REGULAR PAPER

Which sort orders are interesting?

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

Received: 19 March 2010 / Revised: 26 May 2011 / Accepted: 12 June 2011 / Published online: 29 June 2011
© Springer-Verlag 2011

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.

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 query1 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 interesting sort orders can be factorial in the number of attributes involved in the operation. This may not be

1 Contains all attributes required to answer the query.
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 \( (r.c_1 = s.c_1 \text{ and } r.c_2 = s.c_2) \). 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 Sect. 3, we show that a special case of finding optimal sort orders is NP-hard and give a 1/2-benefit approximation algorithm to choose sort orders for a tree of merge-joins.

   - In Sect. 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 Sect. 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 (1) added choices of sort orders for base relations due to the use of query-covering indices (2) sort orders that partially match an order requirement (3) requirement of same sort order from multiple inputs (e.g., merge-based join, union) and (4) common attributes between multiple joins, grouping and set operations.

   - In Sect. 4.2, we also show how to integrate our extensions into a cost-based optimizer.

3. We present experimental results (in Sect. 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.

This article is an extended version of our conference paper [8]. The important additions made in this article are described in Sect. 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**

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

The two catalog tables contain 2 million records each, and have average tuple sizes of 100 and 80 bytes. We assume a disk block size of 4K bytes and 10,000 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

\(^2\) Graefe and McKenna [7] also considers other types of enforcers, collectively called physical property enforcers.