The VLDB Journal manuscript No.
(will be inserted by the editor)

Which Sort Orders are Interesting?

Ravindra Guravannavar · S. Sudarshan · Ajit A. Diwan · Ch. Sobhan Babu

Received: date / Accepted: date

Abstract Sort orders play an important role in query evaluation. Algorithms that rely on sorting are widely used to implement joins, grouping, duplicate elimination and other set operations. The notion of *interesting orders* has allowed query optimizers to consider plans that could be locally sub-optimal, but produce ordered output beneficial for other operators, and thus be part of a globally optimal plan. However, the number of interesting orders for most operators is factorial in the number of attributes involved. 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 query-covering indices, which provide multiple alternative sort orders for relations, motivate us to better address the problem of choosing interesting orders. We show that even a simplified version of the problem is NP-hard and provide a 1/2-benefit approximation algorithm for a special case of the problem. We then present principled heuristics for the general case of choosing interesting orders.

We have implemented the proposed techniques in a Volcano-style cost-based 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.

Ravindra Guravannavar · Sobhan Babu
Department of Computer Science and Engineering,
Indian Institute of Technology, Hyderabad
Andhra Pradesh, India
Tel.: +91-40-2301 6077
Fax: +91-40-2301 6032
E-mail: {ravig,sobhan}@iith.ac.in

S. Sudarshan . Ajit A. Diwan
Department of Computer Science and Engineering,
Indian Institute of Technology, Bombay
Mumbai, India
Tel.: +91-22-2576 7901
Fax: +91-22-2572 0022
E-mail: {sudarsha,aad}@cse.iitb.ac.in

Keywords Query Optimization · Sort Orders

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.

The notion of interesting orders [15] has allowed optimizers to consider plans that could be locally sub-optimal, but produce sort orders that are beneficial for other operators, and thus produce a better plan overall. A sort order on the result of an input sub-expression is considered *interesting* for an operator if it is either required for the evaluation of the operator or reduces the cost of its evaluation when compared to its evaluation with unsorted inputs. However, the number of

[1] contains all attributes required to answer the query

interesting sort orders can be factorial in the number of attributes involved in the operation. This may not be acceptable as queries in the aforementioned 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 on $s.c_1$ or on $s.c_2$) is helpful in getting the desired order efficiently; a secondary index that covers the query has the same effect.
   If a relation (or intermediate result) is already sorted on a prefix of the required sort order, and if the information about such partial sort order is known to the sort operator (also called sort *enforcer* in [7][2]), the cost of sorting can be reduced significantly [5]. In many cases, when the number of duplicates is not too large, such partial sort orders may even eliminate the need to materialize runs on secondary storage, and can complete the sort operation using just one scan of the relation. In this paper, we show how a cost-based optimizer can be extended to generate efficient plans taking into account partial sort orders.
2. We consider operators having more than one interesting sort orders on their inputs, and address the problem of making a coordinated choice of sort orders for multiple such operators in a query plan. We say an operator has a *flexible order requirement* if it has more than one interesting sort order. For example, the merge-join operator has a flexible order requirement since every permutation of the join attributes is an interesting sort order for the operator.
   – In Section 3 we show that a special case of finding optimal sort orders is NP-hard and give a 2-approximation algorithm to choose sort orders for a tree of merge-joins.
   – In Section 4 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 4.1) 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 4.2 we also show how to integrate our extensions into a cost-based optimizer.
3. We present experimental results (in Section 5) 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] [7] also considers other types of enforcers, collectively called physical property enforcers.

This article is an extended version of our conference paper [8]. The important additions made in this article are described in Section 6, along with other related work.

2 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 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 Query 1 shown below. Such queries frequently arise in consolidating data from multiple sources, *e.g.,* in extract-transform-load (ETL) tasks. 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.

Query 1

```
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 bytes and

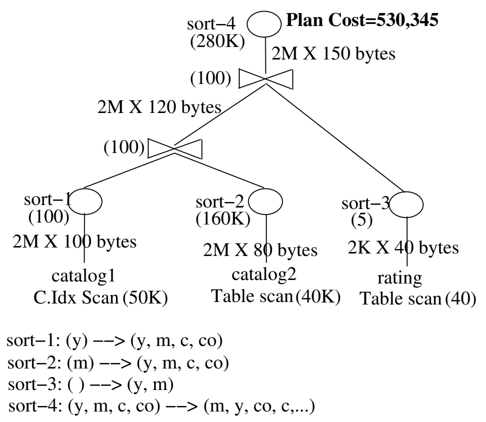


Fig. 1 A Naïve Merge-Join Plan

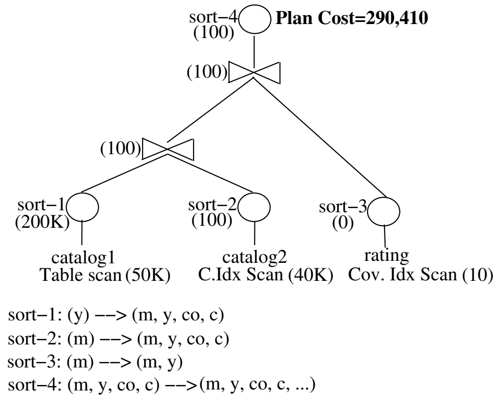

sort–4 (100)
Plan Cost=290,410
(100)
(100)
sort–1 (200K)
sort–2 (100)
sort–3 (0)
catalog1 Table scan (50K)
catalog2 C.Idx Scan (40K)
rating Cov. Idx Scan (10)
sort–1: (y) –-> (m, y, co, c)
sort–2: (m) –-> (m, y, co, c)
sort–3: (m) –-> (m, y)
sort–4: (m, y, co, c) –->(m, y, co, c, ...)

Fig. 2 Optimal Merge-Join Plan

80 bytes. 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, and the index includes the *year* and *rating* columns in its leaf pages (*i.e.,* the index covers the query). Figures 1 and 2 show two different plans for the example query. Numbers in the 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 only 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.

2.1 Changes to External Sort

External sorting algorithms have been studied extensively but in isolation. The standard replacement selection [10] 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 [11] 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. Estivill-Castro and Wood [4] 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 conventions: o, o_1, o_2 *etc.* refer to sort orders. A sort order of size n is a sequence of attributes/columns $(a_1, a_2, \ldots, a_n)$. Sort direction (ascending/descending) is ignored; our techniques are applicable independent of the sort direction.

ϵ	Empty (no) sort order
attrs(o)	The set of attributes in sort order o
$\lvert o\rvert$	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). Further, suppose the number of tuples with any given value for col_1 fit in memory. Standard replacement-selection writes a single run with all the tuples 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. Since sorting of each *partial sort segment* is done independently, the number of comparisons are significantly reduced. 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 $O(n\ log(n/k))$ 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 experimental study presented in Section 5 confirms that the benefits of exploiting partial sort orders can be substantial, and yet none of the systems we evaluated exploited the partial sort orders.

2.2 Optimizer Extensions for Partial Sort Orders

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

$o_1 \wedge o_2$	Longest common prefix between sort orders o_1 and o_2
$o \wedge s$	Longest prefix of sort order o such that each attribute in the prefix belongs to the attribute set s
$o_1 + o_2$	Sort order obtained by concatenating o_1 and o_2
$o_1 - o_2$	Sort 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
$D(e, s)$	Distinct values for attribute(s) s of expression e. $D(e, s) = N(\Pi_s(e))$
$cpu\_cost(e, o)$	CPU cost of sorting the result of e to get order o
M	Number of memory blocks available for sorting

The *Volcano* optimizer framework [7] assumes that 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.

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

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 $N(e') = \lceil N(e)/D(e,\ attrs(o_s))\rceil$ and $B(e') = \lceil B(e)/D(e,\ attrs(o_s))\rceil$. 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.

3 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[3]. 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

[3] We assume merge-join requires sorting on all attributes involved in the join predicate.

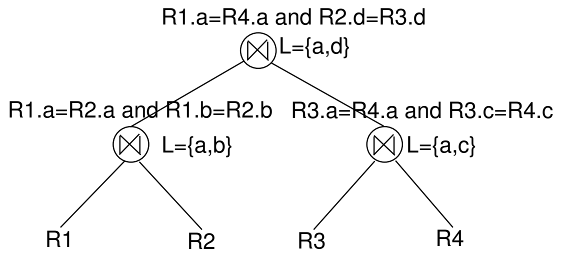


Fig. 3 A Join Tree with Representative Join Attribute Sets

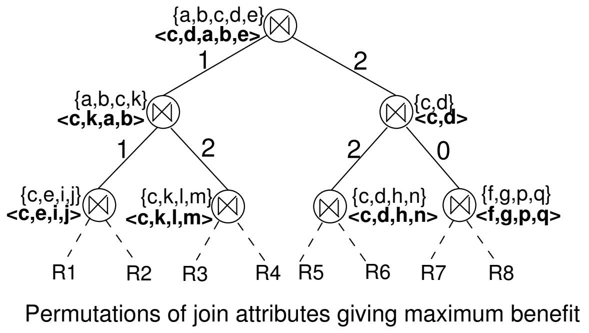


Fig. 4 A Special Case of Choosing Globally Optimal Sort Order

have a partial sort. The assignment of sort orders to each operator in a plan is said to be optimal if no other sort order assignment has a lower cost. 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 1/2 benefit approximation 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 1/2 benefit approximation algorithm.

3.1 Optimal Sort Order Assignment is NP-Hard

We now show that the problem of finding an optimal assignment of sort orders for a given plan is NP-hard by considering a special case of the problem. Let $e = R_1 \bowtie R_2 \bowtie R_3 \ldots \bowtie R_n$ be a join expression with conjunctive join predicates on n relations, where n is a power of 2. Let T be a balanced join order tree for e. Figure 3 shows an example. For each join node v in T, we assign an attribute set L_v (called *representative join attribute set*), which is constructed as follows. If a_i is an attribute involved in the join predicate of v then $\mathcal{H}(a_i) \in L_v$, where $\mathcal{H}(a_i)$ is the *representative* of the attribute equivalence class in the result of e. Two attributes a_i and a_j belong to the same attribute equivalence class if they are equated directly or transitively in the join predicates of e. The *representative* of an equivalence class is an arbitrarily chosen attribute belonging to the equivalence class. For example, if the predicate ($R_1.a_1 = R_2.a_2$ and $R_2.a_2 = R_3.a_3$) is part of the join predicates of e, then $R_1.a_1, R_2.a_2$ and $R_3.a_3$ belong to the same attribute equivalence class, and we will have $\mathcal{H}(R_1.a_1) = \mathcal{H}(R_2.a_2) = \mathcal{H}(R_3.a_3) = R_1.a_1$. In Figure 3 we have shown the representative join attribute sets for each join node. For brevity, we omit the relation name qualifiers for attributes.

Now, suppose all the base relations and intermediate results in T are of the same size and no indices are present on the base relations. The problem of choosing optimal sort orders for each join requires us to choose permutations of representative join attribute sets such that we minimize the cost of intermediate sorts. The cost of sorting depends on the sort order already present on the input and the sort order required. In general, the sort cost on any edge (v_i, v_j) of the tree is a monotonically decreasing function of the length of common prefix between attribute permutations chosen for v_i and v_j. For example, see our cost model for sort presented in Section 2.2. We define the benefit of a solution to be $\sum_{v_i v_j \in E} f(|p_i \wedge p_j|)$, where E is the set of edges in the tree, p_i, p_j are attribute permutations chosen by the solution for vertices v_i, v_j and f is any monotonically increasing function in the length of the common prefix ($|p_i \wedge p_j|$), with 0 at origin (*i.e.,* $f(0) = 0$). Minimizing the sorting cost requires maximizing the total benefit.

Figure 4, shows an example along with a solution, which maximizes the total benefit assuming $f(|p_i \wedge p_j|) = |p_i \wedge p_j|$. The representative join attribute set for each join node is shown in curly braces besides the node. Permutations chosen by the solution are indicated with angle brackets and the number on each edge shows the benefit for that edge. Below we state the problem formally.

Problem 1 *(Common Prefix Problem) Let T be a tree having n vertices, the vertex set being $V(T)$ and the edge set being $E(T)$. Each vertex v_i ($i = 1, \ldots, n$) is associated with an attribute set s_i. Let f be any non-decreasing function with $f(0) = 0$. Find a set of attribute permutations $p_1, p_2 \ldots, p_n$, where p_i is a permutation of set s_i, such that the benefit function $\mathcal{F} = \sum_{\forall v_i v_j \in E(T)} f(|p_i \wedge p_j|)$ is maximized.*

We prove that Problem 1 is NP-hard even for binary trees. To do so we consider the special case where $f(|p_i \wedge p_j|) = |p_i \wedge p_j|$.
The structure of our proof is as follows: we consider a known NP-hard problem, SUM-CUT [3], which is related to graph layouts and rephrase it as MOD-SUM-CUT. We then reduce the MOD-SUM-CUT problem to the Common Prefix Problem on star trees. Finally, we reduce the Common Prefix Problem on star trees to the Common

Prefix Problem on binary trees. In the rest of this section we define each of these problems and then present the proof of NP-hardness.

Problem 2 *(Sum-Cut) [3]* Given a graph G with m vertices, number the vertices of G as $1, \ldots, m$ such that $\sum_{1 \leq i \leq m} c_i$ is minimized, where c_i is the number of vertices numbered $\leq i$ that are adjacent to at least one vertex numbered greater than i.

The Sum-Cut problem can be rephrased as follows: given a graph G with m vertices, number the vertices of G as $1, \ldots, m$ such that $\sum_{1 \leq i \leq m} \bar{c_i}$ is maximized, where $\bar{c_i}$ is the number of vertices numbered $\leq i$ that are adjacent to no vertex numbered greater than i. Let G' be the complement graph of G. The complement graph G' of G contains an edge (v_i, v_j) *iff* (v_i, v_j) is not present in G. On the complement graph G', it is straight-forward to see that the Sum-Cut problem is equivalent to Problem 3 given below.

Problem 3 *(Mod-Sum-Cut)* Given a graph H with m vertices, number the vertices of H as $1, 2, \ldots, m$ such that $\sum_{1 \leq i \leq m} q_i$ is maximized, where q_i is the number of vertices numbered no larger than i that are adjacent to *all* the vertices numbered greater than i.

First, we reduce the *Mod-Sum-Cut* problem to the Common Prefix Problem on star trees. A *star tree* or simply a *star* of n vertices is a tree with a root and $n-1$ leaf vertices.

Lemma 1 *The Common Prefix Problem is NP-Hard for Stars.*

Proof We reduce the *Mod-Sum-Cut* problem to the Common Prefix Problem on stars, with the function f set to $|p_i \wedge p_j|$ (*i.e.,* the length of the longest common prefix). Let graph G with m vertices be the given instance of Mod-Sum-Cut problem. Let $v_1, \ldots, v_m$ be the vertices of G. We construct an instance of the Common Prefix Problem on stars as follows: let S be a star having $m+1$ vertices, with u_r as as its root and $u_1, \ldots, u_m$ as its leaves. The attribute set of root u_r is chosen to be the set of all vertices in G (*i.e.*, $\{v_1, \ldots, v_m\}$), and the attribute set of each leaf u_i is chosen to be $adj(v_i)$, where $adj(v_i)$ is the set of all vertices adjacent to v_i in graph G. A pictorial illustration of the construction is shown in Figure 5.

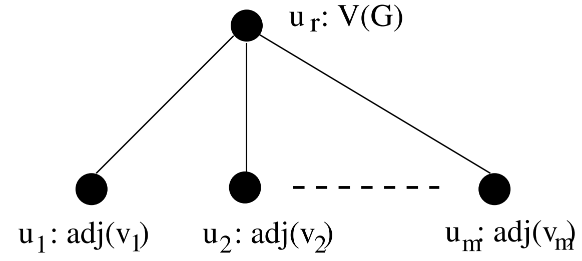


Fig. 5 Reducing Mod-Sum-Cut to Common Prefix on Star

Now, we show that there exists a solution of value k for Mod-Sum-Cut on G *iff* there exists a solution of value k for the Common Prefix Problem on S.

Suppose there exists a solution of value k for Mod-Sum-Cut on G. Let the order of vertices in the solution be $v_{g(1)}, v_{g(2)}, \ldots, v_{g(m)}$, where g is a permutation on $1, \ldots, m$ (*i.e.,* a one-to-one function from $\{1, \ldots, m\}$ to $\{1, \ldots, m\}$). We construct the solution for the corresponding Common Prefix Problem (for star S) as follows: for the root vertex u_r, we choose the attribute permutation to be $o_r = v_{g(m)}, v_{g(m-1)}, \ldots, v_{g(1)}$. For each leaf vertex u_i, we choose a permutation o_i of its attribute set $adj(v_i)$ such that the length of the longest common prefix $|o_i \wedge o_r|$ is maximum.

In the solution ordering for Mod-Sum-Cut, let l_i be the smallest integer such that v_i is adjacent to all vertices in the set $v_{g(l_i+1)}, \ldots, v_{g(m)}$. This implies the following: *(i)* in the solution value for Mod-Sum-Cut, vertex v_i will be counted $m - l_i$ times, *i.e.,* $k = \sum_{v_i}(m - l_i)$, and *(ii)* in the corresponding Common Prefix Problem, there exists a common prefix of length $m - l_i$ between the permutations chosen for u_i and the root u_r. This shows there exists a solution of value k for the corresponding Common Prefix Problem.

Now, suppose there exists a solution of value k for the Common Prefix Problem on S. In the solution, let the attribute permutation chosen for the root vertex u_r be $o_r = v_{h(1)}, v_{h(2)}, \ldots, v_{h(m)}$, where h is a permutation on $1, \ldots, m$. Now, we construct the solution for Mod-Sum-Cut on G by reversing the order of attributes in o_r, *i.e.,* by ordering the vertices of G as $v_{h(m)}, v_{h(m-1)}, \ldots, v_{h(1)}$.

In the solution for the Common Prefix Problem on S, let o_i be the permutation (of set $adj(v_i)$) chosen for leaf u_i. Let l_i denote the length of the longest common prefix between o_i and o_r, *i.e.,* $l_i = |o_i \wedge o_r|$. We observe that, the solution value $k = \sum_{1 \leq i \leq m}(l_i)$. In the corresponding solution for Mod-Sum-Cut on G, l_i will be the smallest integer such that vertex v_i is adjacent to all vertices in the set $v_{h(m)}, \ldots, v_{h(m-l_i+1)}$. Hence, in the solution value for Mod-Sum-Cut on G, vertex v_i will be counted l_i times. Therefore, the solution for Mod-Sum-Cut will have a value of $\sum_{1 \leq i \leq m}(l_i) = k$.

Theorem 1 *Problem 1 is NP-hard even for binary trees.*

Proof We reduce the Common Prefix Problem on stars to the Common Prefix Problem on binary trees. Let S be a star with u_r as its root and $u_1, \ldots, u_m$ as its leaves. Let a_r denote the set of attributes associated with u_r

and $a_1, a_2, \ldots, a_m$ denote the set of attributes associated with vertices $u_1, u_2, \ldots, u_m$ respectively. We now construct an instance of the Common Prefix Problem on binary trees as follows: let T be a binary tree with $2m$ vertices, with $r_1, r_2, \ldots, r_m$ as its internal vertices and $w_1, w_2, \ldots, w_m$ as its leaves. Let the edge set $E(T)$ be $\{r_i r_{i+1} : 1 \leq i < m\} \cup \{r_i w_i : 1 \leq i \leq m\}$. Each internal vertex r_i is assigned the attribute set $\mathcal{A} = a_r \cup \mathcal{L}$, where $\mathcal{L}$ is an arbitrarily chosen set of attributes of size $> m \times |a_r|$ and is disjoint from $a_r \cup a_1 \cup \ldots \cup a_m$. Each leaf vertex w_i is assigned the attribute set a_i. Figure 6 pictorially illustrates the construction. In the figure, the attribute sets a_r and $a_1, \ldots, a_m$ for the star are assumed to be as in Figure 5.

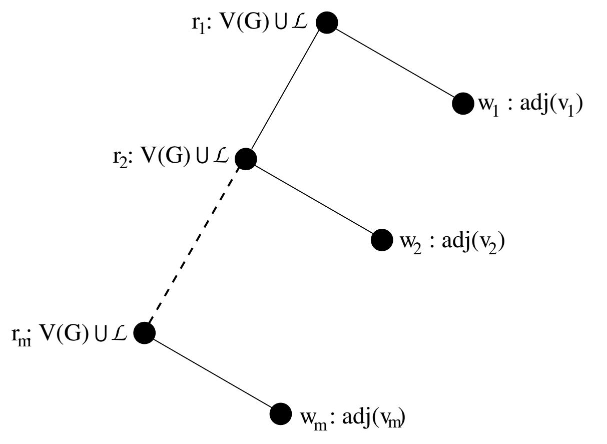


Fig. 6 Reducing the Common Prefix Problem on Stars to the Common Prefix Problem on Binary Trees

Let $\mathcal{Z} = (m-1) \times |\mathcal{A}|$.

First, we show that if there exists a solution of value k for the Common Prefix Problem on S then there exists a solution of value $k + \mathcal{Z}$ for the Common Prefix Problem on T.

Suppose there exists a solution of value k for the Common Prefix Problem on S. Let the o_r be the attribute permutation assigned for u_r and o_i be the attribute permutation assigned for each u_i, $1 \leq i \leq m$. We construct a solution for T as follows: for each intern vertex r_i, we assign the permutation $o_r + o_l$, where o_l is a fixed permutation of $\mathcal{L}$, chosen arbitrarily. For each leaf vertex w_i, we assign the permutation o_i. Since the same permutation is chosen for all the internal vertices, each of the $(m-1)$ pairs of adjacent internal vertices will have a common prefix of length $|\mathcal{A}|$. Further, each of the m pairs of internal and leaf vertices that are adjacent to each other will have a common prefix of length $|o_i \wedge o_r|$. As $k = \sum_{1 \leq i \leq m}(|o_i \wedge o_r|)$, we conclude the solution value for T is $k + \mathcal{Z}$.

Next, we show that if there exists a solution of value k for the Common Prefix Problem on T then there exists a solution of value $k - \mathcal{Z}$ for the Common Prefix Problem on S. To do so, we make use two supporting lemmas, Lemma 2 and Lemma 3. Below we state and prove these lemmas and then continue with the proof of Theorem 1.

Lemma 2 *In any optimal solution for the Common Prefix Problem on T, all the internal vertices are assigned an identical permutation p.*

Proof Let T_{opt} be an optimal solution for T. In the optimal solution, let $p_1, \ldots, p_m$ be the permutations assigned to internal vertices $r_1, \ldots, r_m$ respectively. We prove that $|p_i \wedge p_{i+1}| = |\mathcal{A}|$ for $1 \leq i < m$, which essentially proves this lemma.

Case 1: Suppose $|p_i \wedge p_{i+1}| < |a_r|$ for some i, $1 \leq i < m$. This implies, the total benefit of T_{opt}, $Ben(T_{opt}) < (m-2) \times |\mathcal{A}| + (m+1) \times |a_r|$. Since $|\mathcal{L}| > m \times |a_r|$, we have $|\mathcal{A}| > (m+1) \times |a_r|$. Therefore, $Ben(T_{opt}) < (m-1) \times |\mathcal{A}|$. However, we know that there exists a solution for T with benefit of at least $(m-1) \times |\mathcal{A}|$. This is because each of the internal vertices have the same attribute set of size $|\mathcal{A}|$. This contradicts the given fact that T_{opt} is optimal. Therefore, we conclude that our assumption: $|p_i \wedge p_{i+1}| < |a_r|$ for some i, $1 \leq i < m$, cannot be true.

Case 2: Suppose $|p_i \wedge p_{i+1}| \geq |a_r|$ for all $1 \leq i < m$, but $|p_i \wedge p_{i+1}| < |\mathcal{A}|$ for some i, $1 \leq i < m$.
Given a permutation p, we use the notation $p[j]$ to denote the attribute at the j^{th} position, where $1 \leq j \leq |p|$. The condition for Case 2 implies the following:

(a) $p_1[j] = p_2[j] = \ldots = p_m[j]$ for $1 \leq j \leq |a_r|$.
(b) in T_{opt}, the total benefit of edges incident between internal vertices $\sum_{1 \leq i < m}(|p_i \wedge p_{i+1}|)$ must be less than $|\mathcal{A}| \times (m-1)$.

Now, consider a solution T'_{opt} for T in which each leaf vertex is assigned the same permutation as in T_{opt} and all the internal vertices are assigned an identical permutation p constructed as follows: the first $|a_r|$ attributes of p are chosen in the same order as the first $|a_r|$ attributes in p_i for any $1 \leq i \leq m$ (*i.e.*, $p[j] = p_i[j], 1 \leq j \leq |a_r|$), and the next $|\mathcal{A} - a_r|$ attributes are chosen an in an arbitrary order.

We observe that in both T'_{opt} and T_{opt} the total benefit of edges incident from internal vertices to leaf vertices remains the same. However, in T'_{opt}, the total benefit of edges incident between internal vertices will be $|\mathcal{A}| \times (m-1)$ (this is because all internal vertices are assigned an identical permutation). This implies, the total benefit of T'_{opt} is larger than that of T_{opt}, which contradicts the given fact that T_{opt} is optimal. Therefore, we conclude the assumption made for Case 2 cannot be true.

We thus conclude in every optimal solution T_{opt}, all the internal vertices are assigned an identical permutation, completing the proof of Lemma 2.

Next, we state and prove our second supporting lemma.

Lemma 3 *There exists an optimal solution for T such that, in the permutation p chosen for the internal vertices, every attribute in set a_r occurs before any attribute in set $\mathcal{L}$ occurs.*

Proof Let T_{opt} be an optimal solution for T. In T_{opt}, let $p_1, \ldots, p_m$ be the permutations assigned to the internal vertices $r_1, \ldots, r_m$ respectively. From Lemma 2 we know that all the internal vertices are assigned an identical permutation; let $p_1 = p_2 = \ldots = p_m = p$.

Suppose there exist x, y such that $x < y$, $p[x] \in \mathcal{L}$ and $p[y] \in a_r$. We now modify $p_1, \ldots, p_m$ as follows: in each p_i, we swap $p_i[x]$ with $p_i[y]$. Since there is no attribute common to the set $\mathcal{L}$ and the attribute sets associated with the leaf vertices, this modification cannot decrease the total benefit of T_{opt}. This modification can be repeated until all the attributes in a_r appear before the attributes in $\mathcal{L}$ in the permutation p.

From Lemmas 2 and 3, we can make the following statement: if there exists a solution of value k for the Common Prefix Problem on T, then there exists a solution T_{opt} of value at least k, in which, all internal vertices are assigned an identical permutation p and $|p \wedge a_r| = |a_r|$.

We now construct a solution for the star S as follows: for the root vertex u_r we assign the permutation $p \wedge a_r$. For each leaf vertex u_i, we assign the permutation chosen for the corresponding leaf w_i in the solution T_{opt}. In T_{opt}, the maximum benefit which can be contributed by edges incident between internal vertices of T is $\mathcal{Z}$. Therefore we conclude the corresponding solution on S should have a benefit of at least $k - \mathcal{Z}$.

3.2 A Polynomial Time Algorithm for Paths

We now present an efficient algorithm for solving the Common Prefix Problem, when the tree is a path. The algorithm employs dynamic programming. Note that left-deep and right-deep join plans result in problem instances on paths. The algorithm relies on the following theorem.

Theorem 2 *Let $v_1, v_2, \ldots, v_n$ be a path, where each vertex v_i is associated with an attribute set s_i. The optimal solution value of Common Prefix Problem for any segment (i,j) of the path, $OPT(i,j) = max\{OPT(i,k) + OPT(k+1,j) + f(c(i,j))\}$ over all $i \leq k < j$, where $c(i,j)$ is the number of common attributes for the segment (i,j).*

Proof
Case 1: Let $c(i,j) = 0$, *i.e.*, there exists no attribute common to all vertices $v_i, v_{i+1}, \ldots, v_j$. Consider an optimal solution for the path $v_i, \ldots, v_j$. Let p_x be the attribute permutation assigned by the optimal solution to vertex v_x, $i \leq x \leq j$. The optimal solution must contain two vertices v_k, v_{k+1} such that the benefit for the edge (v_k, v_{k+1}) is 0, *i.e.*, $|p_k \wedge p_{k+1}| = 0$. This directly follows from the assumption of Case 1, $c(i,j) = 0$. Now, the problem can be independently solved for the two segments (v_i, v_k) and (v_{k+1}, v_j) and $OPT(i,j) = OPT(i,k) + OPT(k+1,j)$.

Case 2: Let $c(i,j) \neq 0$. Let $s(i,j)$ be the set of attributes common to all vertices $v_i, \ldots, v_j$. Note that the cardinality $|s(i,j)| = c(i,j)$. Let o_s be an arbitrarily chosen permutation of set $s(i,j)$. We claim that there exists an optimal solution S_{opt} such that, for every vertex v_x $(i \leq x \leq j)$ the attribute permutation p_x chosen by the solution has o_s as its prefix. To see this, consider an optimal solution in which o_s is not a prefix of some p_x. We can then reorder the permutations assigned to the vertices, without a decrease in the total benefit $OPT(i,j)$, so as to have o_s as the prefix of each $p_x, i \leq x \leq j$.

In the above optimal solution S_{opt}, we can see that there must exist an edge $v_k v_{k+1}$ whose benefit is exactly $f(c(i,j))$. The attributes common to path segment $i, \ldots, j$ are also common to path segments $i, \ldots, k$ and $k+1, \ldots j$, *i.e.*, $s(i,j) \subseteq s(i,k)$ and $s(i,j) \subseteq s(k+1,j)$. Therefore, in S_{opt} the permutations assigned for path segments $v_i, \ldots, v_k$ and $v_{k+1}, \ldots, v_j$ give an optimal solution for those two path segments. Hence we have $OPT(i,j) = OPT(i,k) + OPT(k+1,j) + f(c(i,j))$.

Procedure *PathOrder* in Figure 7 computes optimal attribute permutations for any path $(1,n)$, where each vertex i, $1 \leq i \leq n$, is associated with an attribute set $s[i]$. The procedure uses dynamic programming and computes solutions bottom up starting from path segments of length 0 (single vertices). The procedure begins by assigning a benefit of 0 to all path segments (i,i), $1 \leq i \leq n$. It then constructs solutions for paths of increasing length. For each path $(i, i+j)$, of length j, $1 \leq j \leq n-1$, the value of k, which maximizes *ben*=*benefit*(i,k)+*benefit*$(k+1, i+j)$ is identified. The values of k, *ben* and the attributes common to all the vertices of path $(i, i+j)$ are remembered (memoized). Finally, the sub-procedure *MakePermutation* is used to construct the attribute permutations $p[1], \ldots, p[n]$ using the memo structure. The first call to procedure *MakePermutation* is made with parameter i set to 1 (the first vertex on the path) and j set to n (the last vertex on the path), and each of the attribute permutations $p[1], \ldots, p[n]$ initialized with an empty sort order. Procedure *MakePermutation* constructs the attribute permutation for each of the vertices $i, i+1, \ldots, j$ as follows: a permutation *cp* of *commons(i, j)* (*i.e.,* the set of attributes common to all the vertices from $i, \ldots, j$) is chosen at arbitrary. *cp* is then appended to each of $p[i], \ldots, p[j]$. The common attributes for segment (i, j) are then removed from the common attributes of all subpaths of (i, j). The optimal split point m for the path segment (i, j) is read from the memo structure, and the construction of the permutations continues recursively on subpaths (i, m) and $(m+1, j)$, until $i = j$ (i.e., a single vertex). The overall time complexity of procedure *PathOrder* is $O(n^3)$.

3.3 A 1/2 Benefit Approximation Algorithm for Binary Trees

For binary trees we propose an approximation with benefit at least half that of an optimal solution. Note that our approximation guarantee implies at least half the best possible *improvement* over the worst case sort cost. This however, does not imply a 2-approximation on the total cost.

We split the tree into two sets of paths, P_o and P_e. P_o has the paths formed by edges incident from odd levels and P_e has those formed by edges incident from even levels, Figure 8 shows an example. We then find an optimal solution for each of the two sets of paths. Note that this gives us two solutions for the complete tree, because each set of paths covers all the vertices of the tree (for any left over vertices at the leaf level or the root, we choose an arbitrary permutation). Let 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-ben*(S_T) and that of edges in E_e be *even-ben*(S_T). We claim that *ben*$(S_o) \geq$ *odd-ben*(S_T) and *ben*$(S_e) \geq$ *even-ben*(S_T). This claim can be easily proved as follows. Suppose *ben*$(S_o) <$ *odd-ben*(S_T), which means the sum of benefits for edges in P_o is higher in S_T than in S_o. This is not possible since S_o is an optimal solution for P_o (proof by contradiction). Similarly, we can prove that *ben*$(S_e) \geq$ *even-ben*(S_T). Since the total benefit of the optimal solution *ben*$(S_T) =$ *odd-ben*(S_T) + *even-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 8 does not include the root and leaf nodes. For these left over vertices arbitrary permutations can be chosen. Figures 9 and 10 illustrate the 1/2−benefit approximation technique using the example of Figure 4. In Figure 9, we consider a single path which consists of the two edges incident from the root and obtain the optimal solution for this

Procedure PathOrder

```
Input: s[n] : array of attribute sets
Output: p[n] : array of permutations or sort orders
Data Structures:
    benefit[n][n], split[n][n] : arrays of integers
    commons[n][n] : array of attribute sets
    apermute(s) : Function that returns an arbitrarily chosen
    permutation of attribute set s
BEGIN
    // Initialize the arrays.
    for i=1 to n
        benefit[i][i] = 0;          p[i] = $\epsilon$;
        commons[i][i] = s[i];       split[i][i] = -1;
    for j=1 to n-1          // Consider path segments of length j
        for i = 1 to n-j    // Consider path segment (i, i+j)
            Let k be the index such that i $\leq$ k < (i+j) and
            benefit[i][k]+benefit[k+1][i+j] is maximum.

            commons[i][i+j] = commons[i][k] $\cap$ commons[k+1][i+j];
            benefit[i][i+j] = benefit[i][k] + benefit[k+1][i+j] +
                              $f$(|commons[i][i+j]|);
            split[i][i+j] = k;

    // Now, construct the attribute permutations.
    Call MakePermutation(1, n);
END PROC

// Procedure to construct attribute permutations from the
// memo structure, in which the optimal split point and common
// attributes are remembered.
```

Procedure MakePermutation(i, j)

```
BEGIN
    // Choose a permutation of the attributes that are common
    // to all the vertices from i to j.
    Let ca = commons[i][j];
    Let cp = apermute(ca);
    for k=i to j
        Append cp to p[k];
    if (i = j)
        return;
    // Remove the common attributes from all subpaths of (i,j),
    // so that the attributes do not repeat.
    For all i', j' s.t. $i' \geq i$ and $j' \leq j$
        commons[i'][j'] = commons[i'][j'] − ca;
    // Construct the permutations of remaining attributes for the
    // two subpaths, to the left and right of the the split point.
    m = split[i][j];
    MakePermutation(i, m);
    MakePermutation(m+1, j);
END PROC
```

Fig. 7 Optimal Benefit Sort Orders for a Path

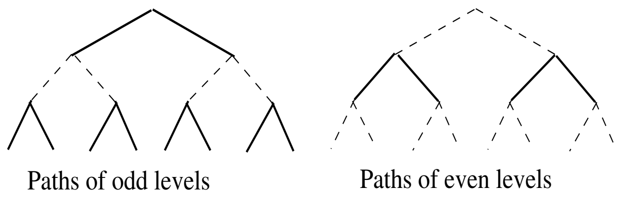


Fig. 8 A 1/2 Benefit Approximation for Binary Trees

path. For the leaf nodes (left over vertices) we choose the permutations arbitrarily, and the figure shows the worst case where the benefit of the edges not included in the set P_o is 0. The total benefit of this solution S_o is 4. Next, we consider the two disjoint paths (P_e) as shown in Figure 10, and obtain optimal solutions independently for these paths. For the root node (the left over vertex) we choose the permutation arbitrarily, and the figure shows the worst case where the benefit of the edges not included in the set P_e is 0. The total benefit of this solution S_e is 5, and we choose this one as our final solution. Note that the benefit of the optimal solution shown in Figure 4 is 8. Kenkre and Vishwanathan [9] have subsequently improved upon our result and have given a $\frac{log\ log\ n}{1\ +\ log\ log\ n}$ factor approximation. Analysis of the general form of the problem and giving a theoretical bound for its approximation are beyond the scope of this paper.

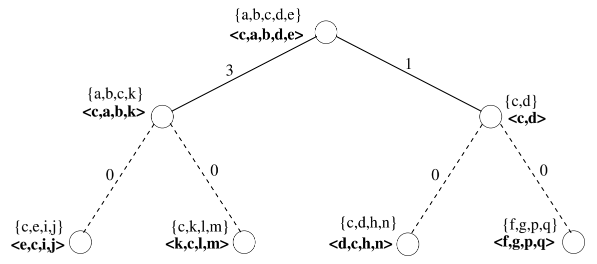


Fig. 9 Solution S_o for the example of Figure 4

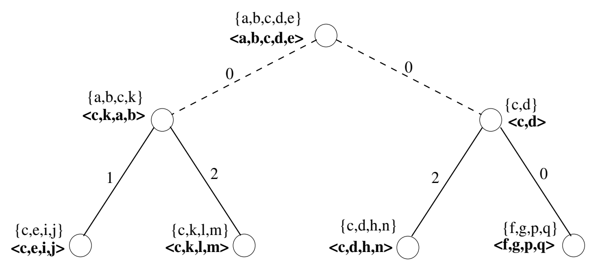


Fig. 10 Solution S_e for the example of Figure 4

4 Optimization with Favorable Orders

The benefit model we presented in the previous section, does not take into account factors such as the physical sort order of a relation, available indices and size of base relations and intermediate results. Moreover, we assumed that the join order is fixed. In this section, we present a two phase approach to address the more general problem. In phase-1, which occurs during plan generation, we use the information about 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 1/2 benefit approximation algorithm of Section 3.3

4.1 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. In the discussion that follows, we use the following notations:

$cbp(e, o)$	Cost of the best plan for expression e with o being the required output sort 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 arbitrarily chosen permutation of set s
$P(s)$	Set of all permutations of 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 sort order can be obtained with lesser cost than a full sort of unordered result. For instance, consider an expression $\sigma_p(r)$. The clustering order of relation r will have a positive *benefit* *w.r.t.* the expression, if the best plan for $\sigma_p(r)$ involves a scan. 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)*.

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

Example:
Consider an expression $e = r \bowtie s$. Suppose the best plan for e (with no sort order requirement on the output) uses a hash-join and its cost is 100 (*i.e., cbp*$(e, \epsilon) = 100$). Suppose the cost of sorting the result of e on column $r.c$ is 60 (*i.e., coe*$(e, \epsilon, (r.c)) = 60$. Now, if the cost of the best plan for obtaining the result of e sorted on $r.c$ is less than 160, $(r.c)$ will be a favorable order *w.r.t* e. For instance, this could happen if there exists an alternative plan using merge-join or nested-loops join that produces the result of $r \bowtie s$ sorted on $r.c$ at a cost of 120. Note that the nested-loops join algorithm preserves the sort order of its left (outer) input, and hence if r is clustered (stored sorted) on $r.c$ a nested-loops join with r as the left input can produce the desired sort order efficiently.

4.1.1 Minimal Favorable Orders

The number of favorable orders for an expression can be very large. For instance, every sort order having the clustering order as its prefix is a favorable order. We call a sort order $o \in$ *ford(e)* as a *minimal favorable order* if the following two conditions hold.

1. $\not\exists\ o' \in$ *ford(e)* such that $o' \leq o$ and *cbp*(e, o') + *coe*(e, o', o) = *cbp*(e, o). Intuitively, sort order o is minimal only if there does not exists a sort order o' such that the cost of obtaining order o equals the cost of obtaining sort order o' followed by an explicit sort to obtain order o.
2. $\not\exists\ o'' \in$ *ford(e)* such that $o \leq o''$ and *cbp*(e, o'') = *cbp*(e, o). Intuitively, sort order o is minimal only if there does not exists a sort order o'' subsuming order o and available at the same cost as o.

We call the set of all minimal favorable orders of an expression e as the *minimal favorable order set* of e and denote it by *ford-min(e)*. Conditions 1 and 2 above, ensure that when a relation has an index that provides sort order o efficiently, orders that are prefixes of o and orders that have o as their prefix are not minimal favorable orders.

Example:
Consider an expression $e = \sigma_p(r)$. Suppose the relation r is stored sorted on $(r.a, r.b)$. If the best plan to evaluate the expression involves a scan, then we will have $(r.a, r.b)$ as a favorable sort order. Apart from $(r.a, r.b)$, we also expect any sort order having $(r.a, r.b)$ as its prefix (*e.g.,* $(r.a, r.b, r.c)$) to be a favorable order. This is because given the sort order $(r.a, r.b)$ it is relatively inexpensive to obtain the sort order $(r.a, r.b, r.c)$. Similarly, we also expect the sort order $(r.a)$ to be a favorable sort order since it is subsumed by the sort order $(r.a, r.b)$. However, in this example, the set *ford-min(e)*, includes only the sort order $(r.a, r.b)$. The sort order $(r.a, r.b, r.c)$ is not minimal because the cost of obtaining the input sorted on $(r.a, r.b, r.c)$ is equal to the cost of obtaining the sort order $(r.a, r.b)$ (which involves a just a scan) followed by a partial sort. Similarly, the sort order $(r.a)$ is not minimal because the cost of obtaining the sort order $(r.a)$ equals that of obtaining $(r.a, r.b)$ and former is subsumed by the later.
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)*$\}$

4.1.2 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 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* (Select-Project-Join-Group-by) 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.
   Let $\mathcal{T}$ =*afm*$(e_1) \cup$ *afm*(e_2)

Then, $afm(e_1 \bowtie e_2) = \mathcal{T} \cup \{o : o' \in \mathcal{T} \cup \{\epsilon\}\ and\ o = o' \wedge S\ +\ \langle S{-}attrs(o' \wedge S)\rangle\}$

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 3.3.

5. $e =_L\mathcal{G}_F(e_1)$, which represents a group-by expression with L as the set of group-by attributes and F as the set of aggregate functions.
   $afm(e) = \{o : o' \in afm(e_1) \cup \{\epsilon\}\ and\ \ o = o' \wedge L\ +\ \langle L{-}attrs(o' \wedge L)\rangle\}$
   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.

Although the approximate *ford-min* of an expression contains all the minimal favorable orders of base relations, it may not contain all the minimal favorable orders of the given expression. As an example, consider an expression $r \bowtie s$, where both r and s are base relations with no clustering or secondary index on them. Now, the *afm(*$r \bowtie s$*)* contains just *one* arbitrarily chosen permutation of the join attributes. However, the minimal favorable orders of $r \bowtie s$ includes all such permutations. In other words, the set *afm* contains all useful sort orders on base relations propagated through intermediate operators, but not all sort orders that could be generated by intermediate operators.

4.2 Optimizer Extensions

We make use of the approximate favorable orders during plan generation (phase-1) to choose a small set of promising *interesting orders* for sort-based operators. 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.

4.2.1 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.

Steps to compute $\mathcal{I}(e, o)$:

1. Collect the favorable orders of inputs plus the required output order
   $\mathcal{T}(e, o) = afm(e_l, S)\ \cup\ afm(e_r, S)\ \cup\ o \wedge S$, 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 $o_1 \le o_2$, 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
   $\mathcal{I}(e, o) = \{o : o' \in \mathcal{T}(e, o)\ and\ o = o'\ +\ \langle S - attrs(o')\rangle\}$

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.

An Example: Consider Query 1 of Section 2. 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 4.1.2 are as follows:
$afm(ct1) = \{(y)\}$, $afm(ct2) = \{(m)\}$, $afm(rt) = \{(m)\}$, $afm(ct1 \bowtie ct2) = \{(y), (m), (y, co, c, m), (m, co, c, y)\}$, $afm((ct1 \bowtie ct2) \bowtie rt) = \{(y), (m), (y, co, c, m), (m, co, c, y), (y, m), (m, y)\}$

For $(ct1 \bowtie ct2) \bowtie rt$ we consider two interesting sort orders $\{(y, m), (m, y)\}$ and for $ct1 \bowtie ct2$ we consider four sort orders $\{(y, co, c, m), (m, co, c, y), (y, m, co, c), (m, y, co, c)\}$. As a result the optimizer will consider the plan shown in Figure 2.

A Note on Optimality: If the set $\mathcal{I}(e, o)$ is computed using the exact set of minimal favorable orders (*ford-min*), then it must contain an optimal sort order, *i.e.*, a sort order, which produces the optimal merge join plan in terms of overall plan cost.

Theorem 3 *The set* $\mathcal{I}(e, o)$ *computed with exact ford-min contains an optimal sort order* o_p *for the optimization goal* $e = (e_l \bowtie e_r)$ *with* (o) *as the required output sort order, under Assumption A.*

Assumption A : If o_1, o_2 are two sort orders on the same set of attributes (*i.e.,* $\text{attrs}(o_1) = \text{attrs}(o_2)$), then the CPU cost of sorting the result of an expression e to obtain o_1 will be same as that for o_2, *i.e.,* $\text{cpu-cost}(e, o_1) = \text{cpu-cost}(e, o_2)$.
The theorem essentially states the following: to identify an optimal sort order, it is sufficient to consider only the minimal favorable orders and not the full set of favorable orders. Appendix A gives the proof of Theorem 3.

4.2.2 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 4.1.2). We then make use of the 1/2 benefit approximation algorithm for trees (described in Section 3.3) 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 $q_i \in afm(v_i.left\text{-}input) \cup afm(v_i.right\text{-}input)$ 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 attribute permutations for the nodes are chosen using the 1/2 benefit approximation algorithm; the chosen sort order for free attributes is then appended to the sort order chosen during plan generation (*i.e.,* $p_i \wedge q_i$) to get a complete order.

The reworking of the sort 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 11 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.

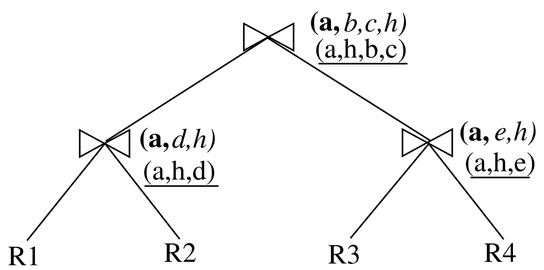


Fig. 11 Post-Optimization Phase

In many cases, there exists a hierarchical relationship between attributes as in the case of the standard example of *order number, line number.* Most indices follow these hierarchies, and thus most promising sort orders for join operations also follow them. The notion of favorable sort orders, which is used in the plan generation phase, formalizes this intuition. The problem of choosing sort orders, which is a fundamental issue in query optimization, must however be addressed in its most general form without entirely relying on the existence of such hierarchical relationship between attributes. The plan refinement phase, which makes use of the results presented in Section 3, achieves this goal. Our experimental section shows that the two phase approach presented in this paper succeeds in producing efficient plans without significant overheads.

5 Experimental Results

We performed experiments to evaluate the benefits due to the proposed ideas. For comparison, we use PostgreSQL (version 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 along with 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.

5.1 Modified Replacement Selection

The first set of experiments evaluate the benefits of exploiting partial sort orders. External sort in PostgreSQL employs the standard replacement selection (SRS) algorithm [10] suitably adapted for variable length records. We modified this implementation to exploit partial sort orders available on the input (as described in Section 2), and we call it Modified Replacement Selection (MRS). We now present experiments comparing the performance of MRS with SRS.

Experiment A1

The first experiment consists of a simple ORDER BY of the TPC-H *lineitem* table on two columns *(l_suppkey, l_partkey).*

Query 2
SELECT l_suppkey, l_partkey
FROM lineitem
ORDER BY l_suppkey, l_partkey;

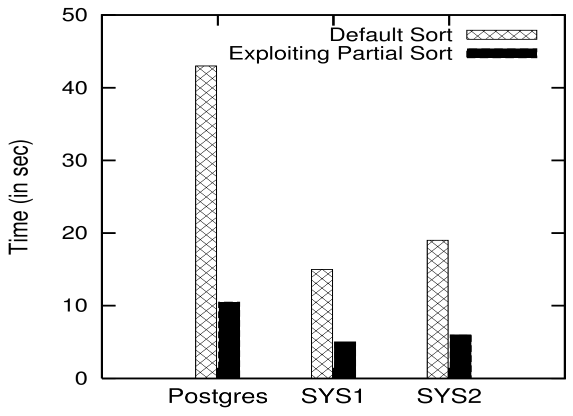


Fig. 12 Performance Results for Experiment A1

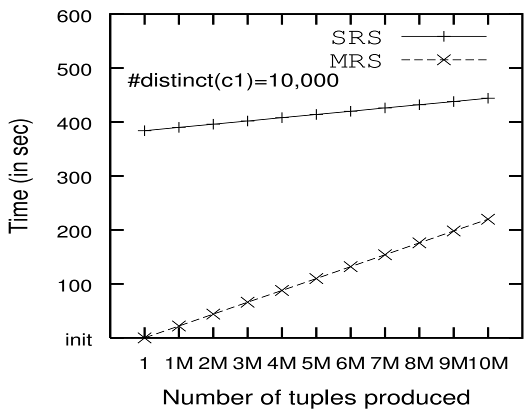


Fig. 13 Rate of Output

A secondary index on *l_suppkey* was available that covered the query (included the *l_partkey* column)[4]. On all 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 then compared the running times with our implementation that exploited partial sort order *(l_suppkey)* and the results are shown in Figure 12.

For SYS1 and SYS2, as we did not have access to their source code, we simulated the partial sorting using a correlated rank query. 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 13 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. Such early output behavior is highly desirable for Top-K queries.

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

Experiment A3

The third experiment shows the effect of *partial sort segment size* on sorting. Eight tables $R_0, \ldots, R_7$, with identical schema of 3 columns $(c1, c2, c3)$ were each populated with 10 million fixed length records each of size 200 bytes. The columns $c1$, $c2$ and $c3$ were all of fixed length character datatype with lengths 10, 10 and 180 respectively. Table R_i had 10^i tuples for each distinct value of $c1$ (*i.e.,* uniform distribution over 10^{7-i} distinct values of $c1$), resulting in a partial sort segment size of 200×10^i bytes. Thus R_0 had a sort segment of size 1 tuple or 200 bytes, and R_7 had a sort segment of size 10 million tuples or 2GB. The data for each table R_i was generated as follows: 10^{7-i} groups of tuples (segments) were produced each with 10^i tuples. Tuples in each group had the same string value for column $c1$ and randomly generated strings for columns $c2$ and $c3$. Each table was clustered on $(c1)$. The query had an order by on $(c1, c2)$. The running times with default and modified replacement selection on PostgreSQL are shown in Figure 14. In the figure, the numbers near the points are the actual execution timings in seconds.

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 $c1$.

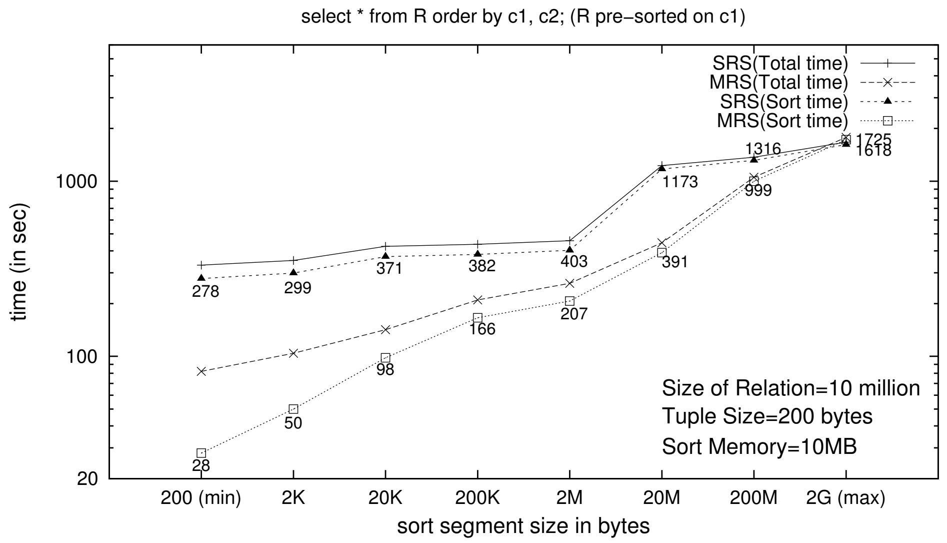


Fig. 14 Effect of Partial Sort Segment Size

Experiment A4

To see the influence of MRS on a query with joins and aggregates, we consider Query 3 shown below. The query finds the number of lineitems and available quantity for each supplier, part pair. The supplier and part key columns were common to the join, group-by and the order-by clauses. Two indices, *lineitem(l_suppkey)* and *partsupp(ps_suppkey)*, were present and covered the query. The indices were thus useful to obtain part of the desired sort order *(suppkey, partkey)*.

Query 3 *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;

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

5.2 Choice of Interesting Orders

We extended PYRO, a Volcano-style cost based optimizer [14], to consider partial sort orders and to use the proposed method for choosing sort orders for merge joins and aggregation. We compare the plans produced by the extended implementation, which we call PYRO-O, with those of PostgreSQL, SYS1 and SYS2.

Experiment B1

For this experiment we used Query 4 shown below, which lists parts for which the total quantity ordered is more than the stock available at the supplier.

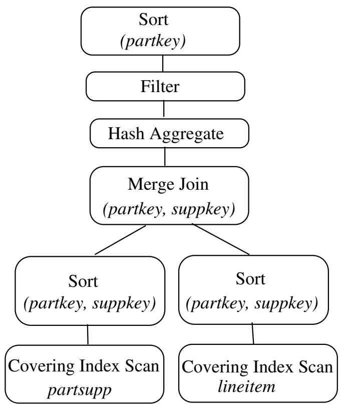
Query 4 *Parts Running Out of Stock*

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;

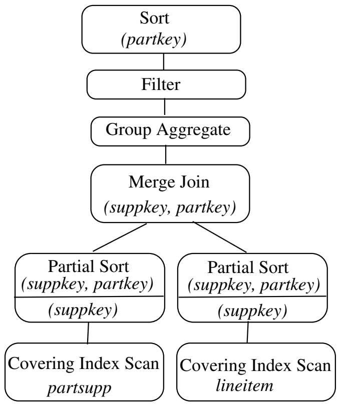
Table *partsupp* had clustering index on its primary key *(ps_partkey, ps_suppkey)*. Two secondary indices, one on *ps_suppkey* and the other on *l_suppkey* were also built on the *partsupp* and *lineitem* tables respectively. The two secondary indices covered all attributes needed for the query.

The experiment shows the need for cost-based choice of interesting orders. The choice of interesting orders for the join and aggregate are not obvious in this case for the following reasons:

1. The order-by clause favors the choice of a sort order where *partkey* appears first.
2. The clustering index on *partsupp* favors the choice of *(partkey, suppkey)*.

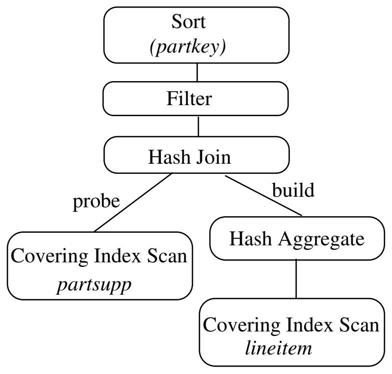


(a) Default Plan (Postgres)

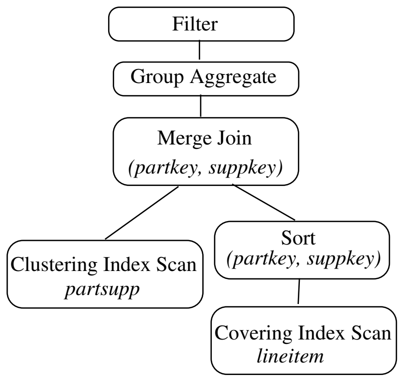


(b) Plan Generated by PYRO–O

Fig. 15 Plans for Query 4 (PostgreSQL and PYRO-O)



(a) Default plan on SYS1



(b) Forced merge–join plan on SYS1 and default plan on SYS2

Fig. 16 Plans for Query 4 (SYS1 and SYS2)

3. The secondary indices favor the choice of *(suppkey, partkey)* that can be obtained by using a low cost partial sort. Note that this option can be much cheaper due to the size of the *lineitem* relation.

Therefore, the optimizer must make a cost-based decision on the sort order to use. Figures 15 and 16 show the plans chosen by PostgreSQL, PYRO-O, SYS1 and SYS2.

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 *(l_partkey, l_suppkey)*. Further, PostgreSQL uses a hash aggregate where a sort-based aggregate would have been much cheaper as the required sort order for the group-by was available from the output of merge-join. Note that the sort order *(ps_partkey, ps_suppkey, ps_availability)*, required by the group-by, can be inferred from the sort order *(ps_partkey, ps_suppkey)*, available on the result of merge-join, due to the presence of the functional dependency $\{ps\_partkey, ps\_suppkey\} \rightarrow \{ps\_availqty\}$. On SYS1, it was possible to force the use of a merge-join instead of hash-join and the plan chosen is shown in Figure 16(b).

We compared the actual running time of PYRO-O's plan with those of PostgreSQL and SYS1 by forcing our plan on the respective systems. Figures 17 and 18 show the execution times. 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 on SYS1. In all cases, the forced PYRO-O plan performed significantly better than the other plans. The main reason for the improvement was the use of a partial sort of *lineitem* index entries as against a full sort. The final sort on *partkey* was not very expensive as only a few tuples needed to be sorted.

For Query 4 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.

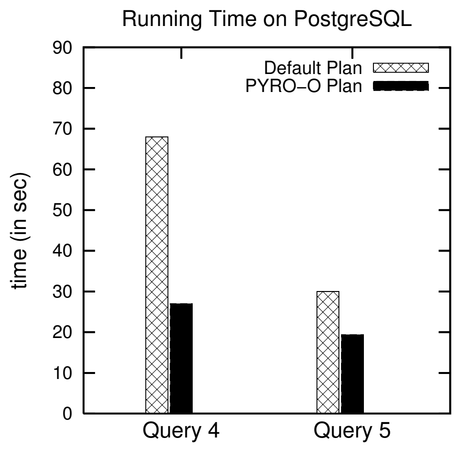


Fig. 17 Performance on PostgreSQL

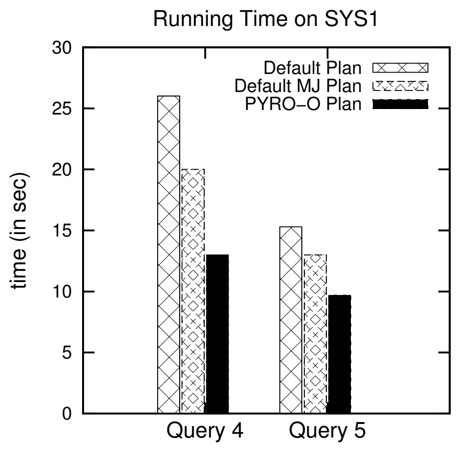


Fig. 18 Performance on SYS1

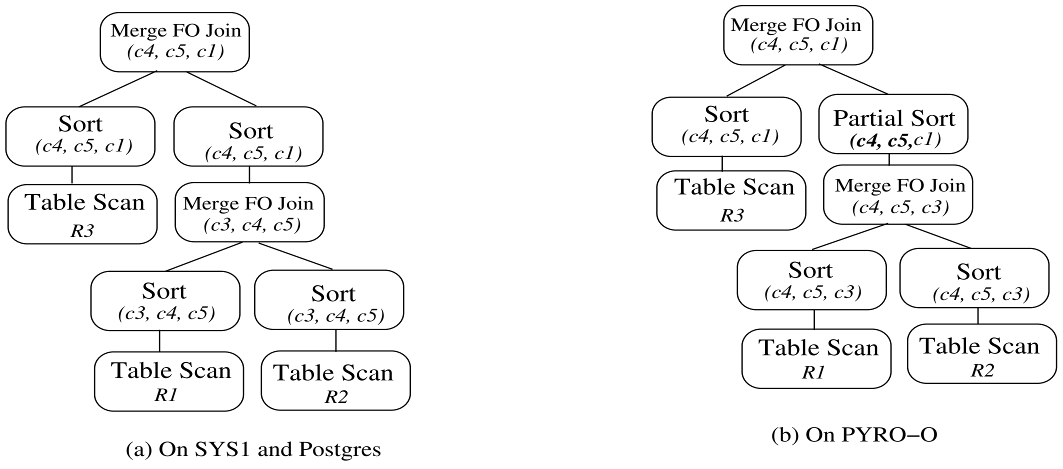


Fig. 19 Plans for Query 5

Experiment B2

This experiment uses Query 5, which has two full outer joins with two common attributes between the joins. We performed this experiment to see whether the systems we compare with are designed to exploit attributes common to multiple sort-based operators.

Query 5 *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 19(a), both SYS1 and PostgreSQL chose sort orders that do not share any common prefix. The plan chosen by PYRO-O is shown in Figure 19(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. The execution timings for Query 5 on PostgreSQL and SYS1 are shown in Figures 17 and 18 respectively.

Experiment B3

In this experiment we compare our approach of choosing orders, PYRO-O, with the exhaustive approach, and a heuristic used by PostgreSQL. PostgreSQL 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

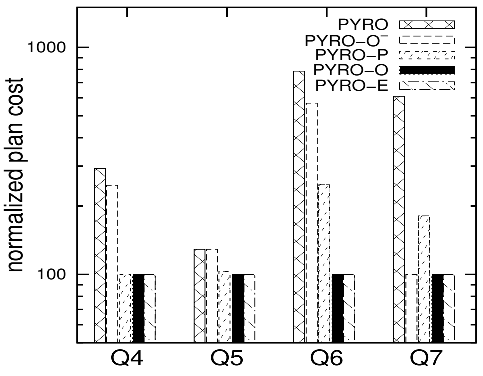


Fig. 20 Evaluating Different Heuristics

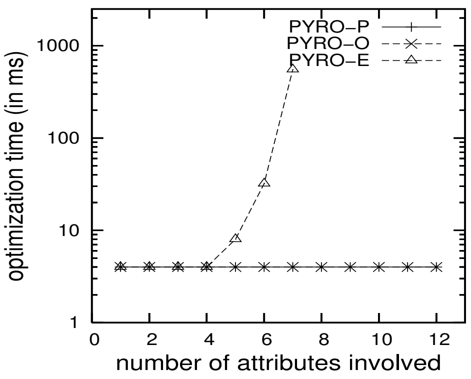


Fig. 21 Optimizer Scalability

$n-1$ attributes are ordered arbitrarily. We implemented PostgreSQL's heuristic in PYRO along with the extensions to exploit partial sort orders and we 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 baseline 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 20 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, Q4 and Q5 stand for Query 4 and Query 5 of Experiments B1 and B2. Q6 and Q7 stand for Query 6 and Query 7, which are given below. For Q4 and Q5, as very few attributes were involved in the join condition, PostgreSQL's 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 well since it makes an arbitrary choice for secondary orders. Although PYRO-O is seen to perform as well as the exhaustive approach on the queries considered, the choice of sort orders in PYRO-O may not be optimal for all queries. The plan refinement algorithm described in Section 4.2.2 uses an approximation and cannot guarantee optimal choice of sort orders.

Query 6 *Total value executed for a given order*

```
SELECT T1.UserId, T1.BasketId, T1.ParentOrderId,
       T1.WaveId, T1.ChildOrderId,
       (T1.Quantity * T1.Price) as OrderValue,
       SUM(T2.Quantity * T2.Price) as ExecValue
FROM TRAN T1, TRAN T2
WHERE T1.UserId=T2.UserId AND
       T1.ParentOrderId=T2.ParentOrderId AND
       T1.BasketId=T2.BasketId AND
       T1.WaveId=T2.WaveId AND
       T1.ChildOrderId=T2.ChildOrderId AND
       T1.TranType='New' AND
       T2.TranType='Executed'
GROUP BY T1.UserId, T1.BasketId, T1.ParentOrderId,
       T1.WaveId, T1.ChildOrderId;
```

Query 7 *Basket Analytics*

```
SELECT * FROM BASKET B, ANALYTICS A
WHERE B.ProdType = A.ProdType AND
       B.Symbol = A.Symbol AND
       B.Exchange = A.Exchange;
```

5.3 Optimization Overheads

Changes to the optimizer implementation to take partial sort orders into consideration are described in Section 2.2. We avoid generating optimization sub-goals for every possible partial sort orders, by making the following change to the optimizer: if the physical operator being considered for a logical operator partially fulfills the sort order requirement of its parent operator, the optimizer adds a partial sort enforcer on top of the child operator. Thus there is no increase in the number of plans explored when considering partial sort orders alone. However, the number of optimization sub-goals generated for operators like merge-join depends on the number of sort orders we consider as interesting.

During plan generation, the number of sort orders we try at each join or aggregate node is of the order of the number of indices present that are useful for answering the query, which in most practical case is expected to be small. Figure 21 shows the scalability of the three heuristics. For this 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 fewer interesting orders than PYRO-P.

The plan-refinement algorithm presented in Section 3.3 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 straight forward to implement. The optimizer extensions neatly integrated into our existing Volcano style optimizer.

6 Related Work

Both System R [15] and Volcano [7] 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, both System R and Volcano assume that 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. Heuristics such as the PostgreSQL heuristic, 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 addressed these two issues.

The seminal work by Simmen et.al. [16] describes techniques to infer sort orders from functional dependencies and predicates applied, and thereby avoids redundant sort enforcers in the plan. Simmen et.al. [16] briefly mention 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. [16] also note that the approach of carrying a flexible order specification increases the complexity of the code significantly. Our techniques do not use flexible order specifications and hence can be incorporated into an existing optimizer with minimal changes. Further, our techniques work uniformly across all types of operators that have a flexible order requirement.

Claussen et.al. [2] explore early-sorting as a means to reduce sorting cost in query plans. The key idea is to avoid sorting of large intermediate join results by pushing sorting to base relations, and using order preserving hash joins. There has been significant work on avoiding redundant sorting by inferring sort orders and groupings using functional dependencies [16,17,12,13]. These techniques are complementary to our work. A more recent work in the same direction is by Zhou et.al. [18], which addresses the problem of generating efficient parallel query execution plans for massive data analysis over clusters of commodity hardware. The key challenge in query optimization for shared-nothing clusters is to reduce data repartitioning, which involves expensive data reshuffling across nodes. They present a unified framework for reasoning about partitioning, grouping and sorting, and show how they incorporated the proposed techniques in the SCOPE optimizer. Optimization with data partitioning is an additional motivation for our work, as coordinated choice of physical properties for binary operators is crucial. It remains important even if data is memory/flash resident reducing the impact of sort order optimization for disk IO.

Graefe et.al. [6] introduce the notion of hash teams, which realizes many of the benefits of *interesting sort orders* in hash based query processing. The key difference between interesting sort orders and hash teams is that input sort orders may be useful even when they do not match the required sort order *exactly.* On the other hand, hash teams, as described in [6], are applicable only when two joins have exactly the same set of attributes involved in the join predicate . Unlike sort orders, partitions created by hashing on attribute set $\{a, b\}$ are not useful to obtain partitions on $\{a, c\}$ (although it is possible to consider approaches where we only hash on $\{a\}$ and use it to obtain the two required set of partitions).

Yu Cao et.al. [1] present a complementary technique, where they consider suffixes of sort orders. If input is sorted on a list of columns, *e.g., (a, b, c)* and the desired sort order is *(c)*, then tuples for each distinct value of *(a, b)* can be interpreted as runs to be merged, thereby avoiding the run generation step.

The present article is an extended version of the conference paper [8], with the following being the important additions. We present the common prefix problem on trees in more detail and give the proof of its hardness, both for stars and binary trees. We present the polynomial-time algorithm for paths. An important property of the minimal favorable orders is stated and proved in the appendix. The experimental section has been enhanced with additional queries from real-world applications and execution plans.

7 Summary

In this paper we addressed the problem of choosing efficient sort orders for sort-based operators such as merge-join and sort-based grouping. We showed that even a simplified version of the problem is *NP-hard*, and proposed principled heuristics for choosing interesting orders. Our heuristics are guided by the notion of favorable orders. We take into account important issues such as partially matching sort orders and attributes common to multiple operators. We then explained how the solution can be used for choosing efficient parameter sort orders for nested queries. We presented a detailed experimental study on widely used database systems, and the results showed significant performance improvements due to the proposed techniques for several queries.

References

1. Yu Cao, Ramadhana Bramandia, Chee-Yong Chan, and Kian-Lee Tan. Optimized Query Evaluation Using Cooperative Sorts. In *Intl. Conf. on Data Engineering*, 2010.
2. J. Claussen, A. Kemper, D. Kossmann, and C. Wiesner. Exploiting Early Sorting and Early Partitioning for Decision Support Query Processing. *VLDB Journal: Very Large Data Bases*, 9(3), 2000.
3. Josep Diaz, Jordi Petit, and Maria Serna. A Survey of Graph Layout Problems. *ACM Comput. Surv.*, 34(3), 2002.
4. Vladimir Estivill-Castro and Derick Wood. A Survey of Adaptive Sorting Algorithms. *ACM Comput. Surv.*, 24(4), 1992.
5. Goetz Graefe. Implementing Sorting in Database Systems. *ACM Comput. Surv.*, 38(3), 2006.
6. Goetz Graefe, Ross Bunker, and Shaun Cooper. Hash Joins and Hash Teams in Microsoft SQL Server. In *Intl. Conf. on Very Large Databases*, pages 86–97, 1998.
7. Goetz Graefe and W.J. McKenna. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *Intl. Conf. on Data Engineering*, 1993.
8. Ravindra Guravannavar and S Sudarshan. Reducing Order Enforcement Cost in Complex Query Plans. In *Intl. Conf. on Data Engineering*, 2007.
9. Sreyash Kenkre and Sundar Vishwanathan. The common prefix problem on trees. *Information Processing Letters*, 105(6):245–248, 2008.
10. Donald E. Knuth. *The Art of Programming, Vol. 3: Sorting and Searching.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.
11. Per-Åke Larson. External Sorting: Run Formation Revisited. *IEEE Trans. Knowl. Data Eng.*, 15(4), 2003.
12. Thomas Neumann and Guido Moerkotte. A Combined Framework for Grouping and Order Optimization. In *Intl. Conf. on Very Large Databases*, 2004.
13. Thomas Neumann and Guido Moerkotte. An Efficient Framework for Order Optimization. In *Intl. Conf. on Data Engineering*, 2004.
14. Prasan Roy. *Multi-Query Optimization and Applications.* PhD thesis, Indian Institute of Technology, Bombay, Department of Computer Sc. & Engg., 2001.
15. P. Griffiths Selinger, M.M.Astrahan, D.D.Chamberlin, R.A.Lorie, and T.G.Price. Access Path Sselection in a Relational Database Management System. In *ACM SIGMOD*, 1979.
16. David Simmen, Eugene Shekita, and Timothy Malkemus. Fundamental Techniques for Order Optimization. In *ACM SIGMOD*, 1996.
17. Xiaoyu Wang and Mitch Cherniack. Avoiding Sorting and Grouping In Processing Queries. In *Intl. Conf. on Very Large Databases*, 2003.
18. Jingren Zhou, Per-Åke Larson, and Ronnie Chaiken. Incorporating Partitioning and Parallel Plans into the SCOPE Optimizer. In *Intl. Conf. on Data Engineering*, 2010.

APPENDIX

A Proof of Optimality with Minimal Favorable Orders

The notion of *minimal favorable orders*, introduced in Section 4, served as the basis for our heuristics for selecting sort orders. Since it is hard to compute the exact set of minimal favorable orders, we used a heuristic approach to compute them approximately. However, it is interesting to study the properties of minimal favorable orders. In this section we give a proof of Theorem 3 stated earlier in Section 4.2.1. The theorem essentially states the following: to identify an optimal sort order, it is sufficient to consider only the minimal favorable orders and not the full set of favorable orders. Below, we repeat the formal statement of the theorem and present a proof. The proof makes use of notation introduced in Sections 2.1 and 4.1.

Theorem 3 *The set $\mathcal{I}(e, o)$ computed with exact ford-min contains an optimal sort order o_p for the optimization goal $e = (e_l \bowtie e_r)$ with (o) as the required output sort order.*

We prove Theorem 3 under the following assumption: If o_1, o_2 are two sort orders on the same set of attributes (*i.e.,* attrs(o_1) = attrs(o_2)), then the CPU cost of sorting the result of an expression e to obtain o_1 will be same as that for o_2, *i.e.,* cpu-cost(e, o_1) =cpu-cost(e, o_2).

Proof Consider the optimization goal for a join expression ($e = e_l \bowtie e_r$, with (o) as the sort order required on the result of e. Let S be the set of join attributes and o' be any sort order on S. The cost of the best merge-join plan for e, when o' is chosen as the sort order for e_l, e_r, is given by:

$$PC(e, o, o') = cbp(e_l, o') + cbp(e_r, o') + coe(e, o', o) + CM(e_l, e_r),$$
$$\text{where } CM(e_l, e_r) \text{ is the cost of merging.} \quad (1)$$

In Equation 1, we note that $CM(e_l, e_r)$ is independent of the sort order o'.

Let o_b be an *optimal sort order* for $el \bowtie e_r$. Assume $o_b \notin \mathcal{I}(e)$. We show that $\exists o_p \in \mathcal{I}(e)$ such that $PC(o_p)$ =$PC(o_b)$.

Case 1: Suppose o_b is such that $o_b \notin ford(e_l) \cup ford(e_r)$.

$$\begin{aligned} PC(e, o, o_b) &= cbp(e_l, o_b) + cbp(e_r, o_b) + coe(e, o_b, o) + \\ &\quad CM(e_l, e_r) \quad (2) \\ &\quad \text{Since, } o_b \notin ford(e_l) \cup ford(e_r) \text{ we can write} \\ &= cbp(e_l, \epsilon) + coe(e_l, \epsilon, o_b) + cbp(e_r, \epsilon) + \\ &\quad coe(e_r, \epsilon, o_b) + coe(e, o_b, o) + CM(e_l, e_r) \quad (3) \end{aligned}$$

Let o_p be a sort order in $\mathcal{I}(e)$ such that $o \wedge S \leq o_p$, where o is the required output sort order in the optimization goal. The existence of such a sort order in $\mathcal{I}(e)$ directly follows from the construction of $\mathcal{I}(e)$, specifically, steps 1 and 2 in Section 4.2.1.

Since both o_b and o_p are sort orders on the same attribute set S, we have

$$coe(e_l, \epsilon, o_b) = coe(e_l, \epsilon, o_p) \text{ and } coe(e_r, \epsilon, o_b) = coe(e_r, \epsilon, o_p) \quad (4)$$

Substituting Equation 4 in Equation 3 we get:

$$\begin{aligned} PC(e, o, o_b) &= cbp(e_l, \epsilon) + coe(e_l, \epsilon, o_p) + cbp(e_r, \epsilon) + \\ &\quad coe(e_r, \epsilon, o_p) + coe(e, o_b, o) + CM(e_l, e_r) \quad (5) \\ &\geq cbp(e_l, o_p) + cbp(e_r, o_p) + coe(e, o_b, o) + \\ &\quad CM(e_l, e_r) \quad (6) \end{aligned}$$

As $(o \wedge S) \leq o_p$, we have $(o_b \wedge o) \leq (o_p \wedge o)$ (because o_b is a permutation of S). Therefore, $coe(e, o_b, o) \geq coe(e, o_p, o)$. From this, we can rewrite Equation 6 as:

$$\begin{aligned} PC(e, o, o_b) &\geq cbp(e_l, o_p) + cbp(e_r, o_p) + coe(e, o_p, o) + CM(e_l, e_r) \\ &\geq PC(e, o, o_p). \end{aligned}$$

By assumption o_b is an optimal sort order. So we conclude - $PC(e, o, o_b) = PC(e, o, o_p)$. In other words, $\mathcal{I}(e)$ contains a sort order o_p having the same plan cost as the optimal sort order o_b.

Case 2: Suppose o_b is such that $o_b \in ford(e_l)$ or $ford(e_r)$ but not both.
Without loss of generality we assume $o_b \in ford(e_l)$. This implies one of the following:

(i) $\exists o' \in$ *ford-min*(e_l) such that $o_b \leq o'$ and $cbp(e_l, o_b) = cbp(e_l, o')$ or
(ii) $\exists o' \in$ *ford-min*(e_l) such that $o' \leq o_b$ and $cbp(e_l, o') + coe(e_l, o', o_b) = cbp(e_l, o_b)$.

We now consider, each of these cases separately.
Case 2-A: Suppose condition (i), repeated below as Equation 7, holds.

$$\exists o' \in \textit{ford-min}(e_l) \text{ such that } o_b \leq o' \text{ and } cbp(e_l, o_b) = cbp(e_l, o') \quad (7)$$

$o' \in$ *ford-min*(e_l) implies $(o' \wedge S) \in$ *ford-min*(e_l, S). Therefore, from the construction of set $\mathcal{I}(e)$, we know:

$$\exists o_p \in \mathcal{I}(e) \text{ such that } (o' \wedge S) \leq o_p \quad (8)$$

$$\text{Since } o_b \leq o', \text{ we know } (o_b \wedge S) \leq (o' \wedge S) \quad (9)$$

Substituting Equation 9 in Equation 8, we get $(o_b \wedge S) \leq o_p$. Since both o_b and o_p are permutations of the same attribute set S, we must have $o_b = o_p$. *i.e.,* the optimal sort order o_b must be in $\mathcal{I}(e)$.
Case 2-B: Suppose condition (ii), repeated below as Equation 10, holds.

$$\begin{aligned} &\exists o' \in \textit{ford-min}(e_l) \text{ such that } o' \leq o_b \text{ and} \\ &\quad cbp(e_l, o') + coe(e_l, o', o_b) = cbp(e_l, o_b) \quad (10) \end{aligned}$$

The plan cost for e, with o_b as as the chosen sort order, is given by:

$$\begin{aligned} PC(e, o_b) &= cbp(e_l, o_b) + cbp(e_r, o_b) + coe(e, o_b, o) + CM(e_l, e_r) \\ &\quad \text{Substituting for } cbp(e_l, o_b) \text{ from Equation 10, we get} \\ &= cbp(e_l, o') + coe(e_l, o', o_b) + cbp(e_r, o_b) + \\ &\quad coe(e, o_b, o) + CM(e_l, e_r) \quad (11) \end{aligned}$$

$o' \in$ *ford-min*(e_l) implies $\exists o_p \in \mathcal{I}(e)$ such that $(o' \wedge S) \leq o_p$. Since $o' \leq o_b$, we know $attrs(o') \subseteq S$. Therefore, we have $o' \wedge S = o'$. And hence, $o' \leq o_p$. Also, since both o_p and o_b are permutations of S, we have $|o_b| = |o_p|$.
Since, $o_b \notin ford(e_r)$, we have $cbp(e_r, o_b) = cbp(e_r, o_p)$. Substituting this in Equation 11, we get:

$$\begin{aligned} PC(e, o_b) &= cbp(e_l, o') + coe(e_l, o', o_b) + cbp(e_r, o_p) + \\ &\quad coe(e, o_b, o) + CM(e_l, e_r) \quad (12) \end{aligned}$$

Since $o' \leq o_b$ and $o' \leq o_p$ and $|o_b| = |o_p|$ we can write Equation 12 as:

$$\begin{aligned} PC(e, o_b) &= cbp(e_l, o') + coe(e_l, o', o_p) + cbp(e_r, o_p) + \\ &\quad coe(e, o_b, o) + CM(e_l, e_r) \\ &\geq cbp(e_l, o_p) + cbp(e_r, o_p) + \\ &\quad coe(e, o_b, o) + CM(e_l, e_r) \quad (13) \end{aligned}$$

Now, we show that $coe(e, o_b, o) \geq coe(e, o_p, o)$ to complete the proof.
Case (a): Suppose, $o' \leq o$.
Since $\mathcal{I}(e)$ contains a sort order which subsumes, $o \wedge S$, it is possible to choose o_p from $\mathcal{I}(e)$ such that $(o \wedge S) \leq o_p$. This implies, $|o_b \wedge o| \leq |o_p \wedge o|$. Hence, $coe(e, o_b, o) \geq coe(e, o_p, o)$. Substituting this in Equation 13, we get:

$$\begin{aligned} PC(e, o_b) &\geq cbp(e_l, o_p) + cbp(e_r, o_p) + coe(e, o_p, o) + CM(e_l, e_r) \\ &\geq PC(e, o_p) \end{aligned}$$

Case (b): Suppose, $o' \not\leq o$.
Now, $o' \wedge o = o_b \wedge o = o_p \wedge o$ (because $o' \leq o_b$ and $o' \leq o_p$). Therefore, $coe(e, o_b, o) = coe(e, o_p, o)$. Substituting this in Equation 13, we get:

$$\begin{aligned} PC(e, o_b) &\geq cbp(e_l, o_p) + cbp(e_r, o_p) + coe(e, o_p, o) + CM(e_l, e_r) \\ &\geq PC(e, o_p) \end{aligned}$$

Case 3: Suppose o_b is present in both *ford*(e_l) and *ford*(e_r)
This implies one of the following:

(i) $\exists o' \in$ *ford-min*$(e_l) \cup$ *ford-min*(e_r) such that $o_b \leq o'$. In this case the proof can proceed as in Case 2-A.
(ii) $\exists o_1 \in$ *ford-min*(e_l) and $\exists o_2 \in$ *ford-min*(e_r) such that *(a)* $o_1 \leq o_b$ and $o_2 \leq o_b$ and *(b)* $cbp(e_l, o_1) + coe(e_l, o_1, o_b) = cbp(e_l, o_b)$ and *(c)* $cbp(e_r, o_2) + coe(e_r, o_2, o_b) = cbp(e_r, o_b)$.

Since $o_1 \leq o_b$ and $o_2 \leq o_b$, either $o_1 \leq o_2$ or $o_2 \leq o_1$. Hence, $\exists o_p \in \mathcal{I}(e)$ such that $o_1 \leq o_p$ and $o_2 \leq o_p$. Choosing such an o_p, and proceeding as in Case 2-B we can prove $PC(e, o_b) \geq PC(e, o_p)$

This completes the proof of Theorem 3.