Cost-Based Query Transformation in Oracle

Rafi Ahmed, Allison Lee, Andrew Witkowski,

Dinesh Das, Hong Su, Mohamed Zait, Thierry Cruanes

Oracle USA
500 Oracle Parkway
Redwood Shores, CA 94065, U.S.A.
First.Last@oracle.com

ABSTRACT

This paper describes cost-based query transformation in Oracle relational database system, which is a novel phase in query optimization. It discusses a suite of heuristic- and cost-based transformations performed by Oracle. It presents the framework for cost-based query transformation, the need for such a framework, possible interactions among some of the transformations, and efficient algorithms for enumerating the search space of cost-based transformations. It describes a practical technique to combine cost-based transformations with a traditional physical optimizer. Some of the challenges of cost-based transformation are highlighted. Our experience shows that some transformations when performed in a cost-based manner lead to significant execution time improvements.

1. INTRODUCTION

Current relational database systems process complex SQL queries involving nested subqueries with aggregation functions, union/union-all, distinct, and group-by views, etc. Such queries are becoming increasingly important in Decision-Support Systems (DSS) and On-Line Analytical Processing (OLAP). Query rewrite has been proposed as a heuristic transformation to optimize such queries. However, there can be many possible variants of transformations even for a simple SQL statement with respect to which and how these transformations are to be applied. There is an added complexity when two or more transformations interact with each other.

In traditional relational database systems, query optimization generally consists of two phases of processing: logical and physical optimization phases. In the logical optimization phase, the given query is rewritten, generally based on heuristics or rules, into an equivalent but potentially more declarative and optimal form. The traditional physical optimizer works within the scope of a single query block, which ranges over a set of base tables with restriction, projection, and join. In the physical optimization phase, access methods, join orders, and join methods are chosen in order to generate an efficient plan for executing the query.

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, to post on servers or to redistribute to lists, requires a fee and/or special permission from the publisher, ACM.
VLDB '06, September 12–15, 2006, Seoul, Korea.
Copyright 2006 VLDB Endowment, ACM 1-59593-385-9/06/09

Query transformations have been implemented either as a rewrite system [16] driven by heuristic rules or as an extension to the plan generation within the physical optimizer [2], [6], [19]. The first approach is general but it does not scale in complex commercial systems, and the second is easily applicable only to a few selected transformations. This work argues for a new approach with a systematic provision for a costing of transformations without the need for modifying the physical optimizer.

Some heuristic rules are essential for transformations to be applied in the logical phase of optimization. In Section 2, we enumerate several transformations that are applied in Oracle based on heuristic rules. However, there remains a large class of transformations, which does not lend itself well to heuristic rules, and may produce sub-optimal execution plans, if the decision to apply these transformations is not made in a cost-based manner.

We motivate these problems with several examples in Section 2. But consider here a simple query, which retrieves employee information and job titles held after 1998 for employees who have an above average salary in their departments that are located in the U.S.

```
Q1
SELECT e1.employee_name, j.job_title
FROM employees e1, job_history j
WHERE e1.emp_id = j.emp_id and
  j.start_date > '19980101' and
  e1.salary >
 (SELECT AVG (e2.salary)
  FROM employees e2
  WHERE e2.dept_id = e1.dept_id) and
        e1.dept_id IN
 (SELECT dept_id
  FROM departments d, locations l
  WHERE d.loc_id = l.loc_id and
        l.country_id = 'US');
```

In this query, both subqueries can be unnested by introducing inline views (derived tables), and further both views can be merged. This presents us with multiple alternatives in which this query can be transformed: unnesting zero or more subqueries and then merging zero or more views. Interestingly, as we will see in Section 2, there can be other transformations that may become applicable to this query. There are no clear heuristic rules to decide which transformations lead to a better execution plan, since a variety of factors (e.g., relative sizes of the tables involved,

selectivity of predicates, existence of indexes on the local columns in the correlation predicates, etc.) come into play.

2. TRANSFORMATION IN ORACLE

Oracle performs a multitude of query transformations, some of which are cost-based and others are heuristic-based. In this section, we discuss several transformations in both the categories, and explain why certain transformations require a cost-based decision and others do not. It should be noted that this is only a subset of transformations performed by Oracle.

Most of our discussion of query transformation is centred on *query trees*, which are different from algebraic *operator trees* in that query trees retain all the declarativeness of SQL. A query tree is converted into an operator tree when it undergoes physical optimization.

2.1 Heuristic Transformations

The traditional heuristic-based transformations are performed with the objectives of doing early evaluation of selection and projection, pruning of redundant operations, and minimizing of query blocks.

Minimizing the number of query blocks by merging them with other query blocks removes restrictions from the set of join permutations that can be generated and thereby allows more tables to be reordered together [16]. A traditional relational optimizer generates only a left-deep (linear) execution tree; it can, therefore, be argued that it will never be able to generate a plan that is equivalent to a bushy-tree plan generated in the untransformed case. However, it is generally believed that a significant fraction of the join trees with low processing cost is to be found in the space of left-deep trees [5].

Generally, minimizing the number of query blocks proves to be a good heuristic as long as it does not require introducing, replicating or re-positioning of distinct or group-by operator. In Oracle, many transformations are driven by heuristics, which we call *imperative* rules, since they always lead to the application of transformations, if they are legal. We briefly describe some of these transformations here.

2.1.1 Subquery Unnesting

Subquery unnesting is an important transformation commonly done by commercial database systems. A non-unnested subquery is evaluated multiple times using the tuple iteration semantics (TIS), which is similar to performing a nested-loop join, and thus many efficient access paths and join orders cannot be considered.

Oracle optimizer can unnest almost all types of subqueries except those that are correlated to non-parents, whose correlations appear in disjunction, or some ALL subqueries with multi-item connecting condition containing null-valued columns. There are two broad categories of unnesting: one, unnesting that generates inline views; two, unnesting that merges a subquery into its outer query. In Oracle, the former is applied in a cost-based manner while the latter is done imperatively.

Consider the following query that retrieves department information for departments that have employees with high salaries.

```
Q2
SELECT d.dept_name, d.budget
FROM departments d
WHERE EXISTS (SELECT 1
              FROM employees e
              WHERE d.dept_id = e.dept_id
                    and e.salary > 200000;
```

This query is transformed into the following equivalent query[1].

```
Q3
SELECT d.dept_name, d.budget
FROM departments d, employees e
WHERE d.dept_id S= e.dept_id and
      e.salary > 200000;
```

Unnesting that merges subqueries allows us to consider additional join methods and additional join orders in the general case. In this example, Oracle can use nested loop as well as hash and sort-merge for the semijoin. Oracle's implementation of antijoin and semijoin has the stop-at-the-first-match property. The Oracle execution engine caches the results of antijoin and semijoin for the tuples in the left table; this caching can be quite beneficial, if there are a large number of duplicates in the joining columns of the left table.

Semijoin, like antijoin and left outerjoin, is a non-commutative join and imposes a partial order on the joining tables; that is, in this example, departments must precede employees in the join order. But we can convert this semijoin into an inner join by applying a sort distinct operator on the selected rows of employees and by relaxing the partial join order restriction [22]. This allows both the join orders – (d semijoin e) and (distinct(e) join d) – to be considered by the optimizer. In Oracle, this transformation has been incorporated into the physical optimizer.

ALL subqueries with non-null connecting conditions and NOT EXISTS subqueries can generally be unnested using antijoin. The next release of Oracle will have another variant of antijoin, a *null-aware antijoin*, which can deal with null-valued columns that appear in the connecting conditions of ALL subqueries.

2.1.2 Join Elimination

Join elimination removes a table from a query, if there are constraints on the join columns of the table, such that the join has no impact on the query results. Consider the following two queries, Q4 and Q5, which retrieve some information about employees:

```
Q4
SELECT e.name, e.salary
FROM employees e, departments d
WHERE e.dept_id = d.dept_id;
```

Since dept_id in employees is a foreign key that references the primary key of departments, the join with departments can be eliminated from Q4.

In the query Q5, the joining column of the right table in the outerjoin is unique. Since outerjoin retains all the tuples of the left table and an equi-join on unique columns does not generate duplicates, the DEPARTMENTS table can be eliminated.

[1] Note that we here use a non-standard notation for semijoin, T1.C S= T2.C, where T1 and T2 are the left and right tables respectively in the semijoin.

```
Q5
SELECT e.name, e.salary
FROM employees e left outer join
departments d on e.dept_id = d.dept_id;
```

Applications of join elimination transformation on Q4 and Q5 produce Q6. If in Q4, e.dept_id can return nulls, a predicate "e.dept_id is not null" must be added to the where clause of Q6.

```
Q6
SELECT e.name, e.salary
FROM employees e;
```

It is obvious that pruning a redundant join will improve the performance of the query, and therefore join elimination is always performed by Oracle, if it is valid.

2.1.3 Filter Predicate Move Around

Filter predicate move around moves inexpensive predicates into view query blocks in order to perform filtering earlier. The filter predicate move around transformation is done with imperative heuristics, since it can open up new access path and reduce the size of data that is processed later by more expensive operations such as joins and aggregation.

Filter predicates can be pulled up, moved across, and pushed down to any levels. An interesting extension to [9] is our capability to push filter predicates through the PARTITION BY of ANSI window functions and ANSI partitioned outerjoin, and through the DIMENSION BY sub-clause of the SQL MODEL clause [25], [26]. Another technique unique to our system is pushing predicates through an aggregate of a window function before evaluating the aggregate [26]. Currently it is applied only within the SQL MODEL clause, but in the next release will be generalized to window functions present in the SELECT list of a query block. For example, consider the table accounts (acct-id, time, balance) recording current balance over time. Consider an inline view calculating a running average balance and an outer query selecting it for the first 12 months for account 'ORCL'.

```
Q7
SELECT acct-id, time, ravg
FROM (SELECT acct-id, time,
     AVG(balance) OVER (PBY acc-id OBY time
         RANGE BETWEEN UNBOUNDED PROCEEDING
         AND CURRENT ROW) ravg
      FROM accounts)
WHERE acct-id = 'ORCL' AND time <= 12;
```

In this query predicates on acct-id and time can be pushed inside the view.

```
Q8
SELECT acct-id, time, ravg
FROM (SELECT  acct-id, time,
     AVG(balance) OVER (PBY acc-id OBY time
         RANGE BETWEEN UNBOUNDED PROCEEDING
         AND CURRENT ROW) ravg
      FROM accounts
      WHERE acct-id = 'ORCL' AND time<=12);
```

Pushing predicates on PARTITION BY (PBY) clauses can always be done. Pushing through ORDER BY (OPBY) requires analysis [26] of the affected range of the window function.

2.1.4 Group Pruning

Group pruning is another imperative transformation, which removes from views groups not needed in the outer query blocks. For example, consider the query Q9.

```
Q9
SELECT sum_sal,country_id,state_id,city_id
FROM (SELECT SUM (e.salary) sum_sal,
      l.country_id, l.state_id, l.city_id
      FROM employees e,departments d,
           locations l
      WHERE e.dept_id = d.dept_id and
            d.location_id = l.location_id
      GROUP BY ROLLUP (country_id,state_id,
                       city_id)
WHERE city_id = 'San Francisco';
```

In the above query, the outer predicate on the city_id filters groups (country_id) and (country_id, state_id). This transformation can be done based on predicates on the group-by columns as well as grouping functions. This transformation is performed after filter predicate move around to move pruning predicates into close proximity to the group-by query.

2.2 Cost-Based Transformations

Here we briefly discuss some of the transformations that are performed by Oracle in a cost-based fashion.

2.2.1 Subquery Unnesting

In case of multi-table EXISTS or ANY subquery, it is generally not possible to simply merge the subquery into its containing query block, since it may have the undesirable effect of generating duplicate rows that were not present in the result of the untransformed query. For a multi-table NOT EXISTS or ALL subquery, it is also not possible to simply merge the subquery into its containing query, since the join of the tables in the subquery must be performed before the antijoin with outer tables. In these cases, an inline view containing the subquery tables must be generated. Unnesting of correlated subqueries containing aggregates also requires generation of inline group-by views. Consider the query Q1 shown before.

```
Q1
SELECT e1.employee_name, j.job_title
FROM employees e1, job_history j
WHERE e1.emp_id = j.emp_id and
  j.start_date > '19980101' and
  e1.salary >
    (SELECT AVG (e2.salary)
     FROM employees e2
     WHERE e2.dept_id = e1.dept_id) and
  e1.dept_id IN
    (SELECT dept_id
     FROM departments d, locations l
     WHERE d.loc_id = l.loc_id and
           l.country_id = 'US');
```

Consider the transformed query Q10 where the first subquery has been unnested by generating an inline view.

```
Q10
SELECT e1.employee_name, j.job_title
FROM employees e1, job_history j,
    (SELECT AVG(e2.salary) avg_sal, dept_id
     FROM employees e2
     GROUP BY dept_id) V
WHERE e1.emp_id = j.emp_id and
      j.start_date > '19980101' and
      e1.dept_id = V.dept_id and
      e1.salary > V.avg_sal and
      e1.dept_id IN
     (SELECT dept_id
      FROM departments d, locations l
      WHERE d.loc_id = l.loc_id
      and l.country_id = 'US');
```

The untransformed query Q1 may perform better under TIS, if the outer query block significantly filters the number of employee tuples for which above average salary needs to be computed; further, TIS can be quite efficient if the local column (i.e., e2.dept_id) in the correlation predicate has an index. On the other hand, the transformed query allows for different join orders and join methods to be considered, and it needs to compute the aggregation and group-by operator only once. Therefore the decision to unnest such subqueries must be cost-based.

The cost-based transformation was introduced in Oracle 10g. Unnesting that generates inline views was heuristic-based in releases prior to Oracle 10g. A somewhat simplified version of that heuristic rule can be given as the following: If there exist filter predicates in the outer query and there are indexes on the local columns in the subquery correlation, then the subquery should not be unnested.

In Section 4.1, we present the performance results of cost-based versus heuristic-based transformations.

2.2.2 Group-By and Distinct View Merging

Group-by view[2] merging (group-by pull-up) allows the view containing a group-by (or distinct) operator to be merged into its outer query block. This allows the optimizer to consider additional join orders and access paths, and delay the evaluation of aggregates until after the joins have been evaluated. Delayed evaluation of aggregates can make performance better or worse depending on the characteristics of the data, such as the reduction in the data set on which aggregation has to be performed. Consider the query Q11, which has been produced by merging the group-by view in Q10.

```
Q11
SELECT e1.employee_name, j.job_title
FROM employees e1, job_history j,
     employees e2
WHERE e1.emp_id = j.emp_id and
      j.start_date > '19980101' and
      e2.dept_id = e1.dept_id and
      e1.dept_id IN
     (SELECT dept_id
      FROM departments d, locations l
      WHERE d.loc_id = l.loc_id
      and l.country_id = 'US')
GROUP BY e2.dept_id, e1.emp_id, j.rowid,
         e1.employee_name, j.job_title,
         e1.salary
HAVING e1.salary > AVG (e2.salary);
```

The untransformed query Q10 requires aggregation to be performed on the entire employees table. The transformed query Q11 performs the joins with job_history and two employees tables and applies the filtering from the second subquery before aggregation is performed. If the joins and filters significantly reduce the size of the data to be aggregated, this could result in an improved execution plan. On the other hand, early aggregation reduces the size of the data to be processed by join operations and the aggregation may need to be performed on a smaller data set in the group-by view. These tradeoffs are the reason why this decision must be cost-based. In queries where the aggregation is performed late, we also consider transforming the query to perform early aggregation, using the group-by placement transformation, which we describe later.

2.2.3 Join Predicate Pushdown

In this transformation, join predicates are pushed down inside a view. The pushed-down join predicates act like correlation once inside the view, thereby opening up new access paths. Join predicate pushdown allows a view to be joined with outer tables by index-based nested-loop join, which is not possible for regular views that can be joined only by hash or sort-merge join methods. The transformation imposes a partial order on the joined tables such that the tables that the view is joined to (via the pushed-down predicates) must precede the view, and the view must be joined by nested-loop.

As an additional optimization, the group-by operator can be removed when join predicates are pushed into a group-by view, if the view has equi-joins on all its group-by items and all those join predicates are valid for pushdown. This is possible because correlation on equality conditions acts as a grouping on those column values. A similar optimization is also possible on distinct views, as shown in query Q13.

Consider the following query, which retrieves employee information and their job histories for employees who work in departments located in the U.K. or in the U.S.

```
Q12
SELECT e1.employee_name, j.job_title
       e2.employee_name as mgr_name
FROM employees e1, job_history j,
     employees e2,
     (SELECT DISTINCT dept_id
      FROM departments d, locations l
      WHERE d.loc_id = l.loc_id and
            l.country_id IN ('UK','US')) V
WHERE e1.emp_id = j.emp_id and
      j.start_date > '19980101' and
      e1.mgr_id = e2.emp_id and
      e1.dept_id = V.dept_id;
```

[2] Note that select distinct is a specific case of the group-by operator.

The query Q12 has been transformed by join predicate pushdown into Q13. This transformation allows us to remove the expensive distinct operator from the view. The inner join is converted internally into a semijoin (this is not shown in the query), which imposes a partial join order where e1 must precede V.

```
Q13
SELECT e1.employee_name, j.job_title
       e2.employee_name as mgr_name
FROM employees e1, job_history j,
     employees e2,
     (SELECT dept_id
      FROM departments d, locations l
      WHERE d.loc_id = l.loc_id and
            l.country_id IN ('UK','US') and
            e1.dept_id = d.dept_id) V
WHERE e1.emp_id = j.emp_id and
      j.start_date > '19980101' and
      e1.mgr_id = e2.emp_id;
```

In Q13, nested-loop can be used for joining the view V; this can be quite efficient if d.dept_id has an index and the number of tuples in the outer query is relatively small. However, a cost-based decision is required to determine which of the queries Q12 and Q13 will produce a more optimal execution plan.

In Oracle join predicate pushdown can be applied on mergeable (e.g., distinct and group-by) views and unmergeable[3] (e.g., union/union-all, anti-/semi-/outer-joined) views.

2.2.4 Group-By Placement

The group-by pushdown [1], [24] transformation pushes the group-by operator down past joins in the query, thereby performing an early evaluation of the group-by operation. Doing an early group-by evaluation may result in a significant reduction in the number of rows on which multiple group-by operators apply as well as in the number of rows later used in the join; hence the overall performance of the query may improve.

Group-by placement can also allow pulling of the group-by operator up past the joins, which we call group-by view merging. A group-by query can undergo different types of group-by placement transformations depending upon its join graph and tables that are referenced in aggregate functions. In Oracle, group-by placement (group-by pushdown) transformation generates one or more group-by views.

The Oracle optimizer first performs group-by view merging (group-by pullup), which is followed by group-by pushdown transformation by-passing those queries that have undergone group-by view merging. A general group-by placement transformation will be available in the next release of Oracle.

2.2.5 Join Factorization

Join factorization applies to UNION/UNION ALL queries where the branches of the UNION ALL contain common join tables. These join tables are pulled out into the containing query block and the UNION ALL query block is turned into a view to which the pulled-out tables are joined. This factorization avoids accessing the common tables multiple times. Using join factorization, query Q14 can be transformed into query Q15.

```
Q14
SELECT e.first_name, e.last_name, job_id,
       d.department_name, l.city
FROM employees e, departments d,
     locations l
WHERE e.dept_id = d.dept_id and
      d.location_id = l.location_id
UNION ALL
SELECT e.first_name, e.last_name, j.job_id,
       d.department_name, l.city
FROM employees e, job_history j,
     departments d, locations l
WHERE e.emp_id = j.emp_id and
      j.dept_id = d.dept_id and
      d.location_id = l.location_id;
Q15
SELECT V.first_name, V.last_name, V.job_id,
       d.department_name, l.city
FROM departments d, locations l,
    (SELECT e.first_name, e.last_name,
            e.job_id, e.dept_id
     FROM employees e
     UNION ALL
     SELECT e.first_name, e.last_name,
             j.job_id, j.dept_id
     FROM employees e, job_history j
     WHERE e.emp_id = j.emp_id) V
WHERE d.dept_id = V.dept_id and
      d.location_id = l.location_id;
```

Interestingly, there are many cases where the common tables can be factorised out but the corresponding join predicates cannot be pulled out. In such cases, the join predicates can be left inside the UNION ALL view, which is then joined by the technique described in the join predicate pushdown section. This transformation will be available in the next release of Oracle.

2.2.6 Predicate Pullup

Filter predicate pullup transformation pulls expensive filter predicates up from the originating view to the containing query of the view. Currently, a predicate is considered expensive if it contains procedural language functions, user-defined operators, or subqueries. The predicate pullup transformation is currently only considered when a rownum[4] predicate is specified in the containing query of the view, and the view contains a blocking operator.

Consider the following query, which contains a view with two expensive predicates.

```
Q16
SELECT  *
FROM (SELECT document_id
      FROM product_docs
```

[3] In some cases, outer/-anti/-semi-joined views can, in fact, be merged, especially if they contain a single table.

[4] Rownum is an Oracle construct that allows users to specify the maximum number of tuples to be retrieved.

```
      WHERE contains(summary,'optimizer',1)
            > 0
      AND contains(full_text,'execution',2)
          > 0
      ORDER BY create_date) V
WHERE rownum < 20;
```

Since there are two expensive predicates in the view, there are three ways the predicate pull-up transformation can be applied, one of which is shown below.

```
Q17
SELECT  *
FROM (SELECT document_id, value(r) AS vr
      FROM product_docs
      WHERE
contains(full_text,'execution',2)> 0
      ORDER BY create_date) V
WHERE contains(summary, 'optimizer', 1) > 0
      AND rownum < 20;
```

Doing a late evaluation of expensive predicates on a significantly reduced data set may, in some cases, improve the performance of the query. If the predicate filters out very few rows, then we can avoid executing the expensive predicate for the full data set. The reduction comes from the presence of a rownum predicate in the containing query. This transformation involves the reverse operation of the common optimization technique of filter predicate pushdown into views. Therefore, in Oracle, the decision to pullup predicates is done in a cost-based manner.

2.2.7 Set Operators Into Join

The set operators MINUS and INTERSECT are converted into antijoin and inner-join/semijoin respectively thereby allowing various join methods and join orders to apply. However, there is a difference in semantics between these set operations and joins; in both INTERSECT and MINUS null values match, whereas in join and antijoins they do not. Further, MINUS and INTERSECT are set operators and therefore they return a duplicate-free result set. A cost-based decision must be made as to whether duplicates should be removed at the inputs or the output of the joins; this problem is similar to that of distinct placement.

2.2.8 Disjunction Into Union All

When filter or join predicates appear in disjunction, the query can be expanded into a UNION ALL query, where each branch contains one of the predicates in the disjunction. In the absence of such a transformation, the disjunctive predicate is applied as a post-filter to the result, which may be a Cartesian product.

3. FRAMEWORK FOR COST-BASED TRANSFORMATION

3.1 Basic Components

In cost-based transformation, logical transformation and physical optimization are combined to generate an optimal execution plan. This is illustrated in Figure 1.

Logical transformation can be thought of as having two distinct components: heuristic-based and cost-based transformations. The cost-based transformation framework comprises the following:

- Transformation algorithms that convert a complete or partial query tree into a semantically equivalent form
- State spaces for various transformations
- State-space search algorithms
- Capability for deep copying query blocks and their constituents
- Cost estimation technique (physical optimizer)
- Transformation directives and cost annotations

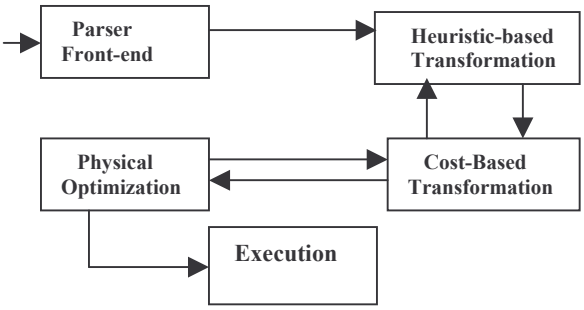

Figure 1. Oracle Query Processing

Different transformations apply on different elements of a query tree. For example, unnesting and view merging apply on subqueries and views respectively; group-by placement applies on the nodes of a join graph. The predicate pull-up transformation applies on expensive predicates.

A query tree is traversed in a bottom-up manner during optimization. Various alternatives for applying one or more transformations on the elements in the query tree generate different states in the state space of the transformation. A deep copy of the (partial) query tree is made before applying a particular state and estimating its cost by invoking the physical optimizer. The evaluation of each state generally requires a different copy of the (partial) query tree. The transformation state that generates the most optimal plan (i.e., the best state) is selected and the directives for the best state are transferred to the original query tree, which is transformed according to these directives.

In Oracle, transformations are generally applied in a sequential manner; that is, each transformation is applied on the entire query tree followed by another transformation, and so on. The sequential order followed for some of the transformations is the following: common sub-expression factorization, SPJ view merging, join elimination, subquery unnesting, group-by (distinct) view merging, group pruning, predicate move around, set operator into join conversion, group-by placement, predicate pullup, join factorization, disjunction into union-all expansion, star transformation, and join predicate pushdown.

However, there are cases where this sequential ordering of transformation is not followed. A transformation can generate constructs, which may necessitate other transformations to be re-applied; for example, conversion of set operator into join may generate an SPJ view and therefore SPJ view merging and filter predicate pushdown may be applied again. Some transformations interact with one another and need to be considered together so that an accurate cost-based decision can be made; this is discussed in Section 3.3.

3.2 State Space Search Techniques

A fundamental question related to cost-based transformation is whether these transformations will lead to a combinatorial explosion of alternatives that need to be evaluated and whether they will provide a trade-off between optimization cost and execution cost.

The source of multiple alternatives is the various transformations themselves as well as the set of objects (e.g., query blocks, tables, join edges, predicates, etc.) on which each transformation may apply. If there are N objects on which a transformation T can apply, then 2^N possible alternative combinations can potentially be generated by the application of T. In general, if there are M transformations, T_1, T_2, ..., T_M, which may apply on N objects, then there are $(1+M)^N$ possible alternative combinations.

For instance, in query Q1, there are four alternatives to consider: no unnesting, unnesting the first subquery (Q_{S1}) only, unnesting the second subquery (Q_{S2}) only, or unnesting both subqueries. We denote a state as an array of bits, where the *n*th bit represents whether the *n*th object (e.g., subquery) is transformed (a value of 1) or not transformed (a value of 0). For instance, the state (0,1) refers to unnesting the second subquery only. When there are M transformations that apply on N objects, the state is represented by an MxN bit matrix.

To cope with combinatorial explosion problem in the case of join permutations, the use of randomised algorithms, such as Simulated Annealing, Genetic Search, Iterative Improvement, and Tabu Search [20], [21], [11] have been proposed. The common idea behind all these strategies is to perform a quasi-random walk in the state space, starting from an initial state and trying to reach a low cost local minimum. Of course, these strategies do not guarantee that the global minimum – the best transformation – can be attained, since only a small fraction of the state space is visited during the walk. Nevertheless, they are of practical interest, since the quality of the solution generally improves with the length of search.

The complexity of cost-based transformation is determined by the number of alternative combinations, the *state space*, which exponentially grows with the number of transformation objects. When the number of transformation objects is small, an enumerative transformation technique with an exhaustive search of the state space may be feasible. In order to limit the potential increase in optimization time, we use several different techniques for enumerating the state space:

- *Exhaustive.* In exhaustive search, all possible 2^N states of the state space for N objects are considered. For example in query Q1, we consider four states: (0,0), (0,1), (1,0), and (1,1). This search is guaranteed to provide the best solution.
- *Iterative.* An iterative improvement technique is used to prune the search space. The general idea in this technique is that we start from an initial state and move to the next neighbouring state by using some method looking for a local minimum by always choosing a downward move; we repeat this search for a local minimum starting with a different initial state in the next iteration. The algorithm stops, if there are no more new states to be found or some terminating condition (i.e., the maximum number of states) has been reached. The number of states enumerated in this technique falls between N+1 and 2^N.
- *Linear.* The underlying idea of this search technique is based on a *dynamic programming* approach, which assumes that for a query consisting of several objects, it suffices to consider only a subset of those objects for transformation and then extend that with additional transformation of another object. In other words, if Cost(1,0) is lower than Cost(0,0), and Cost(1,1) is lower than Cost(1,0), then it is safe to assume that Cost(1,1) is the lowest of the costs of all possible transformations, and thus there is no need to evaluate Cost(0,1). As can be seen, this can significantly cut down the search space. This technique considers N+1 states. Linear search works best when the transformations on different elements are independent of one another.
- *Two-pass*. Two-pass search is the least expensive search technique, where we consider 2 states. We compare the cost of not transforming any of the objects (i.e. the state (0,0,…)) versus the cost of transforming all of the objects (i.e. the state (1,1,…)).

The cost-based transformation framework automatically decides which search technique to use, based on the number of objects to be transformed in the query block, characteristics of the transformation, and the overall complexity of the query. For instance, if a query block contains a small number of subqueries, we use exhaustive search for subquery unnesting, but if the number exceeds a fixed threshold, we use linear search. If the total number of elements subject to transformation (e.g. group-by views, subqueries which can be unnested) in a query exceeds a threshold, then we use two-pass search for all transformations in the query.

3.3 Interaction between Transformations

3.3.1 Interleaving

When two (or more) cost-based transformations apply on the same object such that one transformation becomes applicable only after the other has been applied, then these transformations must be interleaved in order for the optimizer to determine the optimal plan.

For example, unnesting and view merging must be interleaved in some cases, since unnesting might increase the estimated cost of the query; however, the view merging transformation when applied to the view generated in the process of unnesting the subquery may yield an optimal plan indicating that unnesting should be performed on the query and the view so generated must be merged. In query Q10, the aggregate subquery has been unnested into a group-by view. This transformation may be less optimal (that is, Q1 may be cheaper than Q10). However, when view merging is applied on Q10 it yields Q11, which may be less expensive than both Q1 and Q10. In the absence of interleaving of view merging with unnesting, the unnesting transformation will not be applied on Q1 and a sub-optimal plan will be chosen.

3.3.2 Juxtaposition

When two or more cost-based transformations apply on the same object in a way that precludes their sequential application, they must be applied one by one in order for the optimizer to determine the most optimal plan. This comparison of two or more cost-based transformations is called *juxtaposition*.

View merging and join predicate pushdown transformations must be juxtaposed with each other, if they can apply to a view, and the

optimizer must chose among the three options. Consider the example of join predicate pushdown given in Q13. The distinct view in Q12 can be merged to produce the following query.

```
Q18
SELECT DV.employee_name, DV.job_title,
       DV.mgr_name
FROM
(SELECT DISTINCT e1.emp_id, e2.emp_id,
        j.rowid,
        e1.employee_name, j.job_title,
        e2.employee_name as mgr_name
 FROM employees e1, job_history j,
     employees e2, departments d,
     locations l
 WHERE d.loc_id = l.loc_id and
      l.country_id IN ('UK','US') and
      e1.dept_id = d.dept_id and
      e1.emp_id = j.emp_id and
      j.start_date > '19980101' and
      e1.mgr_id = e2.emp_id) DV;
```

Note that in this view merging the distinct operator has been pulled up and keys of the outer tables have been added to the new view, which contains all the tables of the original query Q12.

The costs of queries Q12, Q13, and Q18 will be estimated and the least expensive of the three will determine whether or not these transformations should be applied on the original query.

3.3.3 Impact on State Space

Interleaving and juxtaposing transformations causes an increase in the number of states explored. We could potentially consider an additional state for each subquery that we consider for unnesting. In some cases, the states that we *could* consider are states we would explore later even without interleaving or juxtaposing. If we choose subquery unnesting, because it is cheaper, then there is no need to interleave it with view merging; we will consider view merging in the usual sequential manner. Similarly, if we choose not to do view merging, because it is more expensive, then there is no need to juxtapose it with join predicate pushdown; we will consider join predicate pushdown in the sequential manner later. This mitigates some of the increase in the search space due to interleaving and juxtaposing.

3.4 Optimization Performance

The cost-based transformation technique we have described can be expensive in terms of both optimization time and optimizer memory consumption. Copying query structures repeatedly consumes memory, and optimising each of the transformed queries consumes time. We discuss several techniques to reduce the impact of cost-based transformation on optimization performance.

3.4.1 Cost Cut-Off

During the evaluation of any state, if the cost of the query tree or any of its constituents exceeds the cost of the best state found so far, then the physical optimization of that state is aborted, and the next state is considered by the optimizer.

3.4.2 Reuse of Query Sub-Tree Cost Annotations

Re-optimising each transformed query in its entirety is costly and in many cases unnecessary. Each transformation impacts a few known query blocks, and only these query blocks and the query block containing them in the query tree need to be re-optimized. In other words, we can reuse the results, which we refer to as cost annotations, of optimising two equivalent query sub-trees. For complex queries containing many subqueries or views, this can save a substantial amount of time.

Consider query Q1. Using an exhaustive search space strategy, we would optimize four variations of the query. Each variant contains three query blocks (two inner query blocks and the outer query block, Q_O), so we would optimize twelve query blocks in total. This process is summarized in Table 1. T(Q) refers to the transformed version of query block Q.

Table 1. Re-use and State Space

State	Query blocks optimized
(0,0)	Q_{S1} Q_{S2} Q_O
(1,0)	$T(Q_{S1})$ Q_{S2} Q_O
(0,1)	Q_{S1} $T(Q_{S2})$ Q_O
(1,1)	$T(Q_{S1})$ $T(Q_{S2})$ Q_O

Notice that Q_{S1}, Q_{S2}, $T(Q_{S1})$, and $T(Q_{S2})$ are each optimized twice. We can reuse the cost information instead of optimising each of these the second time. Hence, we can avoid optimising four of the twelve query blocks. The subqueries and their corresponding inline views could themselves contain views and nested subqueries, which would lead to a more substantial improvement.

3.4.3 Memory Management

In Oracle, query structures and optimizer cost and decision structures are customarily not freed until the end of optimization. Cost-based transformation creates many copies of query structures, and many optimizer structures are created to optimize query transformation variants. Hence we must manage memory more intelligently under the cost-based transformation framework. The query structures and optimizer decision structures are freed as each transformation decision is made. Note that optimizer cost annotations cannot be freed, since these may be reused, as previously described.

3.4.4 Caching

Certain optimizer computations are particularly expensive, and may be reusable even if a query block has been altered by a transformation. For instance, Oracle performs dynamic sampling to estimate the single table cardinality of certain tables, such as tables for which optimizer statistics have not been collected, and tables with multiple, possibly correlated, filter predicates. This computation is expensive, and if the single-table predicates on a table are not altered by a transformation, the result can be reused. This and other expensive optimizer computations are cached across multiple calls to the optimizer.

4. PERFORMANCE STUDY

We ran performance experiments for cost-based query transformations. In order to study a realistic workload, we

investigated queries issued by Oracle Applications. Their schema consists of about 14,000 tables representing Human Resources, Financial, Order Entry, CRM, Supply Chain and other applications. Most of the applications have highly normalized schemas.

The workload contained 241,000 queries issued by Oracle Applications. The number of tables in a query varies between 1 and 159 with an average of 8 tables per query. It should be emphasized that most of the queries in this workload are of simple SPJ type. Therefore, only a small fraction – about 8% – of these queries have subqueries, GROUP BY clause, SELECT DISTINCT, or UNION ALL views, which are subject to the cost-based transformations used in these experiments.

4.1 Cost-Based Transformation

In this experiment, we used an in-house database tool to examine the execution plans of all 241,000 queries with cost-based transformation turned off versus cost-based transformation turned on. Of these, only about 19,000 queries are relevant to cost-based transformations. When cost-based transformation was turned off, subquery unnesting, group-by view merging, and join predicate pushdown transformations were applied using heuristic rules; in Section 2.2, we described the heuristic rules used for subquery unnesting.

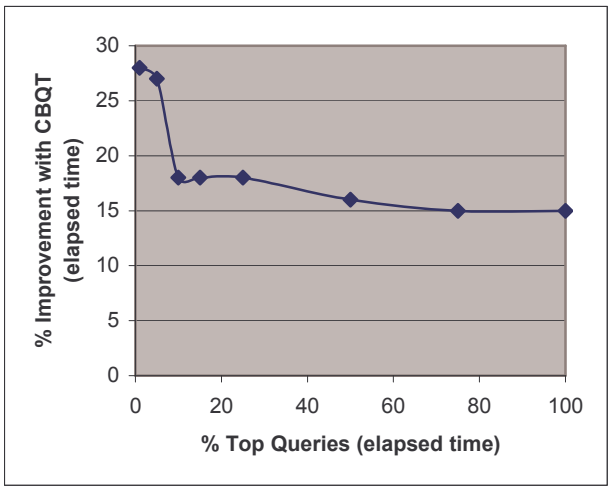

Figure 2. CBQT relative improvements as a function top N% of most expensive queries

The execution plans of 5,910 (i.e., 2.45% of the workload) queries changed; note that for some queries where no execution plan difference was detected, the cost-based and heuristic-based decisions could be identical. We then measured the execution and optimization times of the queries that showed a difference in execution plans. On average, the total run time (optimization + execution) of the queries improved by 20% with cost-based transformation. A small fraction, 18%, of the affected queries degraded by 40%. Figure 2 shows the percentage improvement as a function of the top N longest running queries. Top N is defined as the N longest running queries without cost-based transformation. For example, the top 5% of longest running queries showed a visible improvement of 27%; the top 25% showed an improvement of 18%, and so on. The top 20 queries improved 250% on the average using cost-based transformation. It is worth noting that there was a single outlier query whose total run time was reduced more than 214 times under cost-based transformation. This query is not included in Figure 2. Note that cost-based query transformation tends to benefit better those queries that are more expensive (e.g., the top 5% of the most expensive is benefited more than top 25%).

This reduction came at the expense of optimization time, which increased by only 40%.

4.2 Subquery Unnesting and JPPD

We wanted to assess the impact of two query transformations: subquery unnesting (since it has been studied extensively in the literature and is applicable to our query load) and join predicate pushdown (as it is a relatively new technique, which introduces correlation and thus has the opposite effect of unnesting).

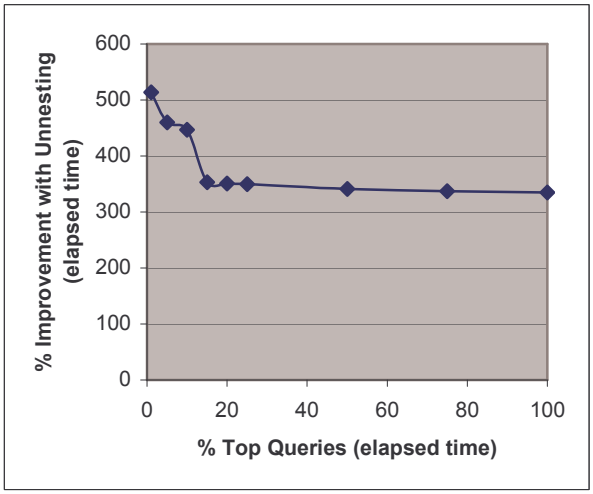

Figure 3. Unnesting relative improvements as a function top N% of most expensive queries

We ran experiments that compared queries where unnesting was completely disabled versus unnesting performed in a cost-based manner. This experiment affected 12,279 queries; i.e., 5% of the entire workload. On average, applying the unnesting transformation improved performance by about 387%. A small fraction of the affected queries, 15%, degraded 50% overall. Figure 3 shows percentage improvement as a function of the top N longest running queries. Top N is defined as before. For example, the top 5% longest running queries showed a substantial improvement of 460%. The top 25% showed an improvement of 350%.

This reduction came at an expense of optimization time, which increased by 31%. Observe that the transformation benefits expensive queries more.

We ran a third set of experiments that compared queries where join predicate pushdown (JPPD) was completely disabled versus JPPD performed in a cost-based fashion. The JPPD transformation affected 1797 queries; i.e., 0.75% of the entire workload.

On the average, the performance improved by about 23% with the JPPD transformation. A small fraction of the queries, 11%,

degraded 15% overall. Figure 4 shows the percentage improvement as a function of the top N% most costly queries where top N is defined as before. For example, the top 5% longest running queries showed an improvement of 15%. The top 25% showed an improvement of 23%. In these experiments, the performance degradation seen for some of the queries is typically due to cost mis-estimation.

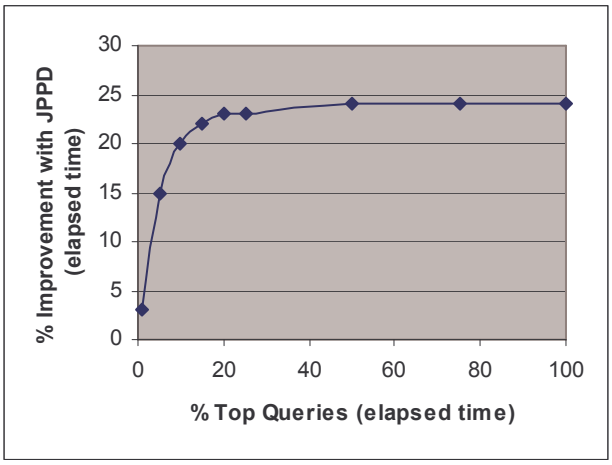

Figure 4. JPPD relative improvements as a function top N% of most expensive queries

Note that in contrast with the unnesting transformation, JPPD benefits less expensive queries more. For example, the top 80% of the most expensive queries benefited more than the top 5%. This reduction came at an expense of optimization time, which increased by 7%.

4.3 Group-By Placement Transformation

In another experiment for group-by placement (GBP) transformation, we compared the workloads with the GBP transformation turned on and off; in Oracle, the GBP transformation is never applied using heuristics. In this experiment, over 2,000 queries were affected. On the average, the performance improved by 21% with GBP. Although some queries degraded, 9 queries improved by more than 200% and 2 queries improved by more than 1,000%.

4.4 Optimization Time

One basic issue with cost-based transformation is the increase in optimization time. We have noted the optimization time increases in the experiments described in the previous sections. In Table 2, we present optimization times and number of states for various state space search strategies (Section 3.2) of a single query that undergoes the unnesting transformation. This query had three base tables and four subqueries, where each subquery contained three base tables, and were of NOT IN, EXISTS, and NOT EXISTS types. All subqueries were valid for unnesting.

The increase in optimization times for various search strategies as compared to the heuristic mode (i.e., no cost-based transformation) is not dramatic, because of the re-use of query sub-tree cost annotations.

Table 2. Increase in Optimization Time for Various State Space Search Techniques

	Optim. Time	#States
Heuristic	0.24 s	1
Two Pass	0.33 s	2
Linear	0.61 s	5
Exhaustive	0.97 s	16

5. RELATED WORK

In a rule-based rewrite system, it is possible to enumerate bushy tree transformations exhaustively [15] and cost the resulting plans; this leads to exponential expansion of plans and may not be practical for commercial optimizers. In [19], an attempt was made to combine transformations with a two-phase optimization; during plan generation, the optimizer suggests a transformation, it tries it, and the rewritten query is re-optimized and re-costed. [6] also had a notion of cost-sensitive re-write. However, there is no systematic treatment of various possible transformations in these works.

Queries with non-aggregated subqueries can be unnested as shown in [10], [22], [4], and [16]. Our system supports almost all of these forms of unnesting techniques. In our experience, unnesting that generates views does not always result in a more efficient plan.

Similar to unnesting of SPJ subqueries, unnesting of aggregate subqueries has been studied extensively before by [22], [7], [10], and [14]. [22] presents an early work on transformations where aggregates from nested subqueries are pulled up the operator tree before joins. Early versions of our system (Version 8.1) applied a similar algorithm. The transformation was triggered by heuristics during the query transformation phase.

[7], [1], [23] and [24] showed transformations where aggregates were pushed down before joins. [1] defines a greedy and conservative heuristic applied during join enumeration which places group-by before the table scan if this results in a cheaper scan or join. [2] defined a similar pull-up, pushdown aggregate transformation, but in addition, showed how to perform the transformation based on cost; it also provided two heuristics to control the expansion. The heuristics were different than ones used initially in our system.

Techniques for pushing predicates on single relations down through views with aggregates and window functions have been known for a long time. [9] proposed a technique that generalizes that by first pulling the predicates up the query tree and then pushing them down resulting in more single table predicates applicable on scans of base tables.

[12] and [13] describe magic set transformation, which adds new views to the expensive query blocks with the aim of reducing the amount of data to process. [19] extends this work to choose magic set rewrite based on cost. Two methods are presented: one (very practical) implements the transformation within the existing DB2 optimizer and the second is based on algebraic transformation for rule-based optimization.

Outerjoins do not commute with inner joins and this limits optimizer choices for join ordering. This problem has been studied

in [22], [18], and [3], where, for certain queries, outerjoin can be commuted with inner joins by introducing a notion of generalized outerjoin. [17] provides a canonical abstraction of outer-join, which allows the optimizer to use various join orders between outer- and inner-joined tables.

6. CONCLUSION

This paper makes two major contributions[5]. First, it proposes a feasible solution for integrating various transformations within a cost-based framework. This scheme has the ability to model most transformations and possible interactions between them thereby allowing the optimizer to choose the most optimal variant of a query. The second contribution is that we propose state-space search algorithms for dealing with the combinatorial explosion of cost-based transformation. We also discuss some transformations (e.g., join predicate push down and join factorization), which have not been discussed in the literature.

We found that for a significant range of OLTP queries, the extensively studied unnesting transformation is a critical transformation. For the affected queries, it improved their total run time by 387%. With join predicate pushdown transformation, there was an improvement of 23% in run time. With group-by placement transformation, we found an improvement of 21% in run time. For subquery unnesting, group-by view merging, and join predicate pushdown, our performance experiments show that cost-based transformations outperform heuristic-based transformations by 20%.

7. ACKNOWLEDGEMENTS

The authors wish to acknowledge Vadim Tropashko, who provided invaluable assistance in running the performance experiments.

8. REFERENCES

[1] S. Chaudhuri and K. Shim, “Including Group-By in Query Optimization”, *Proceedings of the 20th VLDB Conference, Santiago, Chile, 1994.*

[2] S. Chaudhuri and K. Shim, “Optimizing Queries with Aggregate Views”, *EDBT 1996.*

[3] C.A. Galindo-Legaria, et al., “Outerjoin Simplification and Reordering for Query Optimization”, TODS, 1997.

[4] R.A. Ganski and H. K. T. Wong, “Optimization of Nested SQL Queries Revisited”. *Proc. of ACM SIGMOD, 1987.*

[5] H. Garcia-Molina, J. D. Ullman, and J. Widom, "Database System Implementation", *Prentice Hall*, 2000.

[6] G. Graefe and W.J. Mckenna, “The Volcano optimizer generator: Extensibility and Efficient Search”, *Proceedings of the 19th Internatinal Conf. on Data Engineering, 1993.*

[7] A. Gupta, V. Harinarayan, and D. Quass, “Aggregate-Query Processing in Data Warehousing Environments”, *Proceedings of the 21th VLDB Conference,*1995.

[8] J.M. Hellerstein and M. Stonebraker, “Predicate migration: Optimizing Queries with Expensive Predicates,” *Proc. of ACM SIGMOD,* Washington DC, 1993.

[9] A.Y. Levy, I.S. Mumick, and Y. Sagiv, “Query Optimization by Predicate Move-Around,” *Proceedings of the 20th VLDB Conference,* Santiago, Chile, 1994.

[10] W. Kim. “On Optimizing an SQL-Like Nested Query”, *ACM TODS*, September, 1982.

[11] T. Morzy, M. Matysiak, and S. Salza, "Tabu Search Optimization of Large Join Queries", EDBT, 1994.

[12] I.S. Mumick, S. Finkelstain, H. Pirahesh and R. Ramakrishnan, “Magic is Relevant”, *Proceedings of ACM SIGMOD,* 1990.

[13] I.S. Mumick and H. Pirahesh and R. Ramakrishnan, “Implementation of Magic Sets in Relational Database Systems”, *Proceedings of ACM SIGMOD,* 1994.

[14] M. Muralikrishna, “Improved Unnesting Algorithms for Join Aggregate SQL Queries”, *Proceedings of the 18th VLDB Conference*, Vancouver, Canada, 1992.

[15] A. Pellenkoft, C.A. Galindo-Legaria, M. Kersen, “The Complexity of the Transformation-Based Join Enumeration. *Proceedings of the 23rd VLDB Conference,* Athens, Greece, 1997.

[16] J. Rao, H. Pirahesh, and C. Zuzarte, “Cannonical Abstractios for Outerjoin Optimization”, *Proceedings of ACM SIGMOD,* Paris, France, 2004.

[17] H. Pirahesh, J.M. Hellerstein, and W. Hasan, “Extensible Rule Based Query Rewrite Optimizations in Starburst”. *Proc. of ACM SIGMOD,* San Diego, U.S.A., 1992.

[18] A. Rosenthal and Cesar A. Galindo-Legaria, “Query Graphs, Implementing Trees, and Freely-reorderable Outerjoins”, *Proceedings of ACM SIGMOD,* 1990.

[19] P. Seshadri, et al, “Cost-Based Optimization for Magic Algebra and Implementation”, *Proceedings of ACM SIGMOD,* Monteral, Canada 1996.

[20] A. Swami and A. Gupta, “Optimization of Large Join Queries”, *Proceedings of ACM SIGMOD*, Chicago, U.S.A, 1988.

[21] A. Swami, "Optimization of Large Join Queries: Combining Heuristics and Combinatorial Techniques", *ACM SIGMOD,* Portland, Oregon, U.S.A., *1989*.

[22] U. Dayal, “Of Nests and Trees: A Unified Approach to Processing Queries that Contain Nested Subqueries, Aggregates, and Quantifiers”, *Proceedings of the 13th VLDB Conference*, Brighton, U.K., 1987.

[23] W.P. Yan and A.P. Larson, “Performing Group By Before Join”, *IEEE ICDE,* Houston, Texas, 1994.

[24] W.P. Yan and A.P. Larson, “Eager Aggregation and Lazy Aggregation”, *Proceedings of the 21th VLDB Conference,* Zurich, Swizerland, 1995.

[25] A. Witkowski, et al, “Spreadsheets in RDBMS for OLAP”, *Proceedings of ACM SIGMOD,* San Diego,USA, 2003.

[26] A. Witkowski, et al, “Spreadsheets in RDBMS for OLAP”, *ACM TODS*, 2005.

[5] There are U.S. patents pending for some of the work discussed here.