Of Snowstorms and Bushy Trees

Rafi Ahmed
Oracle Corporation
500 Oracle Parkway
Redwood Shores, CA 94065, U.S.A.
rafi.ahmed@oracle.com

Rajkumar Sen
Xplain.Io
226 Airport Parkway
San Jose, CA 95112, U.S.A.
raj@xplain.io

Meikel Poess
Oracle Corporation
500 Oracle Parkway
Redwood Shores, CA 94065, U.S.A.
meikel.poess@oracle.com

Sunil Chakkappen
Oracle Corporation
500 Oracle Parkway
Redwood Shores, CA 94065, U.S.A.
sunil.chakkappen@oracle.com

ABSTRACT

Many workloads for analytical processing in commercial RDBMSs are dominated by snowstorm queries, which are characterized by references to multiple large fact tables and their associated smaller dimension tables. This paper describes a technique for bushy join tree optimization for snowstorm queries in Oracle database system. This technique generates bushy join trees containing subtrees that produce substantially reduced sets of rows and, therefore, their joins with other subtrees are generally much more efficient than joins in the left-deep trees.

The generation of bushy join trees within an existing commercial physical optimizer requires extensive changes to the optimizer. Further, the optimizer will have to consider a large join permutation search space to generate efficient bushy join trees. The novelty of the approach is that bushy join trees can be generated outside the physical optimizer using logical query transformation that explores a considerably pruned search space. The paper describes an algorithm for generating optimal bushy join trees for snowstorm queries using an existing query transformation framework. It also presents performance results for this optimization, which show significant execution time improvements.

1. INTRODUCTION

Current relational database systems process complex SQL queries involving multiple fact tables joined with one another and with corresponding dimension tables. Such queries are becoming increasingly important in Decision-Support Systems (DSS). Generating optimal execution plans for such queries has become critical for a commercial database system. Bushy join trees provide an efficient way to execute these types of queries.

Database researchers, however, have paid scant attention to the problem of bushy join trees. It is well-known that the join order optimization problem is NP-hard [8]. Practical solutions to the problem therefore tend to involve trade-offs that perform a heuristic search of the state space, and therefore, most commercial optimizers restrict their default join enumeration to the space of left-deep trees [12][14].

The following issues with bushy join trees are noted by [4] and [14]: the space of bushy join trees is vastly larger and much more expensive to search than that of left-deep trees; the best among numerous left-deep join tree plans should suffice for all queries; and left-deep join trees interact well with nested-loop and index-based one-pass join algorithms, and therefore execution plans that are based on left-deep trees and are driven by these algorithms tend to be more efficient than the same algorithms used with non-left-deep trees. We address these issues in this paper.

Execution plans based on bushy join trees seem to be the most efficient way to evaluate queries that are formulated for snowstorm schemas, but generating a bushy join tree within a physical optimizer does not come without a price. First, it requires extensive changes to the existing optimizer. Second, query optimization time increases as the search space grows.

In this paper, we introduce an innovative algorithm for generating bushy join trees for queries that are formulated for snowstorm schema. This algorithm generates bushy join trees containing subtrees that are constructed to yield considerably reduced sets of rows and, therefore, their joins with other subtrees are generally much more efficient than joins in the corresponding left-deep trees, where large fact tables must be joined with one another before their sizes could be reduced. The generation of bushy join trees is performed within an existing cost-based transformation framework.

The rest of the paper is organized as follows. We first introduce snowstorm schema and describe the existing cost-based query transformation framework in the Oracle query optimizer. We discuss how bushy join trees interact with star transformation in snowstorm queries. We then present an algorithm for generating optimal bushy join trees using the existing query transformation framework. This is followed by the performance study of the new technique for TPC-DS and commercial workloads. We finally conclude with a discussion on related work.

Proceedings of the VLDB Endowment, Vol. 7, No. 13

1.1 Snowstorm Schema

A *star schema* [6][11] includes a large fact table and several small dimension (lookup) tables. The fact table stores frequently added transaction data such as sales, returns and inventory changes, and it generally represents the relationships among the dimension tables. Each dimension table stores less frequently changed or added data supplying additional information for fact table transactions, such as customers who made purchases. Multiple extensions to the traditional star schema are commonly used in today's data warehouse implementations. The tables belonging to a star schema usually contain data from a subject area such as sales or returns.

An extension to the pure star schema, called a *snowflake schema*, separates static data in the outlying dimension tables from the more dynamic data in the inner dimension tables and the fact tables. That is, in addition to their relationships to the fact table, dimension tables can have relationships to other dimension tables. A dimension table that is joined with only dimension tables but not with any fact table is called a *branch* in our terminology.

A *snowstorm schema* extends the snowflake schema by combining multiple snowflake schemas. In essence, a snowstorm schema combines multiple related subject areas into one comprehensive schema. Since the subject areas are related, some dimension tables are shared among them; e.g., a customer dimension can be shared between sales and returns. It also allows for joins across subject areas, which causes fact tables to be joined with one another directly or through shared dimensions. The join graphs in these cases can be quite complex as illustrated in Section 3.

This snowstorm approach challenges query execution of both star schema and 3NF execution models. Typical executions of queries in a star schema involve bitmap accesses, bitmap merges, bitmap joins and conventional index-driven join operations. The execution plans in a 3NF DSS system are dominated by large hash joins and conventional index-driven joins. In both the systems, large aggregation and group-by operations are quite common. This heterogeneity in execution plans imposes challenges both on hardware and software systems. High sequential I/O-throughput is critical in large hash join operations. At the same time, index-driven queries stress the I/O subsystems' ability to perform small random I/Os. Further, this heterogeneity also challenges a query optimizer in its decision to either use a star-schema approach, such as star transformation (Section 3.1), or a more traditional approach, such as nested-loop and hash joins.

The algorithms devised for processing join graphs such as chain, star, circuit, and clique [10] do not lend themselves well to snowstorm join graphs. This seems to be an area that traditional query optimizers are ill-equipped to deal with.

1.2 Cost-Based Transformation Framework

Query transformations in the Oracle optimizer can be heuristic-based or cost-based. In cost-based transformation, logical transformation (also known as query re-write) and physical optimization are combined to generate optimal execution plans.

A cost-based physical optimizer was first introduced in Oracle 7.1. The *physical* optimizer works within the scope of a single query block, which ranges over a set of tables with restriction, projection, and join. In the physical optimization phase, access methods, join methods, and left-deep tree join permutations are chosen in order to generate an efficient execution plan; the physical optimizer also generates a limited form of right-deep join trees, which originates from swap of build and probe sides in hash join.

In Oracle 10.1, a general framework [1] for cost-based query transformation and several state space search strategies were introduced. During cost-based transformation, a query is copied, logically transformed and its cost is calculated using the existing cost-based physical optimizer. This process is repeated multiple times applying a new set of transformations; and at the end, one or more transformations are selected and applied to the original query, if it results in an optimal cost. The cost-based transformation framework provides a mechanism for the exploration of the state space generated by applying one or more transformations thus enabling the Oracle optimizer to select the optimal transformation in an efficient manner. The cost-based transformation framework can handle the complexity produced by the presence of multiple query blocks in a user query and by the interdependence of transformations.

The availability of the general framework for cost-based transformation has made it possible for other innovative transformations to be added to the vast repertoire of Oracle's query transformation techniques [1][2], such as subquery unnesting, group-by and distinct view merging, join predicate pushdown, join factorization, OR expansion, star transformation, group-by and distinct placement, vector aggregation, etc.

1.2.1 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 sources of multiple alternatives are the various transformations themselves as well as the set of objects (e.g., subquery blocks, view blocks, tables, table groups, join edges, predicates, etc.) on which each transformation may apply.

If there are N independent objects on which a transformation T can apply, then 2^N possible alternative combinations can potentially be generated by the application of T. For simplicity, we here denote a state as an array of bits, where the *n*th bit represents whether the *n*th object (e.g., subquery, view or table group, etc.) is transformed (a value of 1) or not transformed (a value of 0). When there are M transformations that apply on N objects, the state is represented by an MxN bit matrix.

To cope with the combinatorial explosion problem, randomised search algorithms are used. The common idea behind 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.

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. In order to limit the potential increase in optimization time, we use several different techniques for searching the state space of

various transformations. Some examples of search techniques are exhaustive, iterative, linear, two-pass, and perturbation walk.

The cost-based transformation framework automatically decides which search technique to use based on the number of elements to be transformed, the characteristics of the transformation, and the overall complexity of the query.

1.2.2 Re-use of Subtree Cost Annotations

Re-optimizing each transformed query tree in its entirety is costly and in many cases unnecessary. Each transformation impacts a few known query blocks (subtrees), and only these query blocks and the query block containing them in the query tree need to be re-optimized. Therefore, we reuse the cost annotations (i.e., scaled selectivities, estimated costs and cardinalities, etc.) of an already-optimized subtree when optimizing its equivalent subtree during state space exploration. For complex query trees containing many subtrees, this can provide substantial savings of optimization time.

1.2.3 Interleaving of Transformations

The Oracle optimizer generally performs various transformations in a sequential manner. However, there are exceptions to this rule. 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 generally need to be interleaved in order for the optimizer to determine an optimal execution plan.

Consider a query block, Q, and a transformation T1 that applies to Q. If Cost(Q) is 30 and Cost(T1(Q)) is 40, then T1 should be rejected for Q. However, if another transformation T2 is applicable to T1(Q) and Cost(T2(T1(Q))) is 25, then T1 should be applied to Q. That is, T2 must be interleaved with T1 in cases where application of T1 increases the estimated cost of Q, since T2, when applied to the result produced by T1, may yield a lower cost indicating that T1 should be performed on Q.

By the same token, star transformation (discussed in the next section) must be interleaved with the generation of bushy join trees in cases where it increases the cost.

1.3 Processing Star and Snowflake

In the classical method of processing a star query, a Cartesian product of all dimension tables is performed before joining the result with the fact table. The rows from fact table are accessed using an index on the columns that are joined to the dimension tables. This technique avoids a full scan of the fact table. However, a Cartesian product of multiple dimension tables can lead to a large result with numerous failed index probes on the composite index of the fact table, which can be prohibitively expensive.

In addition to the classical method, the Oracle optimizer uses another technique called *star transformation* to evaluate star or snowflake queries efficiently. Star transformation avoids a full table scan of the fact table by retrieving only the relevant set of rows from the fact table and thereby overcoming the drawbacks of the classical method. The presence of filter predicates on dimension tables and the intersection of the fact table rows joining each of the dimension tables may vastly reduce the data set that needs to be accessed from the fact table. Hence, this optimization would prove to be much more efficient than a brute-force full table scan of the fact table. Once the relevant rows are fetched from the fact table, it is joined back to the dimension tables, if needed, and thus avoiding Cartesian products required in the classical star join processing.

This technique [9] is based on bitmap indexes. In a bitmap index on column A, there is a bitmap for each distinct key value of A, where each bit corresponds to a row in the table; the bit is set to 1, if the key value of A appears in that row; otherwise, it is set to 0. In Oracle, B-tree index keys can be dynamically converted into bitmaps during query execution and hence B-tree index can also be used for star transformation.

Consider the following query.

```
Q1.
SELECT D1.x, SUM (F.m)
FROM F, D1, D2
WHERE F.fk1 = D1.pk and F.fk2 = D2.pk and
      D1.a > 5 and D2.b < 77
GROUP BY D1.x;
```

Here F, a large fact table, is joined with small dimension tables, D1 and D2, which have filter predicates. Q1 undergoes star transformation and yields Q2 as shown below.

```
Q2.
SELECT D1.x, SUM (F.m)
FROM F, D1
WHERE F.fk1 = D1.pk and D1.a > 5 and
      F.fk1 IN (SELECT D1.pk
                FROM D1
                WHERE D1.a > 5) and
      F.fk2 IN (SELECT D2.pk
                FROM D2
                WHERE D2.b < 77)
GROUP BY D1.x;
```

If F has bitmap indexes on its join keys –F.fk1 and F.fk2– referenced in the query, the transformation adds subquery predicates corresponding to each dimension table. For a snowflake query, the subquery might refer to more than one table that is joined together. Note that the subquery here may be looked upon as a set membership operation; e.g., F.fk1 IN (8,13, 29 …). When driven by bitmap AND and OR operations on the key values supplied by the dimension subqueries, only the relevant rows need to be retrieved from F. The following operations are performed in Q2 to access and join the fact table, F.

- By iterating over the key values returned by a dimension subquery, the bitmaps are retrieved for a given key value from a bitmap index on table F.
- For a subquery, the bitmaps retrieved for various key values are merged (OR-ed).
- The merged bitmaps supplied by dimension subqueries are AND-ed; that is, a conjunction of the joins is performed.
- From the final bitmap, the corresponding rowid's for F are generated.
- Rows are directly retrieved from F using the rowid's.

The dimension subquery filters the fact table based on the filter predicates on the dimension tables. Therefore it may still be necessary to join the dimension tables back to the relevant rows of the fact table using the original join predicates. The join-back of a dimension table can be avoided, if the dimension table is semi-

joined or if all the predicates on the dimension table are part of the dimension subquery, the column(s) selected from the subquery are unique, and the columns of the dimension table are not referenced anywhere else in the query. In Q1, D2 is not joined back to F, since D2 is not referenced in the SELECT and GROUP-BY clauses and D2.pk is unique.

The existence of a fact table bitmap *join* index, which essentially materializes the join between the fact table and a dimension table, can obviate the necessity of generating a subquery for that dimension. The Oracle optimizer also considers materializing the subquery into a temporary table during query execution. This improves the efficiency of the dimension tables in the subquery, which are accessed multiple times, for example, once for retrieving bitmaps and then for join back.

The Oracle optimizer automatically decides whether to apply the star transformation, which is performed under the transformation framework described in Section 1.2. Bushy join trees can benefit from interleaving star transformation within its subtrees, as this may result in a smaller estimated cost and thereby making it possible for bushy tree plans to be selected.

1.4 Join Predicate Pushdown

The Oracle optimizer performs join predicate pushdown transformation, where join predicates are pushed down inside a view [1]. This allows a view to be joined with outer tables by index-based one-pass join method, which is not possible for regular views that can be joined only by hash or sort-merge join methods.

The pushed-down join predicates act like correlation once inside the view, thereby opening up new index access paths. This transformation imposes a partial join order on the joined tables; that is, the tables that the view is joined with (via the pushed-down predicates) must precede the view in the join permutation; and the view must be joined by index-based one-pass join method.

This transformation is also performed within the cost-based transformation framework (Section 1.2).

One of the issues [4] mentioned in Section 1 was a limited interaction between bushy join trees and index-based one-pass join algorithms. We circumvent this limitation by performing join predicate pushdown on subtrees of bushy join trees.

2. BUSHY JOIN TREES

In a join tree, leaf nodes represent user tables and internal nodes represent join operations. If the right child of every internal node of a join tree is a leaf node, then the tree is called *left-deep* join tree. If the left child of every internal node of a join tree is a leaf node, then the tree is called *right-deep* join tree. If the left or the right child of an internal node of a join tree can be an internal node, then the tree is called a *bushy join* tree.

The number of left-deep trees does not grow nearly as fast as the number of bushy trees for multi-way joins of a given number of tables. For *N* tables, there is only one left-deep tree shape to which tables can be assigned in *N!* ways. The total number of bushy tree (which subsumes left-deep and right-deep tree) *shapes S(N)* for *N* tables is given by the following recurrence function [4].

$$S(1) = 1$$

$$S(N) = \sum_{i=1}^{N-1} S(i) * S(N-i)$$

The second equation states that, for ***i*** between 1 and *N-1* as the number of leaves in the left subtree, those leaves may be re-arranged in *S(i)* ways. Similarly the remaining *N-i* leaves in the right subtree can be re-arranged in *S(N-i)* ways. *S(N)*N!* gives the total number of bushy join trees (i.e., join permutations).

A traditional physical optimizer generates a sequential or a left-deep join tree execution plan. There are many scenarios, however, where bushy join trees can substantially improve query performance.

2.1 Physical vs. Logical

We consider the pros and cons for generating bushy join trees (A) under the transformation framework and (B) in the physical optimizer.

By performing bushy join trees only for snowstorm queries and by devising efficient search strategies, Scheme A can substantively reduce the state space of bushy joins, and thus can render the problem of combinatorial explosion of join permutations more manageable. Scheme B would generate bushy join permutations every time the physical optimizer is invoked. Scheme A does not have this problem.

As the subtrees (views) contained by bushy join trees may be candidates for star transformation (Section 1.3), scheme A can leverage star transformation by performing bushy join before star transformation. This would not be possible in scheme B, which would generate bushy join trees in the physical optimizer after star transformation takes place.

By allowing the existing technique of join predicate pushdown (Section 1.4) on bushy join subtrees (views), scheme A can provide an efficient interaction between index-based nested-loop join method and bushy join subtrees. This may be feasible in Scheme B albeit with considerable effort.

The re-use of cost annotation (Section 1.2.2) allows us to bypass the optimization of a subtree (view) when it reappears in a different state during state space exploration. This is difficult to do in the physical optimizer.

Scheme A provides a cleaner and simpler implementation of bushy join trees, whereas scheme B requires extensive changes to the physical optimizer.

On the con side, scheme A may have some overhead associated with the copying of query structures.

All things considered, scheme A seemed preferable over B.

2.2 Snowstorm Schema and Join Trees

In this paper, the generation of bushy join depends upon the existence of multiple fact and dimension tables in a snowstorm query block. The join graph of a query block and the relative sizes and other statistical properties of the tables and the join columns are used to identify fact and dimension tables.

Consider query Q3, which is posed against a snowstorm schema. In Q3, F1 and F2 are large fact tables and D1, D2, and D4 are small dimension tables.

```
Q3.
SELECT F1.k, F2.n
FROM F1, D1, F2, D2, D4
WHERE F1.a = D1.a AND F2.c = D2.c AND
      F2.d = D4.d AND F1.e = F2.m AND
      D1.g > 1 AND D2.h < 6 AND D4.x = 4;
```

One of the many possible left-deep join permutations for Q3 can be given by (D1, F1, F2, D2, D4), where joins take place in a left-to-right sequential order as shown in Figure 1. Here the intermediate result of join between D1 and F1 must be joined with the large fact table F2, which has not yet been reduced to a smaller size by first joining it with its dimension tables D2 and D4, which have single-table filter predicates. Therefore, left-deep tree execution plans for such a query may turn out to be extremely inefficient. When queries have more than two fact tables, the inefficiency may multiply as shown in Section in 3.1.

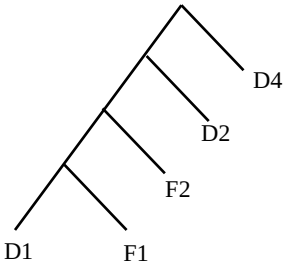

Figure 1. A left-deep join tree for Q3

2.3 Subtrees and Views

In the Oracle optimizer, unmerged views have their independent join subtrees. A bushy join tree execution plan can be generated by introducing subtrees or unmergeable views within a query block. Conversion of Q3 into Q4 illustrates this point.

```
Q4.
SELECT V1.k, V2.n
FROM (SELECT F1.k, F1.e
      FROM F1, D1
      WHERE F1.a = D1.a AND D1.g > 1) V1,
     (SELECT F2.m, F2.n
      FROM F2, D2, D4
      WHERE F2.c = D2.c AND F2.d = D4.d
            AND D2.h < 6 AND D4.x = 4) V2
WHERE V1.e = V2.m;
```

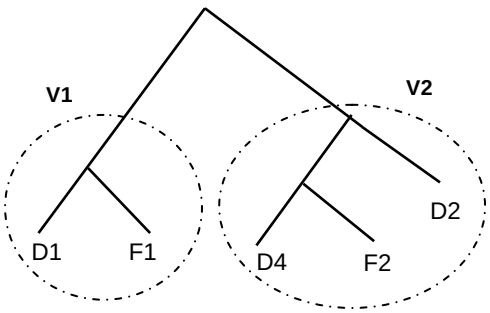

Figure 2. A bushy join tree for Q4

One of the many possible bushy join permutations for Q4 can be given by ((D1, F1), (D4, F2, D2)), as shown in Figure 2. Here the subtrees (or views) impose the desired order of join evaluation. In each subtree, the fact table is joined with its dimension tables first, and then the intermediate results of the subtrees are joined together. This, in effect, produces a bushy join tree execution plan for query Q4.

Note that once the views have been generated, the physical optimizer will choose the best left-deep tree join permutation within each of the views and within the outer query block. For example, it may generate the following join permutations in addition to the one shown above. These bushy tree permutations cannot be generated in a left-deep tree scheme.

((D2, F2, D4), (F1, D1))
((D4, F2, D2), ((D1, F1))
((D1, F1), (D2, F2, D4))

The main idea is to identify table groups for the given set of tables such that each group may form a subtree (view) containing a star/snowflake, which is anchored around one fact table and multiple dimension tables. Each subtree may return a sizably reduced set of rows and, therefore, its join with other subtrees may prove to be much more efficient than joins in the left-deep tree. The *data reduction* yielded by a subtree can be measured as the ratio of its row count and the cardinality of its fact table.

In the Oracle optimizer, bushy join tree generation is performed under the transformation framework described in Section 1.2.

2.4 Search Strategies for Bushy Join Trees

In order to limit the potential increase in optimization time, we use several different techniques for searching the state space for bushy join trees. In all these search techniques the state with the cheapest cost is chosen as the best state, according to which the final transformation is performed.

In bushy join tree generation, the table groups are the objects on which this transformation applies. If there are two table groups, then there are at most four alternatives to consider: no bushy join tree (i.e., left-deep tree), subtree for only the first table group, subtree for only the second table group, or subtrees for both the table groups. The second and third alternatives generate partial bushy join trees, whereas in the fourth alternative a full bushy join tree is produced.

We denote a state as an array of bits, where the *n*th bit represents whether the *n*th object (i.e., table group) is transformed (a value of 1) or not transformed (a value of 0). For instance, the state (0,1,0) for three table groups refers to the generation of subtree for only the second table group.

- *Exhaustive.* In exhaustive search, all possible 2^N states of the state space for N table groups are considered. This search is guaranteed to provide the best solution.
- *Iterative.* In an iterative improvement technique, which is used to prune the search space, we start from an initial state and move to the next neighbouring state 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 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 state space 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. For two transformation objects, starting with the state (0, 0), if we find that Cost(1,0) is lower than Cost(0,0), we use the state (1,0) to generate the next state. And if Cost(1,1) is lower than Cost(1,0), then it is reasonable 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). For bushy trees, we generate the two states (0,0,...) and (1,1,...) as the initial states. After that, at every step, the next state is generated from the best state so far. The strategy significantly cuts down the search space, as it considers at most N+2 states.
- *Two-pass*. Two-pass search is the least expensive search strategy, where only 2 states are considered. The cost of not transforming any object (i.e., the state (0,0,…)) is compared with the cost of transforming all the objects (i.e., the state (1,1,…)), and the cheaper of the two states is selected.

2.4.1 Analysis of Search Space Complexity

Consider N tables and K table groups, $N_1, N_2, \ldots N_K$ such that

$$N = \sum_{j=1}^{K} N_j$$

In our scheme, the total number of join permutations for a state, where every table group forms a subtree, is given by

$$K! + \sum_{j=1}^{K} N_j!$$

The first term above gives the number of left-deep tree permutations in the outer query block and the second term gives the summation over left-deep join tree permutations for all subtrees.

Consider a join graph where $N = 5$. Using the recurrence function given in Section 2, we get the total number of bushy tree *shapes* $S(5)$ as 14 and the total number of bushy join tree permutations as 14*5! (= 1680).

Suppose these N tables are grouped such that $K = 2$, $N_1 = 3$, and $N_2 = 2$. The following shows the number of permutations that are considered with an exhaustive search of the bushy tree state space.

State 1 (0, 0): 5! = 120

State 2 (0, 1): 4! + 2! = 26

State 3 (1, 0): 3! + 3! = 12

State 4 (1, 1): 2! + [2!] + [3!] = 2

Note that in State 4 the permutations within brackets [] are not counted, since the cost annotations (Section 1.2.2) for these subtrees are re-used from States 2 and 3. Our scheme will try a total of 160 join permutations with the exhaustive search, which is an order of magnitude smaller than the full bushy join tree permutations (i.e., 1680). With other search strategies, our scheme would explore far fewer join permutations.

2.5 Bushy Join Tree (BJT) Algorithm

The BJT algorithm searches for a snowstorm pattern in a join graph for a query block. If such a pattern is found, it identifies multiple snowflake subgraphs in the given join graph, where tables in each subgraph may potentially form a subtree in a bushy join tree execution plan.

The join graph must contain (a) at least two fact tables, where (b) each fact table has a join edge with at least one other fact table, (c) a fact table has a join edge with at least one dimension table, and (d) a dimension table has zero or more of their own dimension/branch tables.

By analyzing the join graph, the algorithm first forms table groups such that each group contains one fact table, one or more dimension tables, and zero or more branch tables. A table that does not belong to any table group always remains in the outer query block, but it is has join edges to other tables/views. Note that a table belongs to *at most* one table group; i.e., the table groups do not necessarily partition the set of tables in the join graph.

We define *effective cardinality* of a table T as its cardinality after all its single-table filter predicates have been applied to T. The identification of tables as fact, dimension, or branch depends upon many factors such as join graph properties, a minimum threshold value for effective cardinality of a candidate fact table, a minimum value for the ratio of dimension's effective cardinality to that of its fact table, and a minimum value for the ratio of scaled *number of distinct values* (NDVs) of columns in a join predicate of fact and dimension tables.

Besides the system default values, the user can set a threshold value for the effective cardinality of a fact table and a minimum value for the ratio of dimension's effective cardinality to that of its fact table.

As mentioned before, both star transformation and join predicate pushdown are relevant for the views generated for bushy join trees. These two optimizations are interleaved with bushy join tree generation.

Here, we outline an algorithm for generating bushy join trees in a query block.

Algorithm BJT

{

Input: *a connected join graph for tables $T_1, T_2 \ldots, T_N$.*

Output: *an optimal bushy join tree.*

1. For the given join graph, generate a left-deep tree join order and estimate its cost. (This constitutes the first state.)
2. Use the join graph adjacency properties, effective table cardinalities, etc., to identify fact, dimension and branch tables and generate table groups.
3. If there are less than two table groups, then exit.
4. Using the relevant search strategy, generate a state and build BJT views for the table groups designated by the current state.

4.1 Re-use cost annotations in subtrees, if applicable.

4.2 Get the cost of the current state.

4.3 Perform interleaved star transformation within the outer query block and within each BJT view of the current state, if required, and estimate the new current cost.

4.4 Perform interleaved join predicate pushdown on each BJT view of the current state, if required, and evaluate the new current cost.

4.5 Update the best state so far by comparing the current cost with the cost of the best state so far.

5. If there are more states to be enumerated go to Step 4, else transform the original query based on the best-state directives.

}

3. PERFORMANCE STUDY

We conducted performance studies on two workloads and compared the performance of queries with left-deep and bushy tree execution plans. Since the intent of these experiments is to study *comparative* performance improvements and degradations, we present the results using an unspecified time unit, *U*.

3.1 TPC-DS Workload

We conducted performance experiments on a TPC-DS schema at the scale factor of 300. It contains 99 queries and 25 tables belonging to a snowstorm schema. A 300GB database on an Exadata machine with 4 compute nodes each with 4 CPU cores was used.

While TPC-DS can be applied to any industry application that must transform operational and external data into business intelligence, the workload [6] has been granted a realistic context. It models the decision support tasks of a typical retail product supplier. TPC-DS defines 12 data maintenance operations covering typical DSS query types such as ad-hoc, reporting, iterative (drill down/up) and extraction queries and periodic refresh of the database. TPC-DS query set is designed to cover the entire dataset. This is guaranteed by a sophisticated query template model.

In the TPC-DS benchmark, some queries reference one or more fact tables and multiple dimension tables. In our performance study, we concentrated upon the TPC-DS queries that contain two or more fact tables and multiple dimension tables. The queries used in this experiment were: Q14, Q17, Q18, Q25, Q29, Q50, Q51, Q61, Q64, Q72, Q78, Q91, Q93, and Q97. A somewhat simplified version of the TPC-DS query Q14 is shown below.

```
Q14.
SELECT S.ss_item_s
FROM store_sales S, catalog_sales C,
     web_sales W, item IS, item IC, item IW,
     date_dim DS, date_dim DC, date_dim DW
WHERE S.ss_item_sk = IS.i_item_sk and
  AND C.cs_item_sk = IC.i_item_sk
  AND W.ws_item_sk = IW.i_item_sk
  AND S.ss_sold_date_sk = DS.d_date_sk
  AND C.cs_sold_date_sk = DC.d_date_sk
  AND W.ws_sold_date_sk = DW.d_date_sk
  AND S.ss_cust_sk = C.cs_bill_cust_sk
  AND C.cs_bill_cust_sk = W.ws_bill_cust_sk
  AND IS.i_brand_id = IC.i_brand_id and
  AND IC.i_brand_id = IW.i_brand_id and
  AND IS.i_class_id = IC.i_class_id and
  AND IC.i_class_id = IW.i_class_id and
  AND IS.i_category_id = IC.i_category_id
  AND IC.i_category_id = IW.i_category_id
  AND DS.d_year between 1999 AND 1999 + 2
  AND DC.d_year between 1999 AND 1999 + 2
  AND DW.d_year between 1999 AND 1999 + 2;
```

This query has three fact tables, catalog_sales (431M), store_sales (864M), and web_sales (216M) and each fact table has two dimension tables, date_dim (73K) and item (264K). The instances of the dimension table, item, are joined with each other. The base cardinality of each table is shown within parentheses. The join graph of Q14 in terms of table aliases is shown in Figure 3, where fact tables are indicated by bold circles.

For Q14, the BJT algorithm initially identified three table groups: {W, DW, IW}, {C, DC, IC}**,** and {S, DS, IS}**.** After invoking the exhaustive search, the state where each of the three table groups formed subtrees was found to be the most optimal. The bushy join tree generated in our experiment for Q14 is shown in Figure 4.

There were considerable data reductions (see Section 2.3) in the three subtrees shown in Figure 4, and they ranged from 40% to 60%. These reductions led to the performance gain shown in Figure 5, which gives elapsed times of Q14 for left-deep and bushy join tree execution plans.

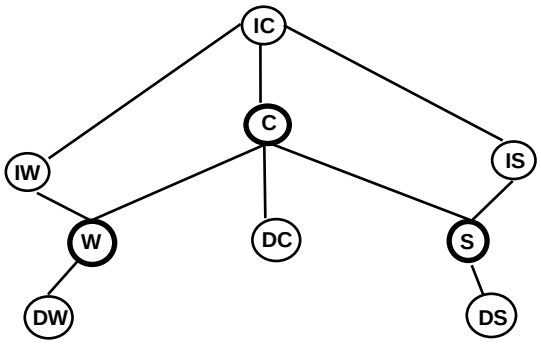

Figure 3. Join Graph for TPC-DS Q14

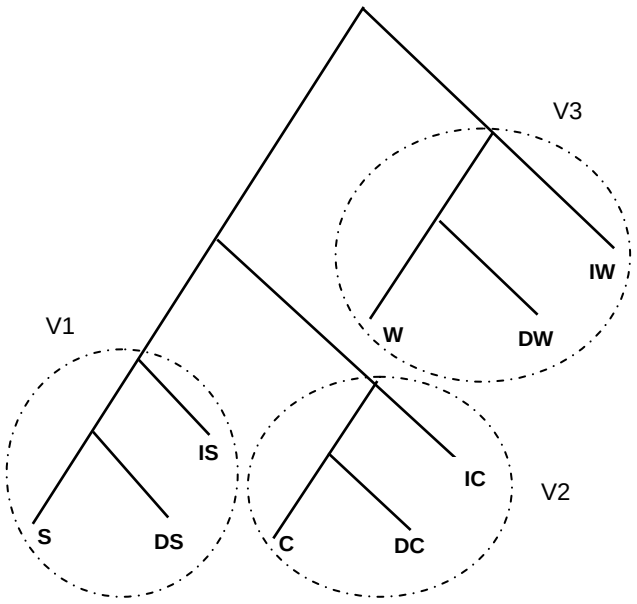

Figure 4. Bushy Join Tree for TPC-DS Q14

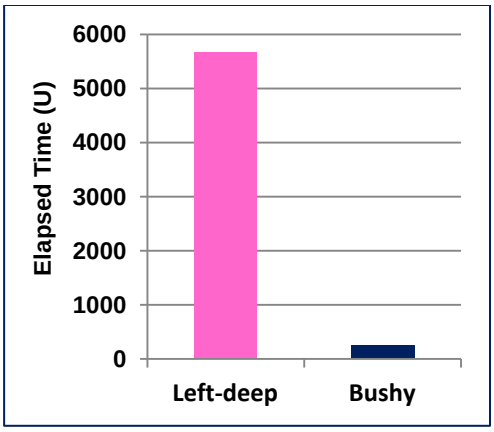

Figure 5. Elapsed Times for TPC-DS Q14

The graph in Figure 6 shows the elapsed times for left-deep and bushy join tree execution plans in our experiments for queries that have two fact tables. Q72, which seems to benefit most from bushy join tree, has two large fact tables, catalog_sales (431M) and inventory (585M); each fact table is joined with several dimension tables. Q29 shows a case where performance degraded due to optimizer cost mis-estimation.

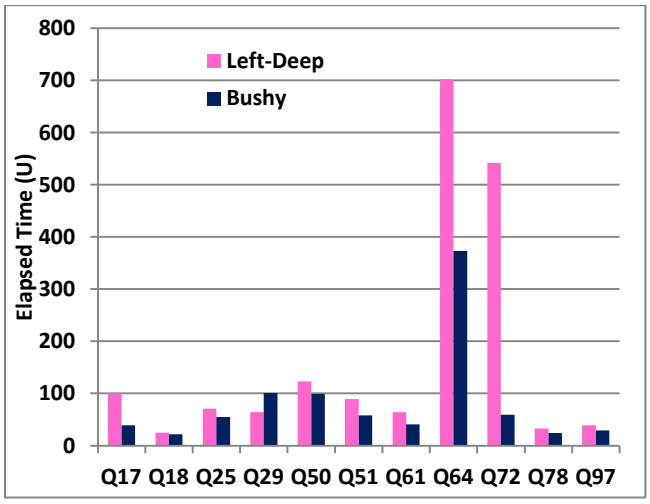

Figure 6. Elapsed time for TPC-DS Snowstorm Queries

3.2 Results for a Commercial Workload

We also conducted performance experiments on a real customer workload. The workload consists of a snowstorm schema with multiple fact tables and numerous dimension tables. The schema consists of 40 tables and 150 queries. The database size was 200GB and the system used was a high-performance parallel machine.

In this workload, there were many long-running snowstorm queries, which bore some structural similarity with one another. Most of these queries contained two large fact tables and several smaller dimension tables. Some queries contained two large fact tables joined with smaller dimension tables, but these fact tables were joined through another fact table that did not have any dimension tables, and therefore it did not belong to any table group (see Section 2.5). The base cardinalites of these fact tables were about 100M.

There were many queries in this workload that benefitted from the bushy join tree plans generated by the BJT algorithm. For the purpose of illustration, we chose query Q101 as a representative. Due to confidentiality reasons, we only present the join graph (Figure 7) for the query and not its SQL text.

The query Q101 had a total of twelve tables T1 to T12. The base cardinalities of these tables are as follows: T1 (100M), T2 (11M), T3 (1M), T4 (110), T5 (41K), T6 (20M), T7 (13K), T8 (60), T9 (29M), T10 (100M), T11 (1M) and T12 (1M). There were single-table filter predicates on tables T3, T9, T11 and T12. In the join graph, the fact tables, T1 and T10, are shown in bold circles. The rest are dimension tables except T3, which is a branch table.

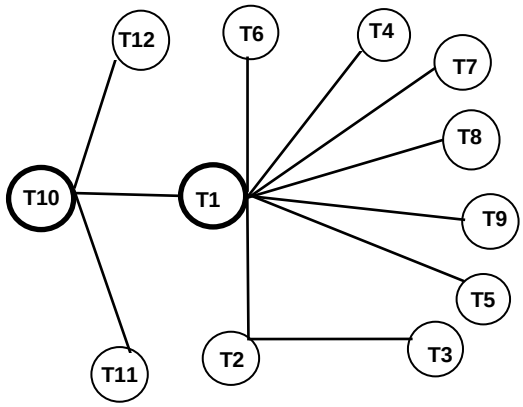

Figure 7. Join Graph for Query Q101

For query Q101, the BJT algorithm (Section 2.5) began by identifying two table groups: {T10, T11, T12} and {T1, T2, T3, T4, T5, T7, T8, T9}. After evaluating the costs of all the possible four states, it chose only the first table group to form a subtree (view). The bushy join tree generated by the Oracle optimizer for Q101 is shown in Figure 8.

In the subtree V1 (Figure 8), the joins of the fact table T10 (100M) with the dimension tables T11 (1M) and T12 (1M), which had single-table filter predicates, resulted in about1M rows; that is, it produced a data reduction of two orders of magnitude.

It should be clear that if two subtrees were formed for the two table groups of Q101, the join order will not be the same as the one shown in Figure 8. By forcing two subtrees in a bushy join tree for Q101 and evaluating its execution plan, we experimentally verified that the execution plan based on the bushy join tree of Figure 8 was the most optimal.

The elapsed times for query Q101 with a left-deep tree and the bushy join tree (Figure 8) execution plans were 622U and 53U respectively. The bushy join tree provided more than an order of magnitude performance improvement for Q101.

Performance improvements were also observed for many snowstorm queries in this workload. The results of an experiment with 50 queries are shown as a scatter plot in Figure 9. Most of the queries, which appear under the diagonal in Figure 9, showed substantial performance gains. There were several queries whose

performances either minimally improved or slightly degraded; these queries appear along or above the diagonal in Figure 9.

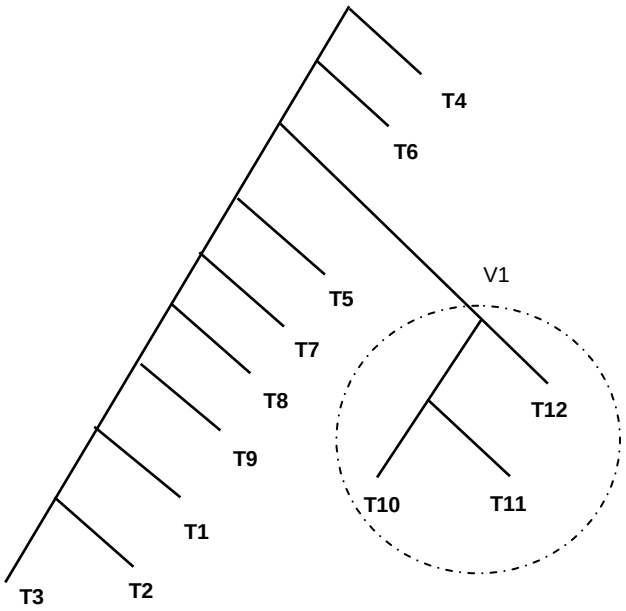

Figure 8. Bushy Join Tree for Query Q101

By analyzing the run-time statistics of queries that showed minimal improvement or degradation with bushy join trees, we discovered that in almost all the cases the subtrees yielded very little data reduction. The optimizer, however, still chose those execution plans due to cost mis-estimation.

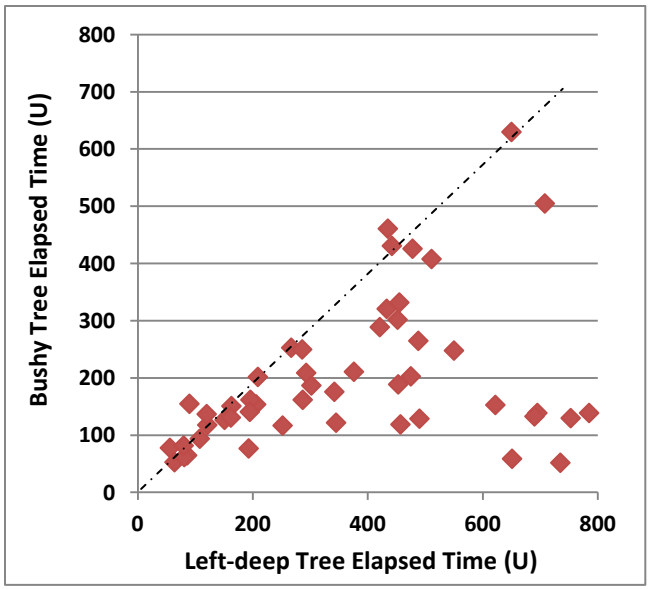

Figure 9. Performance Results for a Commercial Workload

We also did an experiment comparing the optimization times for left-deep and bushy tree plans for 20 queries with varying patterns from the commercial workload. The results are presented in Figure 10. There is an expected increase in the optimization time with bushy join tree plans. But the trade-off we got with the improvements in the elapsed times with bushy join tree plans made this increase in the optimization times almost negligible.

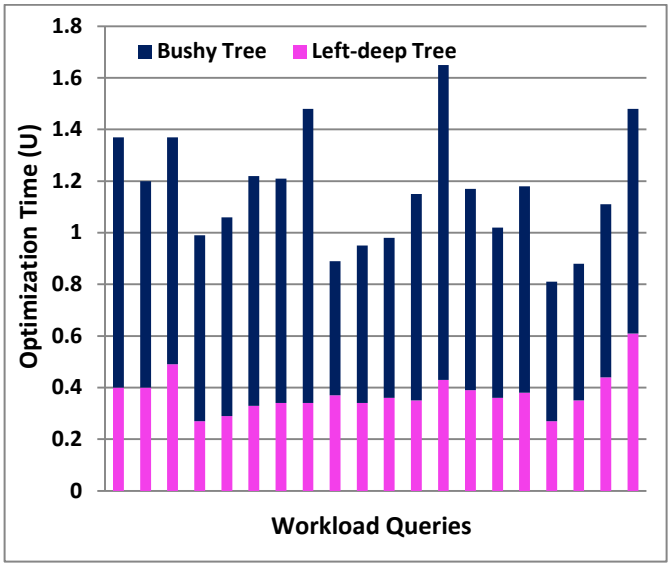

Figure 10. Optimization Times for Bushy and Left-deep Trees

4. RELATED WORK

Ono and Lohman [10] analyze the complexity of dynamic programming algorithms for finding an optimal join permutation. They consider chain, star, and clique queries and point out that an optimal plan for a multi-way join may contain Cartesian products and bushy join trees. When Cartesian products are considered, the number of joins enumerated is $O(3^N)$, but the worst case complexity of the enumerator is $O(4^N)$. Schneider and DeWitt [13] compare the performance of left-deep and right-deep trees in shared-nothing multiprocessor database machines. Vance and Maier [14] proposed an algorithm which generates bushy join tree order containing Cartesian products in a fast manner. Their main idea is to generate partial plans by separating join order enumeration from join predicate analysis. Moerkotte and Neumann [7] propose an algorithm for optimal bushy join trees that adapts to the search space of either chain or clique queries. Ioannidis and Kang [5] present analytical and experimental results for the space of both deep and bushy trees and discuss how iterative improvement, simulated annealing and two-phase search techniques perform on these two spaces; they note that the space of both the deep and bushy trees is easier to optimize than that of the left-deep tree alone. Du, Shan, and Dayal [3] present an algorithm for transforming a left-deep tree into a balanced bushy tree and show how this conversion can reduce response time in a multi-database environment.

5. CONCLUSION

This paper makes important contributions[1] by describing a technique for bushy join tree optimization. Our performance study shows that this optimization provides considerable execution time

[1] U.S. patents have been granted for the techniques presented in this paper.

improvements for complex snowstorm queries. In this technique, the bushy join tree contains subtrees constructed to yield substantial data reduction; and thus, their joins with other subtrees prove to be much more efficient than joins in the corresponding left-deep trees.

The performance study described in Section 3 and our experience with other commercial database workloads demonstrate that the search space of left-deep trees does not suffice for snowstorm queries and that bushy join tree execution plans can result in striking performance gain for this class of queries. By applying join predicate pushdown on bushy subtrees, the Oracle optimizer can render efficient interaction between index-based one-pass join and bushy join trees.

In our scheme, as an added optimization, the star transformation is performed for star/snowflake represented by each subtree formed within bushy join trees.

By generating bushy join trees only for snowstorm queries and by introducing efficient search strategies driven by the number of fact tables, the state space of bushy join trees have been significantly reduced. The Oracle optimizer, therefore, explores the space of bushy trees as the default strategy.

A possible future extension could relax some of the restrictions imposed by snowstorm schema and apply bushy join trees to a wider class of queries.

6. REFERENCES

[1] Ahmed, R., Lee, A., Witkowski, A., Das, D., Su, H., Cruanes, T., and Zait, M. Cost-Based Query Transformation in Oracle. *Proceedings of the 32nd VLDB Conference*, Seoul, S. Korea, 2006.

[2] Bellamkonda, S., Ahmed, R., Witkowski, A., Amor, A., Zait, M., and Lin, C. C. Enhanced Subquery Optimization in Oracle. *Proceedings of the 35th VLDB Conference*, Lyon, France, 2009.

[3] Du, W., Shan, M., and Dayal, U. Reducing Multi-database Query Response Time by Tree Balancing. *Proceedings of ACM SIGMOD*, San Jose, CA, U.S.A., 1995.

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

[5] Ioannidis, Y. E. and Kang, Y. C. Left-Deep Trees vs. Bushy Trees: An Analysis of Strategy Spaces and Its Implication for Query Optimization. *Proceedings of ACM SIGMOD*, 1991.

[6] Kimball, R. and Ross, M. *The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling*. John Wiley and Sons, Inc., 2nd Edition, 2002.

[7] Moerkotte, G., and Neumann, T. Analysis of Two Existing and One New Dynamic Programming Algorithm for the Generation of Optimal Bushy Join Trees with Cross Products. *Proceedings of the 32nd VLDB Conf*erence, Seoul, S. Korea, 2006.

[8] Moerkotte, G., and Scheufele, W. Constructing Optimal Bushy Processing Trees for Join Queries is NP-hard. Technical Report, , University of Mannheim, https://ub-madoc.bib.uni-mannheim.de/795/1/TR-96-011.pdf, 1996.

[9] O'Neil, P. and Graefe, G. Multi-Table Joins Through Bitmapped Join Indices. *SIGMOD Record*, Vol. 24, No.3, September, 1995.

[10] Ono, K., and Lohman, G. M. Measuring the Complexity of Join Enumeration in Query Optimization., *Proceedings of the 16th VLDB, Conference*, Brisbane, Australia, 1990.

[11] Othayoth, R., and Poess, M. The Making of TPC-DS. *Proceedings of the 32nd VLDB Conference*, Seoul, S. Korea, 2006.

[12] Selinger, P., Astrahan, M., Chamberlin, D., Lorie, R., and Price, T. Access Path Selection in a Database Management System. *Proceedings of ACM SIGMOD*, Boston, MA, U.S.A., 1979.

[13] Schneider, D.A., and DeWitt, D.J. Tradeoffs in Processing of Complex Join Queries via Hashing in Multiprocessor Database Machines. *Proceedings of 16th VLDB, Conference*, Brisbane, Australia, 1990.

[14] Vance, B. and Maier, D. Rapid Bushy Join Order Optimization with Cartesian Products. *Proceedings of the ACM SIGMOD*, Montreal, Canada, 1996.