Outer Joins and Filters for Instantiating Objects from Relational Databases Through Views

Byung Suk Lee, Member, IEEE, and Gio Wiederhold, Fellow, IEEE

Abstract—One of the approaches for integrating object-oriented programs with databases is to instantiate objects from relational databases by evaluating view queries. In that approach, it is often necessary to evaluate some joins of the query by left outer joins to prevent information loss caused by the tuples discarded by inner joins. It is also necessary to filter some relations with selection conditions to prevent the retrieval of unwanted nulls.

The system should automatically prescribe joins as inner or left outer joins and generate the filters, rather than letting them be specified manually for every view definition. We develop such a mechanism in this paper. We first develop a rigorous system model to facilitate the mapping between an object-oriented model and the relational model. The system model provides a well-defined context for developing a simple mechanism.

The mechanism requires only one piece of information from users: null options on an object attribute. The semantics of these options are mapped to non-null constraints on the query result. Then the system prescribes joins and generates filters accordingly. We also address reducing the number of left outer joins and the filters so that the query can be processed more efficiently.

Index Terms—Complex object, filter, outer join, relation storage, view.

I. INTRODUCTION

ONE of the approaches for integrating object-oriented programs with relational databases is to instantiate objects from relational databases through views [7]–[9], [10]–[15]. 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. This approach provides an effective mechanism for building object-oriented applications on top of relational databases.

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 III-B, 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 [27]. Outer joins do not discard any tuple in the joined relations by inserting null tuples in place of where a matching tuple would have been inserted if there were one. In particular, we need unidirectional outer joins such as left outer joins. 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 that eliminate the tuples containing null attributes in order to prevent the retrieval of unwanted nulls.

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

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 these two operators can be avoided without affecting the query result for frequent cases we will define in this paper.

We made the following contributions in the context of instantiating objects from relational databases through views.

- Two key operators—a left outer join and a non-null filter—for preventing information loss and the retrieval of unwanted information.
- A simple mechanism for specifying those two operators in a relational view query, given a system model we define. The system model is easily implementable in existing systems.
- Optimization by reducing the number of the two operators without affecting query results.

II. BACKGROUND FRAMEWORK

A. Integration of Object-Oriented Programs and Databases

The desire for integrating object-oriented programs with databases has been increasing recently. This integration enables applications working in object-oriented environment to have shared, concurrent access to persistent storage. Examples are the engineering applications such as computer-aided design and computer-aided software engineering. These are not well supported by conventional databases such as relational 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 [1]-[3], [5], [6]; 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], [7]-[15], and objects are retrieved by evaluating queries to databases.

The relation storage approach incurs the overhead of mapping between different models [10], [16] 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 [10]. 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.

B. Two Perspectives of the Relation Storage Approach

We observed two different perspectives within the relation storage approach: object centered [4], [7]-[9] and relation centered [10]-[15]. 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. The two perspectives may look like the two sides of the same coin, but they differ operationally. In Fig. 1(a), the Project-manager type is mapped to the Project-manager relation. There exists a separate relation for each corresponding object type. In Fig. 1(b), there does not exist a separate Project-manager relation in the given 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 that the derived relation is analogous to the intensional database (IDB) relation [20], [21] used in the integration of the logic-based model and relational model [21]-[23]. For example, the IDB relation of the Project-manager is written as follows using the notion of Datalog [20].

\[
\text{Project-manager (} \text{ssn, \ldots)} \leftarrow \text{Employee (} \text{ssn, \ldots)} \\
& \text{Project (} \text{\ldots, manager-ssn, \ldots)} \land \text{ssn = manager-ssn.}
\]

We use the relation-centered perspective throughout the discussion in this paper, but the result is applicable to the object-centered perspective as well.

C. Instantiating Objects from Relations Through Views

Views provide a user-defined subset of a large database. Thus, as mentioned in Sections II-A and II-B, 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. This mapping information is used by a NEST [21]-[26] operator to restructure a query result into objects. In [17] and [18] appears a description of different methods of implementing the NEST operator. Our concern in this paper is only the mapping of attributes.

The mapping is done by linking object attributes to corresponding relation attributes. Objects have more complex structures than relations. For instance, objects support aggregation hierarchies [34] through an is-part-of relationship. 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 an additional component for the mapping, that is, an attribute mapping function.

Fig. 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 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

---

1 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."
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 oid identifier (oid). In Fig. 2, the key ssn of the Employee relation is retrieved to become the oid of the Employee object. Object id's are not explicitly defined in the type definition but are 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 a null tuple consisting of nulls for all attributes of R2. In other words, R1 w R2 produces the following set of tuples.

\[
\{< t_1, t_2 > | t_1 \in R_1 \land t_2 \in R_2 \land t_1.A \not= B \} \cup \{< t_1, \lambda > | t_1 \in R_1 \land \lambda \not= t_2(t_2 \in R_2 \land t_1.A \not= B) \}
\]

where \(\lambda\) denotes a comparison operator, i.e., \(\lambda \in \{<, \leq, >, \geq, =, \neq\}\).

For the rest of this paper, we use a small size join symbol (\(\bowtie\)) 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:** Non-null Filter: A non-null filter is a conjunction of predicates applicable to a base relation \(R\), defined as follows.

\[
R.A_1 \neq null \land R.A_2 \neq null \land \cdots \land R.A_i \neq null
\]

where \(A_1, A_2, \ldots, A_i\) are the attributes of \(R\) that are not allowed to have nulls.

**B. Motivation**

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 of its attributes are nulls as long as it has an oid. Let us consider the objects of type Employee shown in Fig. 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, it is certain that 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 a 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 paper.

**III. PROBLEM FORMULATION**

**A. The Two Operators**

In the introduction, 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...
Now let us assume the Employee objects prohibit nulls for the dept attribute since a department affiliation is required of every employee, while in the relational database a department affiliation is not required. As mentioned in Section II–C, the dept attribute is retrieved from the dept# of the Employee relation. The join between the Employee relation and the 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# ≠ null. We call this selection condition a non-null filter.

We see from the above examples that we frequently need left outer joins [27] to prevent the loss of wanted objects and non-null filters to prevent the retrieval of unwanted nulls.

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 Fig. 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 are 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.

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 ≠ 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 [30] 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 Fig. 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 [30] from Employee.ssn to Child.ssn. As another example, let us consider the following directed join graph.

\[
R_1 \rightarrow_1 R_2 \rightarrow_2 \ldots \rightarrow_2 R_3 \rightarrow_4 R_4
\]

where the join from R_2 to R_3 is a left outer join and the others are inner joins. If it is known 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 cannot consider reversing the order of 

\[
R_2 \bowtie R_3
\]

and thus cannot 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 a less optimal plan than when left outer joins are evaluated as such and, therefore, cannot be reordered.

C. 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 reasons of efficiency, the number of left outer joins and non-null filters should be reduced whenever possible.

D. Our Approach

The heterogeneity of the object-oriented model and the relational model causes several difficulties in mapping between the two models [31]. Hence we cannot expect a simple solution 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 IV–A. Users do not even have to know what a left outer join 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\(^6\) on object attributes, and they are 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 relations accordingly. Finally, the number of left outer joins and non-null filters are reduced using the integrity constraints of the data model.

In the rest of this paper we first develop a rigorous system model to facilitate the mapping between objects and relations in Section IV. The mechanism is developed in Section V, and the conclusion follows in Section VI.

IV. 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 [32], [33]. Therefore we define a model which is common to many existing object-oriented models [1], [4]–[7]. Note that we do not (yet) deal with methods but focus on object structures. The data model is the relational model proposed by Codd [19]. The view model contains a relational

\[^6\]These constraints require the existence of an object attribute given the oid of an object. We would call this constraint an existence constraint if this term were not already used in [20] to mean the same concept as the referential integrity.
A. Object Type Model

Many existing object-oriented models support aggregation through nested structures and references. For example, the Employee object of Fig. 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 birth Date. The children attribute defines an embedded substructure of the Employee object. Thus our object type has a similar structure to that of the complex object [35]-[40].

We use value-oriented oid’s [44], [45] and retrieve them from the keys of relations. Those relations providing oid’s are called pivot relations [11]-[13]. As discussed in Section II-B, 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. Fig. 3 illustrates these concepts. In Fig. 3(a), the Employee relation is the pivot relation for the Employee object and provides its key ssn as the oid. Figure 3(b) shows the derived relation Project-manager of Fig. 1, which becomes the pivot relation for the Project-manager object. It is defined by Employee type Programmer

Fig. 3. The concept of a pivot relation.

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 oid is mapped from the key of a pivot relation, a subobject also has an associated oid that 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.

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. Attributes without non-null options are allowed to have null values by default.

An example is shown in Fig. 4. The Project attribute defines its own attributes and becomes a subobject of the Programmer object. It has its oid 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 paper.

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 4.1: The O-tree of an object O is a tree that 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 (nonleaf) is labeled by a complex attribute of the object O.

An example of an O-tree is shown in Fig. 5 for the Programmer type.

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

Attribute ::= simple attribute | complex attribute
simple attribute ::= internal attribute | external attribute
complex attribute ::= [ attribute, attribute, ... ]
Fig. 5. The O-tree of the Programmer object type.

Definition 4.2: Given an object $O$, $\text{Oset}(O)$ is defined as a function returning the set of the root of the O-tree and all of its non-leaf descendents.

For example, $\text{Oset}(\text{Programmer})$ returns $\{\text{Programmer}, \text{Project}\}$. Note each element of an Oset has its oid mapped to the key of a pivot relation.

Definition 4.3: Given an object $O$ and an attribute $s_0$ of the object $O$, $\text{Ochain}(O, s_0)$ is defined as a function returning the chain of nodes from the root ($O$) of the O-tree to a node labeled $s_0$, i.e., $O, O_1, \ldots, O_n, s_0$.

For example, $\text{Ochain}(\text{Programmer}, \text{title})$ returns $\text{Programmer}.\text{Project}.\text{title}$ and $\text{Ochain}(\text{Programmer}, \text{Project})$ returns $\text{Programmer}.\text{Project}$.

B. Data Model

Integrity constraints [28]-[30] are a part of the data model. Two kinds of integrity constraints are used in our work: referential integrity constraints and entity integrity constraints. As mentioned in Section III-B, 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 4.4: 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:

$$\forall a \in R.A(a = \text{null} \lor \exists b \in S.B(a = b)) \quad (4)$$

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

Our definition of the entity integrity constraint is more extensive than the definition used in [30].

Definition 4.5: An entity integrity constraint requires one or more of the following conditions to be satisfied:

- Primary key constraint: $R.A \neq \text{null}$ if $A$ is the primary key of $R$.\(^{11}\)
- 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 $\theta_1, \theta_2$ 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 4.5 are sufficient for our work. Fig. 6 shows the schema, the referential integrity constraints and the entity integrity constraints of a sample database.

C. View Model

Fig. 7 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 ($P$) and a pivot mapping function (PMF). The PMF defines the mapping between the pivot relations and the (sub)object(s).

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

Query Part: Fig. 8 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 $\pi$ projected from $R$. For example, the Proj-Assgin relation is labeled with a filter $\text{task} = \text{programming}$ and a set of projected attributes $\text{task}$. Two occurrences of the same relation are distinguished by a tuple variable denoted as a subscript. For example, the two occurrence of the Emp relation in Fig. 8 are distinguished into node $\text{Emp}_1$ and node $\text{Emp}_2$. 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. For example, the directed edge from the Proj-Assgin node to the Proj node denotes a left outer join from the Proj-Assgin relation to the Project relation.

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.

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

$\text{Ochain}(O, s_0)$ was defined in Definition 4.3. 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.

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 4.1 (Attribute Mapping Function):

- $\text{Programmer}.\text{name} \leftrightarrow \text{Emp}.\text{name}$,
- $\text{Programmer}.\text{dept} \leftrightarrow \text{Emp}.\text{dept}$,
- $\text{Programmer}.\text{salary} \leftrightarrow \text{Emp}.\text{salary}$,
- $\text{Programmer}.\text{manager} \leftrightarrow \text{Division}.\text{manager}$,
- $\text{Programmer}.\text{task} \leftrightarrow \text{Proj-Assgin}.\text{task}$,
- $\text{Programmer}.\text{Project}.\text{title} \leftrightarrow \text{Proj-Title}.\text{title}$,
- $\text{Programmer}.\text{Project}.\text{sponsor} \leftrightarrow \text{Sponsor}.\text{name}$,
- $\text{Programmer}.\text{Project}.\text{leader} \leftrightarrow \text{Emp}.\text{name}$,
- $\text{Programmer}.\text{Project}.\text{department} \leftrightarrow \text{Project}.\text{dept}$

\(^{11}\)In [90], only this constraint is used as the entity integrity constraint.

\(^{12}\)Or equivalently, between the keys of the pivot relations and the id's of the (sub)object(s).
/* Underlined attribute 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, -k)
Project (proj#, dept, leader, sponsor)
Sponsor (name, phone#, address)
Proj-Title (proj#, title)

Fig. 6. A sample database. (a) Database schema. (b) Referential integrity constraints. (c) Entity 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 as well.

Fig. 7. Mapping between objects and relations.

As shown in Fig. 3, a pivot relation is either a base relation or a derived relation. If it is a base relation, its key is mapped to the oid. If it is a derived relation, the key of one of its base relations is mapped to the oid. Fig. 8 shows two pivots, Programmer1 and Project1. Here Project1 is the node of a base relation and Programmer1 is the node of a derived relation defined by \( \langle \text{Engineer1}, \{\text{Engineer1} \text{, } \text{Worker1} \text{, } \text{Task} \text{, } \text{Project1} \text{, } \text{Program1} \text{, } \text{Module1} \rangle \). A formal definition of a derived relation is as follows.

**Definition 4.6:** A derived relation of an object type \( O \) is an ordered pair \( (R_b, E) \) 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 expression such that, for all possible instances of the relations in \( E \):

1. \( \Pi_{\text{Key}(R_b)} E' \subseteq \Pi_{\text{Key}(R_b)} R_b \),
2. \( \exists E'' | E'' \subseteq R_b \land E'' \subseteq \Pi_{\text{Key}(R_b)} E' \) where \( E'' \subseteq R_b \) denotes that \( E'' \) is a proper subexpression of \( E \) and have \( R_b \) in common with \( E \).

That is, the result of evaluating \( E \) produces a subset of the keys available from \( R_b \) and there is no proper subexpression \( E' \) that, when evaluated, produces a subset of the keys produced from \( E \). The second property of the above definition is the minimality property. Note that the definition lacks the uniqueness property. Therefore there can be a superexpression \( E'' \) that produces the same set of keys. In this case, we always choose the minimal expression \( E \).

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 mapping between the object set defined in Definition 4.2 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 4.2 (Pivot Mapping Function):**

\[ \text{Programmer} \leftrightarrow \text{Programmer1, Project} \leftrightarrow \text{Project1} \]

As mentioned in Section IV-A, we associate value-oriented oid’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 4.1:**

For each pivot relation \( p \) in PS begin

If \( p \) is a base relation then append ‘Ochain(0, PMF(p)).id to AMF.

else /* \( p \) is a derived relation */ begin

Find the base relation \( R_b \) of \( p \).

Append ‘Ochain(0, PMF(p)).id to AMF.

end.

end.

For example, given the set of pivot relations and the pivot mapping function of the Programmer view, Algorithm 4.1 derives the following mappings between the id’s of the Programmer object and its Project subobject and their corresponding pivot relation keys. These are appended to the
AMF:

\{\text{Programmer}.id \leftrightarrow \text{Engineer}_1.ssn, \}

\text{Programmer}.Project.id \leftrightarrow \text{Project}_1.proj\# \}.

There is a constraint on the definition of the attribute mapping function. Let us consider two object attributes \( s_0 \) and \( s_1 \) that belong to the same level 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 must exist 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.

V. 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.

A. Overview

There are two sources of nulls retrieved from databases. One is from the nulls stored in the tuples, and the other is from the nulls inserted for nonmatching tuples of an outer join. 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, left outer join, as explained in Section III-B. Therefore our strategy is to initialize all joins of the 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 are 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 oid’s and the keys of pivot relations using Algorithm 4.1, and add the result to the attribute mapping function.

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

5) Replace all joins that appear in the definition of derived relations by inner joins. (See Section V-B.)

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 Sections V-C and V-D.)

7) Find the left outer joins that 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 V-E.)

B. Joins Within a Derived Relation

As mentioned in Section II-B, a derived relation is a conceptual relation defined from base relations via a select-join expression, and this provides an abstraction of base relations so that the semantics of the derived relation directly 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 5.1: Let us consider an object type \( O \) and a derived relation \( \langle R_1, E \rangle \) defined according to Definition 4.6. 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 \( i \in [1, n-1] \) while the others are inner joins, then the following is true.

\[
\Pi_{\text{Key}(R_1)}(R_1 \bowtie R_2 \bowtie \cdots \bowtie R_{i} \bowtie R_{i+1} \bowtie \cdots \bowtie R_n)
\]

That is, there exists a proper subexpression that, when evaluated, produces the same set of keys available from \( R_1 \). This violates the second condition required of \( E \) in Definition 4.6. Therefore, all the joins in \( E \) must be inner joins, Q.E.D.

For example, given a derived relation \( \langle \text{Engineer}_1, \{ \text{Engineer}_1 \bowtie \sigma \text{task} = \text{"programming"} \bowtie \text{Proj} \text{-Assign}_1 \rangle \) defined to provide the semantics of the Programmer object, the join between \( \text{Engineer}_1 \) and \( \text{Proj} \text{-Assign}_1 \) must be an inner join. If the join is evaluated as a left outer join, it retrieves all tuples of \( \text{Engineer}_1 \), 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 \( \text{PS} \) of pivot relations:

Algorithm 5.1: 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.

C. 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. It requires there should exist a non-null \( s_0 \) given the oid of the object. Let us denote this non-null constraint as \( O.id \Rightarrow s_0 \). 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 5.1 (Non-null Constraint):** Let us at this point consider $O_{chain}(O, s_0) \equiv O_0, O_1, \ldots, O_n, s_0$. If $s_0$ has a non-null option, then, given $O_n.id$.

- if $s_0$ is a simple attribute, i.e., $O_n.id \Rightarrow s_0$, then $s_0$ cannot be null;
- if $s_0$ is a complex attribute, i.e., $O_n.id \Rightarrow s_0.id$, then $s_0.id$ cannot be null.

For example, given the Programmer object of Fig. 4, the non-null options on name and dept attributes are interpreted as Programmer.id $\Rightarrow$ name and Programmer.id $\Rightarrow$ dept, respectively, because name and dept are simple attributes. Besides, the non-null options on title and depart are interpreted as Project.id $\Rightarrow$ title and Project.id $\Rightarrow$ depart, respectively. Beware they are not interpreted as Programmer.id $\Rightarrow$ title and Programmer.id $\Rightarrow$ 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 $\Rightarrow$ Project.id because Project is a complex attribute.

Can we map the non-null constraint defined by Rule 5.1 to the corresponding non-null constraint on the query result? It 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 4.1 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 Engineer1.ssn $\Rightarrow$ Emp1.name and Engineer1.ssn $\Rightarrow$ Emp1.dept, respectively. Likewise, if Project had the non-null option, its constraint would be mapped to Engineer1.ssn $\Rightarrow$ Project proj#. The non-null option on the title attribute is mapped not to Engineer1.ssn $\Rightarrow$ Proj-title, title but to Project proj# $\Rightarrow$ Proj-title, 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 Project proj# $\Rightarrow$ Project 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 5.2:**

1. $\Omega_{n.n} = O_{chain}(O, s_0) \equiv O_0, O_1, \ldots, O_n, s_0$.
2. $R_p.A := \text{AMF}(\Omega_{n.n}, id)$./* $A$ is always the key of $R_p$.*/
3. If $s_0$ is a simple attribute
   - then $R_s.B := \text{AMF}(\Omega_{n.n}, id)$
   - else $R_s.B := \text{AMF}(\Omega_{n.n}, s_0)$./* If $s_0$ is a complex attribute, $B$ is the key of $R_s$.*/

**D. 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 5.2, it is done as follows:

**Algorithm 5.3:**

1. Replace the filter $f_s$ on $R_s$ by $f_s \land (B \neq \text{null})$. /* 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 2(a), replace all joins on the path by inner joins.

For example, given the non-null constraints established in Section V-C, the following non-null filters are generated in the query of the Programmer object:

$$\text{Emp1.name} \neq \text{null}, \text{Emp1.dept} \neq \text{null}, \text{Project.dept} \neq \text{null}, \text{Proj-title} \neq \text{null}.$$ Besides, the following left outer joins are replaced by inner joins:

$$\text{Engineer1} \bowtie \text{Emp1}, \text{Project} \bowtie \text{Proj-title}.$$ Now we prove the correctness of Algorithm 5.3 with the following theorem.

**Theorem 5.2:** Given a join path $R_1 \bowtie R_2 \bowtie \cdots \bowtie R_n$ and a non-null constraint $A_1 \Rightarrow R_2, A_n$ 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 $A_n \neq \text{null}$.

**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 nonmatching tuples. Then a null $R_n.A_n$ is retrieved from the null tuples appended to the tuples of $R_i$ that 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 $A_n \neq \text{null}$ though 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.

**E. Reducing the Number of Left Outer Joins and Non-null Filters**

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

Considering entity integrity constraints, some non-null filters are removed if they are defined on attributes that 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 $\text{Emp1.name} \neq \text{null}$ and $\text{Emp1.dept} \neq \text{null}$ among the four non-null constraints generated in Section V-D because, as it was shown in Fig. 6(c), those two attributes are prohibited from having nulls.

We can also replace some left outer equijoins with inner equijoins if we consider referential integrity constraints. Since
a referential integrity \( R.A \rightarrow S.B \) allows \( R.A \) to be null, we define a stronger condition by introducing a variable \( \text{min} \) as follows.

**Definition 5.1 (min):** Given a join \( R_i \bowtie R_j \), let \( \text{min}_{ij} \) denote the minimum number of matching tuples in \( R_j \) for each tuple in \( R_i \). Note \( \text{min}_{ij} \) is not necessarily the same as \( \text{min}_{ji} \).

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

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

**Rules 5.2:**
- Given a single join predicate \( A \theta B \) for a join between two relations \( R_i \) and \( R_j \), \( \text{min}_{ij} > 0 \) if \( R_i.A \) is a non-null attribute and one or more of the following conditions are satisfied.
  \[
  \text{MIN}(R_i.A) < \text{MAX}(R_j.B), \quad \text{MIN}(R_i.A) > \text{MAX}(R_j.B), \quad \text{MIN}(R_i.A) = \text{MAX}(R_j.B).
  \]
  Otherwise \( \text{min}_{ij} = 0 \).
- Given a conjunctive join predicate \( A_1 \theta_1 B_1 \land A_2 \theta_2 B_2 \land \cdots \land A_k \theta_k B_k \) for a join between \( R_i \) and \( R_j \), \( \text{min}_{ij} > 0 \) for the conjunction of join predicates if \( \text{min}_{ij} > 0 \) for every single join predicate. Otherwise \( \text{min}_{ij} = 0 \).
- Given a disjunctive join predicate \( A_1 \theta_1 B_1 \lor A_2 \theta_2 B_2 \lor \cdots \lor A_k \theta_k B_k \) for a join between \( R_i \) and \( R_j \), \( \text{min}_{ij} > 0 \) for the disjunction of join predicates if \( \text{min}_{ij} > 0 \) for at least one join predicate. Otherwise \( \text{min}_{ij} = 0 \).
- Given a join path between two relations, such as \( R_i \bowtie R_{i+1} \bowtie \cdots \bowtie R_j \), \( \text{min}_{ij} > 0 \) if \( \text{min}_{k,k+1} > 0 \) for \( k = i, \ldots, j-1 \). Otherwise \( \text{min}_{ij} = 0 \).

If \( \text{min}_{ij} > 0 \) 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 \( \text{min} \).

**Algorithm 5.4:**
1) Find all join paths between pairs of nodes, such as \( R_i \) and \( R_j \), whose \( \text{min}_{ij} > 0 \).
2) For each join path found in Step 1, replace all joins on the path with inner joins.

14In other words, we ignore the fact that \( \text{min} \) may be accidentally greater than zero at the instance level though it is judged to be equal to zero at the semantic level.

15\( \text{min}_{ij} = 0 \) does not mean that \( \text{min}_{ij} \) is always equal to zero. Rather, it means that it is not known at the semantic level whether \( \text{min}_{ij} \) is greater than zero.

For example, in the query of Programmer object, we find a join path from Engineer, to Division1 for which all three joins have \( \text{min} > 0 \) because, as shown in Fig. 6, there are referential integrities Engineer.ssn \( \rightarrow \) Emp.ssn, Emp.dept \( \rightarrow \) Dept1.name, Dept1.name \( \rightarrow \) Division1.name, and there are integrity constraints prohibiting nulls for Engineer.ssn,Emp.dept, and Dept1.name, and none of the relations on the join path has a nonempty filter. We also find a join path from Proj - Assign1 to Project1 for which the \( \text{min} > 0 \). All these joins are replaced by inner joins. Note Project1 \( \bowtie \) Emp2 and Project1 \( \bowtie \) Sponsor1 cannot be replaced with inner joins because Project.leader and Project.sponsor are not non-null attributes.

**F. Summary of the Mechanism**

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

**Algorithm 5.5:**
1) /* Replace all joins within derived relations with inner joins. */
   For each derived relation \((R_b, E)\) in the set of pivot relations \((PS)\), replace all joins in \( E \) by inner joins.
2) /* Generate a non-null filter and prescribe a join. */
   a) /* Replace left outer joins if they prove to be equivalent to inner joins. */
      For each attribute \( s_0 \) of the object \( O \) that has a non-null option,
      i) /* Map the non-null option to a non-null constraint on the query result */
         if \( s_0 \) is a simple attribute
            then \( R_b.B := \text{AMF}(O_{s0.id}) \),/* \( B \) is always the key of \( R_b. */
         else \( R_b.B := \text{AMF}(O_{s0,s0.id}) \). /* If \( s_0 \) is a complex attribute, \( B \) is the key of \( R_b. */
      iii)/* Output the non-null constraint \( R_b.B \rightarrow R_b.B. */
   b) /* Generate a non-null filter. */
      i) Replace the filter \( f_s \) on \( R_b \) by \( f_s \land (B \neq \text{null}) \). /* 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 2(b)iiA,
             replace all joins on the path by inner joins.
3) /* Remove all non-null filters that can be shown to be redundant using the entity integrity constraint. */
   Remove \( R.A \neq \text{null} \) such that \( A \) is a non-null attribute.
4) /* Replace left outer joins if they prove to be equivalent to inner joins. */
   a) /* Find all join paths between pairs of nodes, such as \( R_i \) and \( R_j \), whose \( \text{min}_{ij} > 0 \).
      For each join path found in Step 1,
      replace all joins on the path with inner joins.

The graph of the query for the Programmer object, labeled with joins and non-null filters, is shown in Fig. 9. All the joins of the query except those between Project1 and Emp2 and between Project1 and Sponsor1 have been prescribed as...
Fig. 9. The query graph for the `Programmer` object with joins and non-null filters. (I denotes an inner join, and LO denotes a left outer join.)

inner joins. Two non-null filters have been attached as the selection conditions on the `Project1` and `Proj-Title1` nodes.

VI. CONCLUSION

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.

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, the semantics of which are mapped to the non-null constraint on the query result. 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.

ACKNOWLEDGMENT

The authors thank T. Risch, D. Quass, and other members of the KSYS group for fruitful discussions in the early stage of this work. L. DeMichiel, P. Rathmann, A. Keller, and K. J. Han gave valuable comments on the draft of this paper. Comments from anonymous referees were very helpful in revising this paper.

REFERENCES

[29] C. Date, “Referential integrity,” in Proc. 7th Int. Conf. Very Large Data Bases, Cannes, France, pp. 2-12, Sept. 1981.
Lee and Wiederhold: Outer Joins and Filters for Instantiating Objects


Byung Suk Lee (S'88-M'91) received the B.S. degree in electronics engineering from Seoul National University in 1980, the M.S. degree in electrical engineering from the Korea Advanced Institute of Science and Technology, Seoul, Korea, in 1982, and the Ph.D. degree in electrical engineering from Stanford University, Stanford, CA, in 1991.

He was an Engineer at Goldstar Electric Company, Korea, a Member of the Technical Staff at Bell Communications Research, Piscataway, NJ, and Supervisor at Datacom Global Communications, Princeton, NJ. He is currently an Assistant Professor of Graduate Programs in Software at the University of St. Thomas, St. Paul, MN. His current research interests include information systems in connection with object-oriented databases, distributed databases, and distributed computing.

Dr. Lee is a member of ACM and KSEA.

Gio Wiederhold (M'83-SM'90-F'92) received the degree in aeronautical engineering in Holland in 1957, and then learned to program at the NATO Air Defence Technical Center. He received the Ph.D. degree from the University of California, San Francisco, in 1976.

He is currently a Professor of Computer Science and Medicine (Research) at Stanford University, Stanford, CA. He also holds a courtesy appointment in electrical engineering. As a member of the Computer Systems Laboratory, the Starlab, and the Center for Integrated Systems, he is active in the application and development of knowledge-based techniques to database management. He is currently managing advanced database development within Ads and for Engineering Information Systems. Research into workstation systems for design and experiment planning integrates these interests. He also consults for many governmental and commercial enterprises, among them the United Nations Development Programme, the US Department of Health and Human Services, various US defense agencies, and with the Silicon Valley innovators.

Dr. Wiederhold is a member of the AAAS, AAMS, AIAA, ACL, ACM, ASTM, the Combustion Institute, and TIMS, and is a fellow of the ACM. He has been a member of the Committee for Data Management and Computation of the National Academy's Space Sciences Board. He is the Editor-in-Chief for ACM's Transactions on Database Systems and Associate Editor of Springer-Verlag's M.D. Computing Magazine and the IEEE Expert magazine. He has nearly 200 publications in computing and medicine, including a widely used McGraw-Hill textbook on database design, a 1987 book, File Organization for Database Design, and a monograph on Databases for Health Care with Springer-Verlag. A textbook on medical informatics, coauthored with Ted Shortliffe, is due this year. He has been chairman and program chairman of several conferences.