\Gamma_1 = \{F_1\}$	$\gamma_1 = 1$
$S_2 = \Pi(F_2 \bowtie F_3)$	$\Gamma_2 = \{F_2, F_3\}$	$\gamma_2 = \alpha_{12}\alpha_{23}$
$S_3 = \Pi F_6$	$\Gamma_3 = \{F_6\}$	$\gamma_3 = \alpha_{36}$
$S_4 = \Pi F_4$	$\Gamma_4 = \{F_4\}$	$\gamma_4 = \alpha_{14}$
$S_5 = \Pi F_5$	$\Gamma_5 = \{F_5\}$	$\gamma_5 = \alpha_{25}$
(c) Assembly plan	(d) Γ partition	(e) γ_i

Figure 4.18: An example of Γ_k and γ_k

Since n_s nested subrelations are produced out of n_f relation fragments, $n_f - n_s$ mergings occur. It depends on a query to determine which relation fragments are merged to produce each nested subrelation S_i. Let us consider a set of $n_f - 1$ α_{ij}'s that are defined on n_f relation fragments. We define a partition on this set, i.e., $[\Gamma_1|\Gamma_2|\cdots|\Gamma_{n_s}]$ where each $\Gamma_k, k = 1, 2, \cdots, n_s$, is the set of F_i's that are merged to produce tuples to be inserted into a nested subrelation S_k. Let γ_k denote the combined value of the α_{ij}'s to the F_j's belonging to Γ_k and be defined as follows.

$$\gamma_k = \prod_{F_j \in \Gamma_k} \alpha_{ij} \text{ where } \alpha_{i1} = 1 \tag{4.39}$$

Figure 4.18 shows an example of Γ_k and γ_k. Note the m_k of Equation 4.38 is equal to the number of F_i's in Γ_k.

Given Equation 4.39, the total cost of executing assembly plans is computed as follows.

$$C_{apexec} = \sum_{i=1}^{n_s} M_{f_i} C_{apexec_i}(T_{s_i}, m_i) \tag{4.40}$$

where M_{f_i} is the number of tuples produced for S_i and computed as follows.

$$M_{f_i} = N_{f_1} \prod_{\text{NSRNFT}(S_p) \in P_{1i}} \gamma_p \tag{4.41}$$

where P_{1i} is the path from NSRNFT(S_1), which is the root of the nesting format tree, to NSRNFT(S_i).

Searching (Algorithm 4.3.7) and *Insertion* (Algorithm 4.3.9): The M_{f_i} tuples produced by the execution of assembly plans are attempted to be inserted into a nested subrelation S_i. Then, the searching cost becomes $\sum_{i=1}^{n_s} C_{sisearch}(M_{f_i}, N_{s_i}, N_{s_{par(i)}})$ and the insertion cost becomes $\sum_{i=1}^{n_s} C_{siinsert}(N_{s_i}, T_{s_i}, N_{s_{par(i)}})$ using Equation 4.27 and Equation 4.28.

Thus, the total cost of performing navigational joins on relation fragments is obtained as follows.

$$C_{navjn} = C_{match} + C_{apexec} + \sum_{i=1}^{n_s}(C_{sisearch}(M_{f_i}, N_{s_i}, N_{s_{par(i)}}) + C_{siinsert}(N_{s_i}, T_{s_i}, N_{s_{par(i)}})) \tag{4.42}$$

4.4.2.2.4 SNR Nesting Cost and Assembly Cost *Nesting*: As mentioned in Section 4.3.4, the SNR method uses the same nesting process on a server as the RF method except that the navigational join process is modified so that tuples that are inserted into a single nested relation are transmitted to a client as well. The transmission cost is considered separately in Section 4.4.2.3 and not considered here. Since we ignore the difference between server speed and client speed, the SNR nesting cost is the same as the RF nesting cost.

$$C_{snrnest} = C_{rfnest} \tag{4.43}$$

Assembly (Algorithm 4.3.16): There is an additional cost of assembling the received data stream into a single nested relation on a client. Considering the cost of operations on tuples only, the cost of assembling the received data stream is computed using Equation 4.28.

$$C_{snrassem} = \sum_{i=1}^{n_s} C_{siinsert}(N_{s_i}, T_{s_i}, N_{s_{par(i)}}) \tag{4.44}$$

4.4.2.3 Transmission Cost

We use a simple model [62] of data transmission cost defined as follows .

$$\text{Transmission cost } = C_l + C_b \times \text{Size} \tag{4.45}$$

where Size is the number of bytes of the transmitted data.

In the SFR method, the amount of transmitted data is equal to the size (in bytes) of a single flat relation, i.e., $N_t T_t$, and hence, the transmission cost is as follows.

$$C_{sfrtx} = C_l + C_b N_t T_t \tag{4.46}$$

On the other hand, in the RF method, the amount of transmitted data is the sum of the sizes $(N_{f_i} T_{f_i}, i = 1, 2, \cdots, n_f)$ of relation fragments.

$$C_{rftx} = C_l + C_b \sum_{i=1}^{n_f} N_{f_i} T_{f_i} \tag{4.47}$$

In the SNR method, if we ignore the size of the header and delimiters because it is trivial, the amount of transmitted data is the sum of the sizes $(N_{s_i} T_{s_i}, i = 1, 2, \cdots, n_s)$ of nested subrelations.

$$C_{snrtx} = C_l + C_b \sum_{i=1}^{n_s} N_{s_i} T_{s_i} \tag{4.48}$$

4.5 Comparison of Costs

In this section, we compare the costs of the three different object instantiation methods using the cost model developed in Section 4.4.2. Table 4.4 shows the distribution of cost items which have been used in our cost model. Note $C_{queryproc}$ and C_{refres} are not part of our cost model.

We first discuss the input data parameters that were used for cost comparison and introduce the selectivity (α_{ij}) and EJA ratios (ρ_{f_i}) as the variant input data parameters. Then, we present the results of cost comparison. We carried out the cost comparison in two different ways: sample case test and simulation. We first show the costs of the SFR, RF, and SNR methods by simulations using randomly generated values of data parameters. Then, we compare the costs using sample data

Method	Server	Network	Client
SFR	$C_{queryproc}, C_{sfrde}$	C_{sfrtx}	$C_{sfrnest}, C_{refres}$
RF	$C_{queryproc}, C_{rfde}$	C_{rftx}	C_{rfnest}, C_{refres}
SNR	$C_{queryproc}, C_{rfde}, C_{rfnest}$	C_{snrtx}	$C_{snrassem}, C_{refres}$

Table 4.4: Distribution of cost items

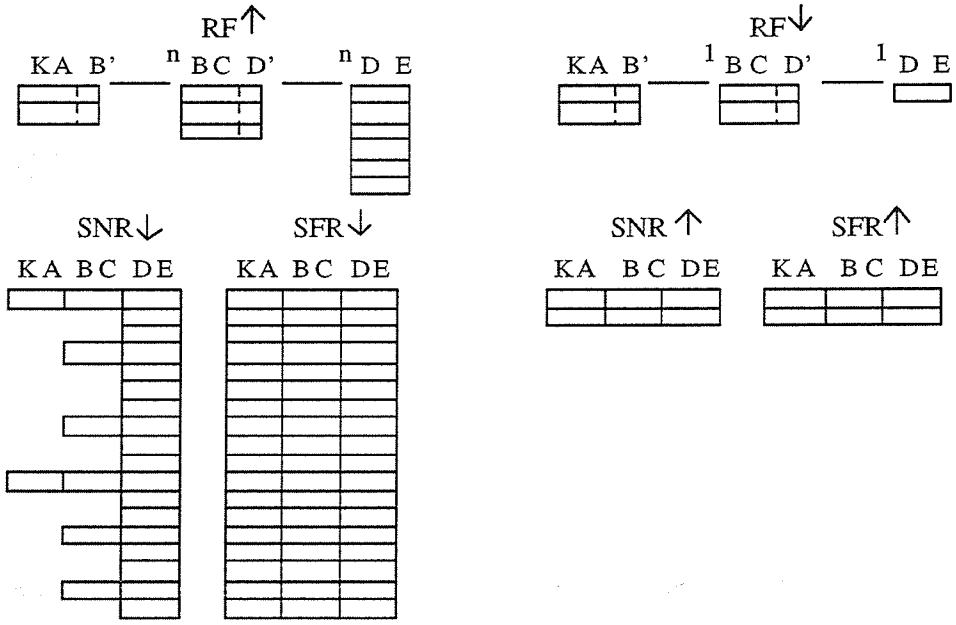
parameters and observe the dependency of costs on the values of selectivities and EJA ratios. The observed result is reinforced by another round of simulation using random values of data parameters, this time with biases given to the domains of the values of selectivities and EJA ratios relatively to the original domains.

4.5.1 Input Data Parameters

We used the data parameters of the RF method as the base set of input data parameters and derived the values of the data parameters of the SFR method and the SNR method using the relationships we have developed in Section 4.4.1.2.2. Besides, based on our discussion of the amount of transmitted data in Section 4.3.5, we have chosen two data parameters, the selectivity (α_{ij}'s) and the extra join attribute (EJA) ratio (ρ_{f_i}'s), as the variant input parameters. The value of α_{ij} is an indicator of the overhead of the duplicate subtuples in the SFR method and the multiply occurring subtuples in the SNR method. The value of ρ_{f_i} is an indicator of the overhead due to the extra join attributes in the RF method. The examples shown in Figure 4.19 and Figure 4.20 illustrate how the values of the selectivity and the EJA ratios affect the costs of the three methods.

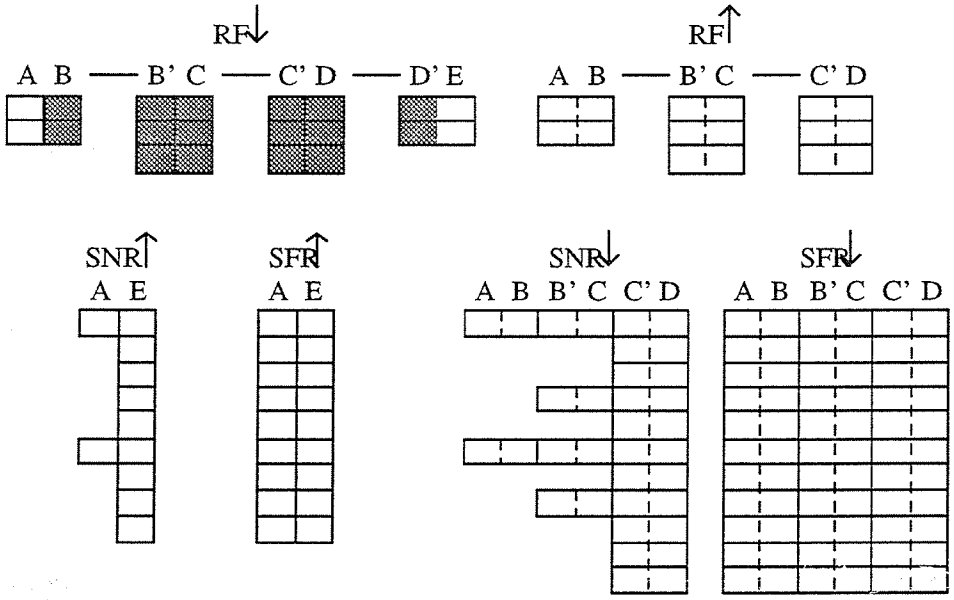
Figure 4.19a shows an example of high selectivities among three relation fragments. Let us assume the two selectivities are the same and equal to $n > 1$. Then, the average cardinality of a corresponding[4] single flat relation is $2 \times n \times n$ where 2 is the cardinality of the first relation fragment. The corresponding single nested relation contains 2 nested tuples, within which there exist $2 \times n \times n$ subtuples of DE. On the other hand, the selectivities are equal to 1 in Figure 4.19b. In that case, both the single flat relation

[4]I.e., materialized for the same query on the same database



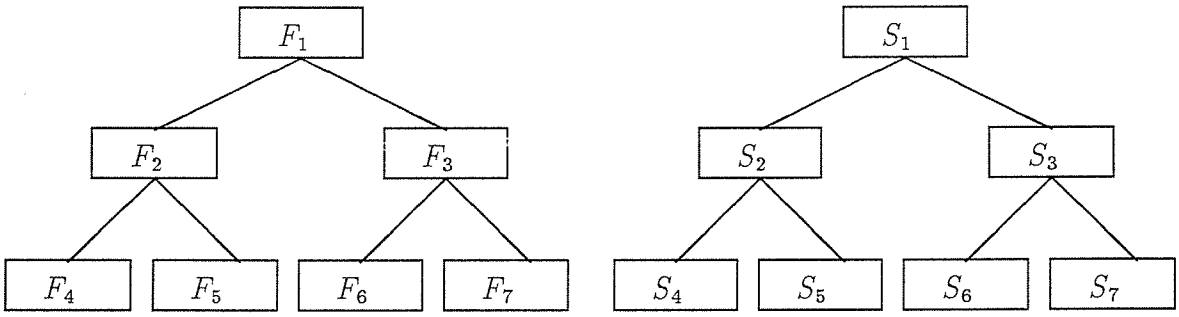
(a) High selectivities (b) Low selectivities

Figure 4.19: Examples of high vs. low values of selectivity



(a) High EJA ratios (b) Low EJA ratios

Figure 4.20: Examples of high vs. low values of EJA ratios



(a) Join tree (b) Nesting format tree

Figure 4.21: A sample query for random values of data parameters

and single nested relation contain only two ($= 2 \times 1 \times 1$) tuples. From these examples, we observe that, given a set of relation fragments, higher selectivities increase the cardinality of the single flat relation and the cardinalities of nested subtuples of a single nested relation, thus increasing the amount of data (in bytes). Figure 4.20a shows an example of relation fragments with high EJA ratios. In the example, all attributes except A and E are extra join attributes. In that case, the corresponding single flat relation and single nested relation contain only the two attributes A and E. On the other hand, the relation fragments shown in Figure 4.20b have no extra join attributes. In this case, all six attributes appear in the corresponding single flat relation and single nested relation. From these examples, we observe that, given a set of relation fragments, higher EJA ratios decrease the tuple sizes of the single flat relation and single nested relation, thus decreasing the amount of data (in bytes). Certainly the costs depend upon the amount of data to be handled to retrieve the same single nested relation. Therefore, higher selectivities and lower EJA ratios are more advantageous to the RF method than the SFR or SFR methods in terms of cost.

4.5.2 Overall Comparison using Simulation

We computed the average costs of the SFR, RF, and SNR methods, and tallied the winning counts – the number of times each method incurred the minimum cost

among the three methods. We used a query whose join tree is a complete binary tree of 7 relations as shown in Figure 4.21. The domains of the random values of input data parameters are as follows. (Let Ψ denote $\{< 1,2 >, < 1,3 >, < 2,4 >, < 2,5 >, < 3,6 >, < 3,7 >\}$.)

- $1.00 \leq \alpha_{ij} \leq 10.00$ for $< i,j > \in \Psi$.
- $0.00 < \rho_{f_i} \leq 1.00$ for $i = 1,2,\cdots,7$.
- $10 \leq N_{f_1} \leq 500, 10 \leq N_{f_j} \leq N_{f_i}\alpha_{ij}$ for $j = 2,3,\cdots,7$. (See Equation 4.10.)
- $10 \leq T_{f_j} \leq 500$ for $j = 2,3,\cdots,7$.
- $0.50\alpha_{1j} \leq \beta_{1j} \leq 1.00\alpha_{1j}$ for $j = 2,3$. (See Equation 4.18.),
  $0.50\alpha_{ij} \leq \beta_{ij} \leq 1.50\alpha_{ij}$ for $< i,j > \in \Psi$ and $i \neq 1$.
- $0.30 \leq d_{f_j} \leq d_t \leq 1.00$ for $j = 1,2,\cdots,7$.

The values of the other data parameters are obtained from these values using the relationships between data parameters discussed in Section 4.4.1.2.2. As a simplification, we assumed no merging of relation fragments in the nesting step. The effect of ignoring the merging cost on the cost comparison result is negligible. Accordingly, we used $\gamma_1 = 1, \gamma_j = \alpha_{ij}$ for $< i,j > \in \Psi$ and $j \neq 1$, and $m_i = 1$ for $i = 1,2,\cdots,7$ (See Equation 4.38).

Table 4.5 shows the average values and the winning counts (in percentage) obtained from 5000 random test cases for the transmission cost and the partial local processing cost, respectively.

It was mentioned in Section 4.3.5 that the SNR transmission cost is always less than the SFR transmission cost, but the RF transmission cost has a trade-off with the SFR transmission cost. Our result showed that indeed the SNR transmission always costed less than the SFR transmission. Moreover, it was observed that the RF transmission costed less than the SFR transmission for *all* test cases, even though there is a theoretical trade-off with the SFR method. The average value of the SFR transmission cost was about 1500 times higher than that of the RF transmission cost and about 1100 times higher than that of the SNR transmission cost. The

Method	Transmission				Partial local processing	
	Average data size	Average cost		#wins	Average cost	#wins
		LAN	WAN			
SFR	3413 Mbytes	3.2 hours	2.4 days	0%	2.9 hours	0%
RF	2.4 Mbytes	8.1 secs	2.4 mins	67%	15.2 secs	100%
SNR	3.2 Mbytes	11.1 secs	3.2 mins	33%	17.5 secs	0%

(Transmission time is elapsed time and local processing time is CPU time.)

Table 4.5: Costs evaluated using Random Data Parameters

transmission costs for the LAN and WAN showed the same relative costs among the different methods except that the WAN incurred about 18 times higher cost than LAN.

Since we assumed in our cost model that the server speed and the client speed are the same, the SNR method always takes the same cost as the RF method and incurs the additional cost (Equation 4.44) of assembling a single nested relation on a client. Therefore, the RF local processing cost is always less than the SNR local processing cost. Furthermore, our result showed that the RF local processing incurred the less cost than the SFR local processing cost as well for all test cases.

For the SFR, RF, and SNR method, the partial local processing cost is 0.9, 1.9, 1.6 times the LAN transmission cost while it is 0.05, 0.1, 0.1 times the WAN transmission cost. If we consider the uncounted cost of query processing and reference resolution, the local processing cost will be the major cost in the LAN environment and hardly ignorable even in the WAN environment.

It is interesting to see that the SFR transmission cost was evaluated to be about 1400 and 1100 times higher than the RF transmission cost and the SNR transmission cost, respectively, while the SFR partial local processing cost was evaluated to be only 590 and 600 times higher than the RF and SNR local processing cost. This difference in the ratios is due to the use of the binary search tree to represent nested subrelations. As mentioned in Section 4.3.3.2, a binary search tree incurs $O(\log_2 N)$ time where N is the number of nodes in the tree. On the other hand, the transmission cost for transmitting those N tuples is linear with respect to N, i.e., $O(N)$. This observation

demonstrates that the benefits of the RF method and the SNR method become more manifest in terms of reducing the transmission cost than the local processing cost.

4.5.3 Dependency on Selectivity and Extra Join Attribute Ratio

4.5.3.1 Observation using Sample Case Test

We performed cost comparisons using sample values of data parameters and observed the dependency of the costs on the values of a single α_{ij} and the set of $\rho_{f_i}, i = 1, 2, \cdots, 5$. Figure 4.22 shows the join tree and the nesting format tree of a sample query. The sample values of the input data parameters are as follows.

- $N_{f_i} = 500, 800, 300, 1200, 300$ for $i = 1, 2, 3, 4, 5$, respectively.
- $\alpha_{12} = 3.0, \alpha_{13} = 1.0 \sim 10.0, \alpha_{34} = 4.0, \alpha_{35} = 1.0$
- $\beta_{12} = 2.7, \beta_{13} = 0.9\alpha_{13}, \beta_{34} = 3.8$
- $T_{f_i} = 200, 300, 250, 100, 400$ for $i = 1, 2, 3, 4, 5$, respectively.
- $\rho_{f_i} = \begin{cases} 0.05, 0.1, 0.15, 0.05, 0.05 \text{ or} \\ 0.8, 0.9, 0.7, 0.6, 0.9 \end{cases}$ for $i = 1, 2, 3, 4, 5$, respectively.

The other data parameters are computed from those input parameters using the relationships between data parameters discussed in Section 4.4.1.2.2. We evaluated the costs using those parameter values while varying the value of α_{13} from 1 through 10. The same evaluation has been repeated for the two different sets of ρ_{f_i}'s.

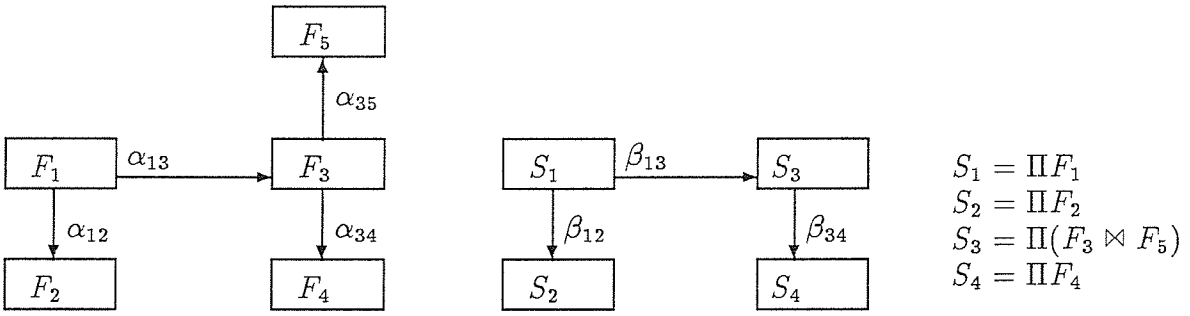
Table 4.6 shows the result of the cost evaluation, and Figure 4.23 shows the graphs of the costs of different methods with respect to the values of α_{13} for the two different sets of ρ_{f_i}'s.

α_{13}: Increasing the value of α_{13} without changing the value of $D_{f_{13}}$ is equivalent to increasing the value of N_{f_3}. In the RF method, the increase of N_{f_3} increases the size of F_3 only and has no effect on the sizes of the other relation fragments. On the other hand, its effect on increasing the number of duplicate subtuples in a single flat

Transmission cost (unit: seconds)												
	LAN						WAN					
	Low ρ_{f_i}			High ρ_{f_i}			Low ρ_{f_i}			High ρ_{f_i}		
α_{13}	SFR	RF	SNR	SFR	RF	SNR	SFR	RF	SNR	SFR	RF	SNR
1.0	18.0	2.2	3.0	3.5	2.2	0.6	317.9	39.4	53.4	62.4	39.4	10.9
2.0	36.0	2.5	4.5	7.1	2.5	1.0	635.8	43.9	79.1	124.7	43.9	18.1
3.0	54.0	2.7	5.9	10.6	2.7	1.4	953.7	48.4	104.9	187.0	48.4	25.3
4.0	72.1	3.0	7.4	14.1	3.0	1.8	1271.6	52.9	130.6	249.4	52.9	32.5
5.0	90.1	3.2	8.9	17.7	3.2	2.3	1589.5	57.4	156.3	311.7	57.4	39.7
6.0	108.1	3.5	10.3	21.2	3.5	2.7	1907.3	61.8	182.1	374.0	61.9	46.9
7.0	126.1	3.8	11.8	24.7	3.8	3.1	2225.2	66.4	207.8	436.4	66.4	54.1
8.0	144.1	4.0	13.2	28.2	4.0	3.5	2543.1	70.9	233.6	498.7	70.9	61.4
9.0	162.1	4.3	14.7	31.8	4.3	3.9	2861.0	75.4	259.3	561.0	75.4	68.6
10.0	180.1	4.5	16.2	35.3	4.5	4.3	3178.9	79.9	285.1	623.3	79.9	75.8

Partial local processing cost (unit: seconds)						
	Low ρ_{f_i}			High ρ_{f_i}		
α_{13}	SFR	RF	SNR	SFR	RF	SNR
1.0	13.3	2.6	3.0	2.9	1.6	1.9
2.0	26.7	3.5	4.0	5.9	2.0	2.5
3.0	40.3	4.4	5.2	9.2	2.6	3.1
4.0	54.3	5.6	6.6	12.8	3.3	4.1
5.0	68.1	6.7	7.9	16.2	4.0	4.9
6.0	82.0	7.8	9.2	19.8	4.6	5.7
7.0	96.0	8.9	10.5	23.4	5.3	6.5
8.0	110.4	10.2	12.2	27.4	6.2	7.7
9.0	124.5	11.4	13.6	31.2	6.9	8.6
10.0	138.6	12.5	15.0	34.9	7.6	9.5

Table 4.6: Costs evaluated using the sample values of data parameters

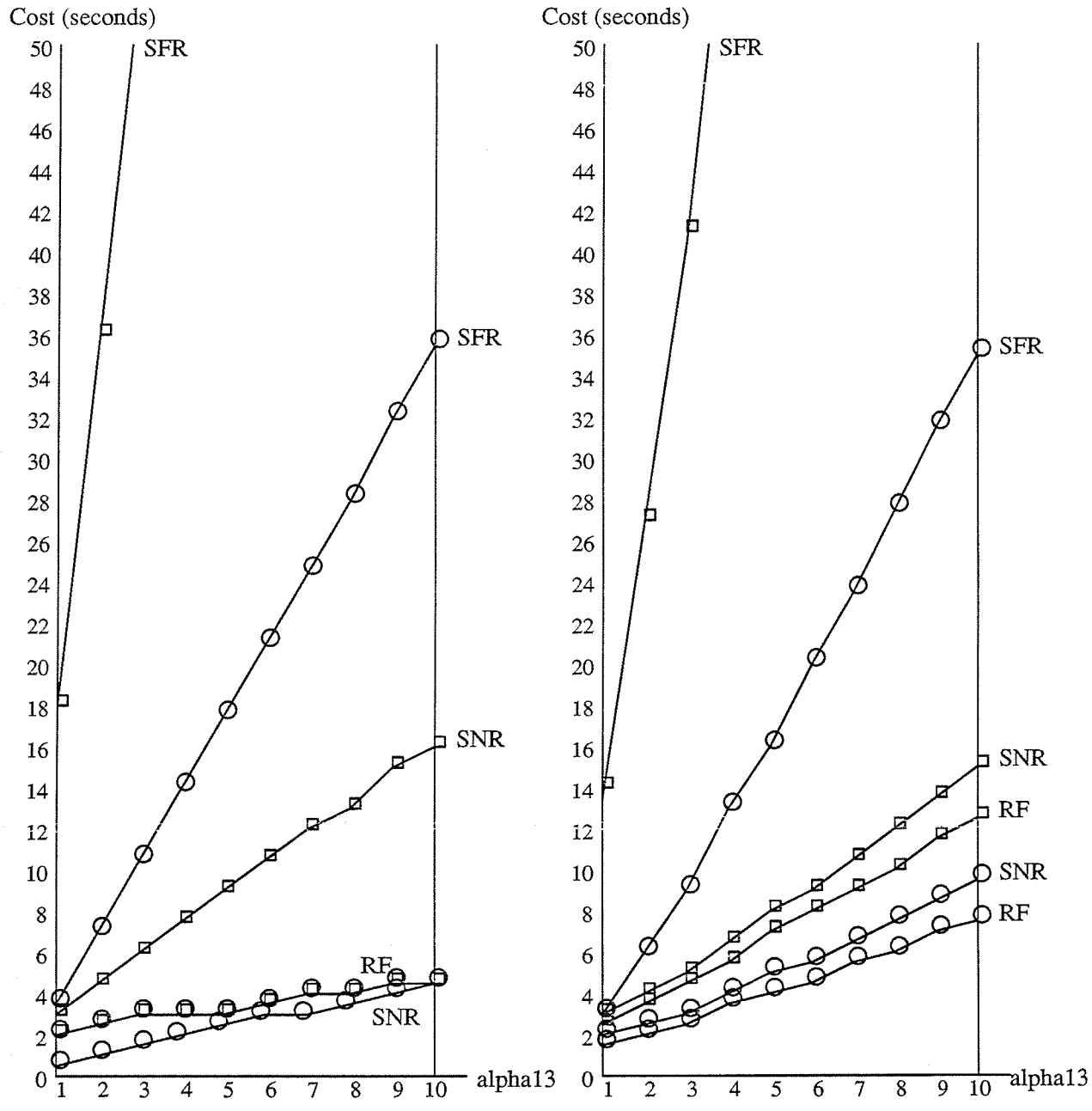


(a) Join tree (b) Nesting format tree (c) Assembly plan

Figure 4.22: A sample query for observing dependency on α_{13} and ρ_{f_3}

relation or multiply occurring subtuples in nested subrelations is more significant. We can verify this fact from Equation 4.11, Equation 4.12, and Equation 4.15. That is, the increase of N_{f_3} not only causes the increase of β_{13} by Equation 4.11, but also increases N_{s_4} according to Equation 4.12. Similarly, the increase of β_{f_3} is 'amplified' by a factor of $N_{s_1}\beta_{12}\beta_{34}(= 5130)$ if we compute N_t according to Equation 4.15. The cost evaluation result showed that both the transmission cost and the partial local processing cost increased linearly with respect to the value of α_{13}, and the slope was in the order of the SFR, SNR, and RF methods, from the highest first.

ρ_{f_3}: As for the values of ρ_{f_3}, higher values of ρ_{f_3}'s increases the overhead due to extra join attributes in the RF method while making the SFR method and the SNR method more efficient by reducing the tuple size of a single flat relation and nested subrelations, respectively, as we can see from Equation 4.14 and Equation 4.16. The cost evaluation result showed that costs were less for the higher ρ_{f_3}'s for both the transmission cost and the local processing cost. One exception is the RF transmission cost, in which case the transmission cost is independent of ρ_{f_3}'s, as we can verify from Equation 4.47. In particular, the SNR transmission incurred less cost than the RF transmission for the higher ρ_{f_3}'s.



(a) LAN Transmission cost

(b) Partial local processing cost

Cost for lower values of extra join attribute ratios.

Cost for higher values of extra join attribute ratios.

(The abscissa denotes the value of α_{13} and the ordinate denotes cost in seconds. Lines labeled with boxes or circles are those obtained for lower or higher values of ρ_{f_i}'s, respectively.)

Figure 4.23: Costs evaluated using the sample values of data parameters

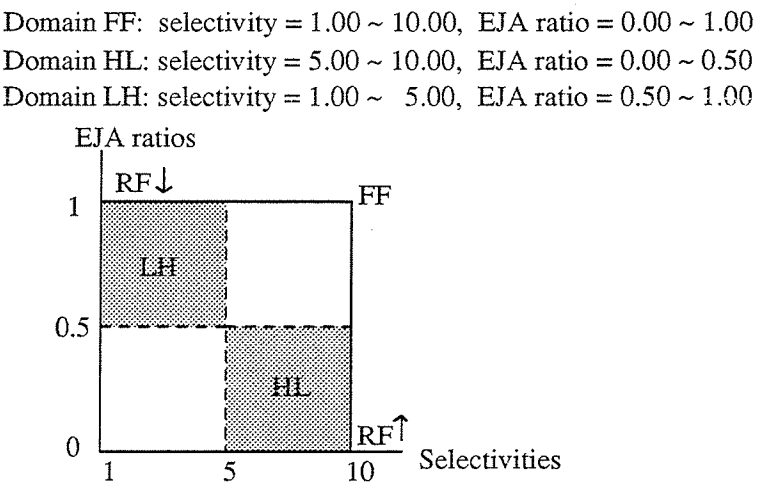


Figure 4.24: Domain HL and domain LH vs. Domain FF (full ranges)

4.5.3.2 Observation using Simulation

We carried out the cost evaluations using random values of data parameters with the same domains as those used in Section 4.5.2, but this time for different domains of α_{ij}'s and ρ_{f_i}'s. The following two different domains were used for generating random values of α_i's and ρ_{f_i}'s.

- Domain HL: (Higher values of α_{ij} and lower values of ρ_{f_i}.)
  $5.00 \leq \alpha_{ij} \leq 10.00$ for $< i, j > \in \Psi$ and $0.00 < \rho_{f_i} \leq 0.50$ for $i = 1, 2, \cdots, 7$.

- Domain LH: (Lower values of α_{ij} and higher values of ρ_{f_i}.)
  $1.00 \leq \alpha_{ij} \leq 5.00$ for $< i, j > \in \Psi$ and $0.50 \leq \rho_{f_i} \leq 1.00$ for $i = 1, 2, \cdots, 7$.

Figure 4.24 contrasts the Domain HL and Domain LH with the domain of the full range of values of α_{ij} and ρ_{f_i} that were used in Section 4.5.2.

Table 4.7 shows the result of cost evaluations. For Domain HL, the RF method showed more favorable result than the result shown in Tables 4.5 in terms of both the average cost and the winning counts than the SFR or SNR method. On the other hand, for domain LH, the RF method showed less favorable result. Thus, we confirmed that the observations made in Section 4.5.3.1 are generally true for other

Method	Transmission				Partial local processing	
	Average data size	Average cost		#wins	Average cost	#wins
		LAN	WAN			
SFR	33878 Mbytes	32.0 hours	23.5 days	0%	30.2 hours	0%
RF	4.1 Mbytes	14.0 secs	4.1 mins	93%	31.7 secs	100%
SNR	8.8 Mbytes	29.9 secs	8.8 mins	7%	36.6 secs	0%

(a) Domain HL $(5.00 \leq \alpha_{ij} \leq 10.00, 0.00 < \rho_{f_i} \leq 0.50)$

Method	Transmission				Partial local processing	
	Average data size	Average cost		#wins	Average cost	#wins
		LAN	WAN			
SFR	47.0 Mbytes	2.7 mins	46.9 mins	0%	2.0 mins	0.8%
RF	0.86 Mbytes	2.9 secs	51.8 secs	22%	4.0 secs	99.2%
SNR	0.53 Mbytes	1.8 secs	31.8 secs	78%	4.6 secs	0%

(b) Domain LH $(1.00 \leq \alpha_{ij} \leq 5.00, 0.50 \leq \rho_{f_i} \leq 1.00)$

(Transmission time is elapsed time and local processing time is CPU time.)

Table 4.7: Costs evaluated using random data parameter values with biased α_{ij}'s and ρ_{f_i}'s

values of data parametes as well. It is interesting to note that, for Domain LH, there were some cases in which the SFR method won over the RF method in the partial local processing cost.

4.6 Summary and Future Work

4.6.1 Summary

We have developed the mechanisms of three different methods for instantiating objects from a remote relational database server by materializing a view query and restructuring the query result into a nested relation and resolving references among them. The three different methods were the single flat relation (SFR) method, the relation fragment (RF) method, and the single nested relation (SNR) method, named after the data structure transmitted from a server to a client in each method.

Rigorous algorithms have been developed for each step of the object instantiation process, mainly focusing on the transmission and the nesting step of the translation, and a partial cost model has been developed. We have excluded the query processing cost and the reference resolution cost to simplify our work, because these two costs are the same in all three methods. We have performed cost comparisons using randomly generated data parameter values; and using sample data parameter values for varying values of a selectivity (α_{ij}) and for higher and lower values of extra join attribute ratios (ρ_{fi}).

The result of the cost comparison demonstrated that the RF method and the SNR method are more efficient than the common SFR method in terms of both the transmission cost and the local processing cost. Therefore, the RF and SNR methods are useful not only for remote database systems but also for local database systems. Besides, the RF and SNR methods are useful for disk-storage database systems as well as main memory database systems although the benefit of the RF and SNR methods is relatively less for the disk-storage database systems due to the significant cost of disk accesses.

The RF method wins over the SNR method more frequently. Therefore, the RF

method is the most preferred method if we have to choose one of the three methods. There remains an optimization issue of choosing between the RF method and the SNR method depending on the query and the speed of a server and a client. (If the server runs slower than the client, it is more favorable to the RF method than the SNR method because the SNR method performs the nesting step on a server. On the other hand, if the server runs faster than the client, it is more favorable to the SNR method. Note that we assumed that the server speed and the client speed are the same for the cost comparison.)

We have not considered the possibility of main memory overflow in case the amount of data retrieved as the result of a query exceeds the amount of available main memory space. Concern about main memory overflow does not discourage the use of the RF or SNR method because it is evident that the SFR method will suffer more severely from the shortage of main memory space than the RF or SNR method because the SFR method carries more redundant data.

4.6.2 Future Work

We discuss further work in two directions. First, the improvement of the efficiency of the RF method and the SNR method, and secondly, handling left outer joins in each of the three methods. Remember that we have dealt with only inner joins for a query in this portion of our work.

4.6.2.1 Improving the Efficiency of the RF and SNR Methods

As mentioned in Section 4.3, we placed more effort in making the SFR method efficient than the RF or SNR method because our objective was to demonstrate that the RF method and the SNR method are more efficient than the SFR method. The RF method and SNR method were designed to be rather simple than utmost efficient. We present here some ideas that are worth pursuing to improve the efficiency of the RF method and the SNR method.

In the current RF method, a client carries out the index creation and navigational joins on relation fragments and hence a server must send extra join attributes to

make those operations possible. As discussed in Section 4.3.5 and demonstrated in Section 4.5.3.1, extra join attributes are the source of redundant data in the RF method. One idea for avoiding the transmission of the extra join attributes is to have a server send the necessary linkage information in the form of physical pointers to the linked tuples of the other relations. In order to map between the heterogeneous address spaces of a server and a client, offset addresses can be used as long as it can be ensured that each relation fragment is allocated in a contiguous memory space. Sending physical pointers will reduce the transmission cost by not sending extra join attributes and the redundant tuples introduced by the extra join attributes. Moreover, all a client has to do is to follow the pointers to build a single nested relation out of the relation fragments. Thus, it reduces the load on a client. However, a server has to pay the price of index creation and navigational join to produce the physical pointers. A direct consequence of this requirement is that the duplicate elimination step cannot be pipelined with the transmission of tuples and more load is placed on a server. Besides, the transmission protocol becomes more complicated because, unlike the case of sending extra join attributes, the number of physical pointers attached to each tuple varies depending on the number of matching tuples.

As for the SNR method, the current SNR method has the overhead of dealing with multiply occurring subtuples in nested subrelations. One idea of eliminating these multiply occurring subtuples is to achieve more compaction of the transmitted data by using backward pointers embedded in the formatted stream of nested tuples. These backward pointers replace the actual tuples with pointers to the previously sent identical tuples. It will make the transmission protocol and the assembly process (Algorithm 4.3.16) more complicated, but will reduce the transmission cost.

The two ideas described so far have their major benefit in reducing the transmission cost. Therefore, these ideas are more useful in the wide area network (WAN) environment where the transmission cost is the dominant cost.

Finally, we have used the RF materialization as an intermediate step of the SNR materialization because, as mentioned in Section 4.3.1, a direct materialization disables the join reordering by a query optimizer. It will be worthwhile to compare the cost reduction achievable by using the direct materialization of a single nested

relation and the cost reduction achievable by utilizing the join reordering available from a query optimizer.

4.6.2.2 Handling Left Outer Joins

We have simplified our work by not considering left outer joins in a query although left outer joins are required frequently to prevent information loss. Thus, it will make our work more complete if we discuss the handling of left outer joins in each step of the different object instantiation methods, before ending this chapter. Since we designed the SNR method using the same query materialization and nesting processes as the RF method, we discuss only the SFR method and the RF method.

The consideration of left outer joins requires the handling of non-matching tuples in the join evaluations of the query materialization step, and the processing of null tuples in subsequent steps. We state briefly the key points of handling left outer joins at each step.

In the SFR method, the query processing algorithm described in Algorithm 4.3.1 should be modified so that if a join is a left outer join and there exists no matching tuple in the destination relation, null tuples are inserted in place of the tuples of the destination relation and its child relations in the join tree.

For each $t_i \in R_i$
  if t_i satisfies Φ_i then continue
  else
    Set t_i and all $t_i \in R_j$'s to null where $R_j, j \neq i$, are the relations in the subtree of the join tree rooted by R_i; Continue.

where 'continue' means to continue the nested loop join on the rest of the relations that have not yet been processed. On the other hand, in the RF method, the query processing algorithm described in Algorithm 4.3.2 should be modified so that if a join is a left outer join and there exists no matching tuple in the destination relation, the joins in a subtree of the join tree rooted by the destination relation are skipped. Consequently, no null subtuple is inserted to any relation fragment.

For each $t_i \in R_i$

if t_i satisfies Φ_i then continue

else

Skip all R_j's where $R_j, j \neq i$, are the relations in the subtree of the join tree rooted by R_i; Continue.

Duplicate elimination process is the same as Algorithm 4.3.3.

A single nested relation which is produced by the nesting step does not contain null subtuples at all. Therefore, the SFR nesting step as described in Algorithm 4.3.5 should be modified so that any decomposed subtuple all of whose column values are nulls is discarded. To achieve this modification, we should place

'If $t_i = \Lambda$ (a null tuple) then return.'

in front of

'$w_r :=$ the root pointed by $w_i.u_i$'

which is the first line of Algorithm 4.3.6.

The RF nesting needs some modifications as well. First of all, the join purge of Section 4.3.3.5.1 is not applicable to a left outer join. Theorem 4.3.1 does not hold for dangling tuples in the source relation of a left outer join. For example, given a left outer join $F_1 \underset{p_1 \wedge p_2 \cdots \wedge p_k}{[\bowtie} F_2$ from a relation fragment F_1 to another relation fragment F_2 with conjunctive join predicates $p_1 \wedge p_2 \wedge \cdots \wedge p_k$, it is possible that some of the dangling tuples in F_1 appear to have matching tuples in F_2 if *only one* of $p_1, p_2 \cdots, p_k$ is evaluated, while in fact there exists no matching tuple for a conjunction of all join predicates, $p_1 \wedge p_2 \wedge \cdots \wedge p_k$. The assembly planning step (Section 4.3.3.5.2) and the index creation (Section 4.3.3.5.3) step need no modification because they have nothing to do with join evaluations. On the other hand, the navigational join step (Section 4.3.3.5.4) performs join evaluations and thus should be modified to distinguish between inner joins and left outer joins. Algorithm 4.3.14 (Match) always returns one or more matching tuples if the evaluated join is an inner join but may return no matching tuple if the evaluated join is a left outer join. Accordingly, Algorithm 4.3.13 should be modified so that if Match(t_{i-1}, F_i, Φ_i) returns no matching tuple and the evaluated join is a left outer join then skip the rest of the 'for each' statements and set $t_i, t_{i+1}, \cdots, t_k$ to nulls before executing Step 3a through Step 3d.

If we had considered the effect of nulls generated by left outer joins in our work, the result of cost comparison would have appeared to be even more favorable to the RF method and the SNR method. The reason is that, in a single flat relation, nulls are duplicated in the same way the other tuples (which are not nulls) are duplicated. Note that there is no duplicate tuple in a relation fragment or a single nested relation.

Chapter 5

Conclusion

In this thesis, we addressed two problems – outer join and instantiation efficiency – in the view-object framework, i.e., in the framework of instantiating objects from relational databases through views. First, we introduced the view-object framework starting from a general framework of integrating objects and databases. Then, given the framework, we made three major contributions as summarized below.

- We defined a rigorous system model in order to embody the concept of interfacing between objects and relations. The system model consists of three parts: an object type model, a data model, and a view model. An object type defines the nested structure of objects. The non-null option is used to specify object attributes that are prohibited from being nulls. Data model uses the relational model and includes integrity constraints as part of the model. A view consists of a relational select-project-join query and an attribute mapping function for mapping between object attributes and relation attributes. It was beyond our scope to formulate a query or derive an attribute mapping function for a given view, so that we assumed that a query and an attribute mapping function were predefined in each pertinent object type. The system model thus developed provided the basis for developing a simple solution to the outer join problem and a part of the system model was used for the instantiation efficiency problem as well.

- We developed a mechanism for having the system decide which join should be an inner join and which join should be a left outer join, given a view-query, and generate non-null filters on the relations specified in the view-query. Users are required only to specify non-null constraints on object attributes whose values should not be null. All joins in a view-query are initialized as left outer joins. Those non-null constraints on object attributes are mapped to non-null constraints on relation attributes of the query result. The non-null constraints on relation attributes are then used to prescribe non-null filters on the attribute of base relations and replace left outer joins sitting on the join path from a pivot relation to the non-null constrained relations by inner joins. The remaining left outer joins are further reduced into inner joins if certain integrity constraints are satisfied. Besides, unnecessary non-null filters are eliminated.

- We developed two new methods of instantiating objects from remote relational databases, which are far more efficient than the conventional method of retrieving a single flat relation (SFR). One of the two new methods retrieves a query result as a set of relation fragments (RF's). The other method retrieves a query result as a single nested relation (SNR). We called the two new methods as the RF method and the SNR method while we called the conventional method as the SFR method. The algorithms of the three object instantiation methods (SFR, RF, and SNR) were described rigorously. Then, we derived cost formulas based on the algorithms and compared the estimated costs of the three methods. Two techniques were used for cost comparison: sample case test and simulation. The cost comparison result showed that the RF method and the SNR method are far more efficient than the SFR method for both the transmission cost and the local processing cost.

Appendix A

Measurement of Cost Parameters

The values of cost parameters were measured using programs that are sufficiently realistic to be part of an actual implementation. As mentioned in Section 4.4.1.1, we use CPU time for main memory cost and an elapsed time for network communication cost.

A.1 Main Memory Cost parameters

We used Unix `clock` system call for measuring the CPU time of the elementary main memory operations shown in Table 4.1. The time resolution of the `clock` is 1/60 seconds while main memory operations take as little as a few microseconds. The poor resolution of `clock` made it impossible to measure the precise values of main memory cost parameters. Moreover, the execution time varies every time the same code is run, depending on the system load. Thus, we obtained the values shown in Tables 4.1 by repeating the same code one million times and computing an average value.

The cost parameter value varies depending on how many subprocedures are called during execution. We can actually define as many subprocedures as we want. According to our experiment on Sun-3, the invocation of a subprocedure which requires four arguments took about 5 to 6 μsecs, which is a large amount of time for a main memory operation. Thus, we excluded the effect of subprocedure invocation from

our measurement by writing a dummy subprocedure requiring the same set of input parameters as a counterpart for each subprocedure and subtracting the time required to invoke dummy subprocedures from the total time. This approach means that our cost parameter values are the minimum values considering only the 'plain' code execution time. One exception is that we did not subtract a subprocedure invocation time if we judged that the code must use a subprocedure, intrinsically independently of who writes the code.

Now we comment on some details of how each cost parameter was obtained.

- C_{bs}: We used an implementation of Algorithm 4.3.7 for tuple sizes of 100 to 500 bytes. Tuples were initialized with pseudo-randomly generated base-64 ASCII strings. The values of C_{bs} using those random tuples were measured to be independent of the tuple size.

- C_{cm}: We measured the time for comparing two tuples of size 100 to 500 bytes where each tuple was initialized with pseudo-random base-64 ASCII string, and obtained the same value independently of the tuple size.

- C_{ci}, C_{cb}: We measured the time for copying a tuple of size 100 to 1000 bytes. The measured time was linear with respect to the tuple size.

- C_e: The time for evaluating equijoins on attributes of type integer was measured using a code written for more general joins including non-equijoins on non-integer attributes. We used the type integer because it frequently happens that joins are performed on key attributes and the key attributes are integers. We used the address of the join attributes, and their sizes and types as input parameters and did not count the time for obtaining those values themselves.

- C_{fl}: Folding was done by dividing a tuple into integer segments and adding up the values of the segments. The tuple size used was 100 to 500 bytes. The measured time was proportional to the tuple size.

- C_{hc}: We measured the time for hashing computation on a psedo-randomly generated integer hashing key using two different hashing methods: the division

method and the multiplicative hashing method [92]. The value shown in Table 4.1 is for the multiplicative hashing method.

- C_{ma}: Our experiment showed that Unix memory allocator (malloc) takes about 130 μsec on Sun-3 without regard to the allocated memory size while the other main memory operations takes only a couple of tens of miscroseconds. Therefore, if we used malloc for our work, the memory management cost would become dominant. However, it is a common practice to pre-allocate a working space [9, 94] to facilitate faster memory allocation and garbage collection. Then, memory allocation takes only the cost of moving a stack pointer within the pre-allocated working space as long as the working space need not be expanded. We assumed the usage of a working space mechanism.

- C_{mp}: The time for reading or writing a pointer value is so small that it hardly affects the cost computation result. Nevertheless we use it for completeness.

- C_{pi}, C_{pb}: We measured time for projecting a tuple of size 500 bytes on a varying number of 32 byte columns. The measured time was proportional to the total size of projected subtuple.

- C_{si}, C_{sn}: We measured the costs of reading a join column of size 8 bytes while scanning a relation, and computing an integer hashing key from the read column value. The size of a column (8 bytes) are reasonable because it is likely that join attributes are of type (short or long) integer. We assumed tuples are allocated contiguously within main memory. The measured time was linear with respect to the number of scanned tuples.

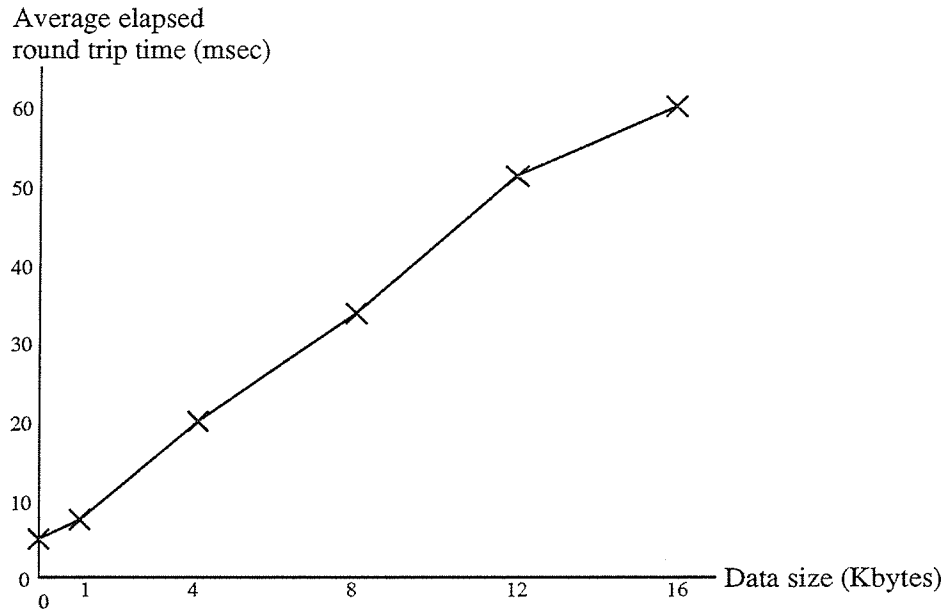
A.2 Network Communication Cost Parameters

The values of network communication cost parameters (C_l, C_b) depend on the communication media. It is well known that the data rate is 10 Mbps for Ethernet [63] used in the LAN environment. Cheriton and Williamson [67] measured the communication latency (C_l) and the per-byte communication cost (C_b) on an idle 10 Mbps Ethernet

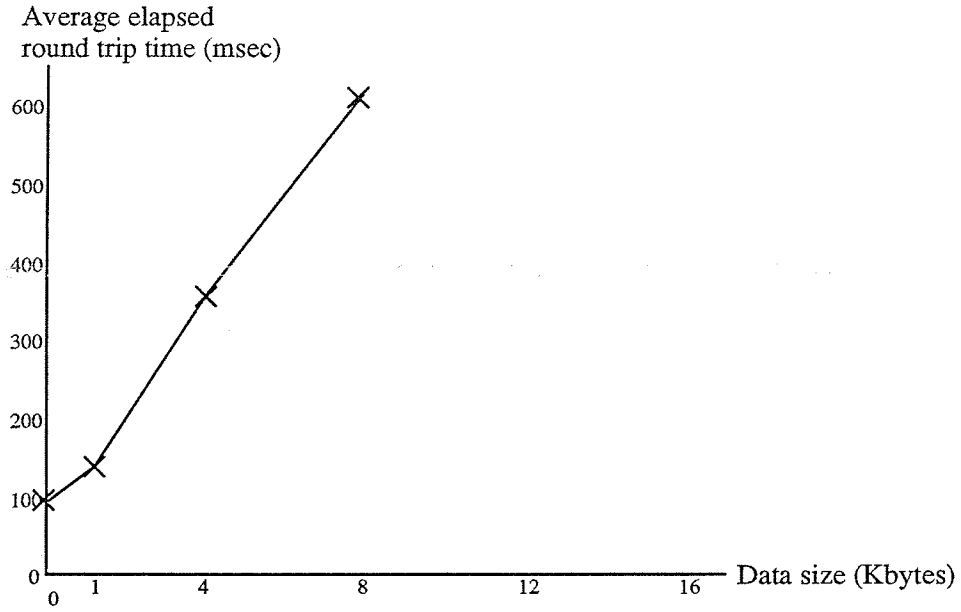
connecting two SUN-3/75's, and obtained $C_l = 2.23$ msec and $C_b = 1.8\mu$sec. As for the WAN, in [64] it is stated that the data rate is about 56 Kbps for the highest speed leased phone line in normal use while 1.544 Mbps T1 NSFnet [65, 66] lines are used in a few places where the high cost (in terms of financial investment) is acceptable. However, the current status of technological development has come to the point that T1 lines are in practical usage for the NSFnet and the availability of T3 lines (45 Mbps) is promised in near future.

We measured the elapsed time for transmitting data from a SUN-3/60 on the Stanford University Ethernet LAN to another SUN-3/60 on the same LAN, and also to a SUN-4 on the University of Illinois via the T1 NSFnet WAN. The client part of the code repeated the transmission of different amounts of data (0, 1K, 4K, 8K, 12K, 16K bytes of data plus 10 bytes of header) 60000 times and averaged the measured round trip times. The server part of the code was written to send an acknowledgement so that the client part can measure the round trip time. The measured time is from main memory to main memory. It does not include any disk access cost but does include main memory execution time for iterations, buffer pointer movements, and sending an acknowledgement.

Figure A.1 shows the average elapsed round trip times measured for vaying data sizes on the LAN and the WAN, respectively. The measured round trip times were almost linear with respect to the amount of transmitted data. We computed the (approximate) values of the two communication cost parameters, C_l and C_b, by equating the measured round trip times to $2 * C_l + C_b \times \text{Size}$ for different values of Size = 10, 1034, 4106, 8202, 12298, and 16394 bytes. (We did not use 12298 and 16394 bytes for the WAN.)



(a) Local area network (on the Stanford Ethernet)



(b) Wide area network (between the Stanford and the Illinois)

(The size of transmistted data is 10 bytes (header) larger than the data size.)

Figure A.1: Average round trip time vs. data size on the LAN and WAN

Bibliography

[1] F. Bancilhon, et al., "The Design and Implementation of O_2, an Object-Oriented Database System," in 'Advances in Object-Oriented Database Systems', Springer-Verlag, September 1988.

[2] O. Deux, et al., "The Story of O_2," IEEE Transactions on Knowledge and Data Engineering, vol. 2, no. 1, March 1990, pp. 91-108.

[3] R. Agrawal, and N. Gehani, "ODE (Object Database and Environment): The Language and the Data Model," Proceedings of the ACM SIGMOD International Conference on Management of Data, Portland, Oregon, May-June 1989.

[4] D. Maier, and J. Stein, "Development of an Object-Oriented DBMS," Proceedings of the OOPSLA International Conference on Object-Oriented Programming Systems, Languages, and Applications, September 1986, pp. 472-482.

[5] S. Ford, et al., "Zeitgeist: Database Support for Object-Oriented Programming," Proceedings of the International Workshop on Object-Oriented Database Systems, 1988, pp. 23-42.

[6] W. Kim, N. Chou, and J. Garza, "Integrating an Object-Oriented Programming System with a Database System," Proceedings of the OOPSLA International Conference on Object-Oriented Programming Systems, Languages, and Applications, September 1988, pp. 142-152.

[7] W. Kim, J. Garza, N. Ballou, and D. Woelk, "Architecture of the OIRON Next-Generation Database System," IEEE Transactions on Knowledge and Data Engineering, vol. 2, no. 1, March 1990, pp. 109-124.

[8] D. Fishman, et al., "Iris: An Object-Oriented Database Management System," ACM Transactions on Office Information Systems, vol. 5, no. 1, January 1987, pp. 48-69.

[9] K. Wilkinson, P. Lyngbaek, and W. Hasan, "The Iris Architecture and Implementation," IEEE Transactions on Knowledge and Data Engineering, vol. 2, no. 1, March 1990, pp. 63-75.

[10] M. Stonebraker, and L. Rowe, "The Design of POSTGRES," Proceedings of the ACM SIGMOD International Conference on Management of Data, 1986, pp. 340-354.

[11] M. Stonebraker, L. Rowe, and M. Hirohama, "The Implementation of POSTGRES," IEEE Transactions on Knowledge and Data Engineering, vol. 2, no. 1, March 1990, pp. 125-142.

[12] T. Learmont, and R. Cattell, "An Object-Oriented Interface to a Relational Database," Proceedings of the International Workshop on Object-Oriented Database Systems, 1987.

[13] G. Wiederhold, "Database Design (2nd ed.)," McGraw-Hill Book Company, 1983.

[14] G. Wiederhold, "Views, Objects, and Databases," IEEE Computer, December 1986, pp. 37-44.

[15] T. Barsalou, and G. Wiederhold, "Knowledge-based Mapping of Relations into Objects," The International Journal of Artificial Intelligence in Engineering, Computational Mechanics Publ., UK, 1989.

[16] B. Cohen, "Views and Objects in OB1: A Prolog-based View-Object-Oriented Database," Technical report TR.PRRL-88-TR-005, SRI, March 1988.

[17] G. Wiederhold, T. Barsalou, and S. Chaudhuri, "Managing Objects in a Relational Framework," Technical report CS-89-1245, Stanford University, January 1989.

[18] A. Paepcke, "PCLOS: A Flexible Implementation of CLOS Persistence," Proceedings of the European Conference on Object-Oriented Programming, Oslo, Norway, August 1988.

[19] Personal communication with Andreas Paepcke, Hewlett-Packard Labs., Palo Alto, California, November 1990.

[20] K. Law, G. Wiederhold, T. Barsalou, N. Siambela, W. Sujansky, D. Zingmond, and H. Singh, "An Architecture for Managing Design Objects in a Sharable Relational Framework," International Journal of Systems Automation, Research and Applications, International Society for Productivity Enhancement.

[21] K. Law, G. Wiederhold, T. Barsalou, N. Siambela, W. Sujansky, and D. Zingmond, "Managing Design Objects in a Sharable Relational Framework," Proceedings of the ASME International Conference on Computers in Engineering, Boston, MA, 1990.

[22] K. Law, T. Barsalou, and G. Wiederhold, "Management of Complex Structural Engineering Objects in a Relational Framework," Engineering with Computers, 6:81-92, 1990.

[23] W. Rubenstein, M. Kubicar, and R. Cattell, "Benchmarking Simple Database Operation," Proceedings of the ACM SIGMOD International Conference on Management of Data, May 1987.

[24] E. Codd, "A Relational Model of Data for Large Shared Data Banks," Communications of the ACM, vol. 13, no. 6, June 1970.

[25] D. Tsichritzis, and A. Klug (eds.), "The ANSI/X3/SPARC DBMS Framework: Report of the Study Group on Data Base Management Systems," Information Systems 3, 1978.

[26] D. Jardine (ed.), "The ANSI/SPARC DBMS Model," Proceedings of the Second SHARE Working Conference on Data Base Management Systems, Montreal, Canada, April 26-30, 1976.

[27] R. Haskin, and R. Lorie, "On Extending the Functions of a Relational Database System," Proceedings of the ACM SIGMOD International Conference on Management of Data, June 1982, pp. 207-212.

[28] R. Lorie, and W. Plouffe, "Complex Objects and Their Use in Design Transactions," Proceedings of the IEEE Annual Meeting-Database Week: Engineering Design Applications, May 1983, pp. 115-121.

[29] K. Dittrich, and R. Lorie, "Object-oriented Database Concepts for Engineering Applications," Technical report RJ 4691 (50029), IBM Research Laboratory, San Jose, CA 95193, May 1985.

[30] J. Ullman, "Principles of Database and Knowledge-Base Systems," Computer Science Press, 1988.

[31] J. Ullman, "Database Theory-Past and Future," Proceedings of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, San Diego, March 1987.

[32] K. Morris, J. Ullman, and A. Van Gelder, "Design Overview of the NAIL! System," Proceedings of the International Logic Programming Conference, 1986.

[33] S. Tsur, and C. Zaniolo, "LDL: A Logic-based Data-Language," Proceedings of the 12th International Conference on Very Large Data Bases, Kyoto, August 1986.

[34] D. Chimenti, A. O'Hare, R. Krishnamurthy, and C. Zaniolo, "An Overview of the LDL System," IEEE Data Engineering, vol. 10, no. 4, December 1987.

[35] C. Date, "The Outer Join," Proceedings of the 2nd International Conference on Databases, Cambridge, Britain, September 1983.

[36] E. Codd, "Extending the Relational Database Model to Capture More Meaning," ACM Transactions on Database Systems, vol. 4, no. 4, December 1979.

[37] C. Date, "Referential Integrity," Proceedings of the 7th International Conference on Very Large Data Bases, Cannes, France, September 1981, pp. 2-12.

[38] C. Date, "An Introduction to Database Systems," vol. 1, Fourth edition, Addison-Wesley Publishing Company, Inc., 1986.

[39] F. Bancilhon, "Object-Oriented Database Systems," Invited lecture, Proceedings of the 7th ACM SIGART-SIGMOD-SIGACT Symposium on Principles of Database Systems., Austin, Texas, March 1988.

[40] D. Maier, "Why Isn't There an Object-Oriented Data Model?," Proceedings of the IFIP 11th World Computer Congress, San Francisco, California, September 1989.

[41] J. Joseph, S. Thatte, C. Thompson, and D. Wells, "Report on the Object-Oriented Database Workshop," SIGMOD Record, vol. 18, no. 3, September 1989.

[42] W. Wilkes, P. Klahold, and G. Schlageter, "Complex and Composite Objects in CAD/CAM Databases," Proceedings of the 5th IEEE International Conference on Data Engineering, Los Angeles, February 1989.

[43] P. Buneman, S. Davidson, and A. Watters, "A Semantics for Complex Objects and Approximate Queries," Proceedings of the ACM Symposium on Principles of Database Systems, 1988.

[44] W. Kim, J. Banerjee, and H. Chou, "Composite Object Support in an Object-Oriented Database System," Proceedings of the OOPSLA International Conference on Object-Oriented Programming Systems, Languages, and Applications, October 1987, pp. 118-125.

[45] "IntelliCorp KEE^{TM} Software Development System User's Manual," Document no. 3.0-U-1, Intellicorp, July 1986.

[46] R. Kempf, and M. Stelzner, "Teaching Object-Oriented Programming with the KEE System," Proceedings of the OOPSLA International Conference on Object-Oriented Programming Systems, Languages, and Applications, October 1987, pp. 11 - 25.

[47] S. Khoshafian, and G. Copeland, "Object Identity," Proceedings of the OOPSLA International Conference on Object-Oriented Programming Systems, Languages, and Applications, 1986.

[48] S. Abiteboul, and P. Kanellakis, "Object Identity as a Query Language Primitive," Proceedings of the ACM SIGMOD International Conference on Management of Data, Portland, Oregon, May-June 1989.

[49] G. Wiederhold, "Binding in Information Processing," Technical report no. STAN-CS-81-851, Department of Computer Science, Stanford University, Stanford, CA 94305, May 1981.

[50] T. Pratt, "Programming Languages: Design and Implementation (2nd ed.)," Prentice-Hall, Inc., 1984.

[51] T. O'Hare, and A. Sheth, "The Interpreted–Compiled Range of AI/DB Systems," Technical Memo (unpublished), Paoli Research Center, Unisys Corp., July 1988.

[52] S. Ceri, G. Gottlob, and G. Wiederhold, "Interfacing Relational Databases and Prolog Efficiently", IEEE Transactions on Software Engineering, February 1989.

[53] S. Finkelstein, "Common Expression Analysis in Database Applications," Proceedings of the of ACM SIGMOD International Conference on Management of Data, 1982.

[54] M. Jarke, "Common Subexpression Isolation in Multiple Query Optimization," in W. Kim, D. S. Reiner, and D. S. Batory (eds), 'Query Processing in Database Systems,' Springer, 1984, pp. 191-205.

[55] P. Larson, and H. Yang, "Computing Queries from Derived Relations," Proceedings of the of the 11th VLDB International Conference on Very Large Databases, August 1985.

[56] T. Sellis, "Multiple-Query Optimization," ACM Transactions on Database Systems, vol. 13, no. 1, March 1988, pp. 23-52.

[57] A. Sheth, D. Buer, S. Russel, and S. Dao, "Cache Management System: Preliminary Design and Evaluation Criteria," Technical Memo TM-8484/000/00, West Coast Res. Center, Unisys Corp., 2400 Colorado Avenue, Santa Monica, CA 90406, October 1988.

[58] M. Nelson, "Caching in the SPRITE Network File System," PhD Thesis, University of California at Berkeley, March 1988.

[59] H. Wedekind, and G. Zoerntlein, "Prefetching in Real time Database Applications," Proceedings of the of ACM SIGMOD International Conference on Management of Data, 1986.

[60] G. Dill, "Peripheral Semiconductor Storage – A Feasible Alternative To Disk and Tape?," Hardcopy, vol. 7, no. 1, January 1987.

[61] B. Lee, and G. Wiederhold, "Outer Joins and Filters for Instantiating Objects from Relational Databases through Views," Technical Report no. 30, Center for Integrated Facility Engineering (CIFE), Stanford University, May 1990.

[62] P. Dwyer, and J. Larson, "Some Experiences with a Distributed Database Testbed System," IEEE Proceedings, vol. 75, no. 5, May 1987, pp. 633-648. Also appear in Gupta, A. (ed.), 'Integration of Information Systems: Bridging Heterogeneous Databases,' IEEE Press, 1989.

[63] A. Tanenbaum, "Computer Networks," 1st edition, Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632, 1981.

[64] A. Tanenbaum, "Computer Networks," 2nd edition, Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632, 1981.

[65] D. Comer, "Internetworking with TCP/IP: Principles, Protocols and Architecture," Prentice-Hall, Inc. 1988.

[66] S. Reddy, "NSFnet Today: A New Implementation of a Vast Research Network" LAN Magazine, June 1989.

[67] D. Cheriton and C. Williamson, "Network Measurement of the VMTP Request-Response Protocol in the V Distributed System," Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, Banff, Alberta, Canada, May 1987, pp. 216-225.

[68] P. Fischer, and S. Thomas, "Operators for Non-First-Normal-Form Relations," Proceedings of the IEEE COMPSAC International Computer Software and Application Conference, November 1983.

[69] M. Roth, H. Korth, and A. Silberschatz, "Theory of Non-First-Normal-Form Relational Databases," Technical report no. TR-84-36, Department of Computer Science, University of Texas at Austin, Austin, Texas 78712, December 1984. < in TODS on "Nested relations">

[70] S. Abiteboul, and N. Bidoit, "Non-First Normal Form Relations to Represent Hierarchically Organized Data," Proceedings of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, April 1984.

[71] D. Bitton, "The Effect of Large Main Memory on Database Systems," Panel report, Proceedings of the ACM SIGMOD International Conference on Management of Data, 1986, pp. 337-339.

[72] A. Ammann, M. Hanrahan, and R. Krishnamurthy, "Design of a Memory Resident DBMS," Proceedings of the IEEE COMPCON Conference, San Francisco, February 1985

[73] K. Whang, et al., "Office-By-Example, An Integrated Office System and Database Manager," ACM Transactions on Office Information Systems, vol. 5, no. 4, October 1987, pp. 393-427.

[74] T. Lehman, and M. Carey, "A Study of Index Structures for Main Memory Database Management Systems," Proceedings of the of the 12th International Conference on Very Large Data Bases, Kyoto, August, 1986, pp. 294-303.

[75] T. Lehman, and M. Carey, "Query Processing in Main Memory Database Management Systems," Proceedings of the ACM SIGMOD International Conference on Management of Data, 1986, pp. 239-250.

[76] H. Garcia-Molina, R. Lipton, and J. Valdes, "A Massive Memory Machine," Proceedings of the IEEE COMPCON Conference, 1984.

[77] D. Bitton and C. Turbyfill, "Performance Evaluation of Main Memory Database Systems," Technical Report 86-731, Department of Computer Science, Cornell University, Ithaca, New York, January 1986.

[78] D. Bitton, M. Hanrahan, and C. Turbyfill, "Performance of Complex Queries in Main Memory Database Systems," Proceedings of the IEEE 3rd International Conference on Data Engineering, 1987, pp. 72-81.

[79] M. Eich, "MARS: The Design of a Main Memory Database Machine," Proceedings of the 5th International Workshop on Database Machine, Karuizawa, October 1987.

[80] A. Swami, "Optimization of Large Join Queries," Ph.D. Thesis, Report no. STAN-CS-89-1262, Department of Computer Science, Stanford University, 1989.

[81] E. Horowitz, and S. Sahni, "Fundamentals of Data Structures," Computer Science Press, Inc., 1976.

[82] W. Mauer, and T. Lewis, "Hash Table Methods," ACM Computing Surveys, vol. 7, no. 1, March 1975, pp. 5-20.

[83] K. Whang, and R. Krishnamurthy, "Query Optimization in a Memory-Resident Domain Relational Calculus Database System," ACM Transactions on Database Systems, vol. 15, no. 1, March 1990.

[84] D. DeWitt, R. Katz, F. Olken, L. Shapiro, M. Stonebraker, and D. Wood, "Implementation Techniques for Main Memory Database Systems," Proceedings of the ACM SIGMOD International Conference on Management of Data, June 1984, pp. 1-8.

[85] L. Shapiro, "Join Processing in Database Systems with Large Main Memories," ACM Transactions on Database Systems, vol. 11, no. 3, September 1986, pp. 239-264.

[86] D. Tsichritzis, and F. Lochovsky, "Data Models," Prentice-Hall, Inc. 1982, pp. 210-225.

[87] W. Kim, "Architectural Issues in Object-oriented Databases," The Journal of Object-oriented Programming, March 1990.

[88] A. Aho, J. Hopcroft and J. Ullman, "The Design and Analysis of Computer Algorithms," Addison-Wesley Publishing Company, 1974.

[89] D. Comer, "The Ubiquitous B-Tree," ACM Computing Surveys, vol. 11, no. 2, June 1979.

[90] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong, "Extendible Hashing: A fast access method for dynamic files," ACM Transactions on Database Systems, vol. 4, no. 3, September 1989, pp. 315-344.

[91] W. Litwin, "Linear Hashing: A New Toolf for File and Table Addressing," Proceedings of the 6th International Conference on Very Large Databases, Montreal, Canada, October 1980.

[92] D. Knuth, "The Art of Computer Programming, Vol 3: Sorting and Searching," Addison-Wesley, 1973.

[93] A. Aho, J. Hopcroft, and J. Ullman, "Data Structures and Algorithms," Addison-Wesley Publishing Company, 1983.

[94] Personal communications with Janet Miller et al., the Iris project group, Hewlett-Packard Company, Cupertino, CA, July 1990.