Query Optimization by Predicate Move-Around

Alon Y. Levy
AT&T Bell Laboratories
Murray Hill, NJ.
levy@research.att.com

Inderpal Singh Mumick
AT&T Bell Laboratories
Murray Hill, NJ.
mumick@research.att.com

Yehoshua Sagiv*
Hebrew University
Jerusalem, Israel.
sagiv@cs.huji.ac.il

Abstract

A new type of optimization, called *predicate move-around,* is introduced. It is shown how this optimization considerably improves the efficiency of evaluating SQL queries that have query graphs with a large number of query blocks (which is a typical situation when queries are defined in terms of multiple views and subqueries). Predicate move-around works by moving predicates across query blocks (in the query graph) that cannot be merged into one block. Predicate move-around is a generalization of and has many advantages over the traditional predicate pushdown. One key advantage arises from the fact that predicate move-around precedes pushdown by pulling predicates up the query graph. As a result, predicates that appear in the query in one part of the graph can be moved around the graph and applied also in other parts of graph. Moreover, predicate move-around optimization can move a wider class of predicates in a wider class of queries as compared to the standard predicate-pushdown techniques. In addition to the usual comparison and arithmetic predicates, other predicates that can be moved around are the `EXISTS` and `NOT EXISTS` clauses, the `EXCEPT` clause, and functional dependencies. The proposed optimization can also move predicates through aggregation. Moreover, the method can also infer new predicates when existing predicates are moved through aggregation or when certain functional dependencies are known to hold. Finally, the predicate move-around algorithm is easy to implement on top of existing query optimizers.

1 Introduction

Current benchmarks (e.g., TPC/D) have exposed a serious weakness of commercial database systems when it comes to query optimization. In some cases, several person months have been spent to optimize queries by hand in order to achieve better performance. Further aggravating the problem is the growing complexity of queries; for example, decision-support queries have become very important in large organizations (e.g., the world's largest private computer is dedicated to running decision-support queries for the retail giant WalMart).

In complex applications, such as decision-support systems, a query usually depends on a large number of subqueries and views. Each of the subqueries and views forms a query block in the query graph. However, most cost-based plan optimizers can only handle one query block at a time. Therefore, it is valuable to merge subqueries and views into one query block. Unfortunately, this is a complicated and sometimes impossible task due to aggregates, the SQL semantics of duplicates, correlations, `EXISTS`, `NOT EXISTS`, `EXCEPT`, `UNION`, and `INTERSECTION`. The work of [PHH92] has investigated this issue and provided some solutions for merging blocks of queries when the duplicate semantics can either be ignored or when all duplicates are eliminated.

When query blocks cannot be merged, it is important to rewrite the query so that predicates can be applied as early as possible. Predicate pushdown [Ull89] is a common and important optimization technique for pushing selection predicates down a query graph, in order to apply those selections as early as possible during evaluation. However, it works only on hierarchical queries, which are nonrecursive queries without common subexpressions. Another approach is the adaptation of the magic-set transformation for an early evaluation of selection and join predicates in nonrecursive SQL queries with common subexpressions [MFPR90a]. The magic-set transformation (see [Ull89] for details) pushes predicates according to the order of doing joins and introduces auxiliary *magic* views.

In this paper, we propose a generalization of the predicate-pushdown technique, called *predicate move-around,* that is capable of pushing predicates down, up and sideways in the query graph; predicates are moved up as an intermediate step before being pushed down.

*Research supported in part by Grant 4766-1-93 of the Israeli National Council for Research and Development.

Proceedings of the 20th VLDB Conference, Santiago, Chile, 1994.

As a result, predicates that appear in the query in one part of the graph can be moved around the graph and applied also in other parts of graph. Moreover, predicate move-around can be applied even if some blocks have aggregates and even if duplicates are retained in some blocks and eliminated in others. Our algorithm for predicate move-around is extensible in the sense that (1) a variety of predicates can be moved around; for example, comparison and inequality predicates, `EXISTS` and `NOT EXISTS` predicates, negated base relations (the `EXCEPT` clause), arithmetic predicates (e.g., $X = Y + Z$), the `LIKE` predicate, functional dependencies and more, and (2) Predicates can be moved through new operators like outer-join. The predicate move-around results in applying a larger number of predicates to base and intermediate relations and doing so as early as possible; hence, the evaluation becomes more efficient. Unlike the magic-set transformation, predicate move-around does not need auxiliary relations (such as the magic and supplementary relations) and does not depend upon the join order. Our move-around algorithm applies to nonrecursive SQL queries, including SQL queries with correlations. It can also be generalized to recursive SQL queries (as defined in the new proposed SQL3 standard [ISO93]), but doing so is beyond the scope of this paper.

To summarize, following are the novel features of our algorithm:

- Moving predicates up, down and sideways in the query graph, across query blocks that cannot be merged.
- Moving predicates through aggregation; in the process, new predicates are deduced from aggregation.
- Using functional dependencies to deduce and move predicates.
- Moving `EXISTS` and `NOT EXISTS` predicates. The `EXCEPT` clause can also lead to a `NOT EXISTS` predicate that can then be moved.
- Removing redundant predicates. This is important, since redundant predicates can lead to incorrect selectivity estimates that may result in access paths and join methods that are far from optimal. Moreover, redundant predicates represent wasted computation.

Our algorithm can be incorporated easily on top of existing query optimizers, since it consists of rewriting the original queries and views. For example, our algorithm would fit well in the Starburst optimizer [PHH92].

The paper is organized as follows. Section 2 illustrates the savings achieved by predicate move-around via a detailed example. Section 3 describes the SQL syntax and the query-tree representation on which the algorithm operates. The query tree is a straightforward parse-tree representation of a query, close to what is used in several systems (e.g., [PHH92]). The predicate move-around algorithm is detailed in Section 4. We describe the general algorithm, and illustrate each step of the algorithm on the example of Section 2. We consider related work in Section 5, and conclude in Section 6.

2 Illustrative Example

We consider a detailed example that illustrates the benefits of the predicate move-around optimization. In particular, this example illustrates how predicates can be moved across query blocks, through aggregation and using knowledge about functional dependencies.

This example is quite typical of the complexity of real world decision-support queries. Further, it illustrates the capabilities of our algorithm to deal with complex queries as compared to more traditional methods. The example uses the following base relations from a telephone database.

`calls`(FromAC, FromTel, ToAC, ToTel, AccessCode, StartTime, Length)
`customers`(AC, Tel, OwnerName, Type, MemLevel)
`users`(AC, Tel, UserName, AccessCode)
`secret`(AC, Tel)
`promotion`(AC, SponsorName, StartingDate, EndingDate)

The `calls` relation has a tuple for every call made. A telephone number is represented by the area code (AC) and the local telephone number (Tel). Foreign numbers are given the area code "011." A `call` tuple contains the telephone number from which the call is placed, the number to which the call is placed, and the access code used to place the call (0 if an access code is not used). The starting time of the call, with a granularity of 1 minute, and the length of the call, again with 1 minute granularity, are included. Due to the time granularity and multiple lines having the same phone number, duplicates are permitted in this relation. In particular, there can be several calls of 1 minute each starting within the same 1 minute, and there can be multiple calls running concurrently between two given numbers.

The `customers` relation gives information about the owner of each telephone number. The information about owners consists of the owner's name, his account type (Government, Business, or Personal), and his membership level (Basic, Silver, or Gold). The key of the `customers` relation is {AC, Tel} and, so, the following functional dependency holds:

$$\{\text{AC, Tel}\} \rightarrow \{\text{OwnerName, Type, MemLevel}\}.$$

A telephone number can have one or more users that are listed in the `users` relation. Each user of a telephone number may have an access code. One user may have multiple access codes, and one access code may be given to multiple people. There are no duplicates in the `users` relation.

A few telephone numbers have been declared secret, as given by the `secret` relation.

The `promotion` relation has, for each planned marketing promotion, the name of the sponsoring organization, the area codes to which calls are being promoted, and the starting and ending dates of the promotion. Note that there may be several tuples with the same area code and same sponsor, but with different dates.

Example 2.1 Consider the query $Q1$ given in Figure 1. Note that this query is defined in terms of two views. The view `fgAccounts`, denoted as $E1$, lists all foreign accounts (i.e., Area-Code="011") of type "Govt" that are not secret accounts. This view includes telephone numbers and names of users of those numbers.

The second view `ptCustomers` (i.e., potential customers), denoted as $F1$, first selects calls longer than 2 minutes and then finds, for each customer with a silver level membership, the maximum length amongst all his calls to each area code other than the customer's own area code.

The query $Q1$ is posed by a marketing agency looking for potential customers amongst foreign governments that make calls longer than 50 minutes to area codes in which some promotion is planned. The query lists the phone number of each relevant foreign government, the names of all users of that phone and the names of the sponsors of the relevant promotions. Note that duplicates are retained, since sponsors may have one or more promotions in one or more area codes.

Since the view `ptCustomers` does aggregation and the view `fgAccounts` generates and eliminates duplicates (while $Q1$ retains duplicates), neither the view $E1$ nor the view $F1$ can be merged into the query block of $Q1$. Consequently, an ordinary optimizer cannot deal effectively with the query $Q1$, since it is forced to optimize and evaluate each view separately and then optimize and evaluate the query. For example, note that the predicate MaxLen $>$ 50 cannot be pushed from the definition of $Q1$ into the definition of the view `ptCustomers` (since that predicate is over a field that is aggregated in `ptCustomers`). Thus, when evaluating the view `ptCustomers`, we can only use the predicate Length $>$ 2; later, when evaluating the query, we can use the predicate MaxLen $>$ 50 to discard those `ptCustomers` tuples that do not satisfy this selection. Our optimization algorithm, in comparison, would do much better, since it is capable of the following.

- Taking the predicate `c`.AC = "011" of the view `fgAccounts` and moving it into the view `ptCustomers`. As a result, the join predicate `c`.AC = `t`.FromAC is replaced with `t`.FromAC = "011" and `t`.ToAC <> `t`.FromAC is replaced with `t`.ToAC <> "011".

- Taking the predicate `c`.Type = "Govt" from the view `fgAccounts` and moving it into the view `ptCustomers`, where it is applied to the `customers` relation. Note that determining soundness of this move requires that we reason with knowledge about functional dependencies. Specifically, in the query $Q1$, the views `fgAccounts` and `ptCustomers` are joined on a key of the `customers` relation and, therefore, the predicate `c`.Type = "Govt" can be moved from `fgAccounts` into the definition of `ptCustomers`.

- Taking the predicate

  ```
  NOT EXISTS (SELECT * FROM secret s
              WHERE s.AC = c.FromAC AND
                    s.Tel = c.FromTel)
  ```

  from the view `fgAccounts` and moving it into the view `ptCustomers`. This leads to a more efficient evaluation of the view `ptCustomers`, since the `customers` relation can be restricted before taking the join with `calls` and before the grouping operation.

- Taking the predicate `c`.MemLevel = "Silver" from the view `ptCustomers` and moving it into the view `fgAccounts`. Again, functional dependencies are used for this move.

- Taking the predicate MaxLen $>$ 50 from the query and inferring that `t`.Length $>$ 50 can be introduced in the `WHERE` clause of `ptCustomers`. As a result, the predicate `t`.Length $>$ 2 can be eliminated from the definition of `ptCustomers` and the predicate `ptc`.MaxLen $>$ 50 can be deleted from the query. Note that this optimization amounts to pushing a selection through aggregation, a novel feature of our algorithm.

The optimized views and query are denoted in Figure 2 as $E1_o$, $F1_o$ and $Q1_o$. Section 4 explains the behavior of the predicate move-around algorithm on this example in detail. □

3 Preliminaries: SQL Notation and the Query-Tree Representation

For simplicity of presentation of the move-around algorithm, we consider here a subset of SQL. Given an SQL query, we first translate it into a *query tree* (see Figure 5 for an example) and then apply the move-around algorithm to the query tree. In this section, we briefly describe the SQL syntax and explain how to build the query tree.

```
calls(FromAC, FromTel, ToAC, ToTel, AccessCode, StartTime, Length)
customers(AC, Tel, OwnerName, Type, MemLevel)
users(AC, Tel, UserName, AccessCode)
secret(AC, Tel)
promotion(AC, SponsorName, StartingDate, EndingDate)
```

```
(E1):   CREATE VIEW fgAccounts(AC, Tel, UserName) AS
            SELECT DISTINCT c.AC, c.Tel, u.UserName
            FROM customers c, users u
            WHERE c.AC = u.AC AND
                  c.Tel = u.Tel AND
                  c.Type = "Govt" AND
                  c.AC = "011" AND
                  NOT EXISTS (SELECT ⋆ FROM secret s
                              WHERE s.AC = c.AC AND s.Tel = c.Tel)

(F1):   CREATE VIEW ptCustomers (AC, Tel, OwnerName, ToAC, MaxLen) AS
            SELECT c.AC, c.Tel, c.OwnerName, t.ToAC, MAX (t.Length)
            FROM customers c, calls t
            WHERE t.Length > 2 AND
                  t.FromAC <> t.ToAC AND                /* <> is the SQL symbol for not equal */
                  c.AC = t.FromAC AND
                  c.Tel = t.FromTel AND
                  c.MemLevel = "Silver"
            GROUPBY c.AC, c.Tel, c.OwnerName, t.ToAC

(Q1):   SELECT ptc.AC, ptc.Tel, fg.UserName, p.SponsorName
        FROM ptCustomers ptc, fgAccounts fg, promotion p
        WHERE ptc.AC = fg.AC AND
              ptc.Tel = fg.Tel AND
              ptc.MaxLen > 50 AND
              p.AC = ptc.ToAC
```

Figure 1: The original views for query $Q1$.

3.1 SQL Syntax

An SQL query consists of a sequence of view definitions (or *blocks*). The following is a GROUPBY block.

```
CREATE VIEW V(A_1, ..., A_l) AS
    SELECT X_1, ..., X_l
    FROM Rel_1 r_1, ..., Rel_m r_m
    WHERE ...
    GROUPBY G_1, ..., G_k
    HAVING ...
```

Each X_i is either an *attribute term* (e.g., $r_j.B$) or an *aggregate term* (e.g., $Max(r_j.B)$); for simplicity, we assume that the X_i's are distinct. If there are no GROUPBY and HAVING clauses, and no *aggregate terms*, then the above block is called a SELECT block; there are also UNION and INTERSECTION blocks. One of the blocks is the *query block;* it is similar to the above block, but without the CREATE-VIEW clause. In this paper, the search condition that appears in a WHERE or HAVING clause is assumed to be in the conjunctive normal form. Each conjunct in the search condition is called a *predicate*. We consider predicates that are built from comparisons (=, <, etc.), AND, OR, NOT, EXISTS, and NOT EXISTS.

3.2 The Query Tree

The nodes of a *query tree* correspond to blocks (the root corresponds to the query block). The children of a node n are the views (i.e., non-base relations) referenced in the block corresponding to node n; e.g., a node for a SELECT block has a child for every occurrence of a view in the FROM clause.

Local and Exported Attributes: The *local* attributes of node n are those appearing in the operands of the corresponding block; the *exported attributes* are

```
(E1_o):  CREATE VIEW fgAccounts_o(AC, Tel, UserName) AS
             SELECT DISTINCT c.AC, c.Tel, u.UserName
             FROM customers c, users u
             WHERE c.AC = u.AC AND
                   c.Tel = u.Tel AND
                   c.Type = "Govt" AND
                   c.MemLevel = "Silver" AND
                   c.AC = "011" AND
                   NOT EXISTS (SELECT * FROM secret s
                               WHERE s.AC = c.AC AND s.Tel = c.Tel)

(F1_o):  CREATE VIEW ptCustomers_o (AC, Tel, OwnerName, ToAC, MaxLen) AS
             SELECT c.AC, c.Tel, c.OwnerName, t.ToAC, MAX (t.Length)
             FROM customers c, calls t
             WHERE t.Length > 50 AND
                   t.ToAC <> "011" AND
                   t.FromAC  = "011" AND
                   c.Tel = t.FromTel AND
                   c.MemLevel = "Silver" AND
                   c.Type = "Govt" AND
                   c.AC = "011" AND
                   NOT EXISTS (SELECT * FROM secret s
                               WHERE s.AC = c.AC AND s.Tel = c.Tel)
             GROUPBY c.AC, c.Tel, c.OwnerName, t.ToAC

(Q1_o):  SELECT ptc.AC, ptc.Tel, fg.UserName, p.SponsorName
         FROM ptCustomers_o ptc, fgAccounts_o fg, promotion p
         WHERE ptc.AC = fg.AC AND
               ptc.Tel = fg.Tel AND
               p.AC = ptc.ToAC
```

Figure 2: The optimized views for query $Q1$.

those appearing in the result of that block (i.e., the attributes of the defined view).

Labels: In the query tree, each node n has an associated *label*, denoted $L(n)$. The label contains predicates that are applicable to attributes of n. Due to functional dependencies, predicates appearing in labels may also contain functional terms; for example, if the attribute $r.A$ functionally determines the attribute $r.B$, then the predicate $f(r.A) = r.B$ is added to nodes having $r.A$ and $r.B$ as attributes. A predicate of a label $L(n)$ is called *local* if all its attributes are local attributes of n; it is called *exported* if all its attributes are exported attributes of n. Next, we explain the four types of nodes in the query tree.

SELECT Nodes

When a view definition consists of a SELECT block[1]

CREATE VIEW V($A_1, \ldots, A_l$) AS
SELECT $r_{k_1}.B_1, \ldots, r_{k_l}.B_l$
FROM $Rel_1\ r_1, \ldots, Rel_m\ r_m$
WHERE ...

we create a SELECT node n. For every Rel_i that is a view, node n has a child for the block of Rel_i. The local attributes of n are all the attributes of the form $r_i.B$, where B is an attribute of Rel_i ($1 \leq i \leq m$). The exported attributes are $V.A_1, \ldots, V.A_l$. Note that the exported attributes are just aliases of the local attributes listed in the SELECT clause; that is, there is a one-to-one correspondence between the local and exported attributes ($V.A_i$ corresponds to $r_i.B_i$).

GROUPBY Triplets and Nodes

A view definition consisting of a GROUPBY block is separated into three nodes (see Figure 3) in order to highlight the movement of predicates through the aggregation operators. The bottom node, n_1, is a SELECT node for the SELECT-FROM-WHERE part of the view definition (it may have children as described

[1] A SELECT DISTINCT block is treated exactly as a SELECT block.

earlier). The middle node, n_2, is a GROUPBY node and it stands for the GROUPBY clause and the associated aggregations. The top node, n_3, is a HAVING node and it stands for the predicate in the HAVING clause.

```
CREATE VIEW V (A1, ..., Al)
  SELECT r.B, s.C, Max(r.D)
  FROM R r, S s
  WHERE  r.B ≤ r.C
         s.C=r.A
  GROUPBY r.A
  HAVING ...
```

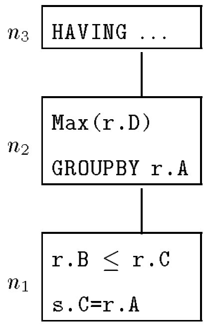

Figure 3: A triplet for a GROUPBY block.

The set of local attributes in n_1 (denoted by L) is defined in the same way as it is defined for an ordinary SELECT node, and is also the set of exported attributes of n_1. Let G denote the set of grouping attributes (i.e., the attributes in the GROUPBY clause), and let A denote the set of aggregate terms (e.g., $Max(r.D)$) used in the SELECT and HAVING clauses. The attributes of the set $L \cup A$ are the local attributes of n_2, whereas $G \cup A$ is the set of exported attributes of n_2 and the set of local attributes of n_3. The exported attributes of n_3 are $\{V.A_1, \dots, V.A_l\}$.

A view definition may have aggregation in the SELECT clause even without having GROUPBY and HAVING clauses. In this case, we still construct a triplet (as if there is an empty GROUPBY clause). Also note that if there is no HAVING clause, then we can omit the top node and let $V.A_1, \dots, V.A_l$ be the exported attributes of n_2.

UNION and INTERSECTION Nodes

If a view definition includes UNION (or INTERSECTION), we create a node n for this operation (see Figure 4). Node n has a SELECT child for every SELECT block in the view definition (some children may be triplets for GROUPBY blocks). For the i^{th} child, the local attributes are defined as usual and the exported attributes are $V_i.A_1, \dots, V_i.A_l$, where V_i is a newly created name. The local attributes of the (UNION or INTERSECTION) node n are the exported attributes from all the children of n. The exported attributes of node n are $V.A_1, \dots, V.A_l$. Note that there is a one-to-one correspondence between the exported attributes of n and the exported attributes of the i^{th} child; that is, $V.A_k \leftrightarrow V_i.A_k$ $(1 \leq k \leq l)$.

DAG Queries

In a DAG query, a view V may have several references in the same or different blocks. In this case, we create a distinct node for each occurrence of V.

```
CREATE VIEW V (A1, ..., Al)

     SELECT ...
     FROM ...
     WHERE ...

     UNION
     :
     UNION
     SELECT ...
     FROM ...
     WHERE ...
```

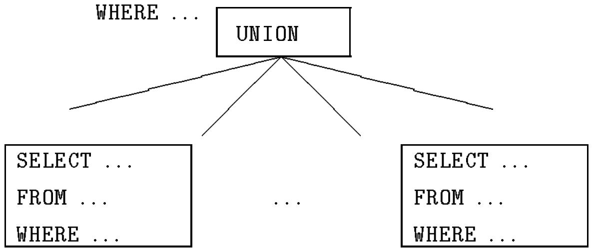

Figure 4: The node structure built for UNION and INTERSECTION.

Example 3.1 Figure 5 shows the query tree for the query $Q1$ of Example 2.1 (the labels in the nodes should be ignored for now). The view ptCustomers is represented on the left by a pair of SELECT and GROUPBY nodes. The GROUPBY block defining view ptCustomers does not have a HAVING clause; hence the top SELECT node of the GROUPBY triplet has been omitted. The view fgAccounts is represented on the right by a single SELECT node. The query view itself is represented by a single SELECT node at the top. The arcs from the ptCustomers and fgAccounts nodes into the query node arise from the usage of the views in defining the query. □

Renamings

To move predicates around in the query tree, we utilize two kinds of *renamings*. An *internal renaming* for node n is a mapping from the local attributes of node n to the exported attributes of n or vice versa. Nodes created for SELECT and GROUPBY blocks have exactly one internal renaming in each direction. In the case of a UNION or an INTERSECTION node n, there is a pair of internal renamings (one in each direction) for each child c; this pair relates the exported attributes of n to the local attributes obtained from the child c. Internal renamings are used to infer exported predicates from predicates with local attributes and vice versa.

An *external renaming* is simply a renaming from the exported attributes of a node n to local attributes referencing them in the parent of n. For example, the attribute fgAccounts.AC is referenced as fg.AC in the root of our example query tree. External renamings are used in order to move predicates from a node to its parent and vice versa.

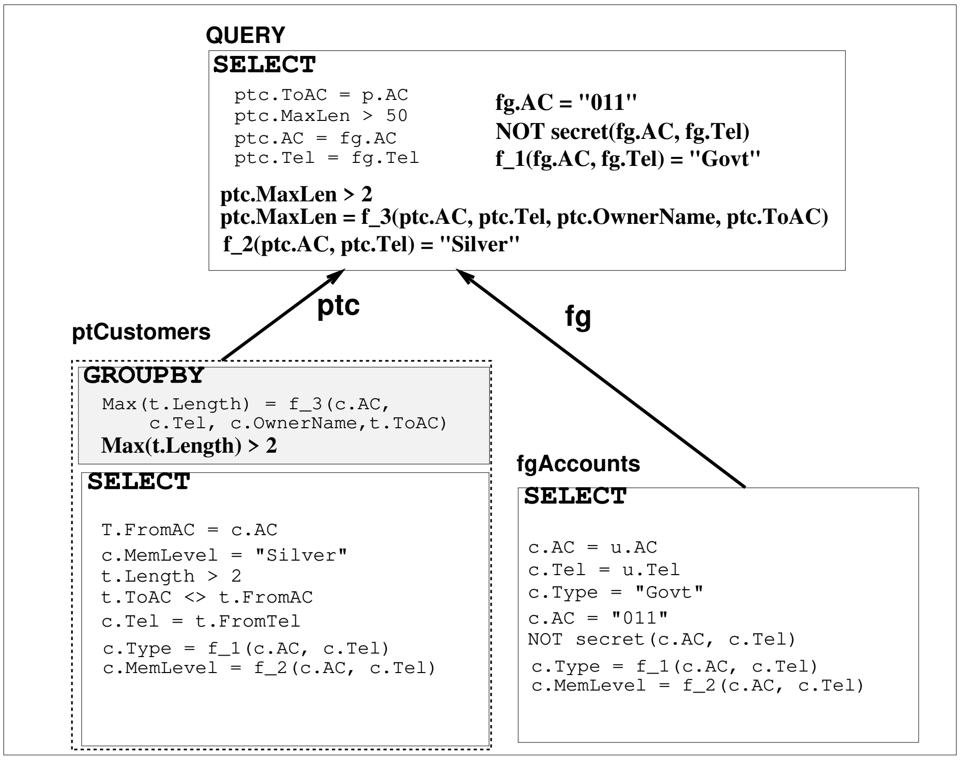

Figure 5: The query tree for Example 2.1. The predicates in roman font are inserted during initialization. The predicates in bold font are added to the labels in the pullup phase.

4 The Move-Around Algorithm

We give an overview of the main steps of the predicate move-around algorithm followed by a detailed description of each step.

4.1 The Main Steps of the Algorithm

1. Label initialization: Initial labels are created from the predicates in the `WHERE` and `HAVING` clauses and from functional dependencies.
2. Predicate pullup: The tree is traversed bottom up. At each node, we infer predicates on the exported attributes from predicates on the local attributes and pull up the inferred predicates into the parent node.
3. Predicate pushdown: The tree is traversed top down. At each node, we infer predicates on the local attributes from predicates on the exported attributes and push down the inferred predicates into the children of that node.
4. Label minimization: A predicate can be removed from a node if it is already applied at a descendant of that node.
5. (Optional:) Convert the query tree into SQL code (the plan optimizer may also work directly with the tree representation of the query).

The algorithm is extensible in the sense that it can be extended to new types of predicates (e.g., `LIKE`), to new types of nodes (e.g., outer join), and to new rules for inferring predicates. Next, we explain each step in detail.

4.2 Label Initialization

`SELECT` Nodes: The initial label of a `SELECT` node consists of the predicates appearing in the `WHERE` clause. For example, in Figure 5, the first five predicates in the `fgAccounts` node come from the `WHERE` clause. Note that (NOT secret(c.AC, c.Tel)) is simply a shorthand for the `NOT EXISTS` subquery.

`GROUPBY` Triplets: In a node triplet for a `GROUPBY` block (see Figure 3), the initial labels of the bottom and top nodes are the predicates from the `WHERE` and `HAVING` clauses, respectively. The initial label of the middle node includes predicates stating that the grouping attributes functionally determine the aggregated values. For example, in the view `ptCustomers`, the predicate

Max(t.Length) = f_3(c.AC, c.Tel, c.OwnerName, t.ToAC)

appears in the `GROUPBY` node (see Figure 5).

`UNION` and `INTERSECTION` nodes: The initial label of a `UNION` or an `INTERSECTION` node n is empty.

Functional Dependencies: Suppose that the following functional dependency holds in a base relation R.

$$fd : \{A_1, \ldots, A_k\} \rightarrow \{B_1, \ldots, B_p\}$$

If a `WHERE` or a `HAVING` clause refers to R, then the predicates $f_{fd_i}(r.A_1, \ldots, r.A_k) = B_i$ $(1 \leq i \leq p)$ are added to the label created for that clause (note that fd_i is an index that depends on fd and i). For example, the functional dependency

$$\{\text{AC, Tel}\} \rightarrow \{\text{OwnerName, Type, MemLevel}\}$$

holds in the `customer` relation; hence, the predicate

$$\text{c.Type} = \text{f\_1(c.AC, c.Tel)}$$

is added to the two `SELECT` leaves in Figure 5, since both reference the `customer` relation.

Example 4.1 The initial labels for the query tree of Example 2.1 are shown in regular font in Figure 5. □

4.3 Predicate Pullup

In the predicate-pullup phase, we traverse the tree bottom up, starting from the leaves. At each node, we infer predicates on the exported attributes from predicates on the local attributes. The inferred predicates are added to the labels of both the given node and its parent. The particular method for inferring additional predicates depends on the type of node under consideration and the types of predicates in the label of that node.

4.3.1 Predicate pullup through `SELECT` nodes

To pull up predicates through a `SELECT` node n, having a label $L(n)$, we proceed as follows.

- Add to $L(n)$ new predicates that are implied by those already in $L(n)$. For example, if both $r_1.A < r_2.B$ and $r_2.B < r_3.C$ are in $L(n)$, then $r_1.A < r_3.C$ is added to $L(n)$. Ideally, we would like to compute the closure of $L(n)$ under logical implications, since that would maximize the effect of moving predicates around. However, the move-around algorithm remains correct even if we are not able to compute the full closure.[2]

- Infer predicates with exported attributes as follows. If α is in $L(n)$, then add $\tau(\alpha)$ to $L(n)$, where τ is the internal renaming from the local attributes to the exported ones. For example, in the `fgAccounts` node of Figure 5, the predicate `fgAccounts`.AC = "011" on the exported attributes is inferred from the predicate c.AC = "011" on the local attributes.

- If α is an exported predicate of $L(n)$, then add $\sigma(\alpha)$ to the label of the parent of n, where σ is the external renaming from the exported attributes of n to local attributes of its parent.

4.3.2 Predicate pullup through `GROUPBY` nodes

In principle, it is enough to perform the three steps of the previous subsection at a `GROUPBY` node. In practice, however, we need some rules for inferring predicates involving aggregate terms. Following is a (sound but not complete) set of such rules; these rules should be applied to the label, $L(n)$, of a `GROUPBY` node n (in all these rules, $\leq$ can be replaced with $<$).

1. If $Min(B)$ is a local attribute of n, then add $Min(B) \leq B$ to $L(n)$ (in words, the minimum value of B is less than or equal to every value in column B). Furthermore, if $(B \geq c) \in L(n)$, where c is a constant, then add $Min(B) \geq c$ to $L(n)$ (in words, if c is less than or equal to every value in column B, then c is also less than or equal to the minimum value of B).

2. If $Max(B)$ is an attribute of n, then add $Max(B) \geq B$ to $L(n)$. Furthermore, if $(B \leq c) \in L(n)$, where c is a constant, then add $Max(B) \leq c$ to $L(n)$. For example, consider the `GROUPBY` node in Figure 5. First, we infer Max(t.Length) $\geq$ t.Length. Since t.Length $>$ 2 is pulled up from the child of the `GROUPBY` node, we infer Max(t.Length) $>$ 2 (by transitivity). Now, MaxLen $>$ 2 is inferred, since MaxLen is an exported attribute that is an alias of Max(t.Length). For clarity, only MaxLen $>$ 2 is shown in the figure.

3. Consider the following three predicates: $Max(B) \geq Min(B)$, $Avg(B) \geq Min(B)$ and $Max(B) \geq Avg(B)$. Each of these predicates is added to $L(n)$ if its aggregate terms are attributes of n.

4. If $Avg(B)$ is an attribute of n and $(B \leq c) \in L(n)$, where c is a constant, then add $Avg(B) \leq c$ to $L(n)$. If $(B \geq c) \in L(n)$, then add $(Avg(B) \geq c)$ to $L(n)$.

4.3.3 Predicate pullup through `UNION` and `INTERSECTION` nodes

Consider a `UNION` (or `INTERSECTION`) node n, as shown in Figure 4. We infer new exported predicates of $L(n)$ as follows. Suppose that node n has m children, denoted $c_1, \ldots, c_m$, and let $\bar{D}_i$ be the conjunction of predicates pushed up from c_i. For $1 \leq i \leq m$, we apply to $\bar{D}_i$ the internal renaming from the attributes in $\bar{D}_i$ to the exported attributes of n, and denote the result as D_i. If n is a `UNION` node, we add the CNF form of $D_1 \vee \ldots \vee D_m$ to $L(n)$; if n is an `INTERSECTION` node, we add the predicates $D_1, \ldots, D_m$ to $L(n)$. As in a `SELECT` node, if α is an exported predicate in $L(n)$, then add $\sigma(\alpha)$ to the

[2] Note that when predicates are conjunctions of comparisons (using $<$, $\leq$ and $=$) among constants and ordinary attributes (i.e., no aggregate terms), then the closure can be computed in polynomial time [Ull89].

label of the parent of n, where σ is the external renaming from the exported attributes of n to local attributes of its parent.

Example 4.2 In Figure 5, the labels generated by the pullup phase are shown in bold font. For clarity, the label of the `GROUPBY` node does not show the predicates pulled up from its child. Also, we do not show all the predicates in the closures of labels. □

4.4 Predicate Pushdown

This phase of the algorithm is a generalization of predicate-pushdown techniques. The combination of pullup and pushdown effectively enables us to move predicates from one part of the tree to other parts. In this phase, we traverse the query tree top down, starting from the root. At each node, we infer new predicates on the local attributes from predicates on the exported attributes and push the inferred predicates down into the children nodes. As earlier, the pushdown process depends on the type of the node.

4.4.1 Predicate pushdown through `SELECT` nodes

In a `SELECT` node n, with label $L(n)$, we do as follows.

- Infer new predicates over the local attributes as follows. For each predicate α in $L(n)$, add $\tau(\alpha)$ to $L(n)$ (if it is not already there), where τ is a renaming from the exported attributes of n to the local ones.
- Add to $L(n)$ new predicates that are logically implied by those already in $L(n)$.
- For each child c of n, if α is a predicate in $L(n)$ that includes only constants and renamings of attributes in c, then add $\sigma(\alpha)$ to $L(c)$, where σ is the external renaming from the local attributes of n to the exported attributes of c.

Example 4.3 In our example, we push the predicate ptc.MaxLen $>$ 50 from the root into the `GROUPBY` node, where it is mapped onto the predicate `MAX` (t.Length) $>$ 50 (see Figure 6; predicates added during the pushdown phase are shown in italic; for clarity, we do not show the full closure at each node.) □

4.4.2 Predicate pushdown through `GROUPBY` nodes

The above three steps should also be performed at the `GROUPBY` nodes. However, we also need rules for inferring new predicates from predicates with aggregate terms. Following is a (sound but not complete) set of such rules; these rules should be applied to the label, $L(n)$, of a `GROUPBY` node n (in all these rules, $\leq$ can be replaced with $<$).

- Suppose that $Max(B) \geq c$ is in $L(n)$, where c is a constant. In this case, we only need to look at tuples satisfying $B \geq c$ in order to compute $Max(B)$. However, if there are other aggregates to compute, we may also have to consider tuples that do not satisfy $B \geq c$. Therefore, if $Max(B) \geq c$ is in $L(n)$, we add $B \geq c$ to $L(n)$ provided that $Max(B)$ is the *only* aggregate term in n. As an example, consider the `GROUPBY` node in Figure 6. The predicate MaxLen $>$ 50 is pushed into this node from the root. By renaming into local attributes, we get Max(t.length) $>$ 50. Since Max(t.length) is the only aggregate term in the `GROUPBY` node, we can infer the predicate t.length $>$ 50. Note that by pushing t.length $>$ 50 down, we discover that we only need tuples satisfying t.length $>$ 50 in the view `ptCustomers`, because the maximum of $t.length$ should be greater than 50.
- If $Min(B) \leq c$ is in $L(n)$, where c is a constant, and $Min(B)$ is the *only* aggregate term in n, then we can add $B \leq c$ to $L(n)$.

When $B \geq c$ cannot be inferred from $Max(B) \geq c$, we can use $Max(B) \geq c$ directly in order to optimize the evaluation; however this extension is beyond the scope of this paper.

4.4.3 Predicate pushdown through `UNION` and `INTERSECTION` nodes

Consider a `UNION` (or `INTERSECTION`) node n, as shown in Figure 4, and let c be a child of n. If α is an exported predicate of $L(n)$, then we add $\tau(\alpha)$ to $L(n)$ and to $L(c)$, where τ is the internal renaming from the exported attributes of n to the local attributes of n (which are also the external attributes of c).

4.5 Label Minimization

At the end of the top-down phase, new predicates appear in labels of nodes. As a result, we can apply predicates earlier than was possible in the original tree. There is the possibility, however, of applying predicates redundantly. In fact, even an evaluation of the original tree could result in redundant applications of predicates; this may happen, for example, when the original query is formulated using predefined views and the user is oblivious to the exact predicates that are used in those views (and, hence, he may redundantly repeat the same predicates in the query). In the move-around algorithm, redundancies are introduced in two ways.

- As a result of renamings between attributes of nodes and the associated pullup (or pushdown), some predicate may appear in a node and in the parent of that node (and, possibly, also in other ancestors of that node). There is no need, however, to apply a

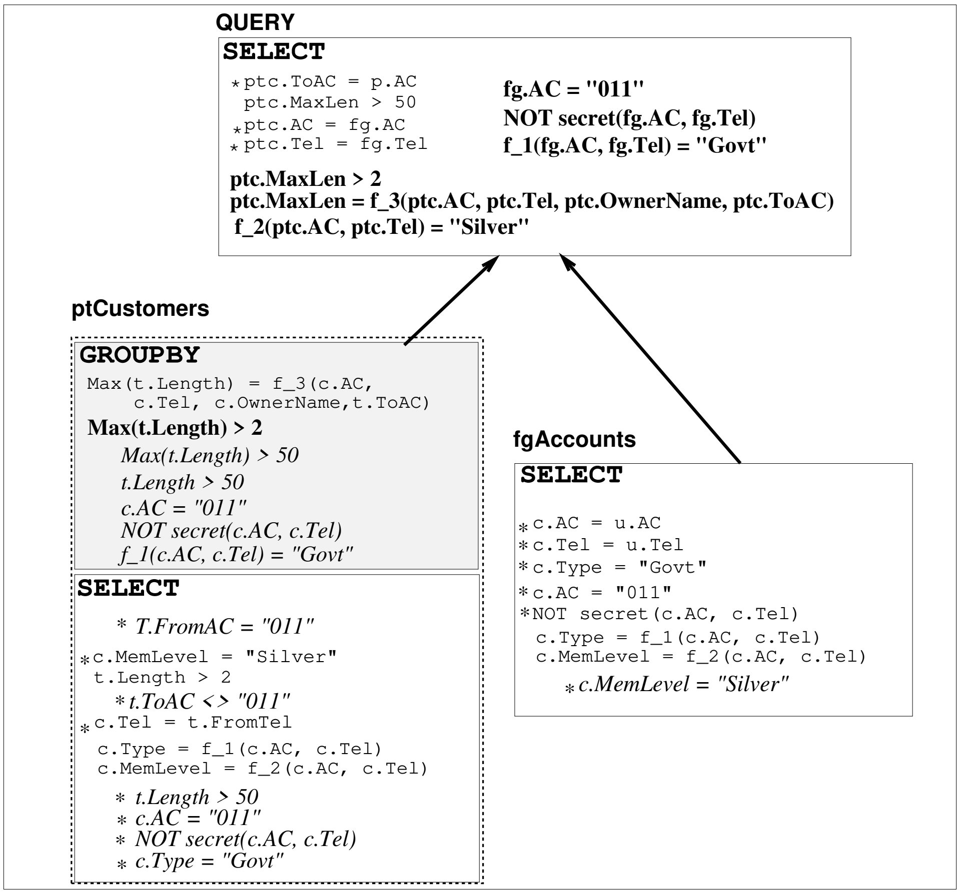

Figure 6: The query tree for Example 2.1 after the pushdown and minimization phases. The predicates in italic font are added during pushdown. Only predicates annotated with a star remain in labels after the minimization phase.

predicate at a node if it has already been applied at a descendant of that node.

- Redundancies are introduced at labels when adding predicates that are logically implied by existing ones.

Removing redundancies is important for two reasons. First, it saves time, since fewer tests are applied during the evaluation of the query. Secondly, redundant predicates might mislead the plan optimizer due to incorrect selectivity estimates.

Redundancies of the first kind are removed as follows. Suppose α is a local predicate of $L(n)$ and that $\sigma(\tau(\alpha))$ is the result of applying to α the internal renaming (to the exported attributes) followed by the external renaming (to the attributes of n's parent). Then a predicate β in the parent of n is redundant if β is logically implied by $\sigma(\tau(\alpha))$. After removing redundancies in this way, we should also discard all predicates that have some exported attributes.

Redundancies of the second kind are removed by the known technique of transitive reduction; we repeatedly remove a predicate from a label if it is implied by the rest of the predicates. We get a nonredundant label when no more predicates can be removed.

Finally, we can completely remove labels of `UNION`, `INTERSECTION` and `GROUPBY` nodes. Moreover, predicates containing functional terms (that were generated from functional dependencies and aggregations) are also dropped from all nodes. In Figure 6, the predicates annotated with a star remain after minimization and form the final labels in our example.

4.6 Translating the Query Tree to SQL

The query tree may be used directly for further rewrite and cost-based optimizations as well as evaluation of the query. In fact, the query tree is similar to the internal representations of queries used by some existing query processors. If desired, however, we can easily translate the query tree back into SQL as follows. `SELECT`, `UNION` and `INTERSECTION` nodes, and `GROUPBY` triplets are translated into the appropriate SQL statements; the `WHERE` and `HAVING` clauses consist of the minimal labels of the corresponding nodes. In our example, the optimized SQL query and views of Figure 2 are the result of applying the above translation to the query tree of Figure 6.

4.6.1 Translating DAG Queries

When a query tree is created from a DAG query, several subtrees of the tree may correspond to the same view. These subtrees are identical at the beginning of the move-around algorithm, but may become different at the end of the algorithm. Consider two subtrees, T_1 and T_2, generated from the same view V. If, at the end of the algorithm, T_1 and T_2 are equivalent (i.e., they have logically equivalent labels in corresponding nodes), then it is sufficient to evaluate just one of T_1 and T_2. If T_1 is contained in T_2, then the view for T_2 may be computed from the view for T_1 by applying an additional selection. If neither one is contained in the other, it may still be possible to compute one view from which the two views can be obtained by additional selections.

4.7 Correctness of the Algorithm

Theorem 4.1 *Let Q be a query and Q' be the rewritten query produced by the predicate move-around algorithm. The queries Q and Q' are equivalent, i.e., they produce the same answer for all databases.* □

Proof: (Sketch:) The proof proceeds by induction on the steps of the algorithm. Let $bu(n)$ and $td(n)$ denote the labels of node n at the end of the pullup and pushdown phases, respectively. A bottom-up induction on the nodes in the query tree shows that any tuple computed at node n must satisfy $bu(n)$. A top-down induction on the nodes of the query tree shows that in order to compute at node n all the tuples that satisfy $td(n)$, it suffices to consider at the children $n_1, \ldots, n_m$ of n only those tuples that satisfy $td(n_1), \ldots, td(n_m)$, respectively. Finally, we show that the label-minimization phase removes only redundant predicates, i.e., predicates that are guaranteed to be applied at lower nodes during the evaluation of the query. □

For queries without aggregation, our algorithm produces an *optimal* query in the following sense. Any attempt to add a predicate to the label of some node either would not change the set of tuples generated at that node or, for some databases, a wrong result would be computed by the query tree. Consequently, predicates are applied as early as possible in the evaluation in the resulting query.

5 Related Work

Our work generalizes predicate-pushdown techniques (e.g., [Ull89]). The main contribution, compared to earlier work, is that we can handle aggregation and other constructs such as `NOT EXISTS` ; our methods can also be extended to recursive queries. In addition, we move predicates both up and down the query tree, thereby enabling predicates from one side of the tree to be moved to applicable places on the other side.

A similar technique for propagating predicates in a query tree was first developed by [LS92] in order to detect and delete redundant Datalog rules. In [LMSS93], this technique was extended to detect satisfiability of Datalog queries in the presence of negated base relations and order predicates. In this paper, we generalize the constraint-propagation techniques of [LS92, LMSS93] to SQL with aggregation operators and other types of constraints (e.g., functional dependencies); in the full paper, we also deal with other SQL constructs (e.g., subqueries). Our optimization algorithm is essentially a rewriting of the query in an optimized form and, hence, is easily implemented on top of existing query optimizers. Finally, by using the termination condition from [LS92] (for terminating the construction of the query-tree), we can extend predicate move-around to recursive SQL queries.

Srivastava and Ramakrishnan [SR92] describe a related technique for pushing predicates in Datalog programs using fold/unfold transformations. Their technique, however, is applicable only when views can be merged and, therefore, cannot be extended to deal with aggregation and relations containing duplicates. Sudarshan & Ramakrishnan [SR91] describe a method for pushing down, through Datalog rules, predicates stemming from aggregate operations. Their method uses a set of rewrite rules and introduces *aggregate selectors* that should be processed directly by the query evaluator; hence, the query evaluator needs to be extended. in order to use their method. In addition, their approach does not combine pushing of other types of predicates with aggregate selectors.

Pirahesh et al. [PHH92] discuss the problem of merging query blocks. Doing so eliminates the need for predicate pushdown. However, that can not always be done (e.g., in the presence of aggregation). Our method can be used in conjunction with the techniques of [PHH92].

Our method complements magic sets [BMSU86, BR87] and GMST [MFPR90b, MFPR90a]. The key differences from the magic-set approach are as follows. First, the magic-set transformation depends upon the join order; it can move predicates up from a relation and then down into another relation that appears later in the join order. In contrast, predicate move-around does not depend upon the join ordering; predicates can be moved up from every relation and down into any other relation. Secondly, predicate move-around pushes predicates defined on single relations (also known as *local predicates*), while the magic-set transformation can also push join predicates defined across relations (however, it introduces auxiliary magic relations even when moving only local predicates). It is, therefore, much better to move local predicates using the predicate move-around algorithm, since we can do so without determining the join order and, moreover, can move predicates in all directions, without creating an additional overhead of auxiliary predicates. Furthermore, doing predicate move-around improves the ability to determine the optimal join order. The magic-set transformation can be applied after predicate move-around in order to move join predicates in the direction of the join order.

There has been a lot of work on optimizing subqueries and eliminating correlations [Kim82, GW87, Day87, Mur92]. Our technique complements well with that work by providing a new powerful means of pushing predicates after correlations are removed.

A predicate is said to be *expensive* if the cost of applying the predicate is high. Placement of expensive predicates has been studied by [HS93, Hel94]. The move-around algorithm, as presented above, assumes that predicates are inexpensive. However, expensive predicates can be handled by modifying the label-minimalization phase.

6 Conclusions

We have described a very general technique for moving predicates around in a query, thus determining the earliest point when predicates can be applied. Our method can handle hierarchical and dag queries. The predicates moved around include arithmetic comparisons, negative predicates (`NOT EXISTS` and `EXCEPT`), functional dependencies and aggregation constraints. Furthermore, we can also handle the `LIKE` predicate of SQL [ISO93] (in a fashion similar to equality) and arithmetic constraints (e.g., $X = Y + Z$). When moving predicates, we can also consider the constraints that hold in database relations. For example, if it is known that the range of an attribute A of a relation R is between 0 and 10, we insert the predicates $r.A \geq 0$ and $r.A \leq 10$ in the label of any node that refers to R.

In many cases, the result of the predicate move-around algorithm is optimal (in the sense that predicates are moved to all parts of the query in which they are applicable). In particular, optimality is guaranteed for queries without aggregation. Achieving an optimal result for queries involving aggregation requires a better understanding of techniques for reasoning about aggregation constraints, which is a subject of current research. The work of [SRSS94] is a first step in that direction. The query-tree technique is a general algorithm and is easily extensible to new kinds of predicates and operators, including recursive SQL queries.

Predicate move-around is a generalization of predicate pushdown techniques. When predicate move-around detects optimizations that cannot be found by ordinary pushdown techniques, the additional savings may be arbitrarily large, depending upon the selectivity of the predicates being moved. Furthermore, such savings are very likely to be discovered in complex queries, such as those encountered in decision-support applications. A significant aspect of the improved performance of predicate move-around is the ability to deal with aggregation operators, which are a major cause for poor performance of current optimizers. The move-around algorithm is easy to implement and can simply replace an existing pushdown module in a query optimizer.

Acknowledgements

We thank Brian Hart and Andrew Witkowski for discussions on the topics discussed here and for comments on earlier drafts of this paper.

References

[BMSU86] F. Bancilhon, D. Maier, Y. Sagiv, and J. Ullman. Magic sets and other strange ways to implement logic programs. In *PODS 1986*, pages 1–15.

[BR87] C. Beeri and R. Ramakrishnan. On the power of magic. In *PODS 1987*, pages 269–283.

[Day87] U. Dayal. Of nests and trees: A unified approach to processing queries that contain nested subqueries, aggregates, and quantifiers. In *VLDB 1987*, pages 197–208.

[GW87] R. Ganski and H. Wong. Optimization of nested SQL queries revisited. In *SIGMOD 1987*, pages 23–33.

[Hel94] J. Hellerstein. Practical predicate placement. In *SIGMOD 1994*.

[HS93] J. Hellerstein and M. Stonebraker. Predicate migration: Optimizing queries with expensive predicates. In *SIGMOD 1993*.

[ISO93] ISO_ANSI. ISO-ANSI working draft: Database language SQL3, 1993.

[Kim82] W. Kim. On optimizing an SQL-like nested query. *ACM TODS*, 7(3), September 1982.

[LMSS93] A. Levy, I. Mumick, Y. Sagiv, and O. Shmueli. Equivalence, query-reachability, and satisfiability in Datalog. In *PODS 1993*, pages 109–122.

[LS92] A. Levy and Y. Sagiv. Constraints and redundancy in datalog. In *PODS 1992*, pages 67–80.

[MFPR90a] I. Mumick, S. Finkelstein, H. Pirahesh, and R. Ramakrishnan. Magic is relevant. In *SIGMOD 1990*, pages 247–258.

[MFPR90b] I. Mumick, S. Finkelstein, H. Pirahesh, and R. Ramakrishnan. Magic conditions. In *PODS 1990*, pages 314–330.

[Mur92] M. Muralikrishna. Improved unnesting algorithms for join aggregate SQL queries. In *VLDB 1992*, pages 91–102.

[PHH92] H. Pirahesh, J. Hellerstein, and W. Hasan. Extensible/rule based query rewrite optimization in starburst. In *SIGMOD 1992*, pages 39–48.

[SR92] D. Srivastava and R. Ramakrishnan. Pushing Constraint Selections. In *PODS 1992*.

[SRSS94] D. Srivastava, K. Ross, P. Stuckey and S. Sudarshan. Foundations of Aggregation Constraints. In *PPCP 1994*.

[SR91] S. Sudarshan and R. Ramakrishnan. Aggregation and Relevance in Deductive Databases. In *VLDB 1991*.

[Ull89] J. Ullman. Principles of Database and Knowledge-Base Systems, Volumes 1 and 2. Computer Science Press, 1989.