Canonical Abstraction for Outerjoin Optimization

Jun Rao
IBM Almaden Research Center
junrao@almaden.ibm.com

Hamid Pirahesh
IBM Almaden Research Center
pirahesh@almaden.ibm.com

Calisto Zuzarte
IBM Toronto
calisto@ca.ibm.com

ABSTRACT

Outerjoins are an important class of joins and are widely used in various kinds of applications. It is challenging to optimize queries that contain outerjoins because outerjoins do not always commute with inner joins. Previous work has studied this problem and provided techniques that allow certain reordering of the join sequences. However, the optimization of outerjoin queries is still not as powerful as that of inner joins.

An inner join query can always be canonically represented as a sequence of Cartesian products of all relations, followed by a sequence of selection operations, each applying a conjunct in the join predicates. This canonical abstraction is very powerful because it enables the optimizer to use any join sequence for plan generation. Unfortunately, such a canonical abstraction for outerjoin queries has not been developed. As a result, existing techniques always exclude certain join sequences from planning, which can lead to a severe performance penalty.

Given a query consisting of a sequence of inner and outer joins, we, for the first time, present a canonical abstraction based on three operations: outer Cartesian products, nullification, and best match. Like the inner join abstraction, our outerjoin abstraction permits all join sequences, and preserves the property of both commutativity and transitivity among predicates. This allows us to generate plans that are very desirable for performance reasons but that couldn't be done before. We present an algorithm that produces such a canonical abstraction, and a method that extends an inner-join optimizer to generate plans in an expanded search space. We also describe an efficient implementation of the best match operation using the OLAP funtionalities in SQL:1999. Our experimental results show that our technique can significantly improve the performance of outerjoin queries.

1. INTRODUCTION

An outerjoin [2] is a modification of an inner join ($\bowtie$) in that it preserves all information from one or both of the input relations. It can be further categorized into *left* ($\longrightarrow$), *right* ($\longleftarrow$) or *full* ($\longleftrightarrow$) outerjoin, depending on which side needs to be preserved. For example, the following SQL query returns all the departments and the employees in them. For those departments without any employees, the department names are still listed, but with the employee name set to *null*.

```
SELECT   department.dname, employee.ename
FROM     department LEFT JOIN employee
         ON department.no=employee.dno
```

Outerjoins are important because they are frequently used in the following traditional applications [10]: (a) certain OLAP queries where we need to preserve tuples from the fact table with unknown (or missing) dimensional values; (b) constructing *hierarchical views* that preserve objects with no children; and (c) queries generated by external tools and query rewriting units. The emergence of XML provides more applications for outerjoins. For example, in information integration, schema mapping [12] involves the discovery of a query or a set of queries that transform the source data into a new structure. When mapping relational data to XML, the transformation queries rely heavily on outerjoins to avoid missing data. In another example, the construction of XML results in XQUERY [1] often needs the outerjoin semantic so that a parent element can survive without a matching child element.

When there are only inner joins in a query, a query optimizer is allowed to consider all join sequences and selects the cheapest execution plan among them. Changing the order of join evaluation is a powerful optimization technique and can improve execution time by orders of magnitude. However, when outerjoins are present in addition to inner joins, changing the join order does not always give the correct results. For example, $R \overset{Prs}{\longrightarrow} (S \overset{Pst}{\bowtie} T) \neq (R \overset{Prs}{\longrightarrow} S) \overset{Pst}{\bowtie} T$. The problem of outerjoin reordering has been studied in [16, 9, 10, 15], with [10] being the most comprehensive. They all try to expand the search space for outerjoin queries in one way or another. However, none of them achieves the goal of allowing all join sequences. For example, conjuncts in the ON clause of an outerjoin are always treated as a whole and cannot be applied separately. As another example, transitivity is never applied across two outerjoins to generate redundant predicates.

To see the impact of this, consider two queries in Figures 1(a) and 1(b). In Q1, the outerjoin uses two conjuncts

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for pro®t or commercial advantage and that copies bear this notice and the full citation on the ®rst page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior speci®c permission and/or a fee.
SIGMOD 2004, June 13-18, 2004, Paris, France.
Copyright 2004 ACM 1-58113-859-8/04/06 ...$5.00.

(a) Q1

```
SELECT  R.k, S.k, T.k
FROM    R LEFT JOIN (S INNER JOIN T ON S.a=T.a)
        ON R.b=S.b and R.c=T.c
```

(b) Q2

```
SELECT  R.k, S.k, T.k
FROM    R LEFT JOIN (S LEFT JOIN T ON S.a=T.a)
        ON R.a=S.a
```

(c)

```
SELECT  R.k, S.k, T.k
FROM    (R LEFT JOIN S ON R.b=S.b) INNER JOIN T
        ON S.a=T.a and R.c=T.c
```

(d)

```
SELECT  R.k, S.k, T.k
FROM    (R LEFT JOIN T ON R.a=T.a) LEFT JOIN S
        ON R.a=S.a and T.a=S.a
```

(e)

```
WITH    Nullified AS            (
SELECT  ...
FROM    (R LEFT JOIN S ON R.b=S.b) LEFT JOIN T
        ON S.a=T.a and S.c=T.c  )
SELECT  R.k, S.k, T.k, ...
FROM    Nullified
WHERE   ...
```

(f)

```
WITH    Nullified AS  (
SELECT  ...
FROM    (R LEFT JOIN T ON R.a=T.a) LEFT JOIN S
        ON R.a=S.a  )
SELECT  R.k, S.k, T.k, ...
FROM    Nullified
WHERE   ...
```

Figure 1: (a), (b) original outerjoin queries Q1 and Q2; (c), (d) incorrect naively rewritten queries; (e), (f) correct rewritten queries using nullification and best match

as the join predicate. Suppose that relations S and T are both large, but relation R is small. One would like to be able to apply one of the conjuncts, say $R.b$=$S.b$, to join relation R with S first, as shown in Figure 1(c). In Q2, the two join predicates (from different ON clauses) share a common attribute. Suppose that S is really big. One would like to be able to generate an additional predicate $R.a$=$T.a$ through transitivity, and to use it for joining relation R with T first, as shown in Figure 1(d). Unfortunately, neither Figure 1(c) nor Figure 1(d) gives the correct results. If we temporarily assume that $R(k, a, b, c) = \{(r1, 1, 1, 1)\}$, $S(k, a, b) = \{(s1, 1, 1)\}$, and $T(k, a, c) = \phi$, the answers for Q1 and Q2 are $A1 = \{(r1, -, -)\}$ and $A2 = \{(r1, s1, -)\}$ respectively ("$-$" for *null*). On the other hand, the query in Figure 1(c) generates ϕ (changing the inner join in Figure 1(c) to a left outerjoin still gives the wrong answer $A2$) and that in Figure 1(d) generates $A1$. Had the outerjoins been changed to inner joins, both transformations would have been valid.

To solve this problem completely, we try to find a canonical abstraction suitable for outerjoins. We exploit two logical operations, *nullification* and *best match* (definitions will be given in Section 3), introduced in [15]. For a given query that has a mixture of inner and outer joins, we then express it in an *outerjoin canonical abstraction*—outer Cartesian products of all relations, followed by a sequence of nullification operations, followed by a best match operation at the end. Our abstraction has the property of both commutativity and transitivity, and hence provides many additional opportunities for query optimization. We will describe in Section 4 an algorithm of generating the outerjoin canonical abstraction. To take advantage of the new opportunities, the planning phase in an optimizer needs to be extended. In Section 5, we explain our plan generation method based on the canonical abstraction. Although a nullification operation is straightforward to implement, a best match operation is fairly complex. Our abstraction will not be useful without an efficient implementation of a best match. Instead of trying to implement it directly, we decompose a best match into one or more simpler building blocks, each of which can then be implemented by exploiting the *OLAP* functionality in the latest SQL standard [14]. Our implementation of a best match operation is described in Section 6.

Using our approach, Q1 and Q2 can now be rewritten as queries in Figures 1(e) and 1(f), respectively. In both cases, the rewritten queries perform a selection over a predefined view $Nullified$ (by the WITH clause). We deliberately leave out some of the details in the queries, which will be provided in Section 6. The important observation here is that the input relations are now joined in a more desirable sequence. Therefore, although syntactically more complex, the rewritten queries can have a huge performance advantage over the original ones. Our experimental results in Section 7 strongly support this claim.

2. BACKGROUND AND PREVIOUS WORK

[7] observes that *null-intolerant* predicates (those that cannot evaluate to true when referencing a *null* value) can simplify queries using outerjoins. For example, if predicate Prs (the subscript contains relations referenced by the predicate) is null-intolerant, the following rule holds.

$$R \overset{Prs}{\bowtie} (S \overset{Pst}{\longleftarrow} T) = R \overset{Prs}{\bowtie} (S \overset{Pst}{\bowtie} T)$$

Dayal's study [5] gives some initial rules on valid evaluation sequences for joins and one-sided outerjoins. Equations below hold when predicates are null-intolerant.

$$R \xrightarrow{Prs} (S \xrightarrow{Pst} T) = (R \xrightarrow{Prs} S) \xrightarrow{Pst} T$$
$$(R \xrightarrow{Prt} T) \xrightarrow{Prs} S = (R \xrightarrow{Prs} S) \xrightarrow{Prt} T$$

[8] describes a normal form that represents an outerjoin query as a sequence of *minimum unions* of join results. The minimum union is responsible for removing tuples that should not be present in the query result. However, this normal form focuses only on relations and ignores the join predicates. Thus, it is not as powerful as the abstraction proposed in this paper. For example, it neither considers breaking the conjuncts in a join predicate nor generates redundant predicates through transitivity.

Galindo-Legaria and Rosenthal [7, 9, 10] have done pioneering work in the area of outerjoin reorderability. The authors identify a special class of query called a *simple* query,

which has the property that the query graph (without specifying the join sequence) unambiguously determines the semantic of the query. Given a simple query, they compute a conflicting set for each join predicate by analyzing the query graph. The conflicting set of a predicate P contains some other join predicates, which if applied after P, will give incorrect results. Based on the conflicting set, they proposed two kinds of approaches in plan generation, one *without compensation* and one *with compensation*. The former only generates plans that contain only valid join sequences, and the later allows plans containing some invalid sequences and then tries to compensate for the incorrect results through a *generalized outerjoin*. However, their approach still does not consider all join sequences, including the two desired ones in Figure 1. Also, no efficient implementation is given for a generalized outerjoin operation.

[15] uses an extended eligibility list (EEL) to represent the conflicts between outerjoins and inner joins. Although EEL is logically equivalent to the conflicting set representation used in [10], it's easily exploited by an optimizer. [15] also introduces the concepts of *nullification* and *best match*, which our canonical abstraction is based on. The two together can perform the right compensation needed for an invalid join sequence.

3. PRELIMINARY

In this section, we specify our assumptions and define or revisit needed terminologies. We want to focus our attention on the most common cases. Therefore, we define an *outerjoin query* to be one that contains any number of left outerjoins (right outerjoins are converted to the left ones) and inner joins. The less common full outerjoins and antijoins will be discussed later in Section 8. An outerjoin query is given by an operator tree where all leaf nodes are base relations (which can contain local predicates) and inner nodes are join operators. For any outerjoin in the tree, the preserving side is always the left and the null-producing side is always the right. We also assume that all join predicates are null-intolerant (this implies no Cartesian products in the query) and that the operator tree has already been simplified (i.e., outerjoins are converted to inner joins if possible). Lastly, we assume that there is always a key attribute KID (the tuple ID can always be used to serve as the KID) for each relation and all $KIDs$ are carried along to the root of the operator tree. We note that a large class of queries are satisfied under our assumptions (in particular, all simple queries are satisfied under our assumptions).

DEFINITION 3.1. Given a relation R, a Boolean predicate p and a set of attributes $a \in R$, a nullification operation $\lambda_{P,a}(R)$ is defined as: $\{r \in R \mid \text{if } p \neq true, \text{ set } r.a \text{ to } null\}$. In such an operation, P is then referred to as a *nullification predicate* and a is referred to as *nullified attributes*. We have to be a little bit careful here. Since SQL [14] has a three-value logic, $p \neq true$ is not the same as $!p = true$. For example, when p evaluates to *unknown*, the former expression is satisfied, but the latter is not. It can be proven that λ has the following properties.

(0) $\lambda_{P1\&P2,a}(R) = \lambda_{P1,a}(\lambda_{P2,a}(R)) = \lambda_{P2,a}(\lambda_{P1,a}(R))$
(& for conjunction)
(1) $\lambda_{P,a1\cup a2}(R) = \lambda_{P,a1}(\lambda_{P,a2}(R)) = \lambda_{P,a2}(\lambda_{P,a1}(R))$
(The equation still holds when P references $a1$ and $a2$.)

DEFINITION 3.2. Tuple $r1$ *dominates* tuple $r2$, if for every non-null attribute in $r2$, $r1$ has the same value in the corresponding attribute, and $r2$ has more attributes with *null* values than $r1$. We also refer to those attributes where $r2$ is *null* and $r1$ is not as *dominated*. For example, tuples $t1=(1,-,3)$ and $t2=(1,2,-)$ are both dominated by $(1,2,3)$. The second attribute in $t1$ and the third attribute in $t2$ are the dominated attributes, respectively. Domination was also known as *subsumption* in [8, 18].

DEFINITION 3.3. Given a relation R, a best match operation $\beta(R)$ is defined as: $\{r \in R \mid r$ is not dominated or duplicated by any other tuples in R and is not an all-null tuple$\}$. We call those tuples that are removed from R by a $\beta(R)$ operation *spurious* tuples. Note that $\beta(R) = R$, when R is a base relation.

DEFINITION 3.4. Given two relations R and S, we define an *outer Cartesian product* (reusing $\times$) as $R \overset{1=1}{\longleftrightarrow} S$. The only difference between an outer Cartesian product and the conventional one is that when one of the relations (say R) is empty, the former returns all tuples in S and pads *null* on attributes in R, while the latter returns an empty set.

DEFINITION 3.5. We can then decompose outerjoins and inner joins using λ, β and $\times$ in (2) and (3) below. Note that we require the outer Cartesian product to handle empty inputs, which was ignored in previous work [15]. For simple presentation, in $\lambda_{P,a}$, we use the relation name as a if a includes all attributes in the relation (we say that the relation is *nullified* by P).

(2) $R \overset{Prs}{\longrightarrow} S = \beta(\lambda_{Prs,S}(R \times S))$
(3) $R \overset{Prs}{\bowtie} S = \beta(\lambda_{Prs,R\cup S}(R \times S))$

The above representation essentially says that an outer join predicate nullifies the null-producing relation, and an inner join predicate nullifies both input relations. Observe that the result of an inner join or an outerjoin does not contain spurious tuples since a β operation is applied in both. Both β and λ take a relation as input and return a relation (with the same schema) in the output. Therefore, they are both composable, i.e., they can be used as the input of other relational operators. The commutative rules ([15]) among β, λ and $\times$ are summarized below:

(4) $\beta(\beta(R)) = \beta(R)$
(5) $\beta(R) \times S = \beta(R \times S)$
(6) $\lambda_{P,a}(R) \times S = \lambda_{P,a}(R \times S)$
P only refers to R and $a \in R$
(7) $\beta(\lambda_{P,a}(\beta(R))) = \beta(\beta(\lambda_{P,a}(R))) = \beta(\lambda_{P,a}(R))$
(rely on our assumption that P is null-intolerant)

In the above rules, (4) says that two consecutive best matches can be reduced to one; (5) and (6) say that an outer Cartesian product is commutative with a best match and with a nullification operation; and (7) says that a best match is commutative with a nullification operation as long as there is a best match at the very end. We should point out here that $\times$ carries an implicit null-tolerant predicate *true*. The reason why $\times$ still commutes with λ is because changing a non-null value to a *null* does not change the result of the implied predicate since it always returns true.

Using the above rules, we can convert an outerjoin query to a bestmatch-nullification representation, and then push all outer Cartesian products as far inside as possible and remove intermediate best matches (such a representation is referred to as BNR). For example, we can have the following BNR transformation.

(8) $(R \xrightarrow{Prs} S) \xrightarrow{Pst} T = \beta(\lambda_{Pst,T}(\lambda_{Prs,S}(R \times S \times T)))$

(9) $R \xrightarrow{Prs} (S \overset{Pst}{\bowtie} T) = \beta(\lambda_{Prs,S \cup T}(\lambda_{Pst,S \cup T}(R \times S \times T)))$

However, it turns out that λ operations themselves are not commutative in general. To see that, let's consider the right side of equation (8) above using a concrete example. Suppose that $R(a) = \{(1)\}$, $S(a) = \{(2)\}$ and $T(a) = \{(2)\}$, respectively. $R \times S \times T$ produces exactly one tuple $\{(1, 2, 2)\}$. Also suppose that Prs and Pst are given by $R.a = S.a$ and $S.a = T.a$, respectively. In Figure 2(a), we show the steps of evaluating the two λ operations in the order given by equation (8), i.e., $\lambda_{Prs,S}$ followed by $\lambda_{Pst,T}$. Since Prs evaluates to *false*, $S.a$ is set to *null* after applying $\lambda_{Prs,S}$. The next step is interesting. Pst now evaluates to *unknown* because of the *null* value set in the previous row. It would have evaluated to *true* if the *null* value were not set. Therefore, $T.a$ is set to *null* in the end. In Figure 2(b), we show that the result has changed when we switch the order of the two λ operations. The reason is that Pst is evaluated earlier and does not see the *null* value it's supposed to see, and therefore evaluates to *true* instead of *unknown*. Consequently, switching the λ operations in (8) gives an incorrect result. Notice that in equation (0), the two λ operations are exchangeable. The difference lies in that the two λ operations in equation (0) nullify the same attribute sets whereas the λ operations in equation (8) do not. Finally, readers can verify that changing the order of the two λ operations in equation (9) does not affect the correctness of the final result, though.

	(R, S, T)
$R \times S \times T$	$\{(1, 2, 2)\}$
$\lambda_{Prs,S}$	$\{(1, -, 2)\}$
$\lambda_{Pst,T}$	$\{(1, -, -)\}$

(a) $\lambda_{Pst,T}(\lambda_{Prs,S}...)$

	(R, S, T)
$R \times S \times T$	$\{(1, 2, 2)\}$
$\lambda_{Pst,T}$	$\{(1, 2, 2)\}$
$\lambda_{Prs,S}$	$\{(1, -, 2)\}$

(b) $\lambda_{Prs,S}(\lambda_{Pst,T}...)$

Figure 2: Effect of Changing the Order of λ

4. CANONICAL ABSTRACTION FOR OUTERJOINS

For a given outerjoin query, we seek a canonical abstraction resembling that of an inner join query. A BNR of an outerjoin query almost gives us such a canonical abstraction except that λ operations are not interchangeable. However, by taking advantage of the fact that all predicates are null-intolerant, we can make λ operations interchangeable.

Again, consider the sequence $\lambda_{Pst,T}(\lambda'_{Prs,S}...)$ in equation (8). To differentiate the two λ, we refer to the inner one as λ'. As we have seen earlier, the reason why λ cannot be moved inside λ' is that Pst then will not see the new *null* values in S updated by λ'. We'd like to stress here that it is the *new null* values that we are concerned about. *Null* values present in relation S itself does not affect the reordering of the two λ operations. Since all predicates

Input	an operator tree T of an outerjoin query
Output	a nullification set (NS_R) for each relation R
Method:	
For each node n traversed in postfix order in T	
If (n is a base relation R)	
$NS_R = \phi$	
Else (n is a join with predicate P)	
Let R_L and R_R be the set of all relations referenced by P in the left and right side respectively	
if (n is an outer join)	
For each relation r in the right side	
$NS_r = NS_r \cup \{P\} \cup \{p \in NS_t \mid t \in R_L\}$	
Else if (n is an inner join)	
For each relation l in the left side	
$NS_l = NS_l \cup \{P\} \cup \{p \in NS_t \mid t \in R_R\}$	
For each relation r in the right side	
$NS_r = NS_r \cup \{P\} \cup \{p \in NS_t \mid t \in R_L\}$	

Table 1: An Algorithm Generating Nullification Sets

are null-intolerant, we identify the following *rippling* effect: $Prs \neq true \Rightarrow S$ is $null \Rightarrow Pst \neq true \Rightarrow T$ is *null*. This implies $Prs \neq true \Rightarrow T$ is *null*, which is equivalent to adding a $\lambda_{Prs,T}$. By adding such an implied λ operation, we do not change the semantic of the original expression. What's more, the relative evaluating order of the λ operations are no longer important because we have short-circuited the rippling effect of the generated *nulls*, the cause that prevents λ from reordering. For example, if we further apply $\lambda_{Prs,T}$ on the last row in Figure 2(b), we will get exactly the same final result as in Figure 2(a). It's easy to verify that applying $\lambda_{Pst,T}$, $\lambda'_{Prs,S}$ and $\lambda_{Prs,T}$ in any other sequence also produces the correct result. The rippling effect is transitive and can pass through multiple predicates. For example, given a sequence $\lambda_{Ptu,U}(\lambda_{Pst,T}(\lambda_{Prs,S}...))$, an implied $\lambda_{Prs,U}$ can be inferred since the *null* value in S (introduced by Prs) first propagates to T through Pst, and then to U through Ptu. Next, we are going to show that for a given BNR, we can precompute all implied λ operations, the addition of which gives us commutativity among λ.

Given an outerjoin query, we associate each relation R with a *nullification set* NS_R, which is used to collect all predicates (including implied ones) that can nullify R if not *true*. We present an algorithm of populating the nullification sets in Table 1. Each node in the operator tree of the query is visited bottom-up in postfix order. If a node corresponds to a relation, we simply initialize its nullification set to empty. If a node is an outerjoin, we note that the join predicate P nullifies each relation r in the right side (remember this is always the null-producing side) and so should be added to NS_r. However, this is not enough. Consider each relation t that is referenced by P and is from the left side. Predicates in NS_t nullify t and thus indirectly nullify each r. Therefore, these predicates (implied) are also added to each NS_r. If a node is an inner join, we note that P nullifies relations in both sides. So we can basically perform the same computation once for relations in the left side and another for those in the right. By induction, it can be shown that the nullification sets we computed is complete in the sense that all predicates that can introduce *null* in R are included in NS_R. We claim two important properties of the nullification sets in Theorem 4.1.

Q1: $R \overset{R.b=S.b \ \& \ R.c=T.c}{\longrightarrow} (S \overset{S.a=T.a}{\bowtie} T)$

NS_R:	ϕ
NS_S:	$\{R.b{=}S.b, \quad R.c{=}T.c, \quad S.a{=}T.a\}$
NS_T:	$\{R.b{=}S.b, \quad R.c{=}T.c, \quad S.a{=}T.a\}$

Q2: $R \overset{R.a=S.a}{\longrightarrow} (S \overset{S.a=T.a}{\longrightarrow} T)$

NS_R:	ϕ
NS_S:	$\{R.a{=}S.a\}$
NS_T:	$\{R.a{=}S.a, \quad S.a{=}T.a, \quad \underline{R.a{=}T.a}\}$

Q3: $(R \overset{R.a=S.a}{\longrightarrow} S) \overset{S.b=T.b}{\longrightarrow} T$

NS_R:	ϕ
NS_S:	$\{R.a{=}S.a\}$
NS_T:	$\{R.a{=}S.a, \quad S.b{=}T.b\}$

Q4: $((R \overset{R.a=S.a}{\longrightarrow} S) \overset{R.b=T.b}{\longrightarrow} T) \overset{S.c=U.c \ \& \ T.d=U.d}{\longrightarrow} U$

NS_R:	ϕ
NS_S:	$\{R.a{=}S.a\}$
NS_T:	$\{R.b{=}T.b\}$
NS_U:	$\{R.a{=}S.a, \quad R.b{=}T.b, \quad S.c{=}U.c, \quad T.d{=}U.d\}$

Figure 3: Nullification Sets Examples

THEOREM 4.1. (1) The nullification sets computed by the algorithm in Table 1 satisfy the following property: applying all $\lambda_{NS_R,R}$ in any order is equivalent to applying the λ sequence in the original BNR. (2) Two equivalent outerjoin queries (always generating the same result) have the same nullification set for each corresponding relation.

Proof: (1) We can show by induction that after processing a node n in the original operator tree, the property holds for the BNR corresponding to the subtree rooted at n. (2) We can show that for each valid outerjoin transformation rule, the nullification sets on the new operator tree remain the same. It has already been proven in [10] that those valid outerjoin transformation rules are complete, i.e., any equivalent query can be derived from the original query by applying a sequence of valid transformation rules. ∎

We illustrate our algorithm on four queries in Figure 3. In Q1, $S.a = T.a$ is added to both NS_S and NS_T by the inner join. The outerjoin then includes $R.b = S.b$ and $R.c = T.c$ in the nullification set of S and T since they are both in the right side of the join. The process for Q2 is similar and the nullification sets are shown below the query. Let's ignore the underlined predicate for the moment. The computation for Q1 and Q2 are relatively straightforward since there is no implied λ. In Q3, predicate $R.a = S.a$ is first added to NS_S by the first join, and is subsequently added to NS_T by the second join because of the implied λ. As an example to show the correctness of the second claim in Theorem 4.1, readers can verify that query $R \overset{R.a=S.a}{\longrightarrow} (S \overset{S.b=T.b}{\longrightarrow} T)$, which is equivalent to Q3, shares the same nullification sets as Q3. Finally we show a four-way join query in Q4. Observe that predicates $R.a = S.a$ and $R.b = T.b$ are carried over to NS_U by the last join.

Collectively, the nullification sets we computed actually give the canonical abstraction we want. It's clear from Theorem 4.1 that an outerjoin query can always be represented by a sequence of outer Cartesian products, followed by a sequence of $\lambda_{NS_R,R}$ on each relation R in the query, followed by a final β operation. Such abstraction gives us many additional opportunities for optimizing an outerjoin query. First of all, since the outer Cartesian products can be evaluated in any binary sequence, this essentially allows us to consider all the join orders for planning. Secondly, the commutativity between λ operations and outer Cartesian products enables us to push a λ as deep as where it first becomes eligible (i.e., all referenced attributes are present in the input). We can then introduce a β operation after a λ and convert the $\beta(\lambda(... \times ...))$ sequence back to either an inner join or an outerjoin, using the reverse of equation (2) and (3) in Section 3. Thirdly, when a nullification set has multiple conjuncts, we can, based on equation (0), split a "big" λ into two smaller ones, each applying a subset of the conjuncts. This means that conjuncts given by the same ON clause in an outerjoin query do not always have to be applied together, which was impossible before. For example, in Q1, it now becomes possible to apply predicate $R.b = S.b$ by itself and therefore to join R and S together first. Lastly, we can compute the transitive closure of predicates within each nullification set. Consider Q2 in Figure 3 as an example, the underlined predicate $R.a = T.a$ is generated through transitivity within NS_T. This additional predicate serves as the only link between R and T, which means that R and T can now be joined using this predicate. This additional conjunct can actually introduce a subtle semantic difference. Consider the situation when $R.a$, $S.a$ and $T.a$ are 5, *null* and 7, respectively. $R.a = S.a$ & $S.a = T.a$ evaluates to *unknown*, while $R.a = S.a$ & $S.a = T.a$ & $S.a = T.a$ evaluates to false. Fortunately, this is fine for a λ operation since the testing on the nullification predicate does not distinguish between *unknown* and false (both are not true). In the next section, we will describe how to extend a query optimizer to take advantage of these new opportunities.

We can organize all the nullification sets for an outerjoin query into a directed graph where each node n corresponds to a unique nullification set (noted as NS_n) and there is an edge from node $n1$ to node $n2$ if $NS_{n1} \subset NS_{n2}$. We then assign each relation R to the node n where $NS_n = NS_R$. We refer to such a graph as DAG_{ns}. It's obvious that DAG_{ns} is acyclic. Otherwise, from a cycle $n1, n2, ...nk, n1$ in the graph, we can derive $NS_{n1} \subset NS_{n2} ... \subset NS_{nk} \subset NS_{n1}$, which is impossible. A *root* node in a DAG_{ns} is defined as the node n whose NS_n has the smallest size. A DAG_{ns} has only one root node and all nodes can be reached by a path starting from the root. Again, this can be proven by induction that in Table 1, after processing each node n in the original operation tree, the DAG_{ns} for the subtree rooted at n has exactly one root node and the rest of the nodes are reachable from the root node. The DAG_{ns} for queries Q1 to Q4 are shown in Figure 4, where each circle represents a node. Relations assigned to a node are shown inside the circle and the corresponding nullification set is next to the circle. Notice that in Q1, two relations S and T are assigned to the same node and the DAG_{ns} for Q4 is not a tree. We make the following interesting observations on the DAG_{ns} for an outerjoin query. Those observations will be useful for the implementation of β operations to be discussed in Section 6.

OBSERVATION 4.1. (a) For each tuple t in the fully nullified query result (i.e., after all $\lambda_{NS_R,R}$ have been applied), if the KID of relation S (assigned to node n) is *null*, then

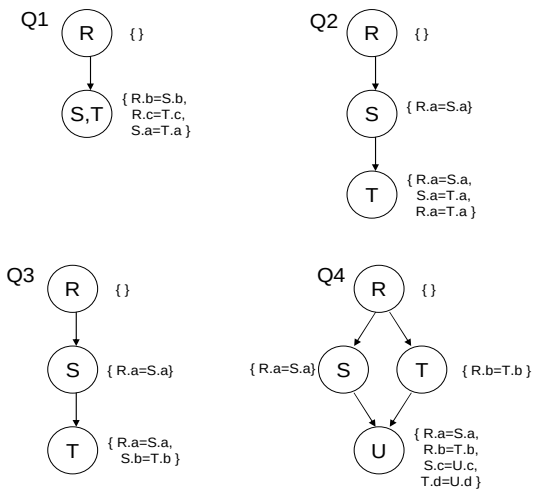

Figure 4: Examples of DAG_{ns}

the KID of each relation T assigned to node n and n's descendants are all *null*. The reasoning is that each T always has an equivalent, or super, set of nullification predicates relative to S. (b) For a node n in DAG_{ns}, predicates in NS_n can only refer to relations assigned to nodes on the path from root to n. (c) Relations assigned to the same node n in a DAG_{ns} must be inner joined together in the original operator tree, using predicates in NS_n. (d) An outerjoin query equivalent to the original one can be composed by first inner joining relations assigned to the same node in a DAG_{ns} and then for each pair of (parent, child) nodes, outer joining the intermediate results on the nodes (results from the child serves as the right side), using predicates in $NS_{child} - NS_{parent}$.

Before moving on, we'd like to point out that the nullification sets could also be useful for some other theoretical analysis of outerjoin queries. For example, given n relations and k predicates, how many different outerjoin queries can be formed? A simple upper bound is k^n since each predicate can belong to one or more nullification sets. A closer estimation can be derived by analyzing all the possible DAG_{ns} that can be composed from the n relations and k predicates.

5. PLAN GENERATION BASED ON CANONICAL ABSTRACTION

In this section, we discuss how to use the outerjoin canonical abstraction for plan generation in a conventional bottom-up join optimizer. For an outerjoin query, we assume that the nullification set of each relation NS_R has been calculated using the algorithm in Table 1 and that transitive closure has been computed within each NS_R. Predicates in NS_R are given as a set of conjuncts C. Because any join order is allowed in our canonical abstraction, the plan search space for outerjoin queries becomes much larger, which increases (but does not guarantee) the possibility of finding a cheaper plan. On the other hand, the larger the search space, the longer the optimization time. In order to avoid wasting too much time on generating unpromising plans, we first limit our search space using the following heuristic rules: (1) A λ operation is always converted back into inner or outer joins whenever possible (by introducing a β after it). (2) Unconvertible λ operations are deferred until all relations are joined. (3) Each conjunct is considered only once as a join predicate when it first becomes eligible. However, the same conjunct can be used again for nullification in the end. (4) No outer Cartesian product is allowed by its own. We feel that those limits preserve most promising plans while reducing the search space considerably, and thus provide a good balance between plan quality and optimization time.

A dynamic programming style join optimizer [17] starts by enumerating each non-overlapping pair of relation sets. It then finds all eligible predicates and uses them to join the two relation sets together. Various physical operators (e.g., hash join) are then considered for the actual join implementation. To avoid generating redundant plans, the optimizer maintains a memory-resident structure (referred to as *MEMO*, following the terminology used in [11]) for holding non-pruned plans. Each MEMO entry corresponds to a unique logical expression, which is typically determined by a relation set and an applied predicate set. Plans corresponding to the same logical expression are inserted into the same MEMO entry. A plan with a higher cost is pruned if there is a cheaper plan with the same or more general properties for the same MEMO entry. The MEMO structure is populated from bottom up such that entries for smaller relation sets are filled before those for larger relation sets. Plan properties are used to distinguish among plans corresponding to the same logical expression. For example, if a plan P produces an *interesting order* [17] that can avoid a required sort subsequently, it might be better than a cheaper unsorted plan with the same logical expression. Keeping tuple ordering as a plan property prevents plans such as P from being pruned too early.

To extend the optimizer for outerjoin support, we reuse the join enumeration process, but for each pair of relation sets (S, L) to be joined together, call a new method presented in Table 2. The method is divided into two parts. The first part is called on every (S, L) pair while the second part is only called when $S \cup L$ contains all relations in the query. We also maintain in each plan an extra property: the nullification sets NS' corresponding to the plan tree. We will explain later why this new plan property is necessary.

The first part of the method iterates through each conjunct c in C that is eligible on (S, L). If c fully nullifies either S (i.e., for each $s \in S$, $c \in NS_s$) or L, it is added to a set C_j (which will be used later as join predicates). Otherwise, c is ignored because it cannot be converted back to a join predicate. In either case, c is removed from C and will not be considered in subsequent joins. The method then decides which type of join should C_j be used for. If C_j fully nullifies both S and L, an inner join will be used. Otherwise, a left outer join will be chosen and the fully nullified relation set is used as the null-producing (right) side of the join. The new plan property NS' is propagated in a similar fashion as the algorithm in Table 1. Theorem 5.1 guarantees that the decision made on the join type is consistent for all conjuncts in C_j.

THEOREM 5.1. For any two conjuncts $c1$ and $c2$ in C_j used to join S and L together in Table 2, the following can never occur: (a) $c1$ fully nullifies both S and L, $c2$ fully nullifies either S or L, but not both; (b) $c1$ only fully nullifies S, $c2$ only fully nullifies L.

Input	S and L, two relation sets to be joined; NS_R, the nullification sets as computed by the algorithm in Table 1)
Output	a list of plans that join S and L, each plan carries an additional property, the nullification sets NS' for the plan tree

Method:
Part I
$C_j = \phi$
For each conjunct c in C that is eligible on S and L
  If ($c \in NS_s$ for each relation s in S) //c fully nullifies S
    add c to C_j
  Else
   if ($c \in NS_l$ for each relation l in L) //c fully nullifies L
    add c to C_j
  Remove c from C
If (C_j is not empty)
  If (C_j fully nullifies S and L) plan $S \overset{C_j}{\bowtie} L$
  Else if (C_j fully nullifies S) plan $L \overset{C_j}{\longrightarrow} S$
  Else if (C_j fully nullifies L) plan $S \overset{C_j}{\longrightarrow} L$
  Compute the nullification sets NS' for the join plan
Else reject this (S, L) pair

Part II
If ($S \cup L$ includes all relations to be joined)
  For each generated plan P (with nullification sets NS')
    For each R where $NS_R - NS'_R \neq \phi$
      P=adding $\lambda_{NS_R - NS'_R, R}$ on top of P
    If (any λ is added above)
      P_f=adding a β operator on top of P

Table 2: Plan Generation using Nullification Sets

Proof: We first make the following observation on the original operator tree T: If a conjunct c refers to a relation R and is not in NS_R, then $R \prec S$ for each $S, c \in NS_S$, where $\prec$ compares the *infix* ordering of the two relations in T. Now, we assume that both $c1$ and $c2$ exist in the original operator tree (generated conjuncts can also be handled accordingly). We consider the following three cases. Case (1): both $c1$ and $c2$ are outerjoin predicates in the original query. Then $c1$ and $c2$ can only fully nullify either S or L, but not both (remember that $c1$ and c are not eligible in S itself, nor in L). Suppose that $c1$ nullifies L and $c2$ nullifies S. There exists an $S_i \in S$ referenced by $c1$ and an $L_j \in L$ referenced by $c2$. Based on the earlier observation, we have the following infix ordering in the original operator tree: $S_i \prec l \in L$ and $L_j \prec s \in S$. However, this implies $S_i \prec S_i$, which is impossible. For the other two cases where at least one of the conjuncts is used as an inner join predicate in the original query, we can show that neither (a) nor (b) is possible in a similar way. ∎

We now move to the second part of the method in Table 2, when all relations are joined together. We have to decide the compensation (if any) needed for each plan P generated. In order to do this, we compare NS'_R with NS_R for each relation R. Observe that NS'_R is always a subset of NS_R (this can be proved by induction on the building process of P). If $NS_R - NS'_R$ is not empty, we know that R is not completely nullified and will add a $\lambda_{NS_R - NS'_R, R}$ operator on top of P to complete the nullification. Such an R is referred to as *further nullified*. If any λ operator is added, a final β operator is added to create the final plan P_f. Note that the nullification sets for P_f is exactly the same as NS for the original operation tree. It's easy to see that our method will always generate a plan corresponding to the join sequence in the original query, and that such a plan does not need further nullification.

(a) Q1: $R \overset{R.b=S.b \ \&\ R.c=T.c}{\longrightarrow} (S \overset{S.a=T.a}{\bowtie} T)$

Planning input	C={c1:$R.b$=$S.b$, c2:$R.c$=$T.c$, c3:$S.a$=$T.a$} $NS_R=\phi$ NS_S={c1, c2, c3} NS_T={c1, c2, c3}
Planning Part I	(R, S) c1[R, $\underline{S}$] $R \overset{c1}{\longrightarrow} S$ (RS, T) c2[$R\underline{S}$, $\underline{T}$] c3[$R\underline{S}$, $\underline{T}$] $RS \overset{c2\ \&\ c3}{\longrightarrow} T$ P: $(R \overset{c1}{\longrightarrow} S) \overset{c2\ \&\ c3}{\longrightarrow} T$
Planning Part II	$NS'_R=\phi$ NS'_S={c1} NS'_T={c1, c2, c3} P_f: $\beta(\lambda_{c2\ \&\ c3, S}((R \overset{c1}{\longrightarrow} S) \overset{c2\ \&\ c3}{\longrightarrow} T))$

(b) Q2: $R \overset{R.a=S.a}{\longrightarrow} (S \overset{S.a=T.a}{\longrightarrow} T)$

Planning input	C={c1:$R.a$=$S.a$, c2:$S.a$=$T.a$, c3:$R.a$=$T.a$} $NS_R=\phi$ NS_S={c1} NS_T={c1, c2, c3}
Planning Part I	(R, T) c3[R, $\underline{T}$] $R \overset{c3}{\longrightarrow} T$ (RT, S) c1[$R\underline{T}$, $\underline{S}$] c2[$R\underline{T}$, S] $RT \overset{c1}{\longrightarrow} S$ P: $(R \overset{c3}{\longrightarrow} T) \overset{c1}{\longrightarrow} S$
Planning Part II	$NS'_R=\phi$ NS'_S={c1} NS'_T={c3} P_f: $\beta(\lambda_{c1\ \&\ c2, T}((R \overset{c3}{\longrightarrow} T) \overset{c1}{\longrightarrow} S))$

Figure 5: Planning for Q1 and Q2

We illustrate our method by showing the plan generation process for queries Q1 and Q2 (from Figure 1). Figure 5(a) describes the planning for Q1. The input to planning consists of the conjunct set C and the nullification set for each relation (as computed in Section 4). In the first part of planning, we start with the relation set pair (R, S). Notice that the only eligible conjunct is $c1$. In the bracket next to a conjunct, we show its nullification pattern by underlining the nullified relation in both inputs. Since $c1$ only nullifies S, we construct the first join as $R \overset{c1}{\longrightarrow} S$. Next, the pair (RS, T) is being considered, $c2$ and $c3$ are eligible. Both conjuncts nullify T, but not RS (neither nullifies R). Therefore, we construct the second join as an outerjoin (T on the right side) using both conjuncts. Notice that $c3$ was an inner join predicate in Q1 and is now automatically converted into an outerjoin predicate. The plan generated at the end of the the first part of planning is given by P. The nullification sets for plan P is given by NS'. Comparing NS' with NS, we find out that $c2$ and $c3$ are missing in NS'_S. Therefore, they are used to further nullify S. Further nullification can introduce spurious tuples, which will be removed by the final β operator in plan P_f.

Figure 5(b) shows the planning for $Q2$. Notice that the conjunct ($c3$) generated through transitivity is also added to C. Suppose that we first consider the relation set (R, T). $c3$ becomes eligible and nullifies T. Thus, $c3$ is used to construct an outerjoin. Had we not been able to exploit

transitivity, the join between R and T would not have been possible. The construction of the second join for (RT, S) is also interesting. Observe that both $c1$ and $c2$ are eligible. $c1$ nullifies S and can be used as an outerjoin predicate. However, $c2$ fully nullifies neither RT (does not nullify R), nor S. Therefore, $c2$ cannot be used as a join predicate and is ignored. In the second part, we discover that $c1$ and $c2$ need to be used together to further nullify T to create the final plan P_f.

Finally, we want to stress the importance of keeping the additional property NS' in each plan. Observe the join plan $P : (R \overset{c1}{\longrightarrow} S) \overset{c2\ \&\ c3}{\longrightarrow} T$ generated for Q1 in Figure 5(a). P contains the same set of relations and applied predicates as the original plan $R \overset{c1\ \&\ c2}{\longrightarrow} (S \overset{c3}{\bowtie} T)$ (referred to as $P0$). Suppose that P is cheaper than $P0$. We cannot just prune $P0$ at this moment because P needs further compensation (through λ and β operations), after which the cost of P can be actually higher than $P0$ (which does not need further compensation). The new plan property NS' serves to distinguish between the two plans P and $P0$ so that they do not prune against each other (notice that NS' is different from NS). Such an extension is an example of the classic space and time tradeoff, in which we trade plan storage for (potentially) better plan quality.

6. IMPLEMENTATION OF BEST MATCH

A λ operation can be easily implemented using the *case* expression in SQL. For example, $\lambda_{P,a}$ can be implemented by an expression "CASE WHEN P THEN a END". Although not specified explicitly, a *null* value will be returned for the expression if the test in WHEN fails. In the rest of this section, we focus on the implementation of β operations.

We first formally define the problem we try to solve. Given a join result of n relations $R_1, ..., R_n$ (referred to as a set R_{1-n}) and a set of k relations $R_{b1}, ..., R_{bk}$ (together referred to as R_b) that are further nullified, we want to implement the final β operation that removes all spurious tuples introduced because of further nullification. The nullication set for each relation R is given by NS_R. The further nullified join result is referred to as a relation *Nullified*. For simple presentation, we assume that *Nullified* only contains the KID (K_i) of each relation. A direct implementation of a β operation is possible, but hard because of the intrinsic complexity of the operation. Instead, we build a β operation using standard SQL functionalities. We refer to a SQL query implementing a β operation as a β query.

Let's first consider a concrete example using query Q1 in Figure 5(a). In Figure 6(a), we show the tuples in three relations R, S and T. The result of evaluating Q1 is given in Figure 6(b). We now consider the plan P generated in Figure 5(a) that chooses a different join sequence. Figure 6(c) shows the join result of P and Figure 6(d) shows the result after P is further nullified. Comparing Figure 6(d) with Figure 6(b), we can see that in Figure 6(d), the second tuple is dominated by the first one and the fourth tuple is duplicated by the third one. Both the second and the fourth tuple need to be removed by the β operation. We deliberately sorted *Nullified* on $\langle rid,sid,tid\rangle$ (*nulls* sort last), which gives the result a *favorable* ordering, i.e., dominating tuples are always sorted before dominated ones and a spurious tuple always finds a dominating or duplicated tuple right before it. We can then perform the β operation through a single pass of

(a) Data in each relation

R(rid,b,c)
(r1,1,5)
(r2,2,6)

S(sid,a,b)
(s1,1,1)
(s2,2,1)
(s3,3,2)
(s4,4,2)

T(tid,a,c)
(t1,1,5)

(b) Q1:$R \overset{c1\ \&\ c2}{\longrightarrow} (S \overset{c3}{\bowtie} T)$

(rid, sid, tid)
(r1, s1, t1)
(r2, -, -)

(c) $P : (R \overset{c1}{\longrightarrow} S) \overset{c2\ \&\ c3}{\longrightarrow} T$

(rid, sid, tid)
(r1, s1, t1)
(r1, s2, -)
(r2, s3, -)
(r2, s4, -)

(d) $\lambda_{c2\ \&\ c3,S}$ on (c)

(rid, sid, tid)
(r1, s1, t1)
(r1, -, -)
(r2, -, -)
(r2, -, -)

Figure 6: Base Data and Results of Q1($c1$:$R.b$=$S.b$, $c2$:$R.c$=$T.c$, $c3$:$S.a$=$T.a$)

Nullified using the following SQL query to filter out the spurious tuples. We assume that when accessing each tuple, we magically have access to its immediate previous (whose attributes are appended with _p) as well, and will discuss how to achieve that a little bit later.

```
SELECT   *
FROM     Nullified
WHERE    rid <> rid_p or
         tid <> tid_p or
         sid <> sid_p
```

Let's ignore the very first tuple for the moment. Observe that if a tuple t is not spurious, it will not dominate the tuple (t_p) before t (otherwise, t_p will sort after t because of the favorable ordering). Thus t will differ from t_p on at least one of three $KIDs$ and the values in the differing KID are not *null*. This is precisely what the three disjuncts in the WHERE clause test (note that each disjunct evaluates to *unknown* on *nulls*). For example, in Figure 6(d), the third tuple (not spurious) differs from the second tuple on rid (neither is *null*). On the other hand, if t is spurious, each of its KID must either be *null* or match the corresponding value in t_p, which dominates or duplicates t. Therefore, none of the disjuncts is satisfied. For example, it's easy to verify that in Figure 6(d), the second tuple (dominated) and the fourth tuple (duplicated) will be filtered out by the disjuncts in the WHERE clause. In order to use such an approach to implement β, we still need to answer the following two questions: (1) how to access the previous tuple together with the current one in SQL; and (2) how to find a sort key that gives a favorable ordering of the tuples.

We start by addressing the first question. SQL has become more and more powerful over the past ten years through various standard extensions. However, many new functionalities have been overlooked and not fully exploited. For example, the OLAP amendment was standardized in SQL:1999 and is now supported by most major database products such as DB2 [3] and Oracle [4]. The OLAP amendment returns ranking, row numbering and existing column function information as a scalar value on a window around the current tuple, through an "OVER ... WINDOW construct" expression. Each window construct can be accompanied by

an "ORDER BY" and a "PARTITION BY" clause, which specifies the ordering and the partitioning for tuples in the window. For example, we can use the following query to compute the five-day average of the IBM stock price on each day.

```
SELECT   day,
         AVG(price) OVER
            (ORDER BY day DESC
             ROWS BETWEEN 2 PRECEDING
             AND 2 FOLLOWING)
            AS 5day_avg
FROM     IBM_stock
```

The OLAP amendment provides us with the answer to the first question. A window construct "ROWS BETWEEN 1 PRECEDING AND 1 PRECEDING" (simply represented as "ROWS ..." in the rest of the section) gives us a handle to the previous tuple. If a favorable order is given by a sort key $FSK = \langle K_1, ..., K_n \rangle$, we can compose a *basic* β query as shown in Figure 7. Again, we assume that *null* values are sorted after the none-null ones. Each K_i_p computes K_i in the previous tuple. We make a special case in dealing with the very first tuple, which does not have a preceding one. Since the planning algorithm we considered in Section 5 won't generate all-null tuples, the very first tuple, if existing, is never spurious. This is handled by the first disjunct in Figure 7 that accepts a tuple if its *rownum* equals to one.

```
SELECT   K1, ..., Kn,
  max(K1)        OVER (   ORDER BY FSK
                          ROWS ...) as K1_p,
                 ...
  max(Kn)        OVER (   ORDER BY FSK
                          ROWS ...) as Kn_p,
  rownumber()    OVER (   ORDER BY FSK
                          ) as rownum
FROM     Nullified
WHERE    rownum=1 or
         K1 <> K1_p or
         ...
         Kn <> Kn_p
```

Figure 7: A Basic β Query using OLAP functions

We now move to the second and more challenging question: how to construct a sort key that gives a favorable ordering of tuples. It's not obvious what constitutes a favorable sort key. For instance, Figure 8(a1) shows a sort key that gives a favorable ordering since the second tuple finds its dominating tuple right above it. For the same set of tuples, Figure 8(a2) shows that a different ordering is not favorable. Notice that the second tuple in Figure 8(a2) now moves to the third and is no longer dominated by the tuple immediately before it. A favorable ordering may not even exist. In Figure 8(b1) and Figure 8(b2), neither ordering on the same set of tuples is favorable. In both cases, the fourth tuple cannot find its dominating tuple above. Since the values in K_1 are all the same, a sort key corresponding to any other permutations of K_i gives the same ordering as in either Figure 8(b1) or Figure 8(b2). Nevertheless, as we will show later, it's possible to remove all spurious tuples by sorting the tuples twice.

$(K_1,$	$K_2,$	$K_3)$
(1,	1,	1)
(1,	1,	−)
(1,	2,	3)

(a1) sorted by $\langle K_1,K_2,K_3 \rangle$

$(K_1,$	$K_2,$	$K_3)$
(1,	1,	1)
(1,	2,	3)
(1,	1,	−)

(a2) sorted by $\langle K_1,K_3,K_2 \rangle$

$(K_1,$	$K_2,$	$K_3)$
(1,	1,	1)
(1,	1,	−)
(1,	2,	3)
(1,	−,	1)

(b1) sorted by $\langle K_1,K_2,K_3 \rangle$

$(K_1,$	$K_2,$	$K_3)$
(1,	1,	1)
(1,	−,	1)
(1,	2,	3)
(1,	1,	−)

(b2) sorted by $\langle K_1,K_3,K_2 \rangle$

Figure 8: Favorable and Unfavorable Ordering

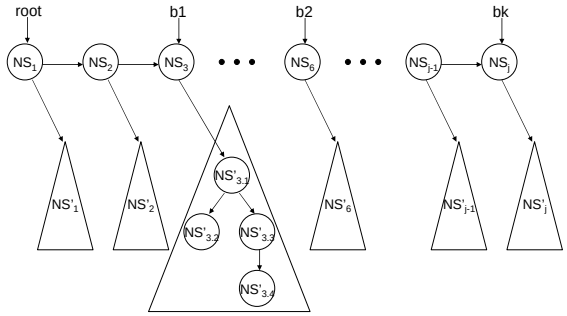

Figure 9: The Construction of a Favorable Ordering

To construct a favorable ordering, we analyze the DAG_{ns} of an outerjoin query. We first consider how to construct a favorable ordering under a common case where R_b satisfies an *inclusion* property: $NS_{R_{b1}} \subseteq NS_{R_{b2}} ... \subseteq NS_{R_{bk}}$, i.e., the nullification sets of R_b include one another. In DAG_{ns}, we can find a directed path $NS_1, NS_2, ..., NS_j$ (represented by the nullification set of the node) such that all $NS_{R_{bi}}$ are on the path, NS_1 correspond to the root node and $NS_j = NS_{R_{bk}}$. This is depicted in Figure 9. Note that the plans we generated in Section 5 never further nullify relations in the root node, which implies $NS_1 \neq NS_{b1}$. The relation set associated with each node NS_i $(i = 1, ..., j)$ is denoted as S_i. We then organize the remaining relations into j sets, where S'_i is defined as $S'_i = \{R \mid NS_i \subset NS_R$ and $NS_{i+1} \not\subseteq NS_R\}$ for $i = 1, ..., j$-1 and $S'_j = \{R \mid NS_j \subset NS_R$ and $R \notin S'_1, ..., S'_{j-1}\}$. Note that each S'_i essentially includes relations whose associated nodes are descendants of NS_i, but not descendants of NS_{i+1} in DAG_{NS} (inside each triangular in Figure 9). When a relation R is a descendant of more than one NS_i, our definition assigns R to the S_i with the largest i (although assigning R to any one of them would also be fine). We then construct a sort key FSK as $\langle K_{S_1}, K'_{S_1}, K_{S_2}, K'_{S_2}, ..., K_{S_j}, K'_{S_j} \rangle$, where K_{S_i} and K'_{S_i} include the KID of relations in S_i and S'_i, respectively. The relative order of $KIDs$ within each K_{S_i} and K'_{S_i} is not important. We claim that sorting on FSK produces a favorable ordering in Theorem 6.1.

THEOREM 6.1. If R_b satisfies the *inclusion* property, the sort key FSK constructed as above produces a favorable ordering on the nullified join result.

Proof: First, duplicated tuples are clustered together since the sort key includes all $KIDs$. Consider any dominated

(a) Q1

```
WITH    Nullified AS (
SELECT  R.k, T.k,
        CASE WHEN R.c=T.c then S.k end as S.k
FROM    (R LEFT JOIN S ON R.b=S.b) LEFT JOIN T
        ON S.a=T.a and R.c=T.c )
SELECT  R.k, S.k, T.k,
        max(R.k) OVER (ORDER BY R.k,S.k
                       ROWS ...) as Rk_p,
        max(S.k) OVER (ORDER BY R.k,S.k
                       ROWS ...) as Sk_p,
        rownumber() OVER (ORDER BY R.k,S.k)
                       as rownum

FROM    Nullified
WHERE   rownum=1 or
        R.k <> Rk_p or
        S.k <> Sk_p
```

(b) Q2

```
WITH    Nullified AS (
SELECT  R.k, S.k,
        CASE WHEN R.a=S.a then T.k end as T.k
FROM    (R LEFT JOIN T ON R.a=T.a) LEFT JOIN S
        ON R.a=S.a )
SELECT  R.k, S.k, T.k,
        max(R.k) OVER (ORDER BY R.k,S.k,T.k
                       ROWS ...) as Rk_p,
        max(S.k) OVER (ORDER BY R.k,S.k,T.k
                       ROWS ...) as Sk_p,
        max(T.k) OVER (ORDER BY R.k,S.k,T.k
                       ROWS ...) as Tk_p,
        rownumber() OVER (ORDER BY R.k,S.k,T.k)
                       as rownum
FROM    Nullified
WHERE   rownum=1 or
        R.k <> Rk_p or
        S.k <> Sk_p or
        T.k <> Tk_p
```

Figure 10: Q1 and Q2 rewritten using β Queries

tuple t. Let K_d be the leftmost KID in FSK that is dominated by another tuple t'. We can show that K_d must belong to a K_{S_i}. The reasoning is that the join result before nullification does not contain any dominated tuples, and the $KIDs$ in K'_{S_i} are not modified after nullification. Suppose that $K_d \in K_{S_i}$. From Observation 4.1(a), we conclude that KID in $K_{S_i}, K'_{S_i}, K_{S_{i+1}}, K'_{S_{i+1}}..., K_{S_j}, K'_{S_j}$ must all be *null*. Because tuples are sorted by FSK, the tuple immediately before t must at least match all $KIDs$ in FSK up to, but not including K_d (t' is an example of such a tuple). Since the rest of the attributes in t are all *null*, t must be dominated by its previous tuple. Therefore, such an ordering is favorable. ∎

When relations in R_b do not satisfy the inclusion property, in general a single favorable ordering of tuples cannot be found. For example, the tuples in Figure 8(b1) and 8(b2) can be the join result where R_2 and R_3 are further nullified and they are not on the same path in DAG_{NS}. However, we can decompose R_b to some smaller relation sets, each of which satisfies the inclusion property and can then be implemented by a basic β query. The problem of finding a minimum number of paths, which cover a subset of nodes in a directed cyclic graph is known to be reducible to a bipartite matching problem [6]. It follows that our problem can be solved in $O(m\sqrt{n})$ time, where n is the number of nodes and m is the number of pairs that are connected in DAG_{NS}. We do not further elaborate on a detailed solution here. To show an example, the β operation for tuples in Figure 8(b1) can be implemented by two β queries using $\langle K1, K2, K3\rangle$ and $\langle K1, K3, K2\rangle$ as the sort key in each. In practice, most queries only require a single basic β query.

We are now ready to complete the queries in Figures 1(e) and 1(f). Q1 and Q2 can be rewritten as in Figures 10(a) and 10(b), respectively. Since the DAG_{NS} (in Figure 4) of both queries contain a single path, we only need one basic β query. The sort key giving a favorable order is $\langle R.k, S.k\rangle$ for Q1 and $\langle R.k, S.k, T.k\rangle$ for Q2. Note that we performed a little optimization in Q1 by excluding $T.k$ from the sort key since T and S are assigned to the same node in DAG_{ns}.

Before closing this section, we'd like to discuss two performance issues in β queries. First of all, a basic β query requires one sort, and a β operation may need more than one basic β query, which means multiple sorts. Although sorting can be expensive, it pays off when a more beneficial join sequence can be selected. It's the responsibility of the optimizer to pick the better plan by comparing the estimated costs. Often, a good join sequence can improve performance by orders of magnitude, whereas the sorting overhead is much less. Also, our approach greatly simplifies the implementation of the complex β operation. Second, we note that it's possible to create a partitioned version of a basic β query by replacing the ORDER BY clause in Figure 7 with a "PARTITION BY K_{S_1}, K'_{S_1} ORDER BY $K_{S_2}, K'_{S_2}, ..., K_{S_j}, K'_{S_j}$" clause. Since $KIDs$ in K_{S_1}, K'_{S_1} are not further nullified, we never assign a dominated tuple to a different partition than that of its dominator. K_i_p then computes the K_i in a previous tuple within each partition. Without showing the details, we note that the WHERE clause also needs to be modified slightly in the partitioned version. This alternative has the benefit of being easily parallelizable and can therefore take advantage of the SMP or the MPP in modern commercial database systems.

7. EXPERIMENTAL RESULTS

In this section, we present our experimental results. The performance benefit of reordering join sequences has been well studied before. Therefore, the experiments in this section are intended to be illustrative, rather than comprehensive. We ran our tests on an IBM 44P server, which has four processors and 3GB of RAM. The operating systems on this machine is AIX 5.1 and the database server we used is `DB2` V8.1 Enterprise Server Edition FP6. We populated a 1GB TPC-H database and created indexes on the key of each relation. We set the bufferpool size to be a little bit over 1GB so that all data can reside in memory. The sort heap size, which controls the amount of memory allocated to each sort and hash-join operator, was set to 40MB.

We design two queries on the TPC-H schema having the

Q1. Find parts of a certain brand and size and show their corresponding line item and available quantity, if any.

```
select p_type, l_orderkey,l_linenumber, ps_availqty
from part left join
     ( lineitem inner join partsupp
       on l_partkey = ps_partkey and
          l_suppkey = ps_suppkey )
     on p_partkey = l_partkey and
        p_partkey = ps_partkey
where p_brand = 'Brand#35' and p_size in (5)
order by p_type, l_orderkey, ps_availqty
fetch first 100 rows only
```

Q2. Find certain parts and show their corresponding line item, if any, and available quantity, if any.

```
select p_partkey, p_type, l_orderkey, ps_availqty
from part_lt_1000 left join
     ( lineitem left join partsupp_gt_995
       on l_partkey = ps_partkey )
     on p_partkey = l_partkey
order by p_partkey, p_type, l_orderkey, ps_availqty
fetch first 100 rows only
```

Figure 11: Testing Queries

same flavor as the two in Figure 1. Therefore, we keep their names as Q1 and Q2. The SQL of the two queries are given in Figure 11. As we can see, relations `part`, `lineitem` and `partsupp` correspond to the relations R, S and T in Figure 1 respectively. In Q1, we use two predicates on relation `part` to reduce its size. Notice that both predicates can be pushed through the joins and evaluated as local predicates on relation `part` (which is indeed what DB2's optimizer does). In Q2, `part_lt_1000` and `partsupp_gt_995` are derived from `part` and `partsupp`, by limiting the `partkey` to less than 1,000 and greater than 995, respectively. For the sake of simplicity, we refer to them by their base relations. We deliberately construct this query to make the join between `part` and `partsupp` more attractive. We use a FETCH FIRST clause in both queries to retrieve only the first 100 tuples. However, since the final result needs to be ordered, all tuples in the join result still need to be computed.

For Q1, we compare two execution plans, one (`Q1`) corresponding to the original query and the other (`Q1_bm`) corresponding to a new query that joins `part` with `lineitem` first, but needs compensation in the end. For Q2, we compare three execution plans, `Q2` itself, `Q2_free` in which `part` joins `lineitem` first, followed by `partsupp` (this plan can be directly derived using the transformation rules and does not need further compensation), and `Q2_bm` where `part` is joined with `partsupp` first and a final β operator is needed. The SQL statements for `Q1_bm` and `Q2_bm` are constructed in a way similar to that in Figure 10. The favorable ordering is given by ⟨`p_partkey,l_orderkey,l_linenumber`⟩ for `Q1_bm` and ⟨`p_partkey,l_orderkey,l_linenumber,ps_partkey, ps_suppkey`⟩ for `Q2_bm`. We should pointer out that neither `Q1_bm` nor `Q2_bm` can be considered by existing techniques. We measure the elapsed time for the five plans a couple of times and report the average time of each in Figure 12. Compared with their closest competitor, the plans using our technique runs 15 times faster for Q1 and 3.5 times faster for Q2 (note the logarithmic scale on the y-axis).

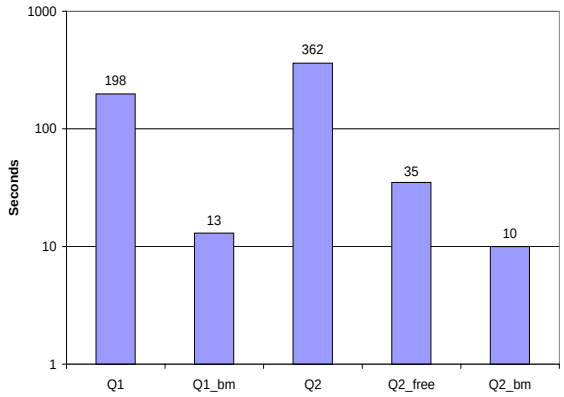

Figure 12: Execution Time of Each Plan

To understand why our technique wins, we'd like to analyze the execution plans in detail. We show the five plans generated by the optimizer in Figure 13. Each node in the plan tree represents an operator, whose type is given by the text inside the node. The number above each node shows the cardinality after the corresponding operator has been applied. Note that the cardinality from the right side (inner) of a nested loop join represents the per outer cardinality. Each operator is also given a unique number, displayed under its type. We omit the final sort operator since it's common in all plans. However, the sort operator needed for the β operator is shown in `Q1_bm` and `Q2_bm`.

In `Q1`, since `lineitem` has to be joined with `partsupp` first, it produces a large intermediate result after the hash join (node 2), which significantly increases the cost for the next hash join (node 1). In `Q1_bm`, since we join `lineitem` first with `part`, the intermediate result (after node 3) becomes much smaller, which makes the second join (node 2) cheaper. Although not shown, the cardinality after node 2 in `Q1_bm` is actually slightly larger than that after node 1 in `Q1`. This is because `Q1_bm` generates a smaller number of spurious tuples, which are subject to removal. The sort operator (node 1) in `Q1_bm` is used for generating the favorable ordering. However, since the number of tuples to be sorted is relatively small, sorting does not add too much additional cost to the plan (almost negligible in this example). The operator used to remove the spurious tuples is a standard filter and is not shown here.

We now move to the second query. Because the way we constructed Q2, `part` and `partsupp` each match many tuples in `lineitem`, but have only a few matches between themselves. Thus, both `Q2` and `Q2_free` have a large intermediate result after the first join, and suffer from a similar performance penalty as in Q1. Joining `part` and `partsupp` first (as shown in `Q2_bm`) is the best, since it reduces the intermediate result size the most, and also enables the second join to use a much cheaper nested loop join.

8. DISCUSSION

One of the assumptions in this paper is that all predicates are null-intolerant. When predicates can tolerate *nulls* (e.g., $R.a$ is null), the rippling of *null* values through predicates (as we have seen in Section 4) breaks, and the process of generating nullification sets in Table 1 is no longer valid. The relative positions of λ operations become important and

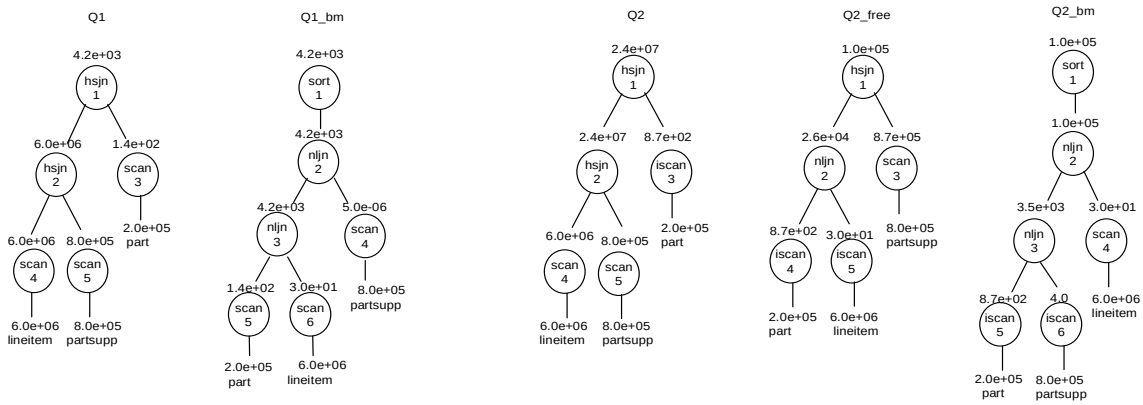

Figure 13: Plan for Q1, Q1_bm, Q2, Q2_free, and Q2_bm

cannot be changed freely.

One solution is to break the original operation tree into multiple smaller trees (blocks), each of which only has null-tolerant predicates at its root. For each smaller tree, we then compute the nullification sets and optimize it separately. During planning, we add one more rule that any λ operation using a null-tolerant conjunct has to be applied last in the λ sequence. This way, we only fix the ordering of null-tolerant conjuncts and still give freedom to null-intolerant ones.

An antijoin [13] is useful for handling negated nested queries. An antijoin returns a tuple t in the outer relation (preserving side) if t cannot find any matching tuples in the inner relation (null-producing side). We use $\triangleright$ to denote an antijoin (the arrow pointing to the null-producing side). Antijoins and full outerjoins can be transformed as follows:

$$R \overset{Prs}{\triangleright} S = \sigma_{S.k\ is\ null}(R \overset{Prs}{\longrightarrow} S)$$
$$R \overset{Prs}{\longleftrightarrow} S = R \overset{Prs}{\longrightarrow} S \cup R \overset{Prs}{\triangleleft} S$$

Since after the transformation there are only left outerjoins, our technique can then by applied. Notice that the predicate "*S.k is null*" is null-tolerant, which limits the reordering of some λ operations.

9. CONCLUSION

In this paper, we propose a novel canonical abstraction for outerjoin queries. Under such an abstraction, an outerjoin query can be decomposed into a sequence of outer Cartesian products, followed by nullification operations and a final best match operation. Our abstraction resembles that of the inner join by allowing all join sequences as well as enabling commutativity and transitivity among predicates. As a result, more powerful optimization can be applied on outerjoin queries to achieve much better performance.

For a given outerjoin query, we provide a method of producing its outerjoin canonical abstraction and extend a conventional optimizer to generate plans in an expanded search space by taking advantage of the abstraction. We also describe an efficient implementation of the best match operation, using the standard OLAP amendment in SQL. Our experimental results on a commercial database system demonstrate the performance advantage of our technique.

10. REFERENCES

[1] XQUERY 1.0. http://www.w3.org/tr/xquery/. 2003.

[2] E. F. Codd. Extending the relational database model to capture more meaning. *Transactions on Database Systems*, 4(4):397–434, 1979.

[3] IBM DB2 universal database version 8.1. 2002.

[4] Oracle Corporation. Oracle 9i. 2002.

[5] Umeshwar Dayal. Processing queries with quantifiers. In *ACM PODS Conference*, 1983.

[6] L.R. Ford, et al. Flows in Networks. Princeton Univ. Press, 1963.

[7] Cesar A. Galindo-Legaria. *Algebraic optimization of outerjoin queries*. PhD thesis, Department of Applied Science, Harvard University, 1992.

[8] Cesar A. Galindo-Legaria. Outerjoins as disjunctions. In *ACM SIGMOD Conference*, pages 348–358, 1994.

[9] Cesar A. Galindo-Legaria, et al. How to extend a conventional optimizer to handle one- and two-sided outerjoin. In *ICDE Conference*, 1992.

[10] Cesar A. Galindo-Legaria, et al. Outerjoin simplification and reordering for query optimization. *Transactions on Database Systems*, 22(1), 1997.

[11] G. Graefe, et al. The volcano optimizer generator: Extensibility and efficient search. In *ICDE*, 1993.

[12] Mauricio A. Hernández, et al. Clio: A semi-automatic tool for schema mapping. In *SIGMOD*, 2001.

[13] Won Kim. On optimizing an SQL-like nested query. *ACM TODS*, 7(3):443–469, 1982.

[14] J. Melton. *Advanced SQL:1999-Understanding Object-Relational and Other Advanced Features*. Morgan Kaufman, 2002.

[15] Jun Rao, et al. Using eels, a practical approach to outerjoin and antijoin reordering. In *Proceedings of the IEEE ICDE Conference*, 2001.

[16] Arnon Rosenthal and Cesar A. Galindo-Legaria. Query graphs, implementing trees, and freely-reorderable outerjoins. In *Proceedings of the ACM SIGMOD Conference*, pages 291–299, 1990.

[17] P. G. Selinger, et al. Access path election in a relational database management system. In *ACM SIGMOD*, 1979.

[18] Jeffrey D. Ullman. *Principles of Database and Knowledge-base Systems, volume II*. Computer Science Press, 1989.