Reducing Order Enforcement Cost in Complex Query Plans

Ravindra Guravannavar* S Sudarshan
Indian Institute of Technology Bombay
{ravig,sudarsha}@cse.iitb.ac.in

Abstract

Algorithms that exploit sort orders are widely used to implement joins, grouping, duplicate elimination and other set operations. Query optimizers traditionally deal with sort orders by using the notion of interesting orders. The number of interesting orders is unfortunately factorial in the number of participating attributes. Optimizer implementations use heuristics to prune the number of interesting orders, but the quality of the heuristics is unclear. Increasingly complex decision support queries and increasing use of covering indices, which provide multiple alternative sort orders for relations, motivate us to better address the problem of optimization with interesting orders.

We show that even a simplified version of the problem is NP-hard and give principled heuristics for choosing interesting orders. We have implemented the proposed techniques in a Volcano-style optimizer, and our performance study shows significant improvements in estimated cost. We also executed our plans on a widely used commercial database system, and on PostgreSQL, and found that actual execution times for our plans were significantly better than for plans generated by those systems in several cases.

1. Introduction

Decision support queries, extract-transform-load (ETL) operations, data cleansing and integration often use complex joins, aggregation, set operations and duplicate elimination. Sorting based query processing algorithms for these operations are well known. Sorting based algorithms are quite attractive when physical sort orders of one or more base relations fulfill the sort order requirements of operators either completely or partially. Further, secondary indices that cover a query[1] are being increasingly used in read-mostly environments. Query covering indices make it very efficient to obtain desired sort orders without accessing the data pages. These factors make it possible for sort based plans to significantly outperform hash based counterparts.

*Work supported by a Bell Laboratories Ph.D. fellowship.

[1] *I.e.,* contain all attributes of the relation that are used in the query.

The notion of interesting orders [10] has allowed optimizers to consider plans that could be locally sub-optimal, but produce orders that are beneficial for other operators, and thus produce a better plan overall. However, the number of interesting orders for most operators is factorial in the number of attributes involved. This is not acceptable as queries in the afore mentioned applications do contain large number of attributes in joins and set operations.

In this paper we consider the problem of optimization taking sort orders into consideration. We make the following technical contributions:

1. Often order requirements of operators are partially satisfied by inputs. For instance, consider a merge-join with join predicate ($r.c_1 = s.c_1$ *and* $r.c_2 = s.c_2$). A clustering index on $r.c_1$ (or on $r.c_2$ or $s.c_1$ or $s.c_2$) is helpful in getting the desired order efficiently; a secondary index that covers the query has the same effect.

   We highlight (in Section 3) the need for exploiting partial sort orders and show how a minor modification to the standard replacement selection algorithm can avoid run generation I/O completely when input is known to have a partial sort order. Further, we extend a cost-based optimizer to take into account partial sort orders.

2. We consider operators with flexible order requirements and address the problem of choosing good interesting orders so that complete or partial sort orders already available from inputs can be exploited.

   - In Sections 4 we show that a special case of finding optimal sort orders is NP-hard and give a 2-approximation algorithm to choose interesting sort orders for a join tree.
   - In Section 5 we address a more general case of the problem. In many cases, the knowledge of indices and available physical operators in the system allows us to narrow down the search space to a small set of orders. We formalize this idea (in Section 5.1) through the notion of *favorable orders*, and propose a heuristic to efficiently enumerate a small set of promising sort

orders. Unlike heuristics used in optimizer implementations, our approach takes into account issues such as (*i*) added choices of sort orders for base relations due to the use of query covering indices (*ii*) sort orders that partially match an order requirement (*iii*) requirement of same sort order from multiple inputs (*e.g.,* merge based join, union) and (*iv*) common attributes between multiple joins, grouping and set operations.

In Section 5.2 we also show how to integrate our extensions into a cost-based optimizer.

3. We present experimental results (in Section 6) evaluating the benefits of the proposed techniques. We compare the plans generated by our optimizer with those of three widely used database systems and show significant benefits due to each of our optimizations.

2. Related Work

Both System R [10] and Volcano [4] optimizers consider plans that could be locally sub-optimal but provide a sort order of interest to other operators and thus yield a better plan overall. However, the papers assume operators have one or few *exact* sort orders of interest. This is not true of operators like merge-join, merge-union, grouping and duplicate elimination, which have a factorial number of interesting orders, in the context of workloads with complex queries having multiple join/grouping attributes. Heuristics such as the PostgreSQL heuristic (described in Section 6), are commonly used by optimizers. Details of the heuristics are publicly available only for PostgreSQL. Further, System R and Volcano optimizers consider only those sort orders as useful that completely meet an order requirement. Plans that partially satisfy a sort order requirement are not handled. In this paper we address these two issues.

The seminal work by Simmen et.al. [11] describes techniques to infer orders from functional dependencies and predicates applied and thereby avoids redundant sort enforcers in the plan. The paper briefly mentions the problem of non-exact sort order requirements and mentions an approach of propagating an order specification that allows any permutation on the attributes involved. Though such an approach is possible for single input operators like group-by, it cannot be used for operators such as merge-join and merge-union for which the order guaranteed by both inputs must match. Moreover, the paper does not make it clear how the flexible order requirements are combined at other joins and group-by operators. Simmen et.al. [11] mention that the approach of carrying a flexible order specification also increases the coding complexity significantly. Our techniques work uniformly across all types of sort-based operators and can be easily incorporated into an existing optimizer. Work on inferring orders and groupings [11] [12] [8] [9] is independent and complementary to our work.

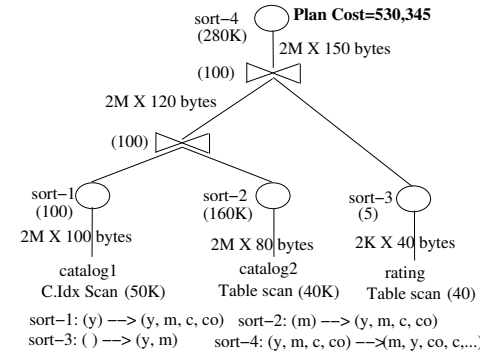


Figure 1. A naïve plan

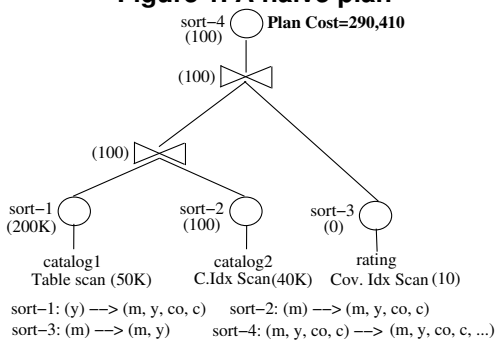


Figure 2. Optimal merge-join plan

3. Exploiting Partial Sort Orders

Often, sort order requirements of operators are partially satisfied by indices or other operators in the input subexpressions. A prior knowledge of partial sort orders available from inputs allows us to efficiently produce the required (complete) sort order more efficiently. When operators have flexible order requirements, it is thus important to choose a sort order that makes maximum use of partial sort orders already available. We motivate the problem with an example. Consider the query shown in Example 1. Such queries frequently arise in consolidating data from multiple sources. The join predicate between the two *catalog* tables involves four attributes and two of these attributes are also involved in another join with the *rating* table. Further, the order-by clause asks for sorting on a large number of columns including the columns involved in the join predicate.

Example 1 *A query with complex join condition*

```
SELECT c1.make, c1.year, c1.city, c1.color, c1.sellreason,
    c2.breakdowns, r.rating
FROM catalog1 c1, catalog2 c2, rating r
WHERE c1.city=c2.city AND c1.make=c2.make AND c1.year=c2.year
    AND c1.color=c2.color AND c1.make=r.make and c1.year=r.year
ORDER BY c1.make, c1.year, c1.color, c1.city,
    c1.sellreason, c2.breakdowns, r.rating;
```

The two catalog tables contain 2 million records each and have average tuple sizes of 100 and 80. We assume a disk block size of 4K bytes and 10000 blocks (40 MB) of main memory for sorting. The table *catalog1* is clustered on *year* and the table *catalog2* is clustered on *make*. The *rating* table has a secondary index on the *make* column with the *year* and *rating* columns included in the leaf pages (a covering index). Figures 1 and 2 show two different plans for the example query. Numbers in parentheses indicate estimated cost of the operators in number of I/Os (CPU cost is appropriately translated into I/O cost units). Edges are marked with the number of tuples expected to flow on that edge and their average size. For brevity, the input and output orders for the sort enforcers are shown using the starting letters of the column names. Though both plans use the same join order and employ sort-merge joins, the second plan is expected to perform significantly better than the first.

3.1. Changes to External Sort

External sorting algorithms have been studied extensively but in isolation. The standard replacement selection [6] for run formation well adapts with the extent to which input is presorted. In the extreme case, when the input is fully sorted, it generates a single run on the disk and avoids merging altogether. Larson [7] revisits run formation in the context of query processing and extends the standard replacement selection to handle variable length keys and to improve locality of reference (reduced cache misses). Estivill-Castro and Wood [2] provide a survey of adaptive sorting algorithms. The technique we propose in this section to exploit partial sort orders is a specific optimization in the context of multi-key external sorting. We observe that, by exploiting prior knowledge of partial sort order of input, it is possible to eliminate disk I/O altogether and have a completely pipelined execution of the sort operator.
We use the following notations: We use o, o_1, o_2 *etc.* to refer to sort orders. Each sort order o is a sequence of attributes/columns $(a_1, a_2, \ldots a_n)$. We ignore the sort direction (ascending/descending) as our techniques are applicable independent of the sort direction.

- ϵ : Empty (no) sort order
- $attrs(o)$: The set of attributes in sort order o
- $|o|$: Number of attributes in the sort order o
- $o_1 \leq o_2$: Order o_2 subsumes order o_1 (o_1 is a prefix of o_2)
- $o_1 < o_2$: Order o_1 is a strict prefix of o_2

Consider a case where the sort order to produce is (col_1, col_2) and the input already has the order (col_1). Standard replacement-selection writes a single large run to the disk and reads it back again; this breaks the pipeline and incurs substantial I/O for large inputs. It is not difficult to see how the standard replacement-selection can be modified to exploit the partial sort orders. Let $o = (a_1, a_2, \ldots a_n)$ be the desired sort order and $o' = (a_1, a_2, \ldots a_k)$, $k < n$ be the partial sort order known to hold on the input. At any point during sorting we need to retain only those tuples that have the same value for attributes $a_1, a_2, \ldots a_k$. When a tuple with a new value for these set of attributes is read, all the tuples in the heap (or on disk if there are large number of tuples matching a given value of $a_1, a_2, \ldots a_k$) can be sent to the next operator in sorted order. Thus in most cases, partial sort orders allow a completely pipelined execution of the sort. Exploiting partial sort orders in this way has several benefits:

1. Let $o = (a_1, a_2, \ldots a_n)$ be the desired sort order and $o' = (a_1, a_2, \ldots a_k)$, $k < n$ be the partial sort order known to already hold on the input. We call the set of tuples that have the same value for attributes $(a_1, a_2, \ldots a_k)$ as a *partial sort segment*. If each *partial sort segment* fits in memory (which is quite often the case in practice), the entire sort operation can be completed without any disk I/O.

2. Exploiting partial sort orders allows us to output tuples early (as soon as a new segment starts). In a pipelined execution this can have large benefits. Moreover, producing tuples early has immense benefits for Top-K queries and situations where the user retrieves only some result tuples.

3. Sorting each *partial sort segment* independently, reduces the number of comparisons significantly. Note that we empty the heap every time a new segment starts and hence insertions into heap will be faster. In general, independently sorting k segments each of size n/k elements, has the complexity $O(k * n/k\ log(n/k)) = O(n\ log(n/k))$ as against $O(n\ log(n))$ for sorting all n elements. Further, while sorting each *partial sort segment* comparisons need to be done on fewer attributes, $(a_{k+1}, \ldots a_n)$ in the above case.

Our experiments (Section 6) confirm that the benefits of exploiting partial sort orders can be substantial and yet none of the systems we evaluated, though widely used, exploited partial sort orders.

3.2. Optimizer Extensions for Partial Sort Orders

In this section we assume order requirements of operators are concrete and only focus on incorporating partial sort orders. We deal with flexible order requirements in subsequent sections. We use the following notations:

- $o_1 \wedge o_2$: Longest common prefix between o_1 and o_2
- $o_1 + o_2$: Order obtained by concatenating o_1 and o_2

- $o_1 - o_2$: Order o' such that $o_2 + o' = o_1$ (defined only when $o_2 \leq o_1$)
- $coe(e, o_1, o_2)$: The cost of enforcing order o_2 on the result of expression e which already has order o_1
- $N(e)$: Expected size, in number of tuples, of the result of expression e
- $B(e)$: Expected size, in number of blocks, of the result of expression e
- $D(e, s)$: Number of distinct values for attribute(s) s of expression e ($= N(\Pi_s(e))$)
- $cpu\_cost(e, o)$: CPU cost of sorting result of e to get order o
- M : Number of memory blocks available for sorting

The *Volcano* optimizer framework [4] assumes an algorithm (physical operator) either guarantees a required sort order fully or it does not. Further, a physical property enforcer (such as sort) only knows the property to be enforced and has no information about the properties that hold on its input. The optimizer's cost estimate for the enforcer thus depends only on the required output property (sort order). In order to remedy these deficiencies we extended the optimizer in the following way: Consider an optimization goal (e, o), where e is the expression and o the required output sort order. If the physical operator being considered for the logical operator at the root of e guarantees a sort order $o' < o$, then the optimizer adds a partial sort enforcer *enf* to enforce o from o'. We use the following cost model to account for the benefits of partial sorting.

$$coe(e, \epsilon, o) = \begin{cases} cpu\text{-}cost(e, o) & \text{if } B(e) \leq M \\ B(e)(2\lceil log_{M-1}(B(e)/M)\rceil + 1) & \text{otherwise} \end{cases}$$

If e is known to have the order o_1, we estimate the cost of obtaining an order o_2 as follows:
$coe(e, o_1, o_2) = D(e, attrs(o_s)) * coe(e', \epsilon, o_r)$, where $o_s = o_2 \wedge o_1$, $o_r = o_2 - o_s$ and $e' = \sigma_p(e)$, where p equates attributes in o_s to an arbitrary constant. Intuitively, we consider the cost of sorting a single *partial sort segment* independently and multiply it by the number of segments. Note that we assume uniform distribution of values for $attrs(o_s)$. Therefore, we estimate $N(e') = N(e)/D(e, attrs(o_s))$ and $B(e') = B(e)/D(e, attrs(o_s))$. When the actual distribution of values is available, a more accurate cost model that does not rely on the uniform distribution assumption can be used.

4. Choosing Sort Orders for a Join Tree

Consider a join expression $e = e_1 \bowtie e_2$, where e_1, e_2 are input subexpressions and the join predicate is of the form: $(e_1.a_1 = e_2.a_1 \; and \; e_1.a_2 = e_2.a_2 \ldots and \; e_1.a_n = e_2.a_n)$. Note that, *w.l.g.*, we use the same name for attributes being compared from either side and we call the set $\{a_1, a_2, \ldots, a_n\}$ as the join attribute set. In this case, the merge join algorithm has potentially $n!$ interesting sort orders on inputs e_1 and e_2[2]. The specific sort order chosen for the merge-join can have significant influence on the plan cost due to the following reasons: (*i*) Clustering and covering indices, indexed materialized views and other operators in the subexpressions e_1, e_2 can make one sort order much cheaper to produce than another. (*ii*) The merge-join produces the same order on its output as the one selected for its inputs. Hence, a sort order that helps another operator above the merge-join can help eliminate a sort or just have a partial sort. In this section we show that a special case of the the problem of choosing optimal sort orders for a tree of merge-joins is *NP-Hard* and provide a 2-approximate algorithm for the problem. In the next section, we describe our heuristics for a more general setting of the problem in which we make use of the proposed 2-approximate algorithm.

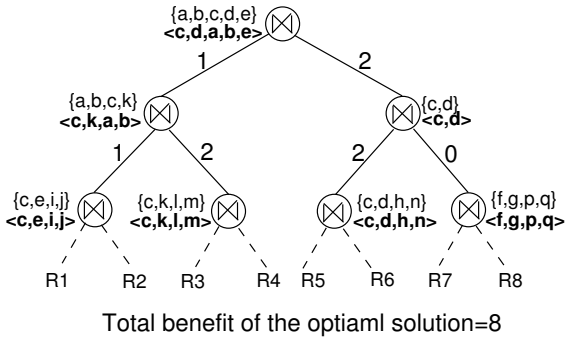


Figure 3. Optimal sort orders (a special case)

4.1. Finding Optimal Sort Orders is NP-Hard

Consider a join expression $e = R_1 \bowtie R_2 \bowtie R_3 \ldots R_n$ and a specific join order tree for the expression. Consider a special case where all base relations and intermediate results are of the same size and no indices built on the base relations. Now, the problem of choosing optimal sort orders for each join requires us to choose permutations of join attributes such that we maximize the length of longest common prefixes of permutations chosen for adjacent nodes. Figure 3, shows an example and an optimal solution under the model where the benefit for an edge is the length of the longest common prefix between the permutations chosen for adjacent nodes and we maximize the total benefit. The join attribute set for each join node is shown in curly braces besides the node. Permutations chosen in the optimal solution are indicated with angle brackets and the number on each edge shows the benefit for that edge. Below we state the problem formally.

[2]We assume merge-join requires sorting on all attributes involved in the join predicate. We do not consider orders on subsets of join attributes since the additional cost incurred at merge-join matches the benefit of sorting a smaller subset of attributes.

Problem 1 *Let T be a binary tree of order n, with vertex set $V(T)$ and edge set $E(T)$. Each node v_i $(i = 1, \ldots n)$ is associated with an attribute set s_i. Find a sequence of permutations $p_1, p_2 \ldots p_n$, where p_i is a permutation of set s_i, such that the benefit function $\mathcal{F}$ is maximum.*

$$\mathcal{F} = \sum_{\forall v_i v_j \in E(T)} |p_i \wedge p_j|$$

Theorem 4.1 *Problem 1 is NP-hard.* □

The known *NP-Hard* problem SUM-CUT [1] is reducible to Problem 1. A formal proof can be found in the technical report [5].

4.2. A 2-Approximate Algorithm

An efficient dynamic programming based algorithm to find the optimal solution (under the benefit model presented in the previous section) for Problem 1 exists when the tree is a *path*. Note that left-deep and right-deep join plans result in paths. We present the solution for paths in brief and make use of the same in the 2-approximation for binary trees. The detailed algorithm can be found in [5].

Consider a path $v_1, v_2, \ldots v_n$, where each vertex v_i has an associated attribute set s_i. The optimal solution for any segment (i, j) of the path, $\text{OPT}(i, j) = \max\{\ \text{OPT}(i, k) + \text{OPT}(k + 1, j) + c(i, j)\ \}$ over all $i \leq k < j$, where $c(i, j)$ is the number of common attributes for the segment (i, j).

For binary trees we propose an approximation with benefit at least half that of an optimal solution. We split the tree into two sets of paths, P_o and P_e. P_o has the paths formed by edges at odd levels and P_e has those formed by edges at even levels, Figure 4 shows an example. We obtain an optimal solution for each of the two sets of paths. Let the the optimal solutions for the two sets of paths be S_o and S_e and the corresponding benefits be $ben(S_o)$ and $ben(S_e)$. Let the set of edges included in P_o and P_e be denoted by E_o and E_e respectively. Consider an optimal solution S_T for the whole tree. In the optimal solution, let the sum of benefits of all edges in E_o be $odd\text{-}ben(S_T)$ and that of edges in E_e be $even\text{-}ben(S_T)$. Note that $ben(S_o) \geq odd\text{-}ben(S_T)$ and $ben(S_e) \geq even\text{-}ben(S_T)$. Since the total benefit of the optimal solution $ben(S_T) = odd\text{-}ben(S_T) + even\text{-}ben(S_T)$, we have $ben(S_o) + ben(S_e) \geq ben(S_T)$. Hence at least one of $ben(S_o)$ or $ben(S_e)$ is $\geq 1/2\ ben(S_T)$. There may be vertices not included in the chosen solution, e.g., the even level split in Figure 4 does not include the root and leaf nodes. For these left over vertices arbitrary permutations can be chosen.

5. Optimization Exploiting Favorable Orders

The benefit model we presented in the previous section and the approximation algorithm do not take into account indices and size of relations or intermediate results. Moreover, we assumed that the join order tree is fixed. In this section we present a two phase approach for the more general problem. In phase-1, which occurs during plan generation, we exploit the information of available indices and properties of physical operators to efficiently compute a small set of promising sort orders to try. We formalize this idea through the notion of *favorable orders*. Phase-2, is a plan refinement step and occurs after the optimizer makes its choice of the best plan. In phase-2, the sort orders chosen by the optimizer are refined further to reap extra benefit from the attributes common to multiple joins. Phase-2 uses the 2-approximate algorithm of Section 4.2

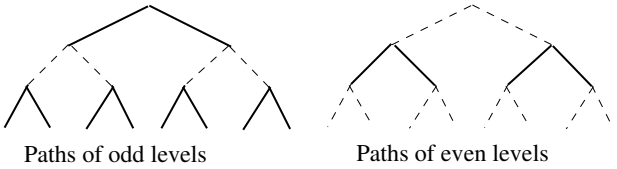


Figure 4. A 2-approximation for binary trees

5.1. Favorable Orders

Given an expression e, we expect some sort orders (on the result of e) to be producible at much lesser cost than other sort orders. Available indices, indexed materialized views, specific rewriting of the expression and choice of physical operators determine what sort orders are easy to produce. To account for such orders, we introduce the notion of *favorable* orders. We use the following notations:

- $cbp(e, o)$: Cost of the best plan for expression e with o being the required output order
- o_R : The clustering order of relation R
- $idx(R)$: Set of all indices over R
- $o(I)$: Order (key) of the index I
- $\langle s \rangle$: An arbitrary order (permutation) on attribute set s
- $o \wedge s$: Longest prefix of o such that each attribute in the prefix belongs to the attribute set s
- $schema(e)$: The set of attributes in the output of e

We first define the ***benefit*** of a sort order o *w.r.t.* an expression e as follows:

$$benefit(o, e) = cbp(e, \epsilon) + coe(e, \epsilon, o) - cbp(e, o)$$

Intuitively, a positive benefit implies the order can be obtained with lesser cost than a full sort of unordered result. For instance, the clustering order of a relation r will have a positive *benefit* for the expression $\sigma_p(r)$. Similarly, query covering secondary indices and indexed materialized views can yield orders with positive benefit. We call the set of all orders, on $schema(e)$, having a positive benefit *w.r.t.* e as the *favorable order set* of e and denoted it as $ford(e)$.

$$ford(e) = \{\ o:\ benefit(o, e) > 0\ \}$$

5.1.1 Minimal Favorable Orders

The number of favorable orders for an expression can be very large. For instance, every order having the clustering order as its prefix is a favorable order. A *minimal favorable order set* of e, denoted by *ford-min(e)*, is the minimum size subset of *ford(e)* such that, for each order $o \in$ *ford(e)*, at least one of the following is true:

1. o belongs to *ford-min(e)*
2. $\exists\ o' \in$ *ford-min(e)* such that $o' \leq o$ and $cbp(e, o') + coe(e, o', o) = cbp(e, o)$ Intuitively, if the cost of obtaining order o equals the cost of obtaining a partial sort order o' followed by an explicit sort to get o, we include only o' in the *ford-min*
3. $\exists\ o'' \in$ *ford-min(e)* such that $o \leq o''$ and $cbp(e, o'') = cbp(e, o)$ Intuitively, if an order o'' subsumes order o and has the same cost, we include only o'' in *ford-min*

Conditions 2 and 3 above, ensure that when a relation has an index on o, orders that are prefixes of o and orders that have o as their prefix are not included unnecessarily.
We define favorable orders of an expression *w.r.t.* a set of attributes s as: *ford(e, s)*= $\{o \wedge s$: $o \in$ *ford(e)*$\}$ Intuitively, *ford(e, s)* is the set of orders on s or a subset of s that can be obtained efficiently. Similarly, the *ford-min* of an expression *w.r.t.* a set of attributes s is defined as: *ford-min(e, s)*= $\{o \wedge s : o \in$ *ford-min(e)*$\}$

5.1.2 Heuristics for Favorable Orders

Note that the definition of favorable orders uses the cost of the best plan for the expression. However, we need to compute the favorable orders of an expression **before** the expression is optimized and without requiring to expand the logical or the physical plan space. Further, the size of the exact *ford-min* of an expression can be prohibitively large in the worst case. In this section, we describe a method of computing approximate *ford-min*, denoted as *afm*, for *SPJG* expressions. We compute the *afm* of an expression bottom-up. For any expression e, *afm(e)* is computable after the *afm* is computed for all of e's inputs.

1. $e = R$, where R is a base relation or materialized view. We include the clustering order of R and all secondary index orders such that the index covers the query.
   $afm(R) = \{o : o = o_R$ or $o = o(I), I \in idx(R)$ and I covers the query$\}$
2. $e = \sigma_p(e_1)$, where e_1 is an arbitrary expression.
   $afm(e) = \{o : o \in afm(e_1)\ \}$
3. $e = \Pi_L(e_1)$, where e_1 is any expression. We include longest prefixes of input favorable orders such that the prefix has only the projected attributes.
   $afm(e) = \{o : \exists o' \in afm(e_1)$ and $o = o' \wedge L\}$
4. $e = e_1 \bowtie e_2$ with join attribute set $S = \{a_1, a_2, \ldots a_n\}$. Noting that nested loops joins propagate the sort order of one of the inputs (outer) and merge join propagates the sort order chosen for join attributes, we compute the *afm* as follows. First, we include all sort orders in the input *afms*. Next, we consider the longest prefix of each input favorable order having attributes only from the join attribute set and extend it to include an arbitrary permutation of the remaining join attributes.
   $afm(e_1 \bowtie e_2) = \mathcal{T} \bigcup \{o : o' \in \mathcal{T} \bigcup \{\epsilon\}$ *and* $o = o' \wedge S + \langle S - attrs(o' \wedge S)\rangle\}$, where $\mathcal{T}$ =$afm(e_1) \bigcup afm(e_2)$

   Note that, for the join attributes not involved in an input favorable order prefix (*i.e.,* $S - attrs(o' \wedge S)$), we take an arbitrary permutation. An exact *ford-min* would require us to include all permutations of such attributes. In the post-optimization phase, we refine the choice made here using the benefit model and algorithm of Section 4.2.
5. $e =_L \mathcal{G}_F(e_1)$
   $afm(e) = \{o : o' \in afm(e_1) \bigcup \{\epsilon\}$ *and* $o = o' \wedge L + \langle L - attrs(o' \wedge L)\rangle\}$
   Intuitively, for each input favorable order we identify the longest prefix with attributes from the projected group-by columns and extend the prefix with an arbitrary permutation of the remaining attributes.

Computing *afms* requires a single pass of the query tree. At each node of the query tree the only significant operation performed is computation of longest common prefix ($o \wedge s$). Although in the worst case the number of such operations is exponential in the number of joins (see [5] for details), in practice we have found it to be quite small even for complex queries (Section 6.3).

5.2. Overall Optimizer Extensions

We make use of the approximate favorable orders during plan generation (phase-1) to choose a small set of promising *interesting orders*. We describe our approach taking merge join as an example but the approach is applicable to other sort based operators. In phase-2, which is a post-optimization phase, we further refine the chosen sort orders.

5.2.1 Plan Generation (Phase-1)

Consider an optimization goal of expression $e = e_l \bowtie e_r$ and required output sort order o. When we consider merge-join as a candidate algorithm, we need to generate sub-goals for e_l and e_r with the required output sort order being some permutation of the join attributes.

Let S be the set of attributes involved in the join predicate. We consider only conjunctive and equality predicates. We compute the set $\mathcal{I}(e, o)$ of interesting orders as follows:

1. Collect the favorable orders of inputs plus the required output order
   $\mathcal{T}(e,o) = afm(e_l,S) \bigcup afm(e_r,S) \bigcup o \wedge S$, where $afm(e,S) = \{o' \wedge S : o' \in afm(e)\}$
2. Remove redundant orders
   If $o_1, o_2 \in \mathcal{T}(e,o)$ and $o_1 \leq o_2$, remove o_1 from $\mathcal{T}(e,o)$
3. Compute the set $\mathcal{I}(e,o)$ by extending each order in $\mathcal{T}(e,o)$ to the length of $|S|$; the order of extra attributes can be arbitrarily chosen
   $\mathcal{I}(e,o) = \{o : o' \in \mathcal{T}(e,o) \text{ and } o = o' + \langle S - attrs(o')\rangle\}$

We then generate optimization sub-goals for e_l and e_r with each order $o' \in \mathcal{I}(e,o)$ as the required output order and retain the cheapest combination.

A Note on Optimality: If the set $\mathcal{I}(e,o)$ is computed using the exact *ford-mins* instead of *afms*, we claim that it must contain an optimal sort order (a sort order that produces the optimal merge join plan in terms of overall plan cost). The detailed proof of this claim can be found in [5].

An Example: Consider Example 1 of Section 3. For brevity, we refer to the two catalog tables as *ct1* and *ct2*, the rating table as *rt* and the columns with their starting letters. The *afms* computed as described in Section 5.1.2 are as follows: $afm(ct1) = \{(y)\}$, $afm(ct2) = \{(m)\}$, $afm(rt) = \{(m)\}$, $afm(ct1 \bowtie ct2) = \{(y,co,c,m),(m,co,c,y)\}$, $afm((ct1 \bowtie ct2) \bowtie rt) = \{(y,m),(m,y)\}$ For $(ct1 \bowtie ct2) \bowtie rt$ we consider two interesting sort orders $\{(y,m),(m,y)\}$ and for $ct1 \bowtie ct2$ we consider the four orders $\{(y,co,c,m),(m,co,c,y),(y,m,co,c),(m,y,co,c)\}$. As a result the optimizer arrives at plan shown in Figure 2.

5.2.2 Plan Refinement (Phase-2)

During the plan refinement phase, for each merge-join node in the plan tree, we identify the set of *free attributes*, the attributes which were not part of any of the input favorable orders. Note that for these attributes we had chosen an arbitrary permutation while computing the *afm* (Section 5.1.2). We then make use of the 2-approximate algorithm for trees (Section 4.2) and rework the permutations chosen for the *free attributes*.

Formally, let p_i be the permutation chosen for the join node v_i. Let q_i be the order such that $q_i \in afm(v_i.\textit{left-input}) \cup afm(v_i.\textit{right-input})$ and $|p_i \wedge q_i|$ is the maximum. Intuitively, q_i is the input favorable order sharing the longest common prefix with p_i. Let $f_i = attrs(p_i - (p_i \wedge q_i))$; f_i is the set of *free attributes* for v_i.

We now construct a binary tree where each node n_i corresponding to join-node v_i is associated with the attribute set f_i. The orders for the nodes are chosen using the 2-approximate algorithm; the chosen order for free attributes is then appended to the order chosen during plan generation (*i.e.,* $p_i \wedge q_i$) to get a complete order.

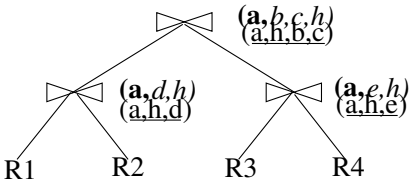


Figure 5. Post-Optimization Phase

The reworking of the orders will be useful only if the adjacent nodes share the same prefix, i.e., $p_i \wedge q_i$ was the same for adjacent nodes. This condition however certainly holds when the inputs for joins have no favorable orders.

Figure 5 illustrates the post-optimization phase. Assume all relations involved ($R_1 \ldots R_4$) are clustered on attribute a and no other favorable orders exist. *i.e.,* $afm(R_i) = \{(a)\}$, for $i = 1$ to 4. The orders chosen by the plan generation phase are shown besides the join nodes with *free attributes* being in *italics*. The reworked orders after the post-optimization phase are shown underlined.

6. Experimental Results

We performed experiments to evaluate the benefits our techniques. For comparison, we use PostgreSQL (v. 8.1.3) and two widely used commercial database systems (we call them SYS1 and SYS2). All tests were run on an Intel P4 (HT) PC with 512 MB of RAM. We used TPC-H 1GB dataset and additional tables as specified in the individual test cases. For each table, a clustering index was built on the primary key. Additional secondary indices built are specified in the test cases. All relevant statistics were built and the optimization level for one of the systems, which supports multiple levels of optimization, was set to the highest.

6.1. Modified Replacement Selection

The first set of experiments evaluate the benefits of modified replacement selection (MRS) as compared to the standard replacement selection (SRS) when the input is known to be partially sorted.

External sort in PostgreSQL employs the standard replacement selection (SRS) algorithm suitably adapted for variable length records. We modified this implementation to exploit partial sort orders available on the input.

Experiment A1: The experiment consists of an ORDER BY of the TPC-H *lineitem* table on two columns *(l_suppkey, l_partkey)*.

Query 1 *ORDER-BY on lineitem*
```
SELECT l_suppkey, l_partkey FROM lineitem
ORDER BY l_suppkey, l_partkey;
```

A secondary index on *l_suppkey* was available that covered the query (included the *l_partkey* column)[3]. On all

[3]On systems not supporting indexes with included columns, we used a table with only the desired two columns, clustered on *l_suppkey*

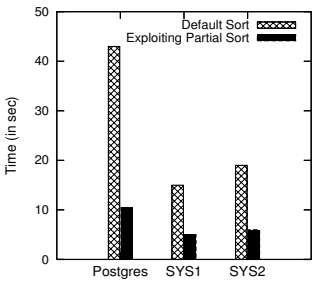


Figure 6. Exp A1

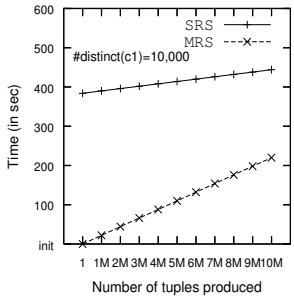


Figure 7. Output Rate

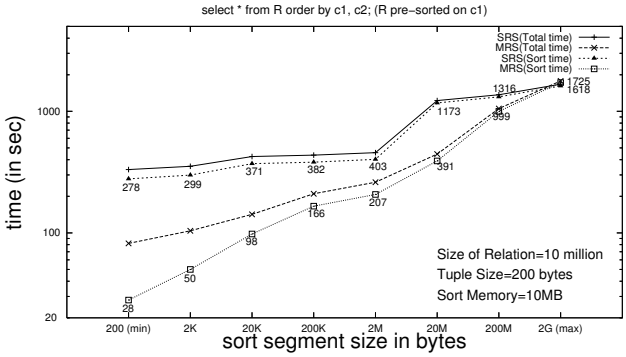


Figure 8. Effect of Partial Sort Segment Size

three systems, the order by on *(l_suppkey, l_partkey)* took almost the same time as an order by on *(l_partkey, l_suppkey)* showing that the sort operator of these systems did not exploit partial sort orders effectively. We compared the running times with our implementation that exploited partial sort order *(l_suppkey)* and the results are shown in Figure 6.

On SYS1 and SYS2 we simulated the partial sorting using a correlated rank query (as we did not have access to their source code). The subquery sorted the index entries matching a given *l_suppkey* on *l_partkey* and the subquery was invoked with all *suppkey* values so as to obtain the desired sort order of *(l_suppkey, l_partkey)*.

By avoiding run generation I/O and making reduced comparisons, MRS performs 3-4 times better than SRS.

Experiment A2: The second experiment shows how MRS is superior in terms of its ability to produce records early and uniformly. Table R_3 having 3 columns $(c1, c2, c3)$ was populated with 10 million records and was clustered on $(c1)$. The query asked an order by on $(c1, c2)$. Figure 7 shows the plot of number of tuples produced vs. time with cardinality of $c1 = 10,000$.

MRS starts producing the tuples without any delay after the operator initialization where as SRS produces its first output tuple only after seeing all input tuples. By producing tuples early, MRS speeds up the pipeline significantly and also helps Top-K queries.

Experiment A3: The third experiment shows the effect of *partial sort segment size* on sorting. 8 tables $R_0 \ldots R_7$, with identical schema of 3 columns $(c1, c2, c3)$ were each populated with 10 million records and average record size of 200 bytes. Each table was clustered on $(c1)$. Table R_i had 10^i tuples for each value of $c1$, resulting in a partial sort segment size of 200×10^i bytes. Thus R_0 had $c1$ as unique and sort segment size of 200 bytes and R_7 had the same value of $c1$ for all 10 million records leading to a sort segment size of 2GB. The query asked for an order by on $(c1, c2)$. The running times with default and modified replacement selection on PostgreSQL are shown in Figure 8.

When the partial sort segment size is small enough to fit in memory (up to 10MB or 50K records), SRS produces a single sorted run on disk and does not involve merging of runs. The modified replacement selection (MRS) gets the benefit of avoiding I/O and reduced number of comparisons. When the partial sort segment size becomes too large to fit in memory, we see a sudden rise in the time taken by SRS. This is because replacement selection will have to deal with merging several runs. MRS however deals with merging smaller number of runs initially as each partial sort segment is sorted separately. As the partial sort segment size increases, the running time of MRS rises and becomes same as that of SRS at the extreme point where all records have the same value for the prefix.

Experiment A4: To see the influence of MRS on a query having other operators, we considered a query that asked for counting the number of lineitems for each supplier, part pair. Two indices, *lineitem(l_suppkey)* and *partsupp(ps_suppkey)*, each of which included other required columns supplied the required sort order partially.

Query 2 *Number of lineitems for each (supplier, part) pair*

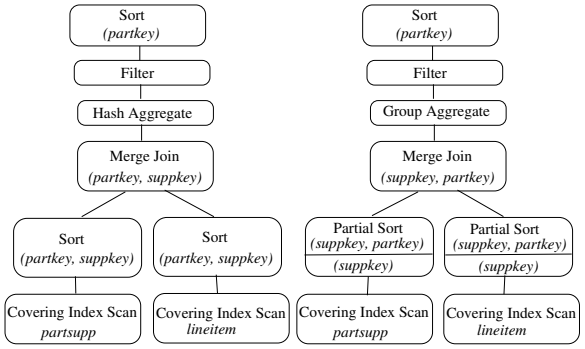
```
SELECT   ps_suppkey, ps_partkey, ps_availqty, count(l_partkey)
FROM     partsupp, lineitem
WHERE    ps_suppkey=l_suppkey AND ps_partkey=l_partkey
GROUP BY ps_suppkey, ps_partkey, ps_availqty
ORDER BY ps_suppkey, ps_partkey;
```

The query took 63 seconds to execute with SRS and 25 seconds with MRS, both on Postgres. The query plan used in both cases was the same - a merge join of the two relations on *(suppkey, partkey)* followed by an aggregate.

6.2. Choice of Interesting Orders

We extended our Volcano-style cost based optimizer, which we call PYRO, to consider partial sort orders and choose good interesting sort orders for merge joins and aggregation. We compare the plans produced by the extended implementation, which we call PYRO-O, with those of Postgres, SYS1 and SYS2.

Experiment B1: For this experiment we used Query 3 given below, which lists parts for which the outstanding order quantity is more than the stock available at the supplier.



(a) Default Plan (Postgres) (b) Plan Generated by PYRO–O

Figure 9. Plans for Query 3

Query 3 *Parts Running Out of Stock*

```
SELECT  ps_suppkey, ps_partkey, ps_availqty, sum(l_quantity)
FROM    partsupp, lineitem
WHERE   ps_suppkey=l_suppkey AND ps_partkey=l_partkey AND
        l_linestatus='O'
GROUP BY ps_availqty, ps_partkey, ps_suppkey
HAVING sum(l_quantity) > ps_availqty ORDER BY ps_partkey;
```

Table *partsupp* had clustering index on its primary key *(ps_partkey, ps_suppkey)*. Two secondary indices, one on $ps\_suppkey$ and the other on $l\_suppkey$ were also built on the *partsupp* and *lineitem* tables respectively. The two secondary indices covered all attributes needed for the query. Figures 9 shows the plans chosen by Postgres and PYRO-O. SYS1 chose a hash-join plan by default. When a merge-join plan was forced with an optimizer hint, SYS1 selected a plan similar to that of Postgres, except that it avoided the sort of *partsupp* by using the clustering index. The default plan on SYS2 was same as the merge-join plan of SYS1.

All plans except the hash-join plan of SYS1 and the plan produced by PYRO-O use an expensive full sort of 6 million lineitem index entries on *(partkey, suppkey)*. Further, Postgres uses a hash aggregate where a sort-based aggregate would have been much cheaper as the required sort order was available from the output of merge-join (note that the functional dependency {ps_partkey, ps_suppkey} → {ps_availqty} holds).

We compared the actual running time of PYRO-O's plan with those of Postgres and SYS1 by forcing our plan on the respective systems. Figures 10 and 11 show the details. It was not possible for us to force our plan on SYS2 and make a fair comparison and hence we omit the same. The only surprising result was the default plan chosen by SYS1 performed slightly poorer than the forced merge-join plan. In all cases, the forced PYRO-O plan performed significantly better than the other plans. The main reasons for the improvement were the choice of a good sort order, and the use of a partial sort of *lineitem* index entries instead of a full sort. The final sort on *partkey* was not very expensive as

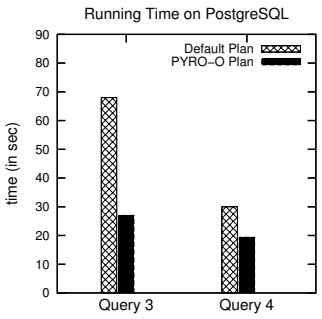


Figure 10. Postgres

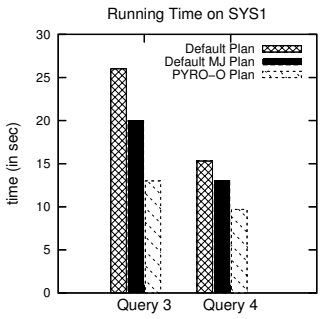
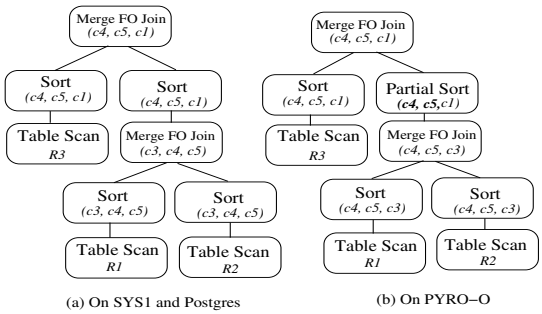


Figure 11. SYS1



(a) On SYS1 and Postgres (b) On PYRO–O

Figure 12. Plans for Query 4

only a few tuples needed to be sorted.

For Query 3 the plan generation phase (phase-1) was sufficient to select the sort orders and phase-2 does not make any changes. We shall now see a case for which phase-1 cannot make a good choice and the sort orders get refined by phase-2.

Experiment B2: This experiment uses Query 4, shown below, which has two full outer joins with two common attributes between the joins.

Query 4 *Attributes common to multiple joins*

```
SELECT * FROM R1 FULL OUTER JOIN R2
    ON (R1.c5=R2.c5 AND R1.c4=R2.c4 AND R1.c3=R2.c3)
    FULL OUTER JOIN R3
    ON (R3.c1=R1.c1 AND R3.c4=R1.c4 AND R3.c5=R1.c5);
```

The tables R1, R2 and R3 were identical and each populated with 100,000 records. No indexes were built. As shown in Figure 12(a), both SYS1 and Postgres chose sort orders that do not share any common prefix. The plan chosen by PYRO-O is shown in Figure 12(b). In the plan chosen by PYRO-O, the two joins share a common prefix of *(c4, c5)* and thus the sorting effort is expected to be significantly less. SYS2, not having an implementation of full outer join, chose a union of two left outer joins. The two left outer joins used to get a full outer join used different sort orders making the union expensive, illustrating a need for coordinated choice of sort orders.

Experiment B3: In this experiment we compare our approach of choosing interesting orders, PYRO-O, with the

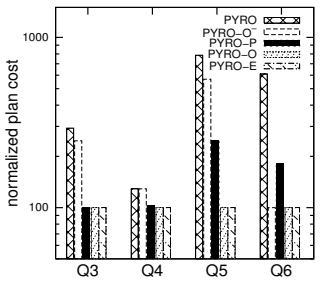


Figure 13. Q3-Q6

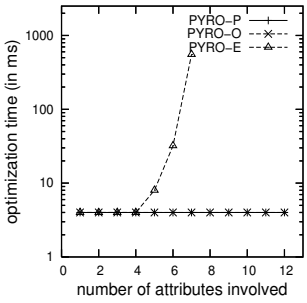


Figure 14. Scalability

exhaustive approach, and a heuristic used by PostgreSQL. Postgres uses the following heuristic: for each of the n attributes involved in the join condition, a sort order beginning with that attribute is chosen; in each order, the remaining $n-1$ attributes are ordered arbitrarily. We implemented Postgres' heuristic in PYRO along with the extensions to exploit partial sort orders and call it PYRO-P. The exhaustive approach, called PYRO-E, enumerates all $n!$ permutations and considers partial sort orders. In addition, we also compare with PYRO, which chooses an arbitrary sort order, and a variation of PYRO-O, called PYRO-O$^-$ that considers only exact favorable orders (no partial sort). Figure 13 shows the estimated plan costs. Note the logscale for y-axis. The plan costs are normalized taking the plan cost with exhaustive approach to be 100. In the figure, Q3 and Q4 are Query 3 and Query 4 of Experiments B1 and B2. Q5 and Q6 were two real-world analytical queries and can be found in the technical report [5]. For Q3 and Q4, as very few attributes were involved in the join condition, Postgres' heuristic along with extensions to exploit partial sort orders, produced plans which were close to optimal. However, for more complex queries the heuristic does not perform as well since it makes an arbitrary choice for secondary orders.

6.3. Optimization Overheads

The optimization overheads due to the proposed extensions were negligible. During plan generation, the number of interesting orders we try at each join or aggregate node depends on the number of indices that are useful for answering the query and is not dependent on the number of join attributes. In most real-life cases this number is fairly small. Figure 14 shows the scalability of the three heuristics. For the experiment a query that joined two relations on varying number of attributes was used. Though PYRO-P and PYRO-O take the same amount of time in this experiment, in most cases, the number of favorable orders is much less than the total number of attributes involved and hence PYRO-O generates significantly fewer interesting orders than PYRO-P.

The plan-refinement algorithm presented in Section 4.2 was tested with trees up to 31 nodes (joins) and 10 attributes per node. The time taken was negligible in each case. The execution of plan refinement phase took less than 6 ms even for the tree with 31 nodes.

Both the optimizer extensions and the extension to external-sorting (MRS) were fairly straight forward to implement. The optimizer extensions neatly integrated into our existing Volcano style optimizer.

7. Conclusion

We addressed the issue of choosing interesting sort orders. We showed that even a simplified version of the problem is *NP-hard* and proposed principled heuristics for choosing interesting orders. Our techniques take into account important issues such as partially matching sort orders and operators that require matching sort orders from multiple inputs. We presented detailed experimental results to demonstrate the benefits due to our techniques.

Unlike merge-join and order-by, operators such as group-by and duplicate elimination actually need grouped but not necessarily sorted input; sorting is just one way of providing grouped input. Extending our techniques to grouped input property is a topic of future work.

Acknowledgments

The *NP-hardness* result and 2-approximation are joint work with Ajit A. Diwan and Ch. Sobhan Babu. We thank C. Santosh Kumar for implementing the extensions to external sorting in Postgres.

References

[1] J. Diaz, J. Petit, and M. Serna. A Survey of Graph Layout Problems. *ACM Comput. Surv.*, 34(3), 2002.

[2] V. Estivill-Castro and D. Wood. A survey of adaptive sorting algorithms. *ACM Comput. Surv.*, 24(4), 1992.

[3] G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Comput. Surv.*, 25(2), 1993.

[4] G. Graefe and W. McKenna. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *ICDE*, 1993.

[5] R. Guravannavar, S. Sudarshan, A. A. Diwan, and C. S. Babu. Reducing Order Enforcement Cost in Complex Query Plans. Technical Report. arXiv:cs.DB/0611094.

[6] D. Knuth. *The Art of Programming, Vol. 3 (Sorting and Searching)*. Addison-Wesley, 1973.

[7] P.-Å. Larson. External sorting: Run formation revisited. *IEEE Trans. Knowl. Data Eng.*, 15(4), 2003.

[8] T. Neumann and G. Moerkotte. A Combined Framework for Grouping and Order Optimization. In *VLDB*, 2004.

[9] T. Neumann and G. Moerkotte. An Efficient Framework for Order Optimization. In *ICDE*, 2004.

[10] P. G. Selinger, M.M.Astrahan, D.D.Chamberlin, R.A.Lorie, and T.G.Price. Access Path Selection in a Relational Database Management System. In *Proceedings of ACM SIGMOD Conference*, 1979.

[11] D. Simmen, E. Shekita, and T. Malkemus. Fundamental Techniques for Order Optimization. In *Proceedings of ACM SIGMOD Conference*, 1996.

[12] X. Wang and M. Cherniack. Avoiding Sorting and Grouping In Processing Queries. In *VLDB*, 2003.