Implementation of Two Semantic Query Optimization Techniques in DB2 Universal Database

Qi Cheng[1] Jarek Gryz[2] Fred Koo[1] Cliff Leung[3] Linqi Liu[2,4]
Xiaoyan Qian[1] Bernhard Schiefer[1]

IBM[1] Toronto
Department of Computer Science[2] York University, Toronto
IBM[3] Almaden
Daedalian Systems Group Inc.[4] Toronto

Abstract

In the early 1980's, researchers recognized that semantic information stored in databases as integrity constraints could be used for query optimization. A new set of techniques called *semantic query optimization* (SQO) was developed. Some of the ideas developed for SQO have been used commercially, but to the best of our knowledge, no extensive implementations of SQO exist today.

In this paper, we describe an implementation of two SQO techniques, Predicate Introduction and Join Elimination, in DB2 Universal Database. We present the implemented algorithms and performance results using the TPCD and APB-1 OLAP benchmarks. Our experiments show that SQO can lead to dramatic query performance improvements. A crucial aspect of our implementation of SQO is the fact that it does not rely on complex integrity constraints (as many previous SQO techniques did); we use only referential integrity constraints and check constraints.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the 25th VLDB Conference, Edinburgh, Scotland, 1999.

1 Introduction

Relational database systems became the predominant technology for storing, handling, and querying data only after a great improvement in the efficiency of query evaluation in such systems. The key factor in this improvement was the introduction and development of query optimization techniques. The traditional types of optimization, however, exploit to a limited extent the semantic information about the stored data. In the late 1970's and early 1980's, researchers [1, 7, 8, 9, 17] recognized that such information could be used for further query optimization, and developed a new set of techniques called *semantic query optimization* (SQO). SQO uses the integrity constraints associated with the database to improve the efficiency of query evaluation. The techniques most often discussed in literature included the following:

1. Join Elimination: A query may contain a join for which the result is known a priori, hence, it need not be evaluated. (For example, for some queries involving a join between two tables related through a referential integrity constraint).
2. Join Introduction: It may be advantegeous to add a join with an additional relation, if that relation is relatively small compared to the original relations as well as highly selective. (This is even more appealing if the join attributes are indexed).
3. Predicate Elimination: If a predicate is known to be always true it can be eliminated from the query.
4. Predicate Introduction: A new predicate on an indexed attribute may allow for a more efficient access method. Similarly, a new predicate on a join attribute may reduce the cost of the join.
5. Detecting the Empty Answer Set: If the query predicates are inconsistent with integrity constraints, the query does not have an answer.

Example 1. Consider the following two queries (both asked against the TPCD [19]). The first query illustrates the technique of Join Elimination.

$\mathcal{Q}_1$: **select** p_name, p_retailprice, s_name, s_address
from tpcd.lineitem, tpcd.partsupp, tpcd.part, tpcd.supplier
where p_partkey = ps_partkey and s_suppkey = ps_suppkey and ps_partkey = l_partkey and ps_suppkey = l_suppkey and l_shipdate between '1994-01-01' and '1996-06-30' and l_discount ≥ 0.1
group by p_name, p_retailprice, s_name, s_address
order by p_name, s_name;

The following referential integrity constraints (parent-child) have been defined in TPCD: **part-partsupp** (on partkey), **supplier-partsupp** (on suppkey), **partsupp-lineitem** (on partkey and suppkey). Given the referential integrity constraints, the intermediate join with **partsupp** can be eliminated from the query, since the tables **part** and **lineitem** can be joined directly. We show in Section 3 that this transformation improves query performance.

The next query illustrates the technique of Predicate Introduction.

$\mathcal{P}_1$: **select** sum(l_extendedprice * l_discount) as revenue
from tpcd.lineitem
where l_shipdate $\geq$ date('1994-01-01') and l_shipdate $<$ date('1994-01-01')+1year and l_discount between $.06 - 0.01$ and $.06 + 0.01$ and l_quantity < 24;

Since the following check constraint, **l_shipdate** $\leq$ **l_receiptdate**, has been defined for TPCD, a new predicate, **l_receiptdate** $\geq$ **date('1994-01-01')**, can be added to the **where** clause in the query without changing its answer set. Now, if the only index on **lineitem** table is a clustered index in which *l_receiptdate* is a major column, a new, potentially more efficient evaluation plan is available for the query. Indeed, we show in Section 4, that the use of this new plan leads to an improved query performance.

The SQO techniques discussed in literature were designed to be a part of a two-stage optimizer. In the first step, queries that are logically equivalent - with respect to the semantics of the database (that is, the stored set of integrity constraints) - to the original query, are generated. In the second step, these queries are submitted to the traditional optimizer which generates access plans for all of them and the query with the lowest estimated evaluation cost is chosen and submitted for evaluation. Since the number of equivalent queries generated in the first phase could be large (in general, exponential in the number of integrity constraints that relate semantically to the query), heuristics were necessary to limit their number [1, 7, 16]. Although several different techniques for SQO have been developed, only simple prototypes have been built. To the best of our knowledge, no extensive, commercial implementations of SQO exist today.[1] There are several reasons why SQO has never caught up in the commercial world. The most prominent one is the fact that SQO was in many cases designed for deductive databases [1, 7, 8] and because of this association, SQO might not appear useful for relational database technology. Second, at the time when SQO techniques were being developed, the relative CPU and I/O speeds were not as dramatically different as they are now. The savings in query execution time (dominated by I/O) that SQO could provide was not worth the extra CPU time necessary to optimize a query semantically. (The analysis presented in [16] shows that the cost of semantic optimization could become comparable to the query execution cost.) Last, it has been usually assumed that many integrity constraints have to be defined for a given database if SQO is to be useful there. Otherwise, only few queries could be optimized semantically. However, this is not the case in most real life databases; except for keys, foreign keys, and check constraints, very few integrity constraints are ever defined. Indeed, many of the integrity constraints considered in early days of SQO are not expressible in most commercial database systems, even today !

We have always believed, however, that many of the SQO techniques could provide an effective enhancement to the traditional query optimization. We show in this paper that this is indeed the case. We developed versions of two SQO techniques: Join Elimination (JE) and Predicate Introduction (PI). Significant portions of these technologies have been implemented in a prototype version of IBM DB2 Universal Database (UDB). We hope that this work will provide valuable lessons for future implementations of SQO.

The paper is organized as follows. Section 2 provides a brief overview of the DB2 UDB optimizer and the general assumptions for the implementation. The algorithms and performance results for Join Elimination and Predicate Introduction constitute Sections 3 and 4 respectively. The paper concludes in Section 5.

[1] This is not to say that *no* semantic information is used for optimization (e.g. the information about keys is routinely used to remove the DISTINCT operator from queries). However, the extent to which semantic information is used in that way is far from what the designers of SQO envisioned.

2 Overview of the Implementation

In traditional database systems, query optimization typically consists of a single phase of processing in which an efficient access plan is chosen for executing a query. In Starburst DBMS [6], on which DB2 UDB is based, the query optimization phase is divided up into *query rewrite optimization* and *query plan optimization* phases; each concentrates on different aspects of optimization. The query rewrite phase transforms queries to other semantically equivalent queries. This is also commonly known as the query modification phase, applying rewrite heuristics. The query plan optimization phase determines the join order, join methods, join site in a distributed database, and the method for accessing each input table.

After an input query is parsed and converted to an intermediate form called *query graph model* (QGM), the graph is transformed by the Query Rewrite Engine [13, 14] into a logically equivalent but more efficient form using heuristics. Query rewrite is a rule based system. Such a system permits keeping the range of optimization methods open-ended, considerably reducing the effort it usually takes to extend the optimization range as user needs evolve. For example, it is important to be able to add a new query rewrite rule into the system without having to modify the existing rules or to have an explanation of how the rule system arrives at the solution for a given query. The growing list of rewrite rules implemented in this system includes predicate pushdown, subquery to join transformation, magic sets transformation, handling of duplicates, merging of views and decorrelating complex subqueries [2, 10, 11, 12, 13]. Rules can be grouped into rule classes for higher efficiency, better understandability and more extensibility. Such grouping of query rewrite rules can help the query rewrite system to converge to a fixpoint faster. Furthermore, each rule class uses a particular control strategy that specifies how rules in the class are selected to fire. An important aspect of the implemented rule based system is its efficiency: experimental results show [14] that less than 1% of the query execution time is spent on the query rewrite phase.

Some of the optimization techniques already present in DB2 UDB optimizer use information about integrity constraints to transform queries, hence implement a form of "semantic" query optimization. The following examples illustrate two such techniques.

Example 2. This example illustrates a simple rule that allows eliminating the DISTINCT keyword. Let $\mathcal{Q}$ be a query in TPCD schema:

$\mathcal{Q}$: **select** DISTINCT nationkey, name
from tpcd.nation;

Since nationkey is a key for the relation nation, the DISTINCT keyword can be eliminated, thus avoiding a potentially expensive sorting.

$\mathcal{Q}'$: **select** nationkey, name
from tpcd.nation;

Example 3. This example illustrates the use of functional dependencies to optimize the order operation [18].

$\mathcal{Q}$: **select** shipdate, commitdate
from tpcd.lineitem
order by shipdate, commitdate;

Assume that there is a functional dependency **shipdate** $\rightarrow$ **commitdate**.[2] Thus, for a given value of **shipdate**, there is only one value of **commitdate**. Hence, $\mathcal{Q}$ can be rewritten into the following query:

$\mathcal{Q}'$: **select** shipdate, commitdate
from tpcd.lineitem
order by shipdate;

Again, after this transformation a potentially expensive sorting operation is avoided.

The design of the query rewrite engine is ideal for the implementation of SQO. Each of the SQO techniques represents a transformation of a query that can be stated as a condition-action rule. This is exactly how transformation rules are implemented in the query rewrite engine. The only restriction that was forced on us as a result of the design of the DB2 optimizer was as follows. Since only a single query can be passed from the query rewrite engine to the plan optimization phase, we could not assume that the plan optimizer would be able to choose the best query from a set of *several* semantically equivalent queries. The disadvantage of this is that the *single* query generated through SQO had to be guaranteed to be better than the original query (this assumption need not be made when there were several candidate queries, since the original query was among them). The advantage of this approach is, however, that less time had to be spent on SQO, so we would not encounter the problem of spending more time on optimization than query execution [16].

The decision to implement JE and PI, out of the choices of SQO techniques, was based on two factors. Our initial experiments with SQO [5], in which we tested all known SQO techniques by rewriting queries by hand, indicated that both JE and PI provided consistent optimization. In addition to this, however, they

[2] This FD is only assumed for the sake of the example; it does not hold in TPCD.

were also the most practical to implement. The transformations they provided relied only on check constraints and referential integrity constraints. Thus, we did not need to change the support mechanism for integrity constraints in DB2 UDB. Moreover, since almost all database systems support these types of integrity constraints, JE and PI can be potentially implemented in other DBMSs.

3 Join Elimination

3.1 Implementation

The Join Elimination (JE) technique discussed in SQO literature was often presented in two different versions [1]. In the first version, the join under consideration is known to be empty (by reasoning over the set of integrity constraints), hence any further join with its result would also be empty. In the second version, if it could be proved by reasoning over the set of integrity constraints that the join is redundant (as in query $\mathcal{Q}_1$ of Example 1) it can be eliminated from the query. Although, JE in its first version is likely to provide very good optimization, we did not think that it was very practical. First, it is cumbersome to express in SQL as an integrity constraint the fact that the join of two tables (possibly with selects) is empty. Second, it is unlikely that such integrity constraints would be stored, since their verification would be costly.

Thus, we concentrate only on the second version of JE and consider the case where redundant joins are discovered through reasoning over referential integrity (RI) constraints. The most straightforward application of our technique is the elimination of a relation (hence a join) where the join is over the tables related through an RI constraint (we refer to such joins as *RI joins*) and the primary key table is referenced only in the join.

Example 4. Assume that the view **Supplier_Info** has been defined over TPCD schema and the query $\mathcal{Q}$ has been asked against that view:

create view Supplier_Info (n, a, c) as
select s_name, s_address, n_name
from tpcd.supplier, tpcd.nation
where s_nationkey = n_nationkey;

$\mathcal{Q}$: **select** s_n, s_a
from Supplier_Info;

Since there is an RI constraint between **supplier** and **nation** on *nationkey*, then every tuple from the **supplier** relation necessarily joins with some tuple in the **nation** relation. Also, no attributes are selected or projected from the **nation** relation. Hence, the query can be rewritten into an equivalent form as $\mathcal{Q}'$:

$\mathcal{Q}'$: **select** s_n, s_a
from tpcd.supplier;

$\mathcal{Q}'$ avoids the join computation, so its evaluation can be more efficient.

Note that even if the user knows that the query $\mathcal{Q}$ can be rewritten as $\mathcal{Q}'$, he may not be able to do so, since he may only have access to the view. Thus, even such simple optimizations have to be performed within the DBMS. Redundancy in RI joins is likely to occur in environments where views are defined with large number of such joins, for example, in data warehousing with a star schema. But it can also appear in ordinary queries if they are not written by a programmer, but are automatically generated, e.g. by GUIs in query managers where hand optimization is impossible.

The JE algorithm we implemented handles not only the removal of explicit RI joins, but also redundant joins that can be inferred through reasoning over the joins explicitly stated in the query and the RIs defined for the database (query $\mathcal{Q}_1$ of Example 1 was transformed in this way). The algorithm has the following steps:

1. Column equivalence classes are built via transitivity from all join predicates in the query. That is, if $A = B$ and $B = C$ are in the query, we can infer $A = C$. All three columns, A, B, C are then in a single equivalence class.
2. All tables in the **from** clause of the query that are related through RI joins are divided into two groups: R group (removable tables) and N group (non-removable tables). The necessary condition for a table to belong to the R group is that it is a parent table for some RI (such tables need to satisfy other conditions as well, their full description, however, is beyond the scope of this paper).
3. All tables in the R group (that is, redundant joins) are eliminated from the query.
4. Since foreign key columns may be nullable, an 'IS NOT NULL' predicate is added to foreign key columns of all tables whose RI parents have been removed and which, in fact, permit nulls.

We present a few examples of the types of transformations that can be performed by the algorithm.

Example 5. For each query specified below we present an optimized query generated by the algorithm and three graphs describing the join structures. The nodes in each graph represent attributes and the edges represent joins between these attributes. The top graph shows the joins explicitly stated in the query. The middle graph shows all the joins that can be inferred from the first graph (that is, all equivalence classes induced by the explicit joins). The RI joins are distinguished from other joins by arrows (from the child to the parent, that is, N:1). The bottom graph shows the structure of the joins after the redundant joins have been eliminated by the algorithm.

• **Query 1**

Let the query be $\mathcal{Q}_1$ of Example 1. The graphs describing the structure of the joins of the query are shown in Figure 1.

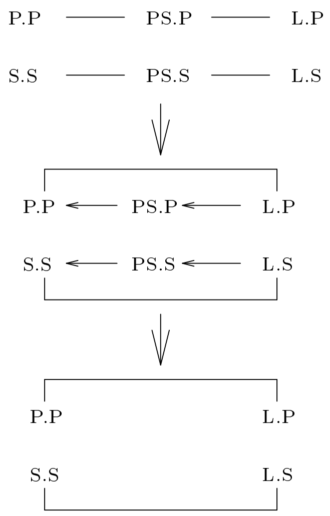

Figure 1: Join graphs for query $\mathcal{Q}_1$.

Thus, $\mathcal{Q}_1$ can be optimized into $\mathcal{Q}_1^{'}$.

$\mathcal{Q}_1^{'}$: **select** p_name, p_retailprice, s_name, s_address
from tpcd.lineitem, tpcd.part, tpcd.supplier
where p_partkey = l_partkey and
s_suppkey = l_suppkey and
l_shipdate between '1994-01-01' and
'1996-06-30' and l_discount ≥ 0.1
group by p_name, p_retailprice, s_name,
s_address
order by p_name, s_name;

We tested $\mathcal{Q}_1$ and $\mathcal{Q}_1^{'}$ in TPCD database of size 100MB in the same environment as described in Section 4. Execution time for the original query $\mathcal{Q}_1$ was 58.5s and for the optimized query $\mathcal{Q}_1^{'}$ 38.25s (a saving of 35%). Since the database was relatively small, the query was CPU bound. We expect that the optimization would be even more prominent for a large database since the I/O cost was reduced by 67% (from 4631 to 1498 page reads).

• **Query 2**

Consider another query in TPCD.

$\mathcal{Q}_2$: **select** ps_partkey as partkey,
avg(ps_supplycost) as supplycost
from tpcd.supplier, tpcd.partsupp,
tpcd.customer, tpcd.orders
where s_suppkey = ps_suppkey and
s_suppkey = c_custkey
and c_custkey = o_custkey and
o_totalprice ≥ 100
group by ps_partkey
order by 2
fetch first 200 rows only;

The join graph for $\mathcal{Q}_2$ is shown in Figure 2 (since *s_suppkey = ps_suppkey* and *c_custkey = o_custkey* are RI joins, they are marked with arrows in the middle graph). The query can be simplified to a single join as indicated in the bottom graph.

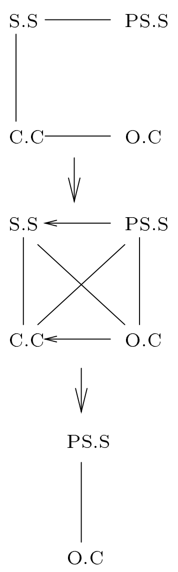

Figure 2: Join graphs for query $\mathcal{Q}_2$.

Thus, the query becomes:

$\mathcal{Q}_2^{'}$: **select** ps_partkey as partkey,
avg(ps_supplycost) as supplycost
from tpcd.partsupp, tpcd.orders
where ps_suppkey = o_custkey and
o_totalprice ≥ 100
group by ps_partkey
order by 2
fetch first 200 rows only;

We tested $\mathcal{Q}_2$ and $\mathcal{Q}_2^{'}$ in the same environment as $\mathcal{Q}_1$. The execution time for $\mathcal{Q}_2$ was $6331.64s$ and for $\mathcal{Q}_2^{'}$, $79.33s$, for a saving of 99%.

• **Query 3**

Consider the following view defined over a star schema (for brevity, we use "..." to indicate sequences of attributes). Cube1Fact is a fact table, each of the Cube1Dim2–Cube1Dim6 is a dimension table, and each of the joins is an RI join (the schema is described in more detail in Section 3.2):

create view olapmain_starview
("Measure","Scenario","Channel_label",
"Customer","Product", "1996", "1995Q3",
"1995Q4","1996Q1","1996Q2","1996Q3",
"1996Q4", "199606YTD", "199501",...,
"199512", "199601",...,"199612") AS
select T2.Membername, T3.Membername,
T4.Membername, T5.Membername,
T6.Membername, F.AN1,...,F.AN32
from Cube1Fact F, Cube1Dim2 T2,
Cube1Dim3 T3, Cube1Dim4 T4,
Cube1Dim5 T5, Cube1Dim6 T6
where T2.RelmemberId = F.COL2 AND
T3.RelmemberId = F.COL3 AND
T4.RelmemberId = F.COL4 AND
T5.RelmemberId = F.COL5 AND
T6.RelmemberId = F.COL6;

Let $\mathcal{Q}_3$ be a query that uses this view. After the view is replaced with its definition, $\mathcal{Q}_3$ can be simplified into $\mathcal{Q}_3^{'}$.

$\mathcal{Q}_3$: **select** "1996", "199606YTD"
from olapmain_starview;

$\mathcal{Q}_3^{'}$: **select** "1996", "199606YTD"
from Cube1Fact
where F.COL2 IS NOT NULL and
F.COL3 IS NOT NULL and
F.COL4 IS NOT NULL and
F.COL5 IS NOT NULL and
F.COL6 IS NOT NULL;

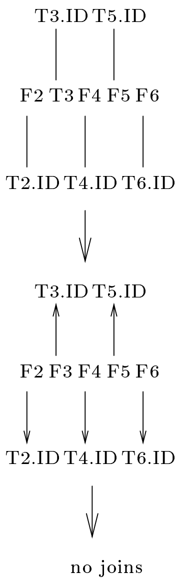

Figure 3: Join graphs for query $\mathcal{Q}_3$.

3.2 Performance Results

The JE technique described above has been implemented in a prototype DB2 UDB installed as part of an OLAP Server. The DB2 OLAP Server is an Online Analytical Processing server that can be used to create a wide range of multidimensional planning, analysis, and reporting applications. DB2 OLAP Server uses the Essbase OLAP engine developed by Hyperion Solutions Corporation. However, DB2 OLAP Server replaces the integrated, multidimensional data storage used by Arbor with a relational storage management, and stores data in the relational data storage using a star schema.

We used the APB-1 OLAP Benchmark [3] schema for the experiments. The benchmark simulates a realistic on-line analytical processing business situation that exercises server-based software. The logical benchmark database structure is made up of six dimensions: time, scenario, measure, and three aggregation dimensions that define the database size (product, customer, and channel).

When an APB-1 OLAP Benchmark database is created using DB2 OLAP Server, it generates a set of relational tables represented as a star schema. The dimension tables (1-6) are: *Time, Measure, Scenario, Channel, Customer, and Product*. One of the dimensions, in our case *Time*, is chosen as the so-called anchor dimension and is joined with the fact table (hence we refer to the result of that join as the fact table). The attributes of the fact table indicate the sales in a given period of time: *199502* stores the sales value for February of 95, *1996Q1* for the first quarter of 96, etc.

In addition, DB2 OLAP Server creates and manages a number of views that simplify SQL application access to the multidimensional data. One of them, *olapmain_starview* (defined in Query 3 of Example 5), is particularly interesting from our point of view since it joins the fact table with five dimension tables (2–6). All of the joins in the view are RI joins. The sizes of the dimension tables (1–6) are respectively: 86, 15, 16, 12, 1001, 10001 rows. The fact table has 2.4 million rows.

We ran the experiments on DB2 OLAP Server in-

Queries		$\mathcal{J}_1$	$\mathcal{J}_2$	$\mathcal{J}_3$	$\mathcal{J}_4$	$\mathcal{J}_5$	$\mathcal{J}_6$	$\mathcal{J}_7$	$\mathcal{J}_8$	$\mathcal{J}_9$	$\mathcal{J}_{10}$
Execution Time (s)	Original	98.2	576.2	11.3	12.5	504.3	586.4	523.5	5.6	5.2	4.7
	Optimized	9.7	23.9	10.9	11.4	167.2	231.0	268.5	4.9	5.1	4.3

Table 1: Performance Results for Join Elimination

stalled in Windows NT Server 4.0 on a 4-way Pentium II Xeon 450 with 4GB of memory. Each of the queries below was run at least 5 times; execution times are averages over all runs. We designed the queries to involve other operations than just the joins in the view to make them more realistic. Queries $\mathcal{J}_1$ and $\mathcal{J}_2$ allowed elimination of all five dimension tables, queries $\mathcal{J}_3$ and $\mathcal{J}_4$ allowed elimination of four dimension tables, queries $\mathcal{J}_5$ and $\mathcal{J}_6$ allowed elimination of three dimension tables, queries $\mathcal{J}_7$ and $\mathcal{J}_8$ allowed elimination of two dimension tables, and queries $\mathcal{J}_9$ and $\mathcal{J}_{10}$ allowed elimination of one dimension table. We also tested queries from which no joins could be eliminated; no deterioration of performance due to extra optimization step was observed.

$\mathcal{J}_1$:
```
select   count("1996Q1")
from     olapmain_starview;
```

$\mathcal{J}_2$:
```
select   sum("199501") as "199501",
         sum("199502") as "199502",
         sum("199503") as "199503",
         sum("199601") as "199601",
         sum("199602") as "199602",
         sum("199603") as "199603"
from     olapmain_starview;
```

$\mathcal{J}_3$:
```
select    "199606", "199605"
from      olapmain_starview
where     channel_label = 'EQ086DVOCPQS'
          and "199606"< "199605"
order by  "199606" , "199605";
```

$\mathcal{J}_4$:
```
select   count("199604") as "199604"
         count("199605") as "199605",
         count("199606") as "199606"
from     olapmain_starview
where    measure = 'Dollar Sales'
         and "199604" > 5000.0
         and "199605" > 5000.0
         and "199606" > 5000.0;
```

$\mathcal{J}_5$:
```
select   sum("1995Q4") as "1995Q4"
from     olapmain_starview
where    scenario = 'Actual'
         and measure = 'Inventory Units';
```

$\mathcal{J}_6$:
```
select    measure, sum("199606") as total
from      olapmain_starview
where     scenario = 'Actual'
group by  measure
order by  measure;
```

$\mathcal{J}_7$:
```
select    channel_label,
          sum("199606") as "199606",
          sum("199506") as "199506"
from      olapmain_starview
where     scenario = 'Actual'
          and measure = 'Dollar Sales'
group by  channel_label
order by  "199606", "199506";
```

$\mathcal{J}_8$:
```
select    scenario,
          sum("1996Q2") - sum("1996Q1")
          as change
from      olapmain_starview
where     measure = 'Dollar Sales'
          and customer = 'VB8NRNCDLNPT'
group by  scenario
order by  change;
```

$\mathcal{J}_9$:
```
select   sum("1995Q4") as "1995Q4",
         sum("1996Q1") as "1996Q1",
         sum("1996Q2") as $
from     olapmain_starview
where    scenario = 'Actual'
         and channel_label = 'VDWRDK3K574X'
         and customer = 'Y8AJBH5KL5LE'
         and measure = 'Dollar Sales';
```

$\mathcal{J}_{10}$:
```
select    customer,
          avg("1996Q1") as "1996Q1"
from      olapmain_starview
where     scenario = 'Actual'
          and measure = 'Dollar Sales'
          and customer = 'VB8NRNCDLNPT'
group by  customer
order by  "1996Q1";
```

The amount of savings provided by the optimization in each of the tested queries is a function of many factors (the selectivity of the predicates, relative cost of the eliminated join versus other operations in a query, the size of the output, etc.). Hence, the optimization ranges from 2% (query $\mathcal{J}_9$) to 96% (query $\mathcal{J}_2$). Nevertheless, for all experiments JE provided consistent optimization for tested queries and for many of them the optimization was substantial. Another point we need to stress here is that for most of the queries the saving in execution time came from reducing the CPU cost, not the I/O (because the dimension tables were small and fit in memory). We expect that we can achieve even more prominent optimization for strictly I/O bound queries.

4 Predicate Introduction

4.1 Implementation

The idea of Predicate Introduction (PI) was discussed in literature as two different techniques: index introduction and scan reduction. The idea behind the index introduction is to add a new predicate to a query if there is an index on the attribute named in the predicate [1, 7, 9, 17] (query $\mathcal{P}_1$ of Example 1 illustrates this transformation). The assumption here is that retrieval of tuples using an index is more efficient than their sequential processing. In general, this assumption is not always true. Thus, the general rule was either accompanied by heuristics [9] or several queries were generated which were then subject to the cost evaluation by the plan optimizer [1]. As we stated in Section 2, we did not have a ready option in our design to get feedback from the plan generator in query rewrite engine. Hence, we needed to restrict the use of PI to situations that would guarantee improvement in the efficiency of query evaluation. Surprisingly, the range of such restrictions had to be more stringent than we expected.

The second use of PI is for table scan reduction. The idea here is to add a predicate to reduce the number of tuples that qualify for a join. As we show later in this section, this provides substantial optimization for all types of join. The problem with scan reduction is, however, that it is not common: it is unlikely that there will be any check constraints or predicates with inequalities about join columns. The example below illustrates scan reduction.

Example 6. $\mathcal{S}_1$ is a query asked against the TPCD database. Given the fact that the check constraint **l_receiptdate** $\geq$ **l_shipdate** holds in TPCD, two new predicates *l2.l_receiptdate > date ('1998-07-01')* and *l1.l_commitdate > date ('1998-07-01')* can be added to the query, thus reducing the number of tuples of **l1** and **l2** that qualify for a join.

$\mathcal{S}_1$: **select** sum(l1.l_extendedprice * l1.discount)
  as revenue
from tpcd.lineitem as l1, tpcd.lineitem as l2
where l1.l_commitdate = l2.l_receiptdate and
  l2.l_shipdate > date('1998-07-01') and
  l2.l_discount between $0.06 - 0.01$ and
  $0.06 + 0.01$ and l2.l_quantity $<$ 24;

Our implementation of PI included also another SQO technique discussed in the literature: detecting an empty query answer set. The idea behind this technique is to check whether the set of logical consequences of the set of query predicates and check constraints is consistent. Consider the following example.

Example 7. Let $\mathcal{E}$ be a query asked against the TPCD database:

$\mathcal{E}$: **select** sum(l_extendedprice * l_discount)
  as revenue
from tpcd.lineitem
where l_shipdate > date('1994-01-01') and
  l_receiptdate < date('1994-01-01')

The query itself does not contain a contradiction. However, given the fact that the check constraint **l_shipdate** $\leq$ **l_receiptdate** holds in TPCD, the query cannot return any answers. Thus, it does not need to be evaluated.

We should note that DB2 UDB already implements a version of predicate introduction [4]. The exisiting theorem prover generates a transitive closure over all equality predicates. The result of this operation is used to provide the optimizer with a choice of additional joins that may be considered to select the best access plan for the query. In addition, query rewrite will also derive additional local predicates based on transitivity implied by equality predicates. This may lead to scan reduction. The current system does not, however, take into account check constraints.

The PI algorithm we implemented has two main components: a theorem prover and a filter. The input to the theorem prover is the set of all check constraints defined for a database and the set of all predicates in a query. Its output is the set of all non-redundant formulas derivable from the input set. These formulas represent new predicates that can be added to the query; they are of the form A *op* B, where A and B are attribute names or constants and *op* is one of the following: $>$, $<$, $\geq$, $\leq$.[3] Clearly, only a few of the new predicates are useful for optimization. The role of the filter is to find them. Our goal in designing heuristics rules for the filter was to *guarantee* that any new predicate added to the query will allow the optimizer to find only better access plans than the ones available for the original query. The sketch of the algorithm is presented below.

Let $\mathcal{N}$ be the set of new predicates computed by the theorem prover.

1. If $\mathcal{N}$ is inconsistent (as in Example 7) an appro-

[3]This set does not include '=' since DB2 UDB already handles reasoning over predicates with equality.

Queries		$\mathcal{P}_1$	$\mathcal{P}_2$	$\mathcal{P}_3$	$\mathcal{P}_4$	$\mathcal{P}_5$
Estimated Cost (internal units)	Original	54031	113316	108676	128342	195098
	Optimized	18308	11222	69623	35974	133665
Execution Time (s)	Original	13.5	24.9	25.1	46.4	56.5
	Optimized	5.4	4.9	58.3	38.6	98.3

Table 2: Performance Results for Index Introduction

priate flag is raised and the query is not evaluated.

2. Else, for each predicate $A\ op\ B \in \mathcal{N}$, it is added to the query if one of the following holds

   a. A or B is a join column

   b.
   - A is a constant, and
   - B is a major column of an index, and
   - no other index on B's table can be used in the plan for the original query

Case 1 of the algorithm handles the discovery of inconsistency in the query. Case 2.a allows for scan reduction. The most important part of the algorithm, Case 2.b introduces an index to the query. Our approach is conservative: we introduce a new predicate to the query if this allows the plan optimizer to use an index (not available before) and this new plan does not preempt the use of another, potentially better plan. Thus, we insist that there is no other index available for the optimizer on the table with the new predicate. Our experiments show that without appropriate enhancements in the query optimizer, even this policy may not be sufficiently restrictive.

4.2 Performance Results

The PI technique described above has been implemented as a prototype in DB2 UDB installed in AIX on machine JRS/6000 Model 590H with 512MB of memory. We used 100MB TPCD Benchmark [19] database for our experiments. The queries we present below are a representative sample of many test cases we considered. As before, each of the queries was run at least five times and the timing results are averages over these runs.

• Detecting an Empty Answer Set

The performance results for this technique were, of course, trivial: query execution time was essentially 0 in each case.

• Index Introduction

Our initial conjecture about when index introduction is useful was as follows: the attribute of the new predicate must be a major column (prefix of the search key) of a clustered index, and no such index is otherwise available for the table of that attribute. A sample set of queries on which we tested this conjecture is presented below.

$\mathcal{P}_1$:
```
select  sum(l_extendedprice * l_discount)
        as revenue
from    tpcd.lineitem
where   l_shipdate ≥ date('1994-01-01') and
        l_receiptdate < date ('1994-01-01')+1 year
        and l_discount between .06 - 0.01
        and .06 + 0.01 and l_quantity < 24;
```

$\mathcal{P}_2$:
```
select  100.00 * sum
        (case
           when p_type like 'PROMO%'
           then l_extendedprice * (1 - l_discount)
           else 0
        end)
        / sum(l_extendedprice * (1 - l_discount))
        as promo_revenue
from    tpcd.lineitem, tpcd.part
where   l_partkey = p_partkey and
        l_shipdate ≥ date('1998-09-01') and
        l_shipdate < date('1998-09-01')+1 month;
```

$\mathcal{P}_3$: same as $\mathcal{P}_2$ except for the indicated lines which are replaced with:

```
        l_shipdate ≥ date('1995-09-01') and
        l_shipdate < date('1995-09-01')+1 month;
```

$\mathcal{P}_4$:
```
select  l_orderkey,
        sum(l_extendedprice * (1 - l_discount))
        as revenue, o_orderdate, o_shippriority
from    tpcd.customer, tpcd.orders, tpcd.lineitem
where   c_mktsegment = 'BUILDING'
        and c_custkey = o_custkey
        and l_orderkey = o_orderkey and
        o_orderdate < date ('1998-03-15') and
        l_shipdate > date ('1998-03-15')
group by l_orderkey, o_orderdate, o_shippriority
order by revenue desc, o_orderdate
fetch   first 10 rows only;
```

$\mathcal{P}_5$: same as $\mathcal{P}_4$ except for the indicated lines which are replaced with:

	Data Reads		Index Reads		CPU Cost	Estimated Number
	Physical	Logical	Physical	Logical	(s)	of Qualifying Tuples
Original Query	21607	22439	12	26	21.9	20839
"Optimized" Query	10680	286516	2687	288326	55.9	12618

Table 3: Comparison of the Evaluation Costs for $\mathcal{P}_3$.

o_orderdate < date ('1995-03-15') and
l_shipdate > date ('1995-03-15')

We created a clustered index with the search key <l_receiptdate, discount, quantity, extendedprice> for the table **lineitem**. Since the major column of the index, *l_receiptdate*, is not used in any of the queries, this index is not used in query evaluation. However, since the check constraint *l_receiptdate* $\geq$ *l_shipdate* is defined in TPCD for the table **lineitem** and each of the queries contain a predicate of the form *l_shipdate* $\geq$ *DATE*, we may add a new predicate *l_receiptdate* $\geq$ *DATE* to each of the queries. Now, a potentially better access plan is available for query evaluation. Indeed, as Table 2 shows, cost estimates generated by the optimizer indicate that the availability of the new predicate should uniformly improve query execution time. Surprisingly, it did not happen: for queries $\mathcal{P}_3$ and $\mathcal{P}_5$ execution time grew with the addition of the new predicate.

In retrospect, there is a simple explanantion for this deterioration. Consider queries $\mathcal{P}_2$ and $\mathcal{P}_3$. The query plan generated for both of the original queries (that is, without predicate introduction) used table scan to retrieve qualifying tuples from the **lineitem** table. The introduction of a new predicate (*l_receiptdate* $\geq$ *date ('1998-09-01')* for $\mathcal{P}_2$ and *l_receiptdate* $\geq$ *date ('1995-09-01')* for $\mathcal{P}_3$), allows the optimizer to use the index. Indeed, the new plan for both queries uses the index to get directly to the tuples within the specified range of *l_receiptdate*. This requires traversing the index leaves in the range of the predicate and retrieving all tuples pointed to by these leaves. (It is often stated in database textbooks that for a range search with a clustered index, it is sufficient to retrieve the first tuple in the range and then scan the rest of the table. For this approach to work, however, the table has to be perfectly sorted at all times. This is often impractical, hence not implemented in DB2 UDB.) What makes the difference between the performance of the two queries is the size of that range: 2% of the tuples fall within the range of *l_receiptdate* $\geq$ *date ('1998-09-01')* and 48% are within the range of *l_receiptdate* $\geq$ *date ('1995-09-01')*. The second range is so large that a simple table scan would be a better plan. Using an index instead involves a large overhead in locking and unlocking index pages. Indeed, I/O cost (number of physical page reads) for $\mathcal{P}_3$ does go down (see Table 3), but the CPU cost increases even more.

So far, we identified two possible reasons why the optimizer may have chosen a more expensive access plan for the query. One is that the cost model underestimates the cost of locking and unlocking index pages. The second, more interesting reason, is the computation of a filter factor for the query.[4] Since a new predicate is added to the query the filter factor of the predicates of the original query is multiplied by the filter factor of *l_receiptdate* $\geq$ *date ('1995-09-01')*. The estimated number of qualifying tuples goes down (see Table 3) and the optimizer chooses an index scan as the best access plan. However, the number of qualifying tuples does *not* decrease in the optimized query, since the query is semantically equivalent to the original one. The problem, of course, is the correlation between *shipdate* and *l_receiptdate*: the filter factor of a conjunction of the predicates with these two attributes is *not* a product of their individual filter factors. This is an important lesson for any implementation of query rewrite involving addition or removal of predicates with correlated attributes.

Queries		$\mathcal{P}_3$	$\mathcal{P}_5$
Execution Time (s)	Original	21.3	52.2
	Optimized	10.9	45.6

Table 4: Performance Results for Index Introduction (modified algorithm)

Once we discovered that our initial conjecture about the usefullness of predicate introduction was incorrect, we needed to restrict it further. Thus, we modified the algorithm so that a new predicate is added to a query only if it contains a major column of an index and a scan of that index is sufficient to answer the query (that is, no table scan is necessary). To verify our hypothesis, we created an index <receiptdate, discount, quantity, extendedprice, shipdate, partkey, suppkey, orderkey> and ran the queries $\mathcal{P}_3$ and $\mathcal{P}_5$ again. As expected, the use of the index-only plan improved the execution time of the original query. The addition of a major column of that index to the query improved

[4] A filter factor of a predicate (also called a reduction factor) is the proportion of tuples in a relation satisfying the predicate.

that even more. The results are presented in Table 4.

• **Scan Reduction**

Since the only check constraint available to us[5] was *l_receiptdate* $\geq$ *l_shipdate*, we had to test queries involving a join with at least one of these attributes. (The queries are admittedly not very meaningful.) The only difference between the two queries is the range of qualifyling tuples (much larger in the second query).

$\mathcal{S}_1$: **select** sum(l1.l_extendedprice * l1.discount) as revenue
from tpcd.lineitem as l1, tpcd.lineitem as l2
where l1.l_commitdate = l2.l_receiptdate and l2.l_shipdate > date('1998-07-01') and l2.l_discount between $0.06 - 0.01$ and $0.06 + 0.01$ and l2.l_quantity < 24;

$\mathcal{S}_2$: **select** sum(l1.l_extendedprice * l1.discount) as revenue
from tpcd.lineitem as l1, tpcd.lineitem as l2
where l1.l_commitdate = l2.l_receiptdate and l2.l_shipdate > date ('1996-07-01') and l2.l_discount between $0.06 - 0.01$ and $0.06 + 0.01$ and l2.l_quantity < 24;

Because of the check constraint, we may add a new predicate *l2.l_receiptdate > date ('1998-07-01')* to $\mathcal{S}_1$ (similarly for $\mathcal{S}_2$) thus limiting the number of tuples that qualify for the join. As shown in Table 5, this provides consistent optimization for the four tested types of join: index nested loops, sort-merge, nested loops, and hash join.

For INL we defined the index <receiptdate, discount, quantity, extendedprice, shipdate, partkey, suppkey, orderkey>. The index was explicitly dropped for all other joins.

Although the results of our experiments were very promising, we could not design more meaningful queries for scan reduction in TPCD. This technique is applicable only when there exist check constraints defined over join attributes of a relation or there are predicates with inequality over join attributes.

5 Conclusions

We developed algorithms for two SQO techniques, Join Elimination (JE) and Predicate Introduction (PI), and implemented siginificant portions of these technologies as a prototype in DB2 UDB. The implementation process and performance analysis provided us with several insights about SQO and query optimization in general.

[5] DB2 does not yet support check constraints across multiple tables.

The most important outcome of our work is the experimental evidence which shows that SQO can provide an effective enhancement to the traditional query optimization. This is particularly striking in the case of JE for which our experiments delivered consistent optimization for all tested queries. Although our implementation of JE was geared towards OLAP environment (in which, we believe, JE can be particularly useful), a few experiments we performed in TPCD were also very promising. The algorithm for JE that we developed handles not only the removal of explicit RI joins, but all redundant joins that can be inferred through reasoning over the query and the RIs defined for the database. We showed on several examples that such inferred joins can be difficult to discover and optimize by hand.

What is novel about our implementation of JE is that it does not depend on the existence of complex integrity constraints. The use of semantic reasoning about referential integrity constraints, which are common in both OLTP as well as OLAP workloads, can lead to dramatic performance improvements for typical queries in these workloads. Moreover, our implementation of JE does not depend on intricate and often unavailable cost information. This makes JE easy to implement and efficient to execute.

Our experiments with PI showed that this technique can be very useful in detecting, by reasoning over integrity constraints, when a query's answer set is empty. The use of PI for index introduction delievered good performance improvements when it was sufficiently restricted. To *guarantee* such improvements, however, these restrictions had to be rather severe, thus limiting the applicability of the technique. Nevertheless, the implementation of PI was very useful for us as an experience. The ramifications of adding or removing a predicate from a query turned out to be more complex than we originally predicted. In particular, the estimate of a filter factor in the optimized query has to take into account the fact that semantically related attributes are likely to be statistically correlated.

There is yet another lesson we learned from this work. We found the necessity of having to generate a single query by the rewrite engine constraining. The conditions used to activate rules that produce such a query are not based on database statistics (this comes into play only at the access plan generation phase) hence have to be sufficiently general to work for all databases. Many rewrites useful for a *particular* database can be missed. (The implementation of magic sets [15] as a cost-based optimization technique proved to be much more successful than as a heuristic rewrite technique.) Thus, we advocate establishing a two-way communication between the two phases of the query optimization, so that the plan generator can provide query rewrite

Query		$\mathcal{S}_1$				$\mathcal{S}_2$			
Join Method		INL	SM	NL	Hash	INL	SM	NL	Hash
Execution Time (s)	Original	107.2	107.5	107.8	107.6	270.8	250.0	250.0	250.3
	Optimized	37.4	52.9	54.3	52.9	218.5	209.3	209.4	209.3

Table 5: Performance Results for Scan Reduction

engine with a feedback on which transformations are useful and which are not. This change will increase the complexity of the design of the optimizer, but we believe it can dramatically increase the effectiveness of the rewrites, in particular, those based on SQO.

Acknowledgements

We would like to thank Jennifer Crawford and Cheryl Greene from IBM Toronto Lab for their help with this project and Parke Godfrey and Renée Miller for helpful comments on the paper.

This research was supported by Center for Advanced Studies, IBM Toronto and NSERC (Grant 216748-98).

References

[1] U. Chakravarthy, J. Grant, and J. Minker. Logic-based approach to semantic query optimization. *ACM TODS*, 15(2):162–207, June 1990.

[2] S. Chaudhuri and K. Shim. Including group-by in query optimization. In *Proc. of VLDB*, pages 354–366, 1994.

[3] OLAP Council. APB-1 OLAP Benchmark Release II, November 1998. (www.olapcouncil.org).

[4] IBM DB2 Universal Database. *Administration Guide.* 1998. Version 5.2 S10J-8157-01.

[5] J. Gryz, L. Liu, and X. Qian. Semantic query optimization in DB2: Initial results. Technical Report CS-1999-01, Department of Computer Science, York University, Toronto, Canada, 1999.

[6] L.M. Haas et al. Starburst Mid-Flight: As the Dust Clears. *IEEE TKDE*, pages 143–160, March 1990.

[7] M.T. Hammer and S.B. Zdonik. Knowledge-based query processing. *Proc. 6th VLDB*, pages 137–147, October 1980.

[8] M. Jarke, J. Clifford, and Y. Vassiliou. An optimizing PROLOG front-end to a relational query system. In *SIGMOD*, pages 296–306, 1984.

[9] J.J. King. Quist: A system for semantic query optimization in relational databases. *Proc. 7th VLDB*, pages 510–517, September 1981.

[10] A.Y. Levy, I. Mumick, and Y. Sagiv. Query optimization by predicate move-around. In *Proc. of VLDB*, pages 96–108, 1994.

[11] I. Mumick and H. Pirahesh. Implementation of magic sets in Starburst. In *Proc. SIGMOD*, 1994.

[12] G. Paulley and P. Larson. Exploiting uniqueness in query optimization. In *Proceedings of ICDE*, pages 68–79, 1994.

[13] H. Pirahesh, J. M. Hellerstein, and W. Hasan. Extensible/rule based query rewrite optimization in Starburst. In *Proc. SIGMOD*, pages 39–48, 1992.

[14] H. Pirahesh, T. Y. C. Leung, and W. Hasan. A rule engine for query transformation in Starburst and IBM DB2 C/S DBMS. In *Proc. ICDE*, pages 391–400, 1997.

[15] S. Seshadri et al. Cost-based optimization for magic: Algebra and implementation. In *SIGMOD*, pages 435–446, 1996.

[16] S. Shekar, J. Srivastava, and S. Dutta. A formal model of trade-off between optimization and execution costs in semantic query optimization. In *Proc.* 14^{th} *VLDB*, pages 457–467, Los Angeles, CA, 1988.

[17] S.T. Shenoy and Z.M. Ozsoyoglu. Design and implementation of a semantic query optimizer. *IEEE Transactions on Knowledge and Data Engineering*, 1(3):344–361, September 1989.

[18] D. Simmen, E. Shekita, and T. Malkems. Fundamental techniques for order optimization. In *Proceedings of SIGMOD*, pages 57–67, 1996.

[19] Transaction Processing Performance Council, 777 No. First Street, Suite 600, San Jose, CA 95112-6311, www.tpc.org. *TPC Benchmark*^{TM} *D*, 1.3.1 edition, February 1998.