Optimized Query Evaluation Using Cooperative Sorts

Yu Cao, Ramadhana Bramandia, Chee-Yong Chan, Kian-Lee Tan

Department of Computer Science
National University of Singapore
{caoyu, bramandia, chancy, tankl}@comp.nus.edu.sg

Abstract— **Many applications require sorting a table over multiple sort orders: generation of multiple reports from a table, evaluation of a complex query that involves multiple instances of a relation, and batch processing of a set of queries. In this paper, we study how multiple sortings of a table can be efficiently performed. We introduce a new evaluation technique, called cooperative sort, that exploits the relationships among the input set of sort orders to minimize I/O operations for the collection of sort operations. To demonstrate the efficiency of the proposed scheme, we implemented it in PostgreSQL and evaluated its performance using both TPC-DS benchmark and synthetic data. Our experimental results show significant performance improvement over the traditional non-cooperative sorting scheme.**

I. Introduction

Sorting is a frequently used and expensive operation in database systems. It is employed not only to produce sorted output, but also in many sort-based algorithms for aggregation, duplicate removal, join, and set operations. As such, it has been extensively studied [1], [2], [3], [4], [5], [6]. The standard technique adopted in most commercial systems is based on the external merge-sort algorithm.

In this paper, we investigate the problem of efficiently sorting a table in multiple sort orders. It turns out that such multiple sortings of a table is not uncommon in many applications. For example, in many organizations, many reports are generated at the end of the day/week/month. Typically, these reports contain the same content but in different sort orders. A bank may produce reports ordered by amount deposited/withdrawn/balance, date, branch, and so on. Similarly, examination schedules are usually printed in different orders - order by course number, dates, examiners, and invigilators. Yet another example arises in decision making applications where a complex query typically contains multiple instances of a base relation or view [7].

Now, in all these examples, the queries can be processed by sorting a table multiple times, once per sorting order. This seems to be wasteful of resources especially since we are sorting a single table (or materialized view) albeit over multiple sort orderings. Intuitively, if we can somehow salvage the (partial) work done to sort a table on a particular order for a subsequent sort order on the same table, then we may reduce the overall processing cost for multiple sortings. This is exactly what we set out to achieve in this paper.

We begin by considering the scenario of sorting a table T in two different sort orders o_1 and o_2, where o_1 and o_2 are sequences of some attributes of T. Based on the relationships between o_1 and o_2, we identify four cases in which we can exploit the (partial) results of sorting T on o_1 to speed up the sorting of T on o_2. In case 1, o_2 is a (strict) prefix of o_1, and in case 2, o_1 and o_2 share a common prefix. Under these situations, the sorted T on o_1 can be re-used directly (for case 1) or via a light-weight reorganization (for case 2) to sort T on o_2. As a result, the cost to sort T on o_2 is totally eliminated (case 1) or significantly reduced (case 2). We refer to these cases as *result sharing* of instance sorts.

While cases 1 and 2 are straightforward, cases 3 and 4 offer sharing opportunities that have not been previously explored. As an example of case 3, a prefix of o_2 appears as a substring of o_1, e.g., o_2 is the attribute a_2 and o_1 is a composite of two attributes (a_1, a_2). In this case, the sorted T on o_1 can be viewed as a concatenation of tuple clusters each of which contains records ordered on o_2. Thus, each tuple cluster of the sorted T on (a_1, a_2) corresponds to a distinct a_1 value. Within this cluster, the a_2 values are sorted. Thus, we could treat these tuple clusters as initial runs for the sort on o_2, and re-use them to facilitate the run merging phase of the merge-sort algorithm without having to perform the (initial) run formation phase. However, a naive exploitation of these readily available runs may not be beneficial as there may be too many of such tuple clusters. To handle this, we propose a *cooperative sort* procedure. It first organizes tuples of T into an intermediate form T' such that (a) T' can be used to produce sorted T on o_1 efficiently with only (possibly) in-memory sorting; (b) T' can be used as initial runs for the sort on o_2 efficiently as T' is structured such that the total number of initial runs for sorting T on o_2 has been significantly reduced (compared to that of using the tuple clusters from the sorted T on o_1). *Cooperative sort* tries to keep as much the benefit of avoiding scanning T and initial run formation for the sort on o_2 as possible by minimizing the cost of the corresponding run merging.

Finally, case 4 covers the most general scenarios where none of the prefixes of o_2 appear as substrings of o_1 and vice versa. As a simple example, we may want to sort T on attribute a_1 as well as attribute a_2. This case can be easily handled by first sorting T on (a_1, a_2) and so reducing this case to case 3.

When a table has to be sorted on more than two orders, we have further opportunity to optimize the performance. Given a table with multiple sorts, we model it as the minimum directed Steiner tree problem. As the number of sortings is manageable, we adopt a brute force algorithm to find the optimal solution on how each sort will be sequenced and completed.

Another direction that our work can be applied is in enriching the plan space for query optimization. For some query plan, it is possible that replacing a hash-join (in an optimal plan) with a sub-optimal sort-merge join can lead to an overall lower cost due to the sort-sharing opportunities either within the same query or among a batch of queries.

In this paper, we make the following key constributions: (1) We systematically study the problem of evaluating multiple sortings on a relation beginning with an analysis of the relationships for two sort orders and generalizing to multiple sort orders; (2) We propose a novel evaluation technique, *cooperative sort*, that optimizes the evaluation of two sort orders by deriving an intermediate hybird sort order; (3) We perform a comprehensive experimental evaluation of our proposed techniques with an implementation in PostgreSQL. Our performance results show that cooperative sort outperforms conventional sort on average by 25% and up to 35%. Moreover, our study shows the potential benefit of enriching the optimizer search space with cooperative sort.

The rest of this paper is organized as follows. In Section II, we present some preliminaries. Section III formally discusses the techniques of result sharing and cooperation between two instance sorts. We elaborate the details of *cooperative sort* in Section IV. Further discussions about general sort sharing are presented in Section V. In Section VI, we generalize cooperative sort to evaluate more than two sort operations. Our experimental study presented in Section VII validates the effectiveness of our proposed techniques. We discuss relevant work in Section VIII and finally conclude in Section IX.

II. Preliminaries

Sort orders are referred as o, o_1, o_2 *etc.*, each of which is a sequence of distinct attributes $(a_1, a_2, \cdots a_n)$, $n \geq 1$, of the relation T[1] to be sorted. In this paper we utilize the following main notations, some of which are borrowed from [8]:

- $s_i = sort(T, o_i)$: a sort operation s_i on T, with order o_i.
- $cost(s)$: the I/O cost (in number of accessed blocks) for sort operation s.
- $attrs(o)$: the set of attributes in sort order o.
- $|o|$: number of attributes in the sort order o.
- $o_1 < o_2$: o_1 is a proper prefix of o_2.
- $o_1 \leq o_2$: o_1 is a prefix of o_2.
- $o_1 \wedge o_2$: the longest common prefix between o_1 and o_2.
- $o_1 + o_2$: sort order obtained by concatenating o_1 and o_2.
- $o - A$: sort order obtained by removing from o those attributes that also appear in the set of attributes A.
- *o-segment*: the cluster of tuples in T that have the same value for $attrs(o)$.
- $B(e)$: size of tuples of expression e, in number of blocks.
- $D(e, o)$: number of distinct values for $attrs(o)$ in tuples of expression e; i.e., $D(e, o) = |\pi_o(e)|$.
- M: number of memory blocks available for sorting.

In this paper, we assume that initial sorted runs are generated using replacement selection, and our cost model assumes that each initial sorted run is of size $2M$ blocks. The external sorting of a relation T is done using the well-known F-way merge sort technique, where F is the merge order (i.e., number of runs that can be merged using M). Our cost model for a sort operation s on T using M blocks of memory is given by

$$cost(s) = 2 \times B(T) \times (\lceil log_F(\frac{B(T)}{2M})\rceil + 1). \tag{1}$$

III. Sort Sharing Techniques

In this section, we present an overview of techniques for optimizing the evaluation of multiple sorts on a relation T. We will first focus on the basic case involving only two sort operations, and then explain how our techniques can be easily extended to the general case in Section VI.

Consider two sort operations $s_1 = sort(T, o_1)$ and $s_2 = sort(T, o_2)$. By exploiting the relationship between o_1 and o_2, the total I/O cost of the two sortings can be reduced with one of two key techniques:

- **Result sharing technique**: the idea is to leverage the output of one sort operation to more efficiently evaluate the other sort operation.
- **Cooperative sorting technique**: the idea is to create "hybrid" sorted runs that can benefit the evaluation of both sort operations.

While the *result sharing technique* has been previously discussed in other contexts [9], [8], to the best of our knowledge, we are the first to investigate the *cooperative sorting technique*.

The relationships between o_1 and o_2 that we are interested in can be classified into four main cases:

- Case 1: o_2 is a prefix of o_1; i.e., $o_2 \leq o_1$.
- Case 2: o_1 and o_2 share a non-empty common prefix which is a proper prefix of o_2; i.e, $0 < |o_1 \wedge o_2| < |o_2|$.
- Case 3: the set of attributes in o_2, where $o_2 = o_{21} + o_{22}$, is a subset of the attributes in a prefix, $o_{11} + o_{12}$, of $o_1 = o_{11} + o_{12} + o_{13}$, such that $o_{21} = o_{12}$, $attrs(o_{22}) \subseteq attrs(o_{11})$, $|o_{11}| > 0$, $|o_{13}| \geq 0$ and $|o_{22}| \geq 0$.
- Case 4: o_1 and o_2 do not satisfy any of the above cases, but $o_1 + (o_2 - attrs(o_1))$ and o_2 satisfy Case 3.

Example 3.1 Consider the following sort orders: $o_1 = (a_1, a_2)$, $o_2 = (a_2)$, $o_3 = (a_1, a_3, a_2)$, $o_4 = (a_2, a_3, a_6, a_7)$, and $o_5 = (a_6, a_3, a_2)$. The pair o_1 and o_3 is an example of case 2 with a common prefix given by (a_1). Two examples of case 3 are: the pair o_1 and o_2, and the pair o_4 and o_5. An example of case 4 is the pair o_3 and o_5. □

The first two cases are the more familiar and simpler cases which can be efficiently evaluated using the result sharing technique. The last two cases are the two new scenarios that we investigate in this paper, and their evaluations can be optimized by our proposed cooperative sorting technique to be presented in the next section. Note that although case 3 seems rather

[1]For simplicity, our discussion assumes T to be a relation, but our techniques also apply when T is the output of some query subplan.

contrived and specialized, this is actually the most fundamental case to optimize as the most general case 4 is evaluated by reducing it to case 3.

In the remainder of this section, we discuss how to apply result sharing technique to evaluate case 1 and case 2.

For the first case, since a relation T sorted on o_1 is trivially also sorted on o_2, it is sufficient to perform only $sort(T, o_1)$; therefore, s_2 is not evaluated explicitly and $cost(s_2) = 0$.

For case 2, suppose $o' = o_1 \wedge o_2$ such that $o_1 = o' + o'_1$, $o_2 = o' + o'_2$, $|o'_1| \geq 0$ and $|o'_2| > 0$. In this case, a relation T sorted on o_1 is also partially sorted on o_2: the output of s_1 can be viewed as a concatenation of o'*-segments*, and each such segment can be sorted independently on o'_2 to form the sorted output for s_2. If the size of each o'*-segment* is no larger than M blocks, then the sorting of each segment on o'_2 can be performed efficiently using internal sorting and s_2 can be evaluated with only a single pass of reading the output of s_1. As noted by [8], the strategy to evaluate s_2 by sorting o'*-segments* also helps to significantly reduce the number of tuple comparisons: the complexity of independently sorting k segments each of size n/k tuples is $O(k \ * n/k \ log(n/k)) = O(n \ log(n/k))$ in contrast to a complexity of $O(n \ log(n))$ for a single sort of all n tuples.

For case 2, s_1 is evaluated using conventional external merge-sort and $cost(s_1)$ is given by Equation 1. Following [8], $cost(s_2) = \sum_{i=1}^{D(T,o')} cost(sort(se_i, o'_2))$, where $cost(sort(se_i, o'_2))$ denotes the cost of sorting the ith o'*-segment* se_i in the sorted output of s_1. If $B(se_i) \leq M$, $cost(sort(se_i, o'_2))$ is simply the cost of performing an internal sorting; otherwise, it is given by Equation 1. If we assume that the values of o' follow a uniform distribution, then $B(se_i) = B(T)/D(T, o')$.

IV. Cooperative Sorting

In this section, we present a novel technique, termed *cooperative sorting*, to efficiently evaluate two sort operations $s_1 = sort(T, o_1)$ and $s_2 = sort(T, o_2)$, when o_1 and o_2 satisfy cases 3 or 4 identified in the previous section. For simplicity, we assume in this section that all the attributes in a sort order are to be sorted in ascending order. We discuss how to handle a combination of ascending and descending sorting orders in Section V and how to optimize evaluation for more than two sort operations in Section VI.

We first explain how the most general case 4 can be evaluated by reducing the problem to case 3. The reduction is achieved by defining a new sort order $o'_1 = o_1 + (o_2 - attrs(o_1))$. Now, the pair of sort orders o'_1 and o_2 satisfy case 3 (by definition), and the sort operations s_1 and s_2 can be computed indirectly by evaluating the pair of sort operations on orders o'_1 and o_2. Since o_1 is a proper prefix of o'_1, the sorted output on o'_1 is also sorted on o_1.

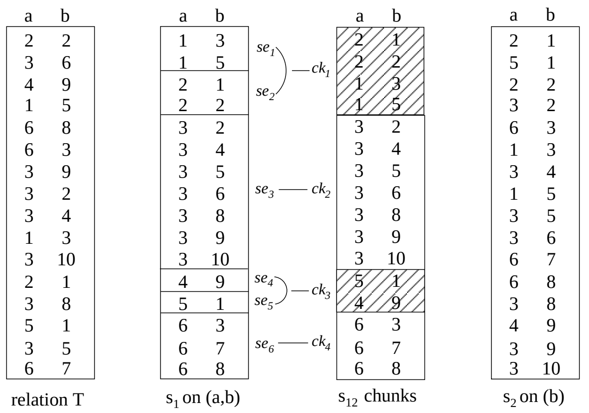
In the rest of this section, we turn our attention to the evaluation of case 3 using cooperative sorting.

A. Key Ideas

Recall that for case 3, we have $o_1 = o_{11} + o_{12} + o_{13}$ and $o_2 = o_{21} + o_{22}$, such that $o_{12} = o_{21}$, $attrs(o_{22}) \subseteq attrs(o_{11})$, $|o_{11}| > 0$, $|o_{13}| \geq 0$ and $|o_{22}| \geq 0$.

Observe that the output of s_1 can be viewed as a concatenation of o_{11}*-segments*, each of which is also sorted on o_2 and can be used as an initial sorted run for s_2. Thus, the *result sharing technique* can actually be applied to evaluate case 3 by first evaluating s_1 followed by merging the resulting o_{11}*-segments* to compute s_2. However, depending on $D(T, o_{11})$ and data skew, the number of o_{11}*-segments* generated by s_1 could be very large with many small segments. In this situation, merging a large number of small sorted runs to evaluate s_2 could lead to an overall performance that is bad or even worse than performing a conventional external sorting of T on o_2. The following example illustrates the limitation of the result sharing technique.

Example 4.1 Consider the relation $T(a, b)$ in Fig. 1, which is used as a running example in this section. Assume the following: each tuple occupies one disk block, the available sorting memory can hold four tuples (i.e., $M = 4$), and the merge order $F = 2$. Consider two sort operations s_1 and s_2 on T, with orders $o_1 = (a, b)$ and $o_2 = (b)$, respectively. Obviously, o_1 and o_2 satisfy case 3 with $o_{11} = (a)$. The output of s_1 is a concatenation of six a*-segments* (se_1 to se_6), each of which is sorted on (b). By applying the result sharing technique to evaluate s_2, the six a-segments of s_1 can be used as initial sorted runs for s_2 and merged to evaluate s_2 using three merge passes with $F = 2$. However, this result sharing is actually more costly than a conventional external sorting: using replacement selection, three initial sorted runs are generated which can be merged with only two merge passes. □



relation T a	relation T b	s_1 on (a,b) a	s_1 on (a,b) b	s_{12} chunks a	s_{12} chunks b	s_2 on (b) a	s_2 on (b) b
2	2	1	3	2	1	2	1
3	6	1	5	2	2	5	1
4	9	2	1	1	3	2	2
1	5	2	2	1	5	3	2
6	8	3	2	3	2	6	3
6	3	3	4	3	4	1	3
3	9	3	5	3	5	3	4
3	2	3	6	3	6	1	5
3	4	3	8	3	8	3	5
1	3	3	9	3	9	3	6
3	10	3	10	3	10	6	7
2	1	4	9	5	1	6	8
3	8	5	1	4	9	3	8
5	1	6	3	6	3	4	9
3	5	6	7	6	7	3	9
6	7	6	8	6	8	3	10

Fig. 1. Cooperative Sorting Example: $M = 4$ and $F = 2$

To avoid the drawback of result sharing technique, we propose a novel technique, *cooperative sorting*, that derives the outputs of both s_1 and s_2 from the output of an intermediate sort operation s_{12} that is based on a "hybrid" sort order. The goal is to take into account the available sorting memory M to generate longer (and thus fewer) initial sorted runs to reduce the cost of run merging phase for s_2.

The key idea behind cooperative sorting is the use of an intermediate sort operation s_{12} that is based on a novel hybrid sort order. The output of s_{12} is a sequence of chunks of tuples which are classified into *natural chunks* and *composite chunks* such that the tuples in the natural chunks are ordered by o_1 and the tuples in the composite chunks are ordered by o_2. The chunks are defined in terms of the o_{11}-segments in the output of s_1 as follows. Conceptually, the output of s_1 is a sequence of o_{11}-segments which can be partitioned into a sequence of chunks each of which consists of a consecutive sequence of o_{11}-segments. A *composite chunk* consists of two or more consecutive o_{11}-segments such that (1) the size of the chunk in terms of the total number of tuples is at most M, and (2) the tuples in the chunk are sorted on o_2. A *natural chunk* consists of exactly one o_{11}-segment and the tuples are sorted on o_1; there is no constraint on the size of a natural chunk.

Example 4.2 Consider again the running example in Fig. 1. Based on $o_1 = (a, b)$ and $o_2 = (b)$, the output of s_1, which is partitioned into six a-segments (se_1 to se_6), can be organized into four s_{12} chunks consisting of two composite chunks, ck_1 and ck_3 (shown shaded), and two natural chunks, ck_2 and ck_4 (shown non-shaded). The size of each composite chunk is at most M (i.e., 4 tuples). □

The chunks in s_{12} are defined to satisfy the following three useful properties. First, the chunks are o_1-*order preserving* in the sense that if a chunk ck_i precedes another chunk ck_j in the output of s_{12}, then every tuple in ck_i has a smaller o_1 value than every tuple in ck_j. Second, since the size of each composite chunk is no more than M, which is the size of the sorting memory, the tuples in a composite chunk can be sorted very efficiently using internal sorting. The above two properties enable s_1 to be efficiently derived from s_{12} by a sequential scan of the chunks. Third, the tuples in each chunk of s_{12} are sorted on o_2.

We now elaborate on how both s_1 and s_2 are derived from the output of s_{12}. To derive s_1, the s_{12} chunks are scanned and processed sequentially: if the chunk is a natural chunk, the tuples are already ordered on o_1 and can simply be output sequentially; otherwise, we first load all the tuples in the chunk into the sorting memory, internally sort the tuples on o_1, and then output the sorted tuples sequentially. To derive s_2 from s_{12}, we treat each chunk as an initial sorted run for s_2 (due to third property) and merge these chunks to derive s_2.

Since the derivation of s_2 from s_{12} is more efficient if there are fewer chunks to be merged, the composite s_{12} chunks should be constructed as large as possible (within the size constraint) to reduce the number of s_{12} chunks. The minimum number of s_{12} chunks can be generated efficiently by a greedy heuristic that sequentially scans the o_{11}-segments of s_1 as follows: if the size of a segment se_i exceeds M, then se_i forms a natural segment; otherwise, determine the longest sequence of consecutive segments $se_i, se_{i+1}, \cdots se_j$ such that its total size is no more than M tuples. If $i = j$, then se_i forms a natural segment; otherwise, create a composite chunk with the segments $se_i, \cdots, se_j$.

By deriving both s_1 and s_2 from the s_{12} chunks, cooperative sorting avoids the I/O cost of independently generating two sets of initial sorted runs for s_1 and s_2. However, there are three potential sources of overhead for cooperative sorting. First, the s_{12} chunks are actually created by merging the initial sorted runs of s_1 (to be explained in the next subsection), which requires a non-trivial extension of the conventional run merging technique. Thus, the total cost of computing s_{12} chunks could be more costly than computing s_1. Second, the number of s_{12} chunks generated by cooperative sorting could still be more than the number of initial sorted runs for s_2 generated by the conventional initial run formation phase, possibly resulting in a more costly run merging phase for cooperative sorting. Third, the derivation of s_1 from s_{12} incurs the computation overhead of performing internal sorts on composite chunks. Therefore, cooperative sorting is not always superior to conventional sorting. Thus, both approaches should be considered in a cost-based manner by the query optimizer for evaluating multiple sorts on a relation.

B. Overview of Hybrid Sort Operation

In this section, we present an overview of our proposed approach to perform s_{12}, the intermediate sort operation whose output is used to derive s_1 and s_2. The computation of s_{12} consists of four main steps. In the first step, we scan the relation T to create initial s_1 runs (i.e., initial sorted runs on o_1). This step is performed using the conventional run formation technique. During the first step, we also collect the following information about T and the initial s_1 runs: the number of distinct o_{11} values in T, and the number of tuples corresponding to each distinct o_{11} value on each initial s_1 run. Thus, at the end of the first step, we know the size of each o_{11}-*segment* and the distribution of each o_{11}-segment's tuples among the initial s_1 runs.

In the second step, based on the information collected from the first step, we apply the previously described greedy algorithm to determine a partitioning of the sequence of o_{11}-segements in T into a minimum number of s_{12} chunks. This partitioning provides the following information about the output of s_{12}: the number and the sequence of s_{12} chunks, the size of each chunk, and the o_{11}-segments that comprise each chunk. Note that the tuples that belong to the same s_{12} chunk are generally distributed across several initial s_1 runs. Since the s_{12} chunks are o_1-order preserving, each initial s_1 run is actually partitioned into a sequence of *chunklets*, where each chunklet is the subset of tuples in the s_1 run that belong to the same s_{12} chunk. We use the notation $ckl_{i,j}$ to denote the chunklet in the j^{th} initial s_1 run that belongs to the i^{th} chunk in s_{12}. Similar to chunks, chunklets are classified into natural and composite chunklets.

Example 4.3 Fig. 2 illustrates the two initial s_1 runs (ordered on $o_1 = a$) generated from the relation T in our running example in Fig. 1. The first initial run consists of chunklets $ckl_{1,1}, ckl_{2,1}, ckl_{3,1}$, and $ckl_{4,1}$, while the second initial run consists of chunklets $ckl_{1,2}, ckl_{2,2}, ckl_{3,2}$, and $ckl_{4,2}$. □

In the third step, we merge the initial s_1 runs to generate the initial s_{12} runs. Essentially, each initial s_{12} run is created

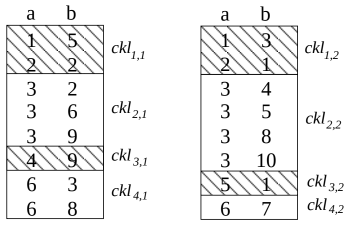


Fig. 2. Initial s_1 Runs for Relation T in Example of Fig. 1

by merging a set of F initial s_1 runs; specifically, the corresponding chunklets in the F input s_1 runs are merged to form a longer chunklet in the output s_{12} run. Thus, each initial s_{12} run is also a sequence of s_{12} chunklets, and the chunklets in the initial s_{12} runs satisfy the same three properties as s_{12} chunks. This merging operation is more complex than the conventional run merging step and will be elaborated in Section IV-C.

In the fourth step, the initial s_{12} sorted runs are merged to generate the output of s_{12}. This is done using the conventional external run merging technique with a minor extension for the tuple comparison operator to take into account the different sorting orders of tuples within natural and composite chunks. Specifically, when comparing two tuples t_1 and t_2 during the merging, if t_1 and t_2 belong to the same composite (resp. natural) chunk, then t_1 precedes t_2 iff t_1 has a smaller value for o_2 (resp. o_1) compared to t_2; otherwise, t_1 precedes t_2 iff t_1 belongs to a chunk that precedes t_2's chunk in s_{12}. To facilitate the subsequent merging of s_{12} chunks to derive s_2, the final pass of this merging step organizes the final single s_{12} run into separate s_{12} chunks. Note that the fourth step is skipped if the the third step produces only one initial s_{12} run.

C. Generating Initial S_{12} Runs

In this section, we elaborate on the procedure to merge initial s_1 runs to generate initial s_{12} runs. Unlike the run merging procedure in conventional external sorting algorithm, this procedure in cooperative sorting is more intricate due to the fact that the input and output runs in our merge operation are of different "types": we are merging input runs that are ordered on o_1 to generate longer output runs that that sorted on the hybrid order for s_{12} defined in Section IV-A. For the case where the chunklets in the input runs to be merged are natural chunklets, the merging procedure is simple and follows the conventional external merge procedure since the input and output orders are the same. However, for the case where the chunklets to be merged are composite chunklets, the merging requires an internal sorting operation to order the tuples in the merged chunklet on o_2. This requires loading into memory all the chunklet tuples that belong to the same chunk for sorting. In this section, we propose an efficient *batched reading* strategy to judiciously load tuples from the sorted runs into the sorting memory to optimize the I/O performance.

To motivate the need for our proposed scheme, consider the following simple conservative approach to merge F initial s_1 runs into an initial s_{12} run. The merging processes the chunklets in the input sorted runs one chunk at a time based on the chunk order. If the current chunk being processed is composite, we first read all the tuples in the chunklets from the F input runs that belong to this chunk into the sorting memory, perform an internal sorting of the tuples, and output the merged chunklet. If the current chunk being processed is natural, we read n tuples from the corresponding chunklet of each input run, where $n = \min\{\text{size of the chunklet}, \lfloor M/F \rfloor\}$, and merge them using the conventional merging technique. While this conservative approach is sound, this approach might be fragmenting the reading of an input run into too many short sequential I/O reads. On the other hand, an arbitrary tuple reading strategy can lead to "deadlock" situations. For example, consider the two initial s_1 runs shown in Fig. 1 and suppose that we are merging chunklets for the first chunk with $M = 4$. If we had read in three tuples from the first initial s_1 run into the sorting memory, then a deadlock situation would arise as the remaining sorting memory space is not adequate to permit loading in all the tuples from the first chunklet of the second initial s_1 run for sorting. Thus, the goal of our proposed *batched reading* strategy is to decompose the reading of each input run into as few batched reads of the tuples as possible to maximize sequential I/O without getting into deadlocks.

Our proposed batched read strategy consists of two main steps. First, we partition each initial s_1 run into a sequence of batches, where each batch corresponds a sequence of tuples in the run. A run that has been partitioned in n batches will be read into the sorting memory using n read requests, each time reading in a complete batch of tuples. The second step then merges the initial s_1 runs based on the schedule of batched reads produced by the first step.

Since the merging of composite chunklets require an internal sorting, all the chunklets belonging to the same composite chunk among the F runs being merged need to fit into the available sorting memory M along with any previously read tuples that have yet to be merged. Let RP_i^m (resp. RP_i^d) denote the set of tuples that belong to the chunk ck_i that are in the sorting memory (resp. still on disk). Thus, for any composite chunk ck_i that has some tuple in the sorting memory which has yet to be merged, ck_i must satisfy the following constraint to avoid the previously described deadlock problem:

$$B(RP_i^m) + B(RP_i^d) + \sum_{k>i} B(RP_k^m) \leq M \qquad (2)$$

We classify a chunk/chunklet as a *small chunk/chunket* if its size is no larger than $\lfloor M/F \rfloor$; otherwise, it is considered to be a *large chunk/chunket*.

For simplicity, our batched read strategy is designed based on the following two rules:

- Each composite chunklet, as well as each small natural chunklet will be completely read in one batched read.
- Each large natural chunklet will be read by a series of batched reads each with size of $\lfloor M/F \rfloor$ followed by a final batched read. Moreover, the first batched read will start from the head of the chunklet.

It follows that each batched read can be classified into one of four types based on its starting and ending points:

1) the batch starts from the head of a composite/natural chunklet and ends at the tail of a composite/natural

chunklet.
2) the batch starts from the head of a natural chunklet and ends inside the same chunklet.
3) the batch starts and ends inside a natural chunklet.
4) the batch starts inside a natural chunklet and ends at the tail of a composite/natural chunklet.

For both type-2 and type-3 batch reads, their sizes are $\lfloor M/F \rfloor$.

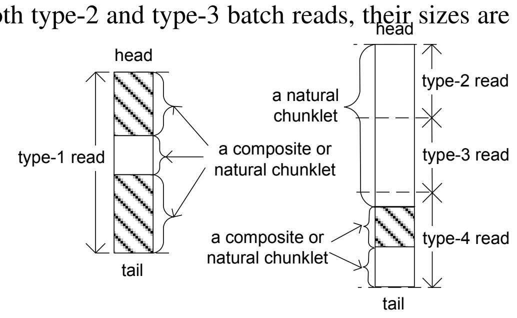


Fig. 3. Illustration of Four Types of Batched Reads

Fig. 3 illustrates the four types of batch reads. A large natural chunklet is read by batched reads of types 2, 3 and 4, while a composite or small natural chunklet is read by batched reads of types 1 and 4. Each type-2 read corresponds to a type-4 read and a (possibly empty) set of type-3 reads.

Moreover, to balance the reading opportunity among natural chunklets among the runs being merged, we make two *further restrictions*: (1) in a type-1 read, the total size of a natural chunklet along with any following tuples must not exceed $\lfloor M/F \rfloor$; (2) the size of a type-4 read is at most $\lfloor M/F \rfloor$.

For the F initial s_1 runs, Algorithm 1 precomputes a sequence BR of all type-1, type-2 and type-4 reads to be conducted during the actual merging. The composition of each type-1 or type-4 read is decided using Algorithm 2. Note that the start point of a type-4 read is fixed within a natural chunklet. BR does not record type-3 reads as they are fixed and can be easily deduced from each pair of type-2 and type-4 reads. Based on BR, Algorithm 3 performs the merging of s_1 runs by reading each run in batches. Note that although the runtime merging process follows the static schedule BR strictly for type-1 and type-2 reads, the actual ordering for type-3 and type-4 reads for natural chunklets might need to be determined dynamically at runtime for correctness. More specifically, a type-3 or type-4 batch read of a run will be dynamically selected as the next read to be executed if the set of in-memory tuples S of the most-recently read batch from this run belong to a natural chunk and S will be earliest batch of in-memory tuples to be exhausted by the merge.

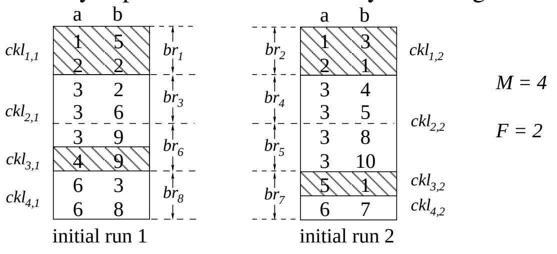


Fig. 4. Batched Read Sequence for Merging Two Initial s_1 Runs in Fig. 2

Fig. 4 shows the sequence of eight batched reads (br_1 to br_8) conducted during the merging of the two initial s_1 runs in Fig. 2. br_1, br_2, br_7 and br_8 are type-1 reads; br_3 and br_4 are type-2 reads, while br_6 and br_5 are their corresponding type-4 reads respectively. There is no type-3 read in this example. Table I shows the calculated BR. Note that since the last tuple (3, 5) read by br_4 is smaller than the last tuple (3, 6) read by br_3, tuples read by br_4 will be exhausted first and thus br_5 is actually read before br_6 at runtime, which cannot be predicted according to BR.

TABLE I
The Entries in BR for Example in Fig. 4

i	1	2	3	4	5	6	7	8
$BR[i]$	br_1	br_2	br_3	br_6	br_4	br_5	br_7	br_8

Algorithm 1 `ComputeBR`

```
Output: a pre-calculated sequence $BR$ of batched reads
1: $idx = 1$
2: for $i = 1$ to $N$ do // $N$ is the number of $s_{12}$ chunks
3:   if $s_{12}$ chunk $i$ is composite then
4:     for $j = 1$ to $F$ do
5:       if $ckl_{i,j}$ is non-empty and has not been processed yet then
6:         $BR[idx] = BatchedRead(ckl_{i,j})$ // determine a type-1 read starting from $ckl_{i,j}$
7:         $idx = idx + 1$
8:   else
9:     for $j = 1$ to $F$ do
10:      if $ckl_{i,j}$ is non-empty and has not been processed yet then
11:        if $size(ckl_{i,j}) > \lfloor M/F \rfloor$ then
12:          $BR[idx]$ is a type-2 read starting from $ckl_{i,j}$
13:          $idx = idx + 1$
14:          $BR[idx] = BatchedRead(ckl_{i,j})$ // determine the corresponding type-4 read
15:          $idx = idx + 1$
16:        else
17:          $BR[idx] = BatchedRead(ckl_{i,j})$ // determine a type-1 read starting from $ckl_{i,j}$
18:          $idx = idx + 1$
```

Algorithm 2 `BatchedRead`

```
Input: $ckl_{i,j}$
Output: a type-1 (or type-4) batched read $br$
1: initialize a type-1 (or type-4) $br$ including complete (or partial) $ckl_{i,j}$
2: $k = i + 1$
3: while true do // check whether $ckl_{k,j}$ can be included in $br$
4:   if $ckl_{k,j}$ is a natural chunklet && $size(ckl_{k,j}) > \lfloor M/F \rfloor$ ||
     including $ckl_{k,j}$ in $br$ violates the further restrictions ||
     including $ckl_{k,j}$ in $br$ violates Inequation 2 for some composite chunk
     $j (i \le j < k)$ then
5:     break
6:   include $ckl_{k,j}$ in $br$
7:   $k = k + 1$
```

D. Cost Model

In this section, we present a simple analytical cost model for cooperative sorting. The total cost of using a cooperative sort s_{12} to evaluate two sort operations s_1 and s_2 consists of three components: (1) the cost $C_{s_{12}}$ of generating s_{12} chunks, which is assumed to be equal to the cost C_{s_1} incurred for evaluating s_1 (given by Equation 1), plus the cost C_{is} of performing internal sortings on composite chunklets within initial s_1 runs; (2) the cost $C_{s_{12} \rightarrow s_1}$ of deriving s_1 is equal to the total cost of performing internal sortings for all the composite s_{12} chunks;

Algorithm 3 `FinalSequence`

```
Input: $BR$
Output: a batched read sequence during actual run merging
1: initialize an empty batched read pool $P$
2: $i = 1$
3: while $i \leq length(BR)$ do
4:     conduct $BR[i]$ when enough memory space is available
5:     $br = BR[i]$ // mark the read for later reference
6:     if $BR[i]$ is a type-1 read then // otherwise it must be type-2
7:         $i = i + 1$
8:     else
9:         add into $P$ the corresponding type-4 read $BR[i+1]$ along with the
           set of type-3 reads between $BR[i]$ and $BR[i+1]$
10:        $i = i + 2$
11:    if br is the last conducted read among the set of type-1 and type-2
       reads for tuples of a natural chunk then // check point
12:        if $P$ is non-empty then
13:            driven by the merge progress, conduct in a specific order all type-3
               and type-4 reads in $P$
14:            restore $P$ to be empty
```

(3) the cost $C_{s_{12} \rightarrow s_2}$ of deriving s_2 is given by $2 \times B(T) \times \lceil log_F N \rceil$, where N is the number of s_{12} chunks.

Assuming a uniform distribution for the values of o_{11}, there are two cases to be considered.

Case 1: $B(T)/D(T, o_{11}) \leq 0.5M$. All s_{12} chunks are composite, and

$$N = \lceil D(T, o_{11})/k \rceil \tag{3}$$

where k is the maximal integer such that $k \times B(T)/D(T, o_{11}) \leq M$.

Let $cpu\_cost(S)$ denote the cost of internally sorting tuples of total size S. We have

$$C_{is} = \frac{B(T)}{2M} \times N \times cpu\_cost(2M/N) \tag{4}$$

$$C_{s_{12} \rightarrow s_1} = N \times cpu\_cost(k \times B(T)/D(T, o_{11})) \tag{5}$$

Case 2: $B(T)/D(T, o_{11}) > 0.5M$. All s_{12} chunks are natural, and

$$N = D(T, o_{11}) \tag{6}$$

$$C_{is} = C_{s_{12} \rightarrow s_1} = 0 \tag{7}$$

Note that in the second case, the total cost of cooperative sorting is actually the same as that of applying result sharing technique. The performance of cooperative sorting depends partially on $D(T, o_{11})$ and the relative sizes of o_{11}-segments. Besides the distinct value cardinality of o_{11}, the statistical value distribution of o_{11} has little impact on the performance.

E. Extensions

In this section, we describe two important extensions of cooperative sorting.

1) Final Merge Optimization: If the external sorting operation is part of a pipelining query plan, a common optimization is to stop the run merge phase just before the final merge step so that the final merge step can be done as part of the generation of the sorted output. In this way, the final merge optimization saves one read and one write scan on T.

If the final merge optimization is enabled, the run merging phase of cooperative sorting will end up with N $(1 < N \leq F)$ s_{12} runs. The output of s_1 is derived by merging the N s_{12} runs on-the-fly, with the *batched reading* strategy being used to sort the tuples in composite chunklets based on o_1 before the merging. As for s_2, each chunklet within the s_{12} runs is treated as an initial run for s_2. For the special case where the number of initial s_1 runs generated for s_{12} is no more than F, these initial s_1 runs can be transformed into initial s_2 runs by simply sorting the composite chunklets based on o_2.

2) Adapting to Other Merge Patterns: Our description of cooperative sorting in Section IV-B has assumed that the sorted runs are merged using k-way merge pattern for ease of presentation. The cooperative sorting approach can be easily adapted to other merge patterns such as *polyphase merge* and *cascade merge* [1]. In the general case, the collection of the runs to be merged could consist of a combination of initial s_1 sorted runs and s_{12} sorted runs. The *batched reading* strategy can be easily modified so that composite chunklets within the s_{12} runs, which have already been sorted on o_2, need not be internally sorted again as part of the merging.

V. Discussions

A. Ascending/Descending Ordering

In this section, we explain how to extend our proposed techniques to handle the general case where a sort order can consist of attributes to be sorted in a combination of ascending and descending orders. For a sort attribute a, let a' and a'' denote the ascending and descending ordering of a, respectively. We can treat a' and a'' as two different attributes in sort orders. For two sort orders o_1 and o_2, we refer to them as a *reverse pair* if (1) $o_1 = o_2$ when ascending/descending orderings are ignored; and (2) for each attribute a' (resp. a'') in o_1, the corresponding attribute in o_2 is a'' (resp. a'). Clearly, for a reverse pair, the result of one order can be easily converted into the result of the other by a backward scan of the sorted output.

We now revisit the four cases for o_1 and o_2 with the additional consideration of ascending/descending order. We only discuss cases where o_1 and o_2 satisfy one of the four cases if all their attributes were to be sorted in ascending order.

For cases 1 and 2, there must exist a longest pair of prefixes, o_{11} and o_{21}, from o_1 and o_2, respectively, such that (o_{11}, o_{21}) forms a reverse pair. By using a backward scan, we can treat o_{11} and o_{21} as a common prefix; thus, the result sharing technique is still applicable. For example, $o_1 = (a', b'')$ and $o_2 = (a'', b')$ still satisfy case 1, while $o_1 = (a', b')$ and $o_2 = (a'', b')$ now satisfy case 2.

For case 3, cooperative sorting is still applicable. For a composite s_{12} chunk, the ascending/descending orders can be handled by internal sorting. For a natural chunk, we generate it as usual with a sorted order o_{12}. To use this natural chunk as an initial run in s_2, its sort order should be o_{21} (each tuple in the chunk has the same value for $attrs(o_{22})$). With a backward scan, o_{12} and o_{21} satisfy either case 1 or case 2. Therefore, we can easily convert the order of the natural chunk on-the-fly from o_{12} to o_{21} when it is merged for s_2.

Since case 4 is handled by reducing to case 3, the discussion for case 4 is similar to case 3.

B. Dynamic Optimization for Cases 3 and 4

Recall that for cases 3 and 4, all the three sorting techniques (conventional sorting, result sharing, and cooperative sorting) are applicable. The choice of which technique to apply can actually be determined dynamically at run-time. Note that all the three techniques share a common step of generating initial s_1 sorted runs. After the initial s_1 runs have been computed, we have precise information on the number of distinct o_{11} values, the number and sizes of s_{12} chunks, and the sizes and distributions of the s_{12} chunklets among the s_1 initial runs. With this information, we can more accurately determine the cost estimates of the three competing techniques and choose the most efficient technique to evaluate s_1 and s_2 at run-time.

VI. Optimization of Multiple Sortings

In this section, we generalize our discussion of evaluating two sort operations to consider the evaluation of a collection of sort operations $S = \{s_1, s_2, \cdots, s_k\}$, $k \geq 2$, where each $s_i = sort(T, o_i)$ is a sort operation on relation T with sort order o_i. We first consider the extension of cooperative sort to handle more than two sort orders, and then explain how to optimize the evaluation of multiple sorts on a relation.

A. K-way Cooperative Sort

In Section IV, we develop cooperative sort to evaluate two sort operations s_1 and s_2. In general, given a collection of k sort operations, a natural question to ask is whether it makes sense to generalize the binary cooperative sort to a k-way cooperative sort so that all k sort operations can be efficiently evaluated. We refer to k as the *order* of cooperative sort.

Given two sort orders o_i and o_j, let $o_i \cdot o_j$ denote the sort order $o_i + (o_j - attrs(o_i))$ as discussed for case 4 in Section IV. Generalizing binary cooperative sort to k-way cooperative sort requires generating $k-1$ intermediate sort operations: $\{s'_2, s'_3, \cdots, s'_k\}$, where the sort order o'_i associated with each s'_i is given by $o'_i = o_1 \cdot o_2 \cdot o_3 \cdot ... \cdot o_i$. Each s'_i corresponds to case 3 with sort orders o'_i and o_i.

k-way cooperative sort works as follows: s_1 is derived from s'_2, and each s_i, $i > 1$ is derived from s'_i. The main idea of the extension is for all intermediate sorts to be derived from a single collection of initial runs that is sorted on o'_k. Fortunately, the following analytical result based on our cost model shows that it is not necessary to consider k-way cooperative sort for $k > 2$ (the proof can be found in [10]).

Theorem 6.1: For each query Q involving multiple sorts on some relation, there exists an optimal query evaluation plan P for Q where the maximum order of cooperative sort used in P is at most 2.

Based on Theorem 6.1, we only consider *binary cooperative sort* in subsequent discussions and we simply use the term *cooperative sort* to refer to it.

B. Multiple Sorting Optimization

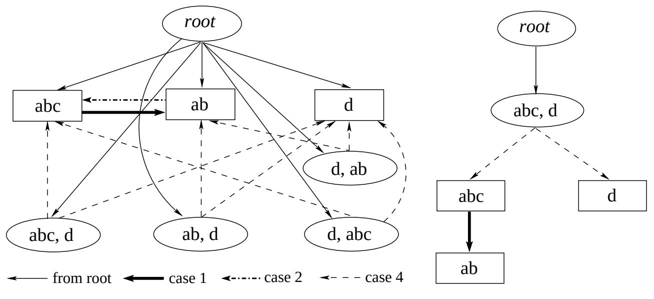
Given a collection S of k sort operations, there are many ways in which these operations can be ordered to exploit cooperative sort. In this section, we model this optimization problem as a graph problem. Given S, we construct a directed search graph $G(V, E)$, where $V = V_a \cup V_b$, V_a represents the set of *sort nodes* and V_b represents the set of *cooperative sort operator nodes*.

Each sort node $u \in V_a$ is associated with a sort order, denoted by $order(u)$. For each sort operation $s = sort(T, o)$ $\in S$, we create a sort node $u \in V_a$ and set $order(u)$ to o. Each directed edge (u, v) from sort node u to sort node v is associated with $cost(u, v)$ equal to the cost of sorting T that currently satisfies $order(u)$ to satisfy $order(v)$. There are two types of directed edges between sort nodes, corresponding to case 1 and case 2 in Section III.

For each pair of sort nodes u and v such that $order(u)$ and $order(v)$ satisfy case 3 or case 4, we create a new cooperative sort operator node $w \in V_b$ and associate $operand(w) = (u, v)$. This node represents a cooperative sort operation from which u and v can be derived. From w, we add two directed edges: (w, u) and (w, v). Both $cost(w, u)$ and $cost(w, v)$ are labeled based on the cost model in Section IV.

Finally, an artificial $root \in V_a$ node is added to represent the relation T without a particular order. We add an edge from $root$ to each existing node v in V, with $cost(root, v)$ equal to the cost of a conventional sorting operation.

Once the search graph has been constructed, the optimal solution can be found by computing the minimum directed Steiner tree of G. The terminal nodes of the Steiner tree are the set of sort nodes in V_a.



(a) The Sample Search Graph G (b) Steiner Tree of G

Fig. 5. An Example of Multiple Sorting Optimization

Example 6.1 Consider three sort operations $sort(T, ab)$, $sort(T, abc)$ and $sort(T, d)$, where a, b, and c are attributes of T. The search graph is depicted in Fig. 5(a), where the sort (resp. cooperative sort) nodes are represented by rectangles (resp. ellipses). The computed Steiner tree for this search graph is shown in Fig. 5(b). Based on the Steiner tree, a feasible evaluation plan is as follows: first perform cooperative sort between $sort(T, abc)$ and $sort(T, d)$, and derive $sort(T, ab)$ from $sort(T, abc)$. □

Although finding the minimum directed Steiner tree is an NP-hard problem [11], applying a brute-force algorithm is actually adequate if $|V_b|$ is small. Basically, we enumerate every subset of V_b to be used in the spanning tree and find one with the minimum cost. The complexity of finding the directed minimum spanning tree is $O(N^2)$ where N is the number of nodes in the graph [12]. Hence, the total complexity of the algorithm is $O(2^{|V_b|}|V|^2)$. In our context, since $|V_a|$ is small and $|V_b| \leq |V_a|^2$ is also small, a brute-force solution is reasonable; otherwise, heuristic/approximation algorithms [13], [14] can be applied here.

VII. Performance Study

We validated our ideas using a prototype built in PostgreSQL 8.3.5 [15]. All experiments were performed on a Dell workstation with a Quad-Core Intel Xeon 2.66GHz processor, 8GB of memory, one 500G SATA disk and another 750GB SATA disk, running Linux 2.6.22. Both the operating system and PostgreSQL system are built on the 500GB disk, while the databases of PostgreSQL are stored on the 750GB disk.

This performance study focused on the effect of cooperative sort. In our implementation, the cooperative sort is integrated into PostgreSQL as a standard operator. It adopts k-way merge pattern and is capable of final merge optimization. For the purpose of fair comparison, we also converted the run merge pattern of the original sort operation in PostgreSQL from polyphase to k-way (We have evaluated the performance of our k-way merge implementation against PostgreSQL's polyphase merge scheme. Our results showed no significant differences between the two schemes. Readers may refer to [10] for details.). Moreover, we added a post-optimizer that implements the optimization techniques in Section VI. The post-optimizer receives an execution plan from the original query optimizer, exploits sharing and cooperation opportunities between sorts in a cost-based manner and, whenever possible, generates a cheaper plan enhanced with cooperative sorts. By switching on and off the post-optimizer, we can easily compare the cost of processing a query under the cooperative sort operation against that of the corresponding two independent sort operations.

A. Micro-benchmark Test

In this section, we use a micro-benchmark test to compare the performance of cooperative sort against two independent sort operations. We define a query template Q:

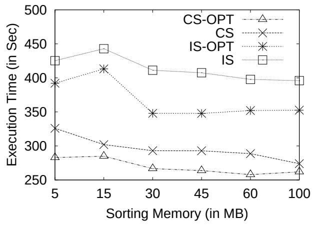
(select *attr1*, *attr2* from T order by *attr1*, *attr2*)
union all
(select *attr1*, *attr2* from T order by *attr2*)

This template also serves to simulate two queries in a batch. The execution plan of Q is an immediate result union of two sorts, s_1 and s_2, on the same relational table T. The sort orders of s_1 and s_2 are $(attr1, attr2)$ and $(attr2)$ respectively and thus satisfy case 3.

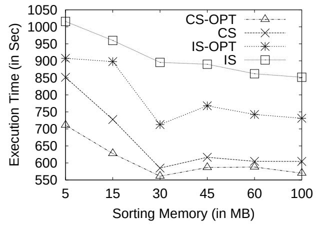
We generated six concrete queries with the above query template by using three different relations from the TPC-DS [16] benchmark for T and two different scale factors (denoted by SF) to vary the size of T. The statistical information about three relations, along with their sort attributes, are shown in Table VII-A. The scale factor SF values used are 40 GB and 100 GB. Another experimental parameter that we varied is the available sorting memory dedicated to each sorting operation (denoted by M) with values ranging from 5 MB to 200 MB. The sorting memory values are chosen such that at least half of them will result in a single run merge step.

We compare the performance of two basic evaluation techniques for sorting: the conventional technique of using *two independent sorts* (denoted by IS) and our proposed *cooperative sort* (denoted by CS). We also enable/disable the final merge optimization to study the combined effectiveness of this optimization with the basic techniques. We use CS-OPT and IS-OPT to denote the variants that have the optimization enabled, and CS and IS to denote the variants that have the optimization disabled.

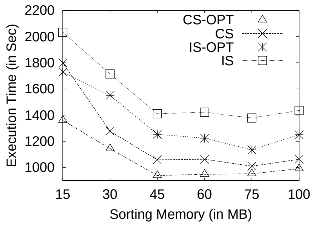
Each total execution time reported refers to the total query evaluation time including the I/O cost of reading the sorted outputs of s_1 and s_2. Each query timing is measured with the query running alone in the database system; and the operating system is restarted between queries to clear the system cache.



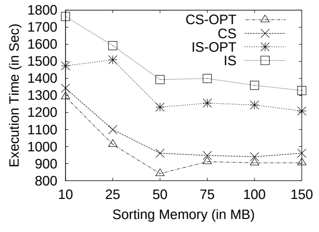
(a) web_sales, SF 40



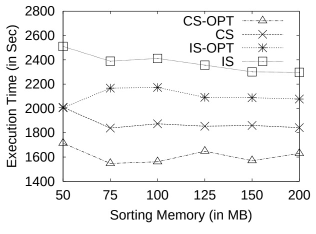
(b) catalog_sales, SF 40



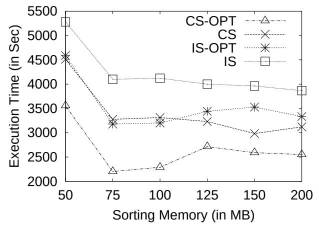
(c) store_sales, SF 40



(d) web_sales, SF 100



(e) catalog_sales, SF 100



(f) store_sales, SF 100

Fig. 6. Performance Comparison on TPC-DS Dataset

1) General Results: Fig. 6 compares the performance of the four evaluation strategies as a function of the sorting memory size; the comparison for each query is shown on a separate graph. The detailed breakdown of the various cost components for CS and IS are shown in Table VII-A.1; only SF 40 dataset is shown due to space limitation, the complete results can be found in [10]. The meanings of these cost components are given in Table VII-A.1.

Due to space limitations, we shall not present detailed query-by-query analysis. Instead, we will summarize the more interesting findings here.

First, we observe that CS(-OPT) offers significant performance improvement over IS(-OPT) in most queries. The savings range from a few seconds to 1,033 seconds which is achieved by CS-OPT over IS-OPT for the query on *store_sales* with $M = 50$ and $SF = 100$ in Fig. 6. In terms of relative

TABLE II

TESTED TPC-DS DATASET

relation	*attr1*	*attr2*	number of tuples (in million)	tuple size (in byte)
web_sales	`ws_item_sk`	`ws_sold_time_sk`	$0.72 \times SF$	226
catalog_sales	`cs_item_sk`	`cs_sold_time_sk`	$1.44 \times SF$	226
store_sales	`ss_item_sk`	`ss_sold_time_sk`	$2.88 \times SF$	164

TABLE III

THE COMPONENT COSTS OF CS AND IS

	notation	**description**
CS	$RF_{cs}(s_{12})$	initial run formation cost for s_{12} (i.e., creating initial s_1 sorted runs)
	$RM_{cs}(s_{12})$	run merge cost for s_{12} (i.e., creating s_{12} chunks)
	$RM_{cs}(s_2)$	run merge cost for s_2 (i.e., merging s_{12} chunks to derive s_2)
	$SC_{cs}(s_{12})$	cost of internal sorting to create initial s_{12} runs from initial s_1 runs
	$SC_{cs}(s_1)$	cost of internal sorting during the derivation of s_1 output from s_{12}
IS	$RF_{is}(s_1)$	initial run formation cost for s_1 (i.e., creating initial s_1 sorted runs)
	$RM_{is}(s_1)$	run merge cost for s_1 (i.e., merging s_1 sorted runs)
	$RF_{is}(s_2)$	initial run formation cost for s_2 (i.e., creating initial s_2 sorted runs)
	$RM_{is}(s_2)$	run merge cost for s_2 (i.e., merging s_2 sorted runs)

improvement, the average percentage improvement is around 25% and the highest improvement is 35% achieved by CS over IS for the query on *catalog_sales* with $M = 30$ and $SF = 40$.

Second, although operating on the same set of initial runs, the run merge phase of s_{12} incurs a higher CPU cost than that of $s_1$due to the additional tuple comparison steps. Note that $RM_{cs}(s_{12})$ does not include the internal sorting cost $SC_{cs}(s_{12})$. However, for all six queries, $RM_{cs}(s_{12})$ is close to or even less than $RM_{is}(s_1)$. This observation validates the I/O effectiveness and efficiency of our *batched reading* strategy.

Third, for all six queries, $RF_{cs}(s_{12})$, $RF_{is}(s_1)$ and $RF_{is}(s_2)$ are always more or less the same with any amount of sorting memory. This is due to the fact that during the initial run formation phase, the reading and writing of tuples to the disk files are interleaved and the cost of incurred random I/O is independent of the size of the sorting memory. On the other hand, $RM_{cs}(s_{12})$, $RM_{cs}(s_2)$, $RM_{is}(s_1)$, and $RM_{is}(s_2)$ all decrease when the sorting memory increases, as the larger sorting memory makes the run merging more I/O-efficient.

Finally, for all 6 tables, $SC_{cs}(s_{12})$ and $SC_{cs}(s_2)$ increase along with the size of sorting memory. The reason is two-fold: on the one hand, the larger sorting memory means that more tuples will be combined into composite chunks/chunklets and more tuples need to be internally sorted; on the other hand, with the fixed total number of tuples, it is cheaper to independently sort many smaller composite chunks/chunklets than independently sort fewer larger composite chunks/chunklets, which is similar to the analysis of case 2 in Section III.

2) Effect of Result Sharing: As discussed at the beginning of Section IV-A, the result sharing technique (denoted by RS) can actually be applied to evaluate case 3. In this section, we compare the effectiveness of RS against CS for the six queries. Fig. 7 compares the performance of the query on web_sales with SF 40; the comparison for other quries have similar trends and are omitted.

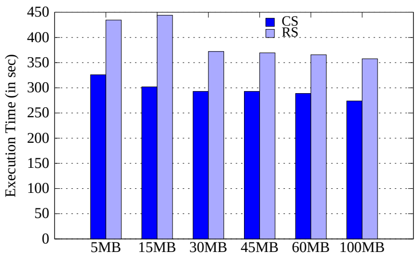


Fig. 7. Comparison of CS with RS on web_sales, SF 40

The results clearly demonstrate that CS significantly outperforms RS in all sorting memory settings. The performance of RS is just a little better than IS (see Fig. 6(a)).

B. Synthetic Data

We also utilize synthetic data to investigate the sensitivity of CS. We generated synthetic tables for the *web_sales* relation in TPC-DS benchmark using $SF = 40$; each table has 28.8 million tuples. We run template query Q defined in the previous section on the synthetic tables to compare the performance of CS and IS.

1) Varying Total Number of s_{12} Chunks: Under CS, there will be n initial runs for s_2 if n chunks are formed by s_{12}. The purpose of this experiment is to learn how the total number of s_{12} chunks will affect the run merge cost for s_2. We vary the number n of distinct `ws_item_sk` (the o_{11}) values inside a *web_sales* table. Six values of n are used: 15, 25, 50, 100, 150 and 200. A uniform distribution is used for the values of `ws_item_sk`. We fix the sort memory to 20MB, so that even when n is 200 the tuples with the same `ws_item_sk` value cannot fit in memory and thus will form a natural chunk. As a result, there will be totally n natural chunks.

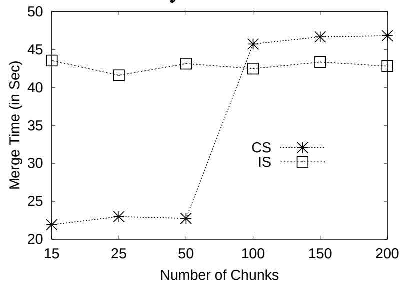


Fig. 8. Varying Total Number of s_{12} Chunks

The experimental result is shown in Fig. 8. The y-axis denotes the run merge time for s_2. With 20MB sort memory, the merge order F is 73. Moreover, the number of initial runs to merge for s_2 under IS is 56. Therefore, with all the different n values, the number of merge passes for IS on s_2 is always 1 and the merge costs are more or less the same. As for CS, the merge cost increases significantly when n becomes larger than 73. This is because the number of merge passes changes from 1 to 2. This confirms the expectation that the merge costs of CS remains largely unchanged with varying n as long as the number of merge passes required stays the same. We also

TABLE IV

COMPONENTAL COSTS OF SORTS IN MICRO-BENCHMARK TEST (IN SECONDS)

		CS					IS			
	Memory	$RF_{cs}(s_{12})$	$RM_{cs}(s_{12})$	$RM_{cs}(s_2)$	$SC_{cs}(s_{12})$	$SC_{cs}(s_1)$	$RF_{is}(s_1)$	$RM_{is}(s_1)$	$RF_{is}(s_2)$	$RM_{is}(s_2)$
web_sales SF 40	5MB	129.25	70.29	59.15	3.45	22.98	127.39	70.90	128.71	54.43
	15MB	126.62	69.47	32.57	8.30	23.57	126.36	75.54	125.70	71.79
	30MB	129.62	58.64	28.12	11.47	23.80	126.52	60.05	126.24	53.60
	45MB	130.18	53.87	27.46	15.45	24.51	129.92	55.22	125.84	53.24
	60MB	126.27	47.89	28.81	18.41	24.85	126.23	50.96	129.36	47.61
	100MB	125.64	34.90	24.52	22.11	25.32	125.93	49.26	129.59	46.88
catalog_sales SF 40	5MB	256.60	221.96	230.49	7.58	45.38	259.03	219.82	255.64	192.93
	15MB	260.75	229.75	91.48	16.65	46.29	263.36	188.90	254.87	164.14
	30MB	254.66	121.97	58.62	20.65	47.19	257.42	155.15	260.35	136.16
	45MB	258.27	149.05	55.25	25.48	47.21	260.98	150.07	258.29	132.78
	60MB	255.65	132.31	54.76	32.59	47.71	258.62	137.59	261.16	118.33
	100MB	262.61	118.78	51.89	40.62	48.43	261.75	126.01	269.65	106.86
store_sales SF 40	15MB	352.36	934.28	244.45	19.21	70.28	410.03	539.52	399.86	492.84
	30MB	377.94	385.95	236.96	28.94	72.11	392.31	399.09	370.46	381.49
	45MB	362.83	195.38	224.75	39.27	72.91	351.04	277.77	358.31	259.73
	60MB	384.26	291.87	102.73	49.96	73.91	354.45	279.03	384.18	242.23
	75MB	377.61	243.56	93.21	64.51	75.17	380.68	256.74	360.36	217.99
	100MB	393.62	263.99	99.12	67.59	76.32	385.29	270.82	375.42	232.20

notice that with the same number of merge passes, the merge cost of CS is always lower than that of IS, which is consistent with the observation in the micro-benchmark test.

2) Varying Number of Composite s_{12} Chunks: In this experiment, we examine the contributions of internal sorting cost to the total CS cost. These internal sorts are applied to composite chunklets and chunks. We fix the total number m of chunks generated and vary the number n of composite chunks. We set the sort memory to 50 MB and m to 55. Five values of n are used: 0, 13, 27, 42 and 55.

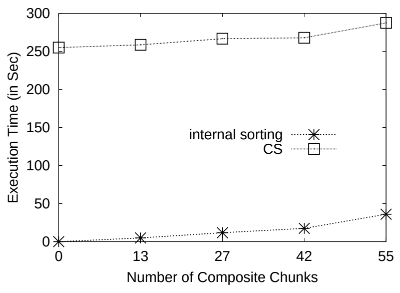


Fig. 9. Varying Number of Composite s_{12} Chunks

Fig. 9 shows the internal sorting cost as well as the overall CS cost. As expected, the internal sorting cost increases along with the number of composite chunks. When all the 55 chunks become composite, this cost takes a non-trivial percentage of the total CS cost.

C. Synthetic Database and Queries

So far, we have evaluated cooperative sort for the basic scenario of processing 2 sort operations on different orders. In this section, we evaluate the effectiveness of cooperative sort on queries. We generate a synthetic database with three relations `Employee`*(id, name, country_id, supervisor_id)*, `Sales`*(employee_id, item_id, quantity, profit)* and `Item`*(id, name)*. `Employee` records the information of employed salesperson and has 10 million 32-byte tuples; `Sales` records the transactions and has 50 million 12-byte tuples; `Item` records the products on sale and has 10 million 24-byte tuples.

We define two queries on this database:

Q1: Find the name of each salesperson and its supervisor.

Q2: Find each salesperson that has sold more than 1000 units of a product in a single transaction or his supervisor has done so.

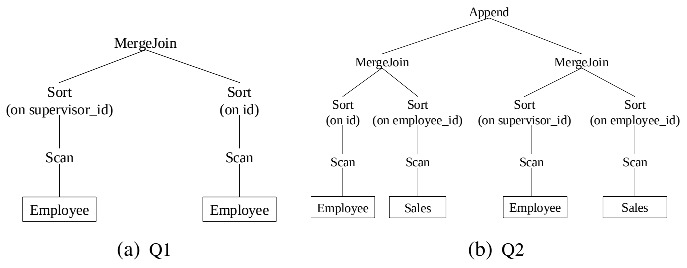


Fig. 10. The Optimal Plans for Q1 and Q2

With 50MB sort memory, the optimal plans generated by PostgreSQL for these two queries are shown in Fig. 10.

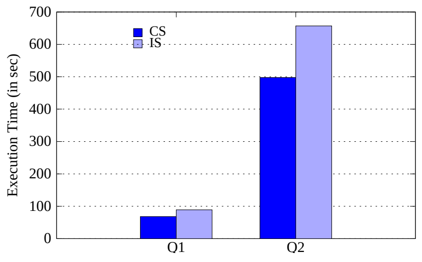


Fig. 11. Query Execution Times of Q1 and Q2

We run Q1 and Q2 with both CS and IS. For Q2, one redundant sort on *sales* is also skipped by the post-optimizer, which contributes about 90 seconds' saving. The overall query execution times are shown in Fig. 11. The results clearly show that both queries can be processed in lesser time under CS.

D. Impact on Optimizer Search Space

In this experiment, we study the potential benefit of enriching the optimizer search space with cooperative sort. We make use of the synthetic database and Q1 from Section VII-C. In PostgreSQL, each sorting and hashing operation has a dedicated operator memory. We vary this operator memory and compare various execution plans for Q1: Hybrid Hash Join (HHJ), Sort Merge Join (SMJ) and Sort Merge join with Cooperative Sort (SMJ-CS). As shown in Fig. 12, the original optimizer of PostgreSQL generates either SMJ or HHJ as the optimal plan for Q1. We can find that with 10MB operator memory, SMJ-CS beats the optimal HHJ, which indicates that a cooperative-sort-aware query optimizer is promising.

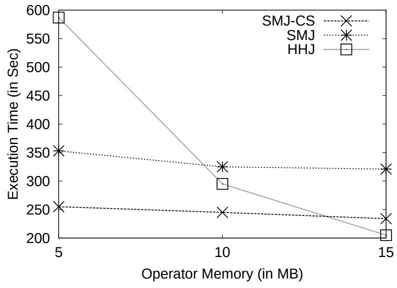


Fig. 12. Optimization Potential of CS

VIII. Related Work

Sorting is one of the most extensively studied problems in computing. Knuth's classical text [1] provides extensive coverage of the fundamentals of sorting, including both replacement selection for run formation and run merge patterns.

Larson [17] introduced a cache-aware replacement selection that works for various length keys. There are also lots of techniques to speed up the run merge phase [4], [5], [6], focusing on how to improve I/O performance during the merge phase because this phase is typically I/O bound. These techniques are however complementary to our *batched reading* strategy, which relies more on the pre-collected knowledge about input data distribution. Our current implementation only applies simple forecasting technique to the type-3 reads. But it is possible to incorporate other optimization techniques like double buffering [1], read-ahead [5], etc. Much research has been done on adaptive sorting [18] exploiting near-sortedness. Graefe's survey [19] discussed how sorting is implemented in database systems with many tricks and optimizations.

Simmen et.al [20] described how to determine the ordering propagation from the inputs to the outputs of joins, based on functional dependencies and selection conditions. Their work was followed and extended by [21], [22], [23], which are all independent and complementary to our work.

In [8], Sudarshan et. al observed that the order requirements of operators are often partially satisfied by the inputs. They proposed to maximize the benefit of such partial sort order by modifying the standard replacement selection algorithm and improving the selection of interesting orders. We instead consider the opportunity of partial sorting sharing between two distinct sort operations (case 2 in Section III). A similar idea to partial sorting was considered previously in [9] for the CUBE operator, which computes group-bys corresponding to all possible combinations of a list of attributes. Consider two group-bys $B = \{a_1, a_2, \ldots, a_j\}$ and $S = \{a_1, a_2, \ldots, a_{l-1}, a_{l+1}, \ldots, a_j\}$. With sort-based aggregation, the result of B can be viewed as a concatenation of one or more *partitions* and the result of S is the union of independently computing aggregation within each partition.

Finally, there have been a few previous works on optimizing multiple scans on the same table, such as `MAPLE` [7] and *cooperative scan* [24], etc.

IX. Conclusion

In this paper, we have examined the problem of sorting a relational table on multiple sort orders. Such collections of sortings are common in many applications. We have identified several cases in which the (partial) work done in sorting a table on a particular order can be re-used for a subsequent sort of the same table on a different order. We proposed the cooperative sort technique to efficiently handle sorting of a table on two orders. We also proposed a post-optimizer to exploit cooperative sort in a traditional query evaluation plan. We have implemented our techniques in PostgreSQL, and our extensive performance study indicated a performance gain of upto 35% over the naive strategy of processing each sort independently.

Acknowledgements This research is supported in part by NUS Grant R-252-000-271-112.

References

[1] D. E. Knuth, *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. Addison-Wesley, 1998.
[2] V. Pai and P. Varman, "Prefetching with multiple disks for external mergesort: simulation and analysis," in *ICDE*, 1992.
[3] B. Salzberg, "Merging sorted runs using large main memory," *Acta Informatica*, vol. 27, no. 3, 1989.
[4] W. Zhang and P. Larson, "Dynamic memory adjustment for external mergesort," in *VLDB*, 1997.
[5] W. Zhang and P.-A. Larson, "Buffering and read-ahead strategies for external mergesort," in *VLDB*, 1998.
[6] L. Zheng and P. Larson, "Speeding up external mergesort," *IEEE Transactions on Knowledge and Data Engineering*, vol. 8, no. 2, 1996.
[7] Y. Cao, G. C. Das, C.-Y. Chan, and K.-L. Tan, "Optimizing complex queries with multiple relation instances," in *SIGMOD*, 2008.
[8] R. Guravannavar and S. Sudarshan, "Reducing order enforcement cost in complex query plans," in *ICDE*, 2007.
[9] S. Agarwal, R. Agrawal, P. Deshpande, A. Gupta, J. F. Naughton, R. Ramakrishnan, and S. Sarawagi, "On the computation of multidimensional aggregates," in *VLDB*, 1996.
[10] Y. Cao, R. Bramandia, C.-Y. Chan, and K.-L. Tan, "Optimized query evaluation using cooperative sorts, technical report," in *www.comp.nus.edu.sg/∼caoyu/icde-TR.pdf.*
[11] R. M. Karp, "Reducibility among combinatorial problems," in *Complexity of Computer Computations*. Plenum Press, 1972.
[12] L. Georgiadis, "Arborescence optimization problems solvable by edmonds' algorithm," *Theor. Comput. Sci.*, vol. 301, no. 1-3, 2003.
[13] M.-I. Hsieh, E. H.-K. Wu, and M.-F. Tsai, "Fasterdsp: A faster approximation algorithm for directed steiner tree problem," *J. Inf. Sci. Eng.*, vol. 22, no. 6, 2006.
[14] M. Charikar, C. Chekuri, Z. Dai, A. Goel, S. Guha, and M. Li, "Approximation algorithms for directed steiner problems," in *Journal of Algorithms*, 1999.
[15] "Postgresql Offical Website," http://www.postgresql.org/.
[16] "TPC BENCHMARK Decision Support," http://www.tpc.org/tpcds/.
[17] P.-A. Larson, "External sorting: Run formation revisited," *IEEE Transactions on Knowledge and Data Engineering*, vol. 15, no. 4, 2003.
[18] V. Estivill-Castro and D. Wood, "A survey of adaptive sorting algorithms," *ACM Computing Surveys (CSUR)*, vol. 24, no. 4, 1992.
[19] G. Graefe, "Implementing sorting in database systems," *ACM Comput. Surv.*, vol. 38, no. 3, 2006.
[20] D. Simmen, E. Shekita, and T. Malkemus, "Fundamental techniques for order optimization," in *SIGMOD*, 1996.
[21] T. Neumann and G. Moerkotte, "A combined framework for grouping and order optimization," in *VLDB*, 2004.
[22] X. Wang and M. Cherniack, "Avoiding sorting and grouping in processing queries," in *VLDB*, 2003.
[23] T. Neumann and G. Moerkotte, "An efficient framework for order optimization," in *ICDE*, 2004.
[24] M. Zukowski, S. Héman, N. Nes, and P. A. Boncz, "Cooperative scans: Dynamic bandwidth sharing in a dbms," in *VLDB*, 2007.