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

Byung Suk Lee and Gio Wiederhold

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.

1 Introduction

One of the approaches for integrating object-oriented programs with relational databases is to instantiate objects from relational databases through views [10, 11, 12, 13, 14, 7, 8, 9, 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 3.2.1, 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* [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 which eliminate the tuples containing null attributes 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 specifies 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 of 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.

2 Background Framework

2.1 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 [2, 3, 1, 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, 8, 9, 10, 11, 12, 13, 14, 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 are stored in a single relation `Employee(name, salary, department)`, and their owners are distinguished by the database view mechanism.

(a)	
Type Employee $\uparrow$is-a Type Project-manager	
$\Downarrow$generates	
Relation Employee(ssn, ...) Relation Project-manager(ssn, ...)	

(a) Object-centered perspective

(b)	
Type Employee $\uparrow$is-a Type Project-manager	
$\Uparrow$defined-from	
Relation Employee(ssn, ...) Relation Project(..., manager-ssn, ...)	

(b) Relation-centered perspective

Figure 1: Two perspectives of relation storage approach

2.2 Two Perspectives of the Relation Storage Approach

We observed two different perspectives within the relation storage approach: *object-centered* [4, 7, 8, 9] and *relation-centered* [10, 11, 12, 13, 14, 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[1]. The two perspectives may look like the two sides of the same coin, but they differ operationally. Figure 1 shows the two perspectives. In Figure 1a, the `Project-manager` type is mapped to the `Project-manager` relation. There exists a separate relation for each corresponding object type. In Figure 1b, 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, 22, 23]. For example, the IDB relation of the `Project-manager` is written as follows using the notion of Datalog [20].

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

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

2.3 Instantiating Objects from Relations through Views

Views provide a user-defined subset of a large database. Thus, as mentioned in Section 2.1 and Section 2.2, 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

[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'.

Database schema: /* Underlined attributes are keys. */

Employee($\underline{\text{ssn}}$, e_name, salary, dept#)

Deparment($\underline{\text{dept\#}}$, d_name, manager_ssn)

Child($\underline{\text{ssn}}$, $\underline{\text{c\_name}}$, sex, birth_date)

Object Type Employee /* [ ] denotes a tuple. */

[name: string, dept: Department,

children: [name: string, birthDate: string]]

View:

- Query expressed in relational algebra:

$$\Pi_{\{\text{ssn,e\_name,dept\#,c\_name,birth\_date}\}} \text{ Employee} \underset{\text{ssn=ssn}}{\bowtie} \text{Child}$$

- Mapping between object attributes and relation attributes:

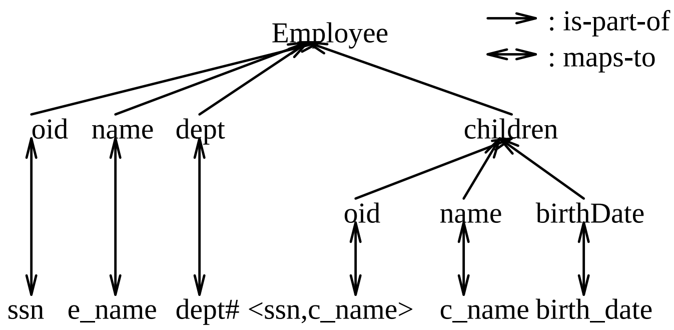


Figure 2: An example of instantiating an object type through views

structures of the two models. This mapping information is used by a NEST [24, 25, 26] operator to restructure a query result into objects. In [17, 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 structure than relations. For instance, objects support aggregation hierarchies [34] through an is-part-of relationship.[2]. 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*.

Figure 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*

[2]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.

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, 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 assumed to exist implicitly. The `dept` attribute of an `Employee` object has type `Department`. We call an attribute whose type is another object type as 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, the `dept` attribute of an `Employee` object is retrieved from the `dept#` of the `Department` relation[3]. 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[4]. 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 5.3.

3 Problem Formulation

3.1 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 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[5]. Formal definitions of the left outer join and the non-null filter are as follows.

Definition 3.1 (Left Outer Join) Given two relations R_1 and R_2, a left outer join from R_1 to R_2, denoted by $R_1 \text{⟕} R_2$, is defined as follows.

$$R_1 \text{⟕} R_2 = (R_1 \bowtie R_2) \cup ((R_1 - \Pi_{R_1}(R_1 \bowtie R_2)) \times \Lambda) \tag{1}$$

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, $R_1 \underset{A\theta B}{\text{⟕}} R_2$ produces the following set of tuples.

$$\{< t_1, t_2 > | t_1 \in R_1 \wedge t_2 \in R_2 \wedge t_1.A\theta t_2.B\} \cup \{< t_1, \Lambda > | t_1 \in R_1 \wedge \nexists t_2(t_2 \in R_2 \wedge t_1.A\theta t_2.B)\} \tag{2}$$

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

[3]Let us assume there is a type `Department` whose object id is retrieved from the `dept#` of the `Department` relation.

[4]Do not confuse 'subobjects' as the instances of a subtype in an is-a hierarchy of object types.

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

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 ($⟕$).

Definition 3.2 (Non-null filter) A non-null filter is a conjunction of predicates applicable to a base relation R, defined as follows.

$$R.A_1 \neq \text{null} \wedge R.A_2 \neq \text{null} \wedge \cdots \wedge R.A_i \neq \text{null} \tag{3}$$

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

3.2 Motivation

3.2.1 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 the objects of type `Employee` shown in Figure 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 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.

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 2.3, 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 [27] to prevent the loss of wanted objects, and non-null filters to prevent the retrieval of unwanted nulls.

3.2.2 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 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.

3.2.3 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 [30] is preseved.

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, 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.

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

where the join from R_2 to R_3 is a left outer join and the others are inner joins. If it is known there always exists a matching tuple of R_3 for every tuple of R_2, then the result of $R_1 \bowtie R_2 \;\text{⟕}\; R_3 \bowtie R_4$ 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 $R_2 \;\text{⟕}\; R_3$ 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.

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

3.4 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 4.1. 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 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 the paper, we first develop a rigorous system model to facilitate the mapping between objects and relations in Section 4. The mechanism is developed in Section 5, and the conclusion follows in Section 6.

4 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, 5, 6, 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 query[7] and defines a mapping between objects and relations. We restrict the query to an *acyclic select-project-join* query.

4.1 Object Type Model

Many existing object-oriented models support *aggregation* through nested structures and references. For example, the `Employee` object of Figure 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 [35, 36, 37, 38, 39, 40].

We use value-oriented object id's [44, 45] and retrieve them from the keys of relations[8]. Those relations providing object id's are called *pivot relations* [11, 12, 13]. As discussed in Section 2.2, 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 illustrates these concepts. In Figure 3*a*, the `Employee` relation is the pivot relation for the `Employee` object and provides its key `ssn` as the object id. Figure 3*b* shows the derived relation `Project-manager` of Figure 1, which becomes the pivot relation for the `Project-manager` object. It is defined by $\texttt{Employee} \underset{\text{ssn=manager-ssn}}{\bowtie} \texttt{Project}$, 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 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

[6] 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 [20] to mean the same concept as the referential integrity.

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

[8] 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.

$\boxed{\text{Relation Employee}}$ (ssn) $\xleftrightarrow{1:1}$ (id) Object Employee

(a) A pivot relation as a base relation

Relation Employee
| ssn$\bowtie$manager-ssn
Relation Project

(ssn) $\xleftrightarrow{1:1}$ (id) Object Project-manager

(b) A pivot relation as a derived relation

Figure 3: The concept of a pivot relation

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[9].

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

An example is shown in Figure 4. The `Project` attribute defines its own attributes and becomes a subobject of the `Programmer` object. It has its object id 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 (O-tree) The O-tree of an object O is a tree which has the following properties.

- Its root is labeled by 'O'.

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

[10] Many commercial tools for building object-oriented system applications, KEE[41, 42, 43] 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 4: An example object type

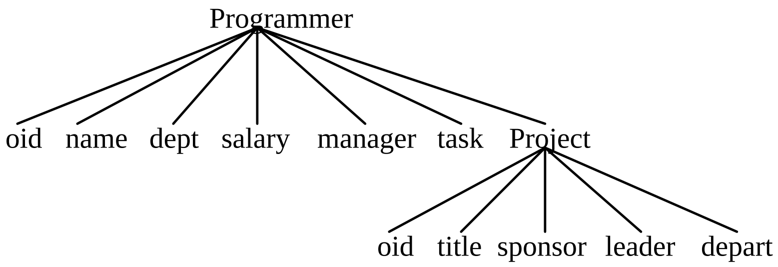


Figure 5: The O-tree of the `Programmer` object type

- 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 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.

Definition 4.2 (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 each element of an Oset has its object id mapped to the key of a pivot relation.

Definition 4.3 (Ochain) Given an object O and an 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 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`.

4.2 Data Model

Integrity constraints [28, 29, 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 3.2.3, 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 4.4 (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:

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

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 [30].

Definition 4.5 (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[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 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 4.5 are sufficient for our work. Figure 6 shows the schema, the referential integrity constraints and the entity integrity constraints of a sample database.

4.3 View Model

Figure 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 (PS) and a pivot mapping function (PMF). The PMF defines the mapping between the pivot relations and the (sub)objects[12].

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

4.3.1 Query Part

Figure 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 π projected from R. For example, the `Proj-Assign` relation is labeled with a filter `task = ''programming''` and a set of projected attributes {`task`}. Two occurrences

[11] In [30], 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)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.ssn
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 6: A sample database

of the same relation are distinguished by a tuple variable denoted as a subscript. For example, the two occurrence of the `Emp` relation in Figure 8 are distinguished into node $\texttt{Emp}_1$ and node $\texttt{Emp}_2$. 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 $\texttt{Proj-Assign}_1$ node to the $\texttt{Project}_1$ node denotes a left outer join from the `Proj-Assign` relation to the `Project` relation.

4.3.2 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.

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

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.

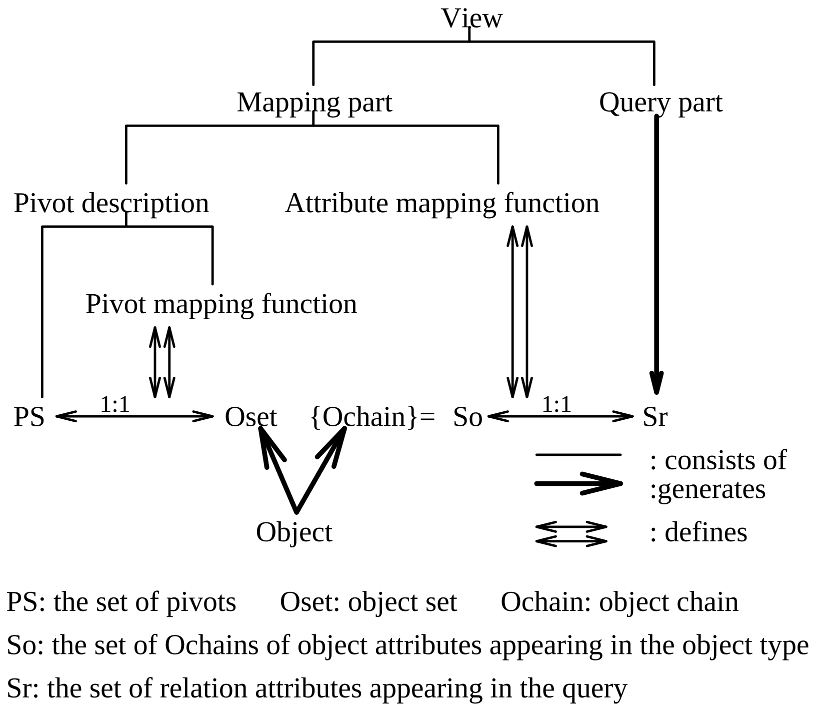


Figure 7: Mapping between objects and relations

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)

`Programmer.name` $\leftrightarrow$ `Emp`$_1$`.name`,

`Programmer.dept` $\leftrightarrow$ `Emp`$_1$`.dept`,

`Programmer.salary` $\leftrightarrow$ `Emp`$_1$`.salary`,

`Programmer.manager` $\leftrightarrow$ `Division`$_1$`.manager`,

`Programmer.task` $\leftrightarrow$ `Proj-Assign`$_1$`.task`,

`Programmer.Project.title` $\leftrightarrow$ `Proj-Title`$_1$`.title`,

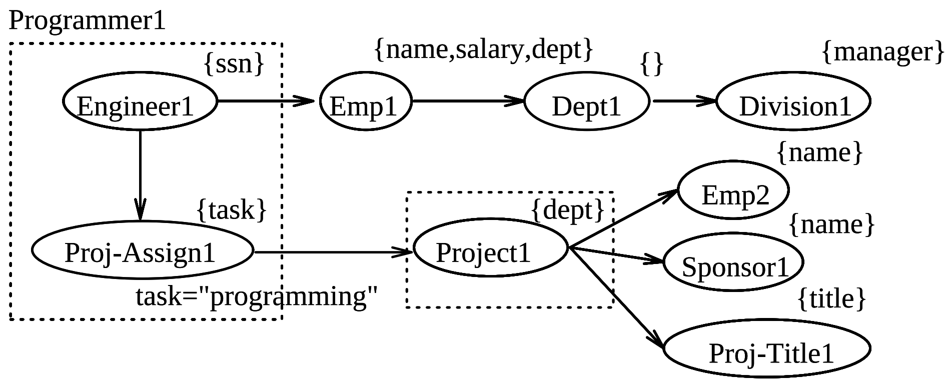
`Programmer.Project.sponsor` $\leftrightarrow$ `Sponsor`$_1$`.name`,

`Programmer.Project.leader` $\leftrightarrow$ `Emp`$_2$`.name`,

`Programmer.Project.depart` $\leftrightarrow$ `Project`$_1$`.dept`

As shown in Figure 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 object id. If it is a derived relation, the key of one of its base relations is mapped to the object id. Figure 8 shows two pivots, `Programmer`$_1$ and `Project`$_1$. Here `Project`$_1$ is the node of a base relation and `Programmer`$_1$ is the node of a derived relation defined by $\langle$ `Engineer`$_1$, {`Engineer`$_1$ $\underset{\text{ssn=ssn}}{\bowtie}$ $\sigma_{\texttt{task = ``programming''}}$ `Proj-Assign`$_1$}$\rangle$. A formal definition of a derived relation is as follows.

Definition 4.6 (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



(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 8: The query graph for the `Programmer` object

a *select-join*[13] expression such that, for *all possible instances* of the relations in E:

- $\Pi_{\mathrm{Key}(R_b)} E \subseteq \Pi_{\mathrm{Key}(R_b)} R_b$
- $\neg\exists E'(E' <_{R_b} E \wedge \Pi_{\mathrm{Key}(R_b)} E' \subseteq \Pi_{\mathrm{Key}(R_b)} E)$ where $E' <_{R_b} E$ 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 poper subexpression E' which, 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'' which 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)
`Programmer` $\leftrightarrow$ `Programmer`$_1$, `Project` $\leftrightarrow$ `Project`$_1$

As mentioned in Section 4.1, 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 4.1
For each pivot relation $p \in$ PS begin

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

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 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.

{ `Programmer.id` $\leftrightarrow$ $\texttt{Engineer}_1\texttt{.ssn}$, `Programmer.Project.id` $\leftrightarrow$ $\texttt{Project}_1\texttt{.proj\#}$ }.

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 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 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.

5 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.

5.1 Overview

There are two source of nulls retrieved from databases. One is from the nulls stored in the tuples, 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 3.2.1. 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 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 5.2.)
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 5.3 and Section 5.4.)
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 5.5.)

5.2 Joins within a Derived Relation

As mentioned in Section 2.2, 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 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.

$$\Pi_{\mathrm{Key}(R_1)}(R_1 \bowtie R_2 \bowtie \cdots \bowtie R_i \text{⟕} R_{i+1} \bowtie \cdots \bowtie R_n) = \Pi_{\mathrm{Key}(R_1)}(R_1 \bowtie R_2 \bowtie \cdots \bowtie R_i) \quad (5)$$

That is, there exists a proper subexpression which, 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 $\langle$ `Engineer`$_1$, { `Engineer`$_1$ $\bowtie_{\mathrm{ssn=ssn}}$ $\sigma_{\texttt{task = ``programming''}}$ `Proj-Assign`$_1$ } $\rangle$ defined to provide the semantics of the `Programmer` object, the join between `Engineer`$_1$ and `Proj-Assign`$_1$ must be an inner join. If the join is evaluated as a left outer join, it retrieves all tuples of `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:

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.

5.3 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. It requires there should exist a non-null s_0 given the oid of the object. Let us denote this non-null constraint as $O.\mathrm{id} \Rightarrow s_0$. 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 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., $O_n.\text{id} \Rightarrow s_0.\text{id}$, then s_0.id cannot be null.

For example, given the `Programmer` object of Figure 4, 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 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 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 $\texttt{Engineer}_1\texttt{.ssn} \Rightarrow \texttt{Emp}_1\texttt{.name}$ and $\texttt{Engineer}_1\texttt{.ssn} \Rightarrow \texttt{Emp}_1\texttt{.dept}$, respectively. Likewise, if `Project` had the non-null option, its constraint would be mapped to $\texttt{Engineer}_1\texttt{.ssn} \Rightarrow \texttt{Project}_1\texttt{.proj\#}$. The non-null option on the `title` attribute is mapped not to $\texttt{Engineer}_1\texttt{.ssn} \Rightarrow \texttt{Proj-Title}_1\texttt{.title}$ but to $\texttt{Project}_1\texttt{.proj\#} \Rightarrow \texttt{Proj-Title}_1\texttt{.title}$ 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 $\texttt{Project}_1\texttt{.proj\#} \Rightarrow \texttt{Project}_1\texttt{.dept}$.

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. $\Omega_{0,n}.s_0 := \text{Ochain}(O, s_0) \equiv O_0.O_1.\cdots.O_n.s_0$.
2. $R_p.A := \text{AMF}(\Omega_{0,n}.\text{id})$. /* A is always the key of R_p. */
3. 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. */
4. Output the constraint '$R_p.A \Rightarrow R_s.B$'.

5.4 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 5.2, it is done as follows.

Algorithm 5.3

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 5.3, 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 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 $R_1.A_1 \Rightarrow R_n.A_n$ 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.

5.5 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 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 $\mathtt{Emp_1.name} \neq \mathtt{null}$ and $\mathtt{Emp_1.dept} \neq \mathtt{null}$ among the four non-null constraints generated in Section 5.4 because, as it was shown in Figure 6c, those two attributes are prohibited from having nulls.

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

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

Using only the semantics of `min` without considering the instances of relations[14], we define the following rules for deciding whether `min` is greater than zero or not. MIN($R.A$) denotes the minimum non-null value $R.A$ can have, and MAX($R.A$) denotes the maximum non-null value $R.A$ can have. MIN($R.A$) and MAX($R.A$) are known from the range constraints or value constraints, if there are any, on $R.A$.

Rule 5.2

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

  $\theta =$ '$=$' and $R_i.A \mapsto R_j.B$ and the filter f_j on R_j is empty, or
  $\theta =$ '$>$' and $\mathrm{MIN}(R_i.A) > \mathrm{MAX}(R_j.B)$, or
  $\theta =$ '$\geq$' and $\mathrm{MIN}(R_i.A) \geq \mathrm{MAX}(R_j.B)$, or
  $\theta =$ '$<$' and $\mathrm{MAX}(R_i.A) < \mathrm{MIN}(R_j.B)$, or
  $\theta =$ '$\leq$' and $\mathrm{MAX}(R_i.A) \leq \mathrm{MIN}(R_j.B)$, or
  $\theta =$ '$\neq$' and $(\mathrm{MIN}(R_i.A) > \mathrm{MAX}(R_j.B)$ or $\mathrm{MAX}(R_i.A) < \mathrm{MIN}(R_j.B))$.

  Otherwise $\mathtt{min}_{ij} = 0$[15].

- 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 a join between R_i and R_j, $\mathtt{min}_{ij} > 0$ for the conjunction of join predicates if $\mathtt{min}_{ij} > 0$ for *every* single join predicate. Otherwise $\mathtt{min}_{ij} = 0$.

- 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 a join between R_i and R_j, $\mathtt{min}_{ij} > 0$ for the disjunction of join predicates if $\mathtt{min}_{ij} > 0$ for *at least one* join predicate. Otherwise $\mathtt{min}_{ij} = 0$.

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

[14] 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.

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

If $\mathtt{min}_{ij} > 0$ 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.

Algorithm 5.4

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

For example in the query of Programmer object, we find a join path from $\mathtt{Engineer}_1$ to $\mathtt{Division}_1$ for which all three joins have min > 0 because, as shown in Figure 6, there are referential integrities $\mathtt{Engineer}_1\mathtt{.ssn} \mapsto \mathtt{Emp}_1\mathtt{.ssn}$, $\mathtt{Emp}_1\mathtt{.dept} \mapsto \mathtt{Dept}_1\mathtt{.name}$, $\mathtt{Dept}_1\mathtt{.name} \mapsto \mathtt{Division}_1\mathtt{.name}$, and there are integrity constraints prohibiting nulls for $\mathtt{Engineer}_1\mathtt{.ssn}$, $\mathtt{Emp}_1\mathtt{.dept}$, and $\mathtt{Dept}_1\mathtt{.name}$, and none of the relations on the join path has a non-empty filter. We also find a join path from $\mathtt{Proj\text{-}Assign}_1$ to $\mathtt{Project}_1$ for which the min > 0. All these joins are replaced by inner joins. Note $\mathtt{Project}_1$ ⟕ $\mathtt{Emp}_2$ and $\mathtt{Project}_1$ ⟕ $\mathtt{Sponsor}_1$ can not be replaced with inner joins because Project.leader and Project.sponsor are not non-null attributes.

5.6 Summary of the Mechanism

Given a query with initial left outer joins, the overall mechanism developed in Section 5 is as follows.

Algorithm 5.5

1. /* 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.
2. 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 2(b)iiA,
            replace all joins on the path by inner joins.

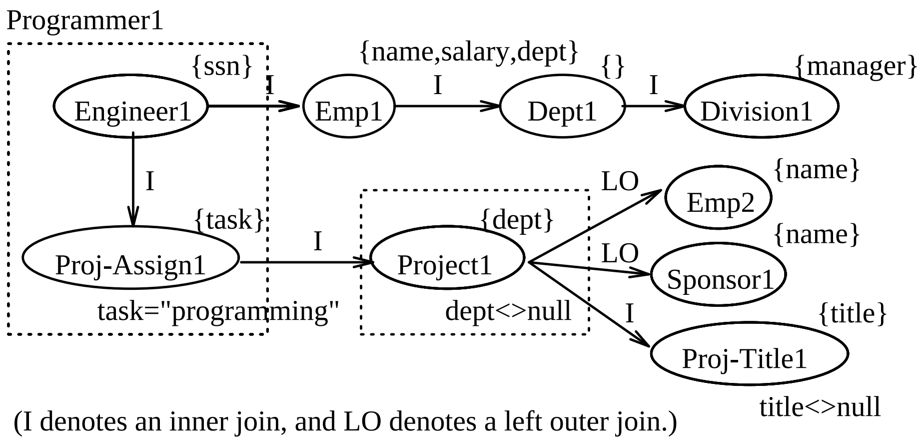


Figure 9: The query graph for the `Programmer` object with joins and non-null filters

3. /* Remove all non-null filters which can be shown to be redundant using the entity integrity constraint. */
   Remove '$R.A \neq$ 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 $\mathtt{min}_{ij} > 0$.
   (b) 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 Figure 9. All the joins of the query except those between $\mathtt{Project}_1$ and $\mathtt{Emp}_2$ and between $\mathtt{Project}_1$ and $\mathtt{Sponsor}_1$ have been prescribed as inner joins. Two non-null filters have been attached as the selection conditions on the $\mathtt{Project}_1$ and $\mathtt{Proj\text{-}Title}_1$ nodes.

6 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, whose semantics is 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.

Acknowledgement

We had fruitful discussions with Tore Risch, Dallan Quass, and other members of the KSYS group in the early stage of this work. Linda DeMichiel, Peter Rathmann, Arthur Keller, and Ki-Joon Han gave valuable comments on the draft of this paper. Comments from anonymous referees were very helpful to revise this paper. This work was performed as part of the KBMS project, supported by DARPA Contract No. N039-84-C-02111, and was partially sponsored by the Center for Integrated Facility Engineering at the Stanford University.

References

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

[2] R. Agrawal and N. Gehani, "ODE (Object Database and Environment): The Language and the Data Model," in *Proc. ACM SIGMOD Int. Conf. Management Data*, Portland, Oregon, May 1989.

[3] A. Paepcke, "PCLOS: A Flexible Implementation of CLOS Persistence," in *Proc. European Conf. Object-Oriented Programming*, Oslo, Norway, Aug. 1988.

[4] D. Maier and J. Stein, "Development of an Object-Oriented DBMS," in *Proc. Int. Conf. Object-Oriented Programming Syst., Languages, Applications*, pp. 472-482, Sept. 1986.

[5] S. Ford, et al., "Zeitgeist: Database Support for Object-Oriented Programming," in *Proc. Int. Workshop Object-Oriented Database Syst.*, pp. 23-42, 1988.

[6] W. Kim, N. Chou, and J. Garza, "Integrating an Object-Oriented Programming System with a Database System," in *Proc. Int. Conf. Object-Oriented Programming Syst., Languages, Applications*, pp. 142-152, Sept. 1988.

[7] D. Fishman, et al., "Iris: An Object-Oriented Database Management System," *ACM Trans. Office Information Syst.*, vol.5, no.1, pp. 48-69, Jan. 1987.

[8] M. Stonebraker and L. Rowe, "The Design of POSTGRES," in *Proc. ACM SIGMOD Int. Conf. Management Data*, pp. 340-354, 1986.

[9] T. Learmont and R. Cattell, "An Object-Oriented Interface to a Relational Database," in *Proc. Int. Workshop Object-Oriented Database Syst.*, 1987.

[10] G. Wiederhold, "Views, Objects, and Databases," *IEEE Computer*, pp. 37-44, Dec. 1986.

[11] T. Barsalou, W. Sujansky, and G. Wiederhold, "Expert database systems in medicine—The PENGUIN project," in *Proc. AAAI Spring Sym. on AI in Medicine*, Stanford University, CA, pp. 14-18, Mar. 1990.

[12] T. Barsalou, and G. Wiederhold, "Complex objects for relational databases," *Computer Aided Design* (Special issue on object-oriented techniques for CAD), vol. 22, no. 8, pp. 458-468, 1990.

[13] 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," *Int. J. of Syst. Automation: Research and Applications*, Int. Society for Productivity Enhancement, 1991.

[14] G. Wiederhold, T. Barsalou, and S. Chaudhuri, "Managing Objects in a Relational Framework", Tech. rep. STAN-CS-89-1245, Stanford University, Jan. 1989.

[15] W. Premeriani, M. Blaha, J. Rumbaugh, and T. Varwig, "An Object-oriented Relational Database," *Communications of the ACM*, vol. 33, no. 11, Nov. 1990.

[16] W. Rubenstein, M. Kubicar, and R. Cattell, "Benchmarking Simple Database Operation," in *Proc. ACM SIGMOD Int. Conf. Management Data*, May 1987.

[17] B. Lee and G. Wiederhold, "Efficiently Instantiating View-objects from Remote Relational Databases," submitted for publication, June 1991.

[18] B. Lee, "Efficiently Instantiating Objects from Relational Databases through Views", Ph.D. thesis, Department of Electrical Eng., (Also published as Tech. rep. STAN-CS-90-1346, Computer Science Department), Stanford University, Dec. 1990.

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

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

[21] K. Morris, J. Ullman, and A. Van Gelder, "Design Overview of the NAIL! System," in *Proc. Int. Logic Programming Conf.*, 1986.

[22] S. Tsur and C. Zaniolo, "LDL: A Logic-based Data-Language," in *Proc. 12th Int. Conf. Very Large Data Bases*, Kyoto, Aug. 1986.

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

[24] P. Fischer and S. Thomas, "Operators for Non-First-Normal-Form Relations," in *Proc. IEEE Int. Computer Software and Application Conf.*, Nov. 1983.

[25] M. Roth, H. Korth, and A. Silberschatz, *Theory of Non-First-Normal-Form Relational Databases*, Tech. rep. TR-84-36, Department of Computer Science, University of Texas at Austin, Dec. 1984.

[26] S. Abiteboul and N. Bidoit, "Non-First Normal Form Relations to Represent Hierarchically Organized Data," in *Proc. ACM Symp. Principles of Database Syst.*, April 1984.

[27] C. Date, "The Outer Join," in *Proc. 2nd Int. Conf. Databases*, Cambridge, Britain, Sept. 1983.

[28] E. Codd, "Extending the Relational Database Model to Capture More Meaning," *ACM Trans. Database Syst.* vol.4, no.4, Dec. 1979.

[29] C. Date, "Referential Integrity," in *Proc. 7th Int. Conf. Very Large Data Bases*, Cannes, France, pp. 2-12, Sept. 1981.

[30] C. Date, *An Introduction to Database Systems*, fourth edition, vol.1, Addison-Wesley Publishing Company, 1986.

[31] F. Bancilhon, "Object-Oriented Database Systems," invited lecture in *Proc. 7th ACM Symp. Principles of Database Syst.*, Austin, TX, Mar. 1988.

[32] D. Maier, "Why Isn't There an Object-Oriented Data Model?," in *Proc. IFIP 11th World Computer Congress*, San Francisco, CA, Sept. 1989.

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

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

[35] R. Haskin and R. Lorie, "On Extending the Functions of a Relational Database System," in *Proc. ACM SIGMOD Int. Conf. Management Data*, pp. 207-212, June 1982.

[36] R. Lorie and W. Plouffe, "Complex Objects and Their Use in Design Transactions‘," in *Proc. IEEE Annual Meeting - Database Week: Eng. Design Applications*, pp. 115-121, May 1983.

[37] K. Dittrich and R. Lorie, "Object-oriented Database Concepts for Engineering Applications", Tech. rep. RJ4691(50029), IBM Research Lab., San Jose, CA, May 1985.

[38] W. Wilkes, P. Klahold, and G. Schlageter, "Complex and Composite Objects in CAD/CAM Databases," in *Proc. 5th IEEE Int. Conf. Data Eng.*, Los Angeles, February 1989.

[39] P. Buneman, S. Davidson, and A. Watters, "A Semantics for Complex Objects and Approximate Queries," in *Proc. ACM Symp. Principles of Database Syst.*, 1988.

[40] W. Kim, J. Banerjee, and H. Chou, "Composite Object Support in an Object-Oriented Database System," in *Proc. Int. Conf. Object-Oriented Programming Syst., Languages, Applications*, pp. 118-125, Oct. 1987.

[41] *IntelliCorp KEE™ Software Development System User's Manual*, Document No. 3.0-U-1, Intellicorp Inc., Mountain View, CA, July 1986.

[42] R. Kempf and M. Stelzner, "Teaching Object-Oriented Programming with the KEE System," in *Proc. Int. Conf. Object-Oriented Programming Syst., Languages, Applications*, pp. 11-25, Oct. 1987.

[43] "KEEconnection: A Bridge between Databases and Knowledge Bases," unpublished paper, Intellicorp Inc., Mountain View, CA, 1987.

[44] S. Khoshafian and G. Copeland, "Object Identity," in *Proc. Int. Conf. Object-Oriented Programming Syst., Languages, Applications*, 1986.

[45] S. Abiteboul and P. Kanellakis, "Object Identity as a Query Language Primitive," in *Proc. ACM SIGMOD Int. Conf. Management Data*, Portland, Oregon, May 1989.