Noname manuscript No.
(will be inserted by the editor)

Sort-Sharing-Aware Query Processing

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

Received: date / Accepted: date

Abstract Many database applications require sorting a table (or relation) over multiple sort orders. Some examples include creation of multiple indices on a relation, 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 to optimize multiple sortings of a table. We investigate the correlation between sort orders and exploit sort sharing techniques of reusing the (partial) work done to sort a table on a particular order for another order. Specifically, we introduce a novel and powerful evaluation technique, called cooperative sorting, that enables sort sharing between seemingly non-related sort orders. Subsequently, given a specific set of sort orders, we determine the best combination of various sort sharing techniques so as to minimize the total processing cost. We also develop techniques to make a traditional query optimizer extensible so that it will not miss the truly cheapest execution plan with the sort sharing (post-) optimization turned on. We demonstrate the efficiency of our ideas with a prototype implementation in PostgreSQL and evaluate the performance using both TPC-DS benchmark and synthetic data. Our experimental results show significant performance improvement over the traditional evaluation scheme.

Yu Cao
Department of Computer Science
School of Computing
National University of Singapore
E-mail: caoyu@comp.nus.edu.sg

Ramadhana Bramandia
Department of Computer Science
School of Computing
National University of Singapore
E-mail: bramandia@comp.nus.edu.sg

Chee-Yong Chan
Department of Computer Science
School of Computing
National University of Singapore
E-mail: chancy@comp.nus.edu.sg

Kian-Lee Tan
Department of Computer Science
School of Computing
National University of Singapore
E-mail: tankl@comp.nus.edu.sg

1 Introduction

Sorting is a frequent 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 ([17, 22, 25, 31–33]). The standard technique adopted in most commercial systems is based on the external merge-sort algorithm that consists of two phases: an initial run formation phase that creates sorted subsets, called runs, and a merge phase that repeatedly merge runs into larger and larger runs, until a single run has been created.

In this paper, we investigate the problem of efficiently sorting a table on multiple sort orders. It turns out that such multiple sortings of a table is not uncommon in many applications. For example, in data warehousing, a fact table typically has two types of attributes: those that contain facts and those that are foreign keys pointing to dimension tables. According to the workload, the index selection program may recommend to create both the primary key index and foreign key indices on the fact table, which requires the table to be sorted multiple times to bulk load the various indices. In many organizations, many reports are generated at the end of the day/week/month. Typically, these reports contain the same content but on 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 on different orders - such as course number, dates, examiners, and invigilators. Yet another example arises in decision making applications where a complex query typically contains multiple instances of a relation or view [4], and the execution plan may introduce sortings on the instances.

In the above examples, the table could be separately sorted multiple times, once per sort order. However, intuitively this is wasteful of resources (mainly I/O cost) especially when the table is huge, as after all we are manipulating the same set of tuples. On the contrary, it seems promising to execute these sortings in a more collaborative manner so as to reduce the overall processing cost, by somehow salvaging the (partial) efforts spent on sorting the table on a particular order to speed up the sortings on other orders. Such *sort sharing* is exactly what we set out to achieve in this paper.

We begin by considering sorting a table T on two sort orders o_1 and o_2, both of which are sequences of some attributes of T. When o_1 and o_2 share a common prefix, it is obvious that, once T has been sorted on o_1, the sorting output can be either re-used directly (if one order is a prefix of the other) or be re-organized in a light-weight way (if neither order is a prefix of the other) in order to derive the sorted T on o_2. We refer to such kind of optimizations as *result sharing*, which leverages the output of one sorting to more efficiently evaluate the other sorting. The result sharing technique has been well recognized [3, 14].

However, when o_1 and o_2 do not share a common prefix, the potential sort sharing opportunities have not been explored previously. In this paper, we introduce a new property between a pair of sort orders called *subset-prefix* and design a novel sorting technique called *cooperative sorting* that can be applied to optimize two sort operations if their sort orders satisfy the subset-prefix property. Cooperative sorting first organizes the tuples of T into an intermediate form T' such that subsequently (a) T' can be used to produce the sorted T on o_1 efficiently with only (possibly) in-memory sorting; (b) T' can also be viewed as a set of initial sorted runs on o_2, which can be efficiently merged to derive the sorted T on o_2. In so doing, cooperative sorting saves the initial run formation phase for o_2. Furthermore, for the general case of two seemingly non-related sort orders, we show that the pair of sort operations could still be optimized by first applying cooperative sorting on a derived pair of sort orders followed possibly by using result-sharing optimization to achieve the desired sortings. Consequentially, when sorting a table on an arbitrary pair of sort orders, we can always optimize the evaluation by utilizing result sharing and/or cooperative sorting.

With the result sharing and cooperative sorting techniques, we then tackle the optimization problem of evaluating more than two sortings on the table. We model this problem as the minimum directed Steiner tree problem, which unfortunately is NP-Hard. When the number of sortings is manageable, we will adopt a brute force algorithm to find the optimal solution on how each sorting should be sequenced and accomplished. Otherwise, we will resort to heuristic or approximation algorithms.

So far, we have implicitly assumed that the sortings on the table are the optimization decision of a conventional query optimizer which is unaware of sort sharing optimization. Further modifications of query plans generated by such a sort-sharing-blind optimizer, such as replacing a hash join with a sort-merge join and replacing a hash-based aggregation with a sort-based aggregation, may enable additional sort-sharing opportunities and thereby lead to a lower query execution cost. Therefore, it would be beneficial to let sort sharing be explicitly considered during query optimization. As a result, we propose solutions for the standard query optimizer to directly generate optimal sort-sharing-aware query plans. Our techniques are generally applicable to different types of query optimizer, such as the System-R style and the Volcano style.

We have performed a comprehensive experimental evaluation of our proposed techniques with an implementation in PostgreSQL. We ran a micro-benchmark test, on both TPC-DS dataset and our own synthetic dataset, to compare the performance of cooperative sorting against two independent sort operations. The performance results showed that cooperative sorting improved the performance on average by 25% and up to 35%. We also conducted a case study of *cooperative index building*, where the standard cooperating sorting technique is slightly extended and then exploited when creating multiple indices on a single table. The corresponding performance study on TPC-DS dataset illustrated that cooperative sorting is very helpful. The highest and the average performance improvement were 37% and 24% respectively. Finally, we studied the overall benefits of sort sharing techniques and the enhanced sort-sharing-aware query optimizer when executing normal queries.

The rest of this paper is organized as follows. In Section 2, we present some preliminaries. In Section 3, we introduce our new sort order property, subset-prefix property, and categorize the relationship between two sort orders into four cases. These four cases can be optimized by applying the existing result-sharing sorting technique and/or our new cooperative sorting technique. We elaborate on cooperative sorting in Section 4. In Section 5, we generalize cooperative sorting to evaluate more than two sort operations, explain how to optimize the evaluation of multiple sortings on a table, and discuss sort-sharing-aware query optimization. Further general discussions about sort sharing are presented in Section 6. Our experimental study presented in Section 7 validates the effectiveness of our proposed techniques. We discuss relevant work in Section 8 and finally conclude in Section 9.

2 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 [14]:

- $s_i = sort(T, o_i)$: 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 the attributes that also appear in the set of attributes A.
- *o-segment*[2]: 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

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

3 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 setting involving only two sort operations, and then explain how our techniques can be easily extended to the general setting in Section 5. For simplicity, we assume 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 sort orders in Section 6.

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 pair of sort operations can be optimized for two well-known cases. The first case is when o_2 is a prefix of o_1 (i.e. $o_2 \leq o_2$), and the second case is when o_1 and o_2 share a non-empty common prefix which is a proper prefix of o_2 (i.e. $0 < |o_1 \wedge o_2| < |o_2|$).

In this paper, we introduce a new property between two sort orders termed *subset-prefix* that forms the basis of our novel cooperative sorting technique. Given two sort orders o_1 and o_2, o_2 is defined to be a *subset-prefix* of o_1 if they satisfy two conditions:

1. some prefix o_{21} of $o_2 = o_{21} + o_{22}$ is the substring (but not prefix) o_{12} of $o_1 = o_{11} + o_{12} + o_{13}$, and
2. the set of attributes in the suffix o_{22} of o_2 is a subset of the attributes in the prefix o_{11} of o_1; i.e., $o_{12} = o_{21}$, $attrs(o_{22}) \subseteq attrs(o_{11})$, $|o_{11}| > 0$, $|o_{13}| \geq 0$ and $|o_{22}| \geq 0$.

As the name of the property suggests, if o_2 is a subset-prefix of o_1, then the set of attributes in o_2 is a subset of the set of attributes in a prefix of o_1.

Example 1 Consider the following four sort orders: $o_1 = (a_1, a_2)$, $o_2 = (a_2)$, $o_3 = (a_2, a_3, a_4, a_5)$, and $o_4 = (a_4, a_3, a_2)$. We have three pairs of sort orders that satisfy the subset-prefx property: o_2 is a subset-prefix of o_1, o_2 is a subset-prefix of o_4, and o_4 is a subset-prefix of o_3. □

Based on the new subset-prefix property, we can classify the relationship between o_1 and o_2 into four disjoint cases:

- Case 1: o_2 is a prefix of o_1.
- Case 2: o_1 and o_2 share a non-empty common prefix which is a proper prefix of o_2.
- Case 3: o_2 is a subset-prefix of o_1.
- Case 4: o_1 and o_2 do not satisfy any of the above three cases.

The first two cases are the more familiar and simpler cases, where s_1 and s_2 can be efficiently evaluated using the **result sharing technique**, which has been previously discussed in other contexts [3, 14]. The idea is to leverage the output of one sort operation to more efficiently evaluate the other sort operation.

For case 1, 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

[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.

[2] It is also known as *value packet* [18].

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 [14], 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. s_1 is evaluated using the conventional external merge-sort and $cost(s_1)$ is given by the Equation 1. Following [14], $cost(s_2) = \sum_{i=1}^{D(T,o')} cost(sort(se_i, o'_2))$, 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 $B(se_i) = B(T)/D(T,o')$.

The cases 3 and 4 are the new scenarios that we investigate in this paper. For case 3, the evaluations of s_1 and s_2 can be optimized by our newly proposed **cooperative sorting technique**, whose idea is to create “hybrid” sorted runs that can benefit the evaluation of both sort operations. We shall discuss the details of cooperative sorting in the next section.

For the most general case 4, s_1 and s_2 can be optimized as follows. First, we derive two new sort orders o'_1 and o'_2, where o'_2 is the longest prefix of o_2 such that o'_2 is a subset-prefix of $o'_1 = o_1 + (o'_2 - attrs(o_1))$. Note that the derivation of o'_1 and o'_2 is always possible; in particular, the trivial o'_2 containing only the first attribute of o_2 is a subset-prefix of the corresponding o'_1. Second, we apply cooperative sorting to evaluate two sort operations $sort(T, o'_1)$ and $sort(T, o'_2)$. Since o_1 is a prefix of o'_1, the output of $sort(T, o'_1)$ is also sorted on o_1 and thus can be directly utilized as the output of s_1. Since o'_2 is a prefix of o_2, there are two cases to be considered for the evaluation of $sort(T, o_2)$: if $o'_2 = o_2$, then the output of $sort(T, o'_2)$ can be directly utilized as the output of s_2; otherwise, we can derive the output of s_2 by independently sorting each o'_2-*segment* within the output of $sort(T, o'_2)$ on order $o_2 - attrs(o'_2)$. In order to optimize the independent sorting of the o'_2-*segments*, we choose o'_2 to be the longest prefix of o_2 that meets the subset-prefix requirement.

4 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_2 is a subset-prefix of o_1 (i.e. case 3) as defined in the previous section.

Recall that in this case, 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 $|o_{22}| \geq 0$.

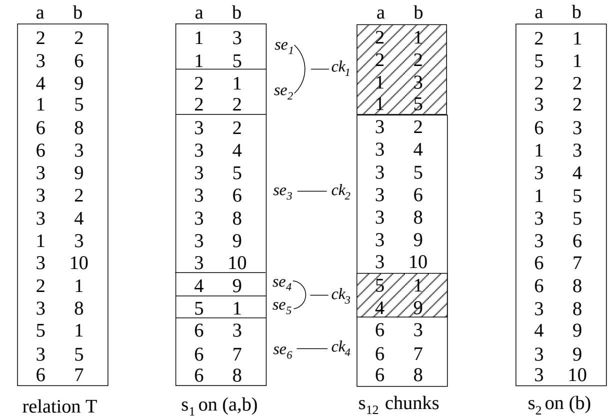
4.1 Overview

Observe that the output of s_1 can be viewed as the concatenation of o_{11}-*segments* (i.e., a set of tuples with identical o_{11} values), each of which is also sorted on o_2 and thus is a sorted run for s_2. As a result, the result sharing technique can actually be applied to this case by first evaluating s_1 followed by merging the resultant o_{11}-*segments* to compute s_2. However, depending on the number of distinct o_{11} values and the extent of data skew in T, 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 this drawback of applying the result sharing technique for case 3.

Example 2 Consider the relation $T(a,b)$ in Fig. 1, which will serve 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_2 is a subset-prefix of o_1 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). These six *a-segments* can be merged for s_2 with three I/O passes of reading and writing T tuples. However, this is actually not better than a conventional external sorting: the replacement selection incurs one I/O pass and generates three initial runs, which can be merged with only two I/O passes. As a result, both approaches for evaluating s_2 will incur three I/O passes. □

Cooperative sorting is proposed in order to retain the benefit of result sharing, i.e. avoiding scanning T to generate initial sorted runs for s_2, and also overcome as much as possible the drawback of result sharing. The core of cooperative sorting is an intermediate sort operation s_{12} based on a special hybrid sort order, such that the outputs of both s_1 and s_2 can be efficiently derived from the output of s_{12}.

We will discuss how to perform the intermediate sort operation s_{12} in Sections 4.2 and 4.3. The output of s_{12} will be a sequence of *tuple chunks* which are either *natural* or *composite*. Tuples of a natural chunk are ordered by o_1, while tuples of a composite chunk are ordered by o_2. For each composite chunk, it consists of tuples from two or more *consecutive* o_{11}-*segments* in the output of s_1, and its size is no larger than the sorting memory (i.e. M blocks). For each natural



relation T		s_1 on (a,b)		s_{12} chunks		s_2 on (b)	
a	b	a	b	a	b	a	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 $F = 2$

chunk, it consists of tuples from exactly one o_{11}-*segment* in the output of s_1, and there is no constraint on its size. Moreover, the tuple chunks are o_1-*order preserving*, which means that if a chunk ck_i precedes another chunk ck_j in the output of s_{12}, then every tuple in ck_i has an o_1 value smaller than that of every tuple in ck_j.

Example 3 Look at the running example in Fig. 1. The output of s_{12} contains four tuple chunks, two composite (ck_1 and ck_3 shown shaded) and two natural (ck_2 and ck_4 shown non-shaded). The output of s_1 contains six *a-segments*, se_1 to se_6. In the output of s_{12}, se_1 and se_2 are combined into ck_1, se_3 is exactly ck_2, se_4 and se_5 are combined into ck_3, and se_6 is exactly ck_4. Both ck_1 and ck_3 are no larger than $M = 4$ blocks, while ck_2 is larger than M and ck_4 is smaller than M. □

To derive the output of 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. Since the chunks are o_1-order preserving, the whole resultant tuple stream will be ordered by o_1.

Notice that the tuples in each natural s_{12} chunk are also ordered by o_2. Therefore, to derive the output of s_2, all the s_{12} chunks can be treated as initial sorted runs on o_2 and merged recursively.

Compared with result sharing, cooperative sorting generates longer and thus fewer initial sorted runs for s_2 to merge. Although the evaluation cost of s_{12} is slightly more expensive than the normal cost of s_1 and deriving the output of s_1 from the output of s_{12} requires additional internal sorting cost, the saving on run merge cost for s_2 makes cooperative sorting competitive. As indicated by both the cost model in Section 4.4 and the experimental results in Section 7, cooperative sorting is at least as good as and often better than result sharing.

However, the number of s_{12} chunks generated in cooperative sorting could still be more than the number of initial sorted runs generated by a conventional initial run formation phase for s_2, and thus cooperative sorting may incur a more costly run merging phase for s_2. As a result, cooperative sorting is not guaranteed to be always superior to evaluating s_1 and s_2 independently. Both cooperative sorting and conventional sorting should be considered in a cost-based manner by the query optimizer for evaluating multiple sorts on a relation.

4.2 Intermediate Sort Operation s_{12}

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) with the conventional initial run formation technique. We also collect the set of distinct o_{11} values, and count the number of tuples corresponding to each distinct value, in each initial s_1 run at runtime when it is being generated. After all initial s_1 runs have been generated, we combine statistics for each initial s_1 run to acquire the global statistics on the distinct o_{11} values in T. 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.

We allocate a very small portion of memory for the purpose of the above statistics collection, and flush the memory content to disk files when necessary (e.g., the statistics for one initial s_1 run will be written to disk before the generation of the next run starts). The global statistics will be computed from the disk files, which are also very small and thus incur negligible I/O cost.

The above accurate statistics collection procedure works well when the domain of o_{11} values is not large. As we shall see, in our experimental study, with 0.5MB of memory, the scheme performs well for 50k distinct o_{11} values. Alternatively, we can estimate the statistics using approximation techniques such as [5,11]. In this case, the subsequent three steps of computing s_{12} (to be described shortly) need to be modified to handle estimated o_{11} statistics. This extension is straightforward, and does not affect the correctness of our proposed scheme. However, some composite chunks might have to be externally sorted due to an underestimation of their sizes.

In the second step, we determine the output information 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. Intuitively, the composite s_{12} chunks should be as large as possible (within the size constraint), so as to minimize the total number of s_{12} chunks. We thereby apply a *greedy algorithm* that utilizes the statistics collected from the first step and sequentially checks the o_{11}-*segments* as follows. If the size of an o_{11}-*segment* se_i exceeds M, then se_i forms a natural chunk; otherwise, determine the longest sequence of consecutive o_{11}-*segments* $se_i, se_{i+1}, \cdots se_j$ such that their total size is no more than M. If $i = j$, then se_i forms a natural chunk; otherwise, $se_i, \cdots, se_j$ form a composite chunk. Repeat the above procedure from se_{j+1} unless se_j is the last o_{11}-*segment*.

Note that the tuples belonging to a s_{12} chunk are generally distributed across multiple initial s_1 runs. Since the s_{12} chunks are o_1-order preserving, each initial s_1 run consists of a sequence of *tuple chunklets*, each of which represents a subset of tuples of a distinct s_{12} chunk. Chunklets are also correspondingly classified as natural and composite.

Example 4 Fig. 2 illustrates the two initial s_1 runs ordered by $o_1 = (a, b)$ and generated from the relation T in Fig. 1. Based on the sizes of *a-segments*, the above greedy algorithm decides to form four s_{12} chunks. The first initial s_1 run consists of chunklets $ckl_{1,1}$, $ckl_{2,1}$, $ckl_{3,1}$, and $ckl_{4,1}$; the second initial s_1 run consists of chunklets $ckl_{1,2}$, $ckl_{2,2}$, $ckl_{3,2}$, and $ckl_{4,2}$. Here $ckl_{i,j}$ denotes the chunklet in the j^{th} initial s_1 run that corresponds to the i^{th} s_{12} chunk. □

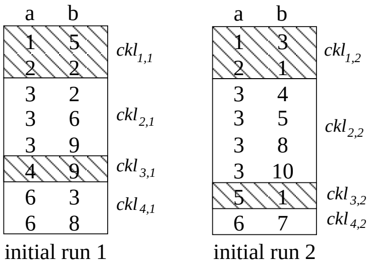


Fig. 2 Initial s_1 Runs for Relation T in Example of Fig. 1

In the third step, we merge the initial s_1 runs to generate the initial s_{12} runs. Each initial s_{12} run is created by merging a set of F initial s_1 runs. Specifically, the chunklets in the F initial s_1 runs that correspond to the same s_{12} chunk are merged to form a longer chunklet in the initial s_{12} run. Consequentially, each initial s_{12} run is also a sequence of chunklets, where tuples of a natural chunklet are ordered by o_1, tuples of a composite chunklet are ordered by o_2 and the chunklets are o_1-order preserving. This merging operation is different from the conventional run merging procedure and will be elaborated in Section 4.3.

Example 5 When merging the two initial s_1 runs in Fig. 2, each pair of chunklets $ckl_{i,1}$ and $ckl_{i,2}$ ($i \in \{1, 2, 3, 4\}$) are merged respectively. The resultant initial s_{12} run is exactly the final s_{12} chunks as shown in Fig. 1. □

In the fourth step, the initial s_{12} runs are recursively merged to generate the s_{12} chunks. This is done by the conventional external run merging technique with a minor extension for the tuple comparison operator. 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. Note that the fourth step is skipped if the third step produces only one initial s_{12} run as in the above example.

4.3 Generating Initial s_{12} Runs

In this section, we elaborate on the procedure of merging F initial s_1 runs into an initial s_{12} run.

Merging a set of natural chunklets in the initial s_1 runs is simple and just follows the conventional external merge procedure, since the input and output orders are the same.

However, as mentioned in Section 4.2, for each composite chunklet in the generated initial s_{12} run, its tuples will be ordered by o_2, while the set of composite chunklets in the initial s_1 runs are all sorted on o_1. Therefore, before we can merge these composite chunklets in the initial s_1 runs, we need to internally sort each of them on o_2[3], which requires that our tuple reading strategy, i.e. the way we read tuples from different initial s_1 runs into the sorting memory during run merging, should ensure that these composite chunklets will be able to co-exist in the sorting memory when it is their turn to be merged. Assume that these composite chunklets correspond to the i^{th} s_{12} chunk. When tuples in these composite chunklets are being read into the sorting memory, the following constraint must always be satisfied until these composite chunklets are completely read:

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

where RP_i^m (resp. RP_i^d) denotes the set of tuples in the input initial s_1 runs that belong to the i^{th} s_{12} chunk and currently are in the sorting memory (resp. still on the disk). Equation 2 prevents too many tuples of s_{12} chunks after the i^{th} chunk from occupying the sorting memory space but not being merged, while some tuples of the i^{th} chunk are still remaining on the disk.

Tuple reading strategies violating the above Equation 2 can lead to "deadlock" situations. For example, consider the two initial s_1 runs shown in Fig. 2 and suppose that we are merging the two composite chunklets $ckl_{1,1}$ and $ckl_{1,2}$ for the first s_{12} chunk with $M = 4$. If we had read the first three tuples, (1,5), (2,2) and (3,2), of the first initial s_1 run into the sorting memory, then a deadlock situation would arise as the remaining memory space is not adequate for loading the two

[3] Internal sortings are feasible as by design the total size of these composite chunklets will not exceed the sorting memory M.

tuples of $ckl_{1,2}$, (1,3) and (2,1), in the second initial s_1 run for internal sorting.

On the other hand, a sound yet conservative tuple reading strategy might fragment the reading of the initial s_1 runs into too many short sequential I/O reads. For example, consider the following approach to merge F initial s_1 runs into an initial s_{12} run. The merging reads and processes the chunklets in the initial s_1 runs for one s_{12} chunk at a time based on the chunk order. If the current chunk being processed is composite, we first read all the chunklets that belong to this chunk into the sorting memory, perform an internal sorting on each chunklet, and then merge the sorted chunklets. If the current chunk being processed is natural, we first read n tuples from the corresponding chunklet in each initial s_1 run, where $n = \min\{\text{size of the chunklet, } \lfloor M/F \rfloor\}$, to initialize the merging. Each subsequent read includes at most $\lfloor M/F \rfloor$ sequential tuples of a chunklet in some initial s_1 run. By applying this approach to merge the two initial s_1 runs in Fig. 2 with $M = 4$, a total of 10 sequential reads is required, which is suboptimal: we shall later illustrate how this can be reduced to 8 sequential reads.

We thereby propose an efficient *batched tuple reading* strategy for loading tuples from the F initial s_1 runs into the sorting memory. Our strategy consists of two main steps. First, we partition each initial s_1 run into a sequence of tuple batches. An initial s_1 run containing n tuple batches will be read with n sequential reads, each of which reads a complete tuple batch. Second, we schedule the reading of tuple batches from different initial s_1 runs to do the tuple merging. Our goal is to minimize the total number of tuple batches (i.e., maximize the sequential I/O) without violating Inequation 2.

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

- In an initial s_1 run, a composite chunklet, or a natural chunklet that is no larger than $\lfloor M/F \rfloor$, will be completely included by a single tuple batch, where the total size of any natural chunklet along with all the following tuples must not exceed $\lfloor M/F \rfloor$.
- In an initial s_1 run, a natural chunklet that is larger than $\lfloor M/F \rfloor$ will be partitioned into a series of consecutive tuple batches, each of which, except the last one, has a size of $\lfloor M/F \rfloor$. The last tuple batch may contain tuples from other chunklets, but its size is at most $\lfloor M/F \rfloor$.

It follows that each tuple batch can be classified into one of four types based on its starting and ending points in the initial s_1 run:

1. the batch starts from the head of a composite/natural chunklet and ends at the tail of a (possible different) 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 both inside the same natural chunklet.
4. the batch starts inside a natural chunklet and ends at the tail of a (possibly different) composite/natural chunklet.

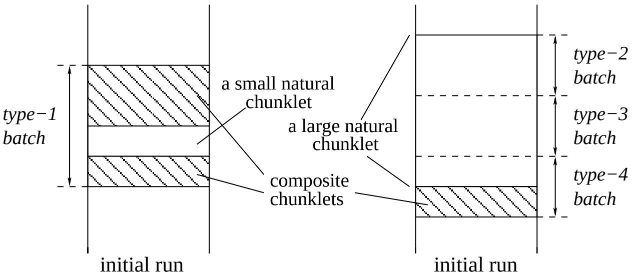


Fig. 3 Illustration of Four Types of Tuple Batches in Initial s_1 runs

Fig. 3 illustrates the four types of tuple batches. A natural chunklet that is larger than $\lfloor M/F \rfloor$ will be partitioned into one type-2 tuple batch, zero or several type-3 tuple batches, and one type-4 tuple batch. A composite chunklet, or a natural chunklet that is no larger than $\lfloor M/F \rfloor$, will be included by one type-1 or type-4 tuple batch. The size of a type-2 or type-3 tuple batch is exactly $\lfloor M/F \rfloor$. The size of a type-4 tuple batch is at most $\lfloor M/F \rfloor$. The size of a type-1 tuple batch could be larger than $\lfloor M/F \rfloor$ but is under constraint of the first rule above.

Given F initial s_1 runs, Algorithm 1 generates the complete set of tuple batches and records them in an array TB. Algorithm 1 essentially involves two nested computation loops. In the outer loop, each time it checks all the chunklets in the initial s_1 runs that belong to one s_{12} chunk, based on the chunk order; in the inner loop, it sequentially checks each chunklet of the current s_{12} chunk, and decides the specific tuple batch(es) that will include this chunklet. In TB, a type-4 tuple batch immediately follows the corresponding type-2 tuple batch, and the set of type-3 tuple batches in between are not recorded, as they can be easily deduced at runtime. The composition of each type-1 (or type-4) tuple batch starting from the head (or interior) of a chunklet is determined by using Algorithm 2, which tries to maximize the batch size by including as many tuples following this chunklet in the initial s_1 run as feasible.

At runtime of run merging, Algorithm 3 schedules the reading of tuple batches. For type-1 and type-2 tuple batches, they are read in the same order as in TB but are possibly interleaved with dynamically arranged type-3 and type-4 tuple batches. Moreover, when merging the chunklets for a natural chunk, a type-3 or type-4 tuple batch associated with a specific chunklet will be selected as the next one to read if and only if in the sorting memory tuples belonging to the same chunklet will be exhausted most quickly by the merging. This ensures the correctness of merging and is consistent with the run merging procedure in conventional external sorting.

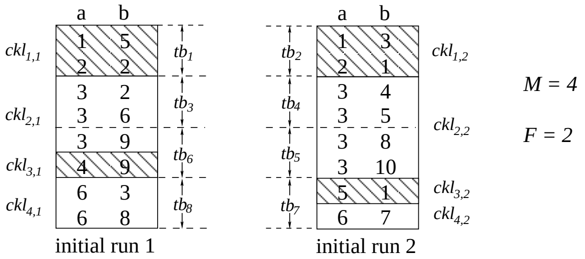


Fig. 4 Tuple Batches of the Two Initial s_1 Runs in Fig. 2

i	1	2	3	4	5	6	7	8
$TB[i]$	tb_1	tb_2	tb_3	tb_6	tb_4	tb_5	tb_7	tb_8

Table 1 The Entries in TB for Example in Fig. 4

Example 6 Fig. 4 shows the eight tuple batches (tb_1 to tb_8) comprising the two initial s_1 runs in Fig. 2. Table 1 shows the tuple batch array TB. tb_1, tb_2, tb_7 and tb_8 are type-1 tuple batches; tb_3 and tb_4 are type-2 tuple batches, and tb_6 and tb_5 are their corresponding type-4 tuple batches respectively. There are no type-3 tuple batches in this example. During run merging, these eight tuple batches will be read in the following sequence: $tb_1, tb_2, tb_3, tb_4, tb_5, tb_6, tb_7, tb_8$. Note that since the last tuple (3,5) in tb_4 is smaller than the last tuple (3,6) in tb_3, tuples in tb_4 will be exhausted first and thus tb_5 will be read before tb_6 at runtime. This situation cannot be predicated before runtime. □

4.4 Cost Model

In this subsection, we present an analytical cost model for cooperative sorting. The total cost of utilizing cooperative sorting 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 estimated as the cost C_{s_1} of independently 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 $C_{s_{12} \to s_1}$ of deriving s_1 which is equal to the total cost of performing internal sortings for all the composite s_{12} chunks; (3) the cost $C_{s_{12} \to s_2}$ of deriving s_2 by merging s_{12} chunks which 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 only two cases to consider.

Algorithm 1 `ComputeTB`

```
Output: a tuple batch array $TB$ according to the to be merged $F$ initial $s_1$ runs
$idx \leftarrow 1$
for $i \leftarrow 1$ to $N$ do // $N$ is the total number of $s_{12}$ chunks
    if the $i^{th}$ $s_{12}$ chunk is composite then
        for $j \leftarrow 1$ to $F$ do
            if $ckl_{i,j}$ is non-empty and has not been assigned to a tuple batched yet then // $ckl_{i,j}$ denotes the chunklet in the $j^{th}$ initial $s_1$ run that corresponds to the $i^{th}$ $s_{12}$ chunk
                $TB[idx] \leftarrow TupleBatch(ckl_{i,j})$ // form a type-1 tuple batch starting from the head of $ckl_{i,j}$
                $idx \leftarrow idx + 1$
    else
        for $j \leftarrow 1$ to $F$ do
            if $ckl_{i,j}$ is non-empty and has not been assigned to tuple batches yet then
                if $size(ckl_{i,j}) > \lfloor M/F \rfloor$ then
                    $TB[idx] \leftarrow$ a type-2 tuple batch starting from the head of $ckl_{i,j}$
                    $idx \leftarrow idx + 1$
                    $TB[idx] \leftarrow TupleBatch(ckl_{i,j})$ // form the corresponding type-4 tuple batch
                    $idx \leftarrow idx + 1$
                else
                    $TB[idx] \leftarrow TupleBatch(ckl_{i,j})$ // form a type-1 tuple batch starting from the head of $ckl_{i,j}$
                    $idx \leftarrow idx + 1$
```

Algorithm 2 `TupleBatch`

```
Input: $ckl_{i,j}$
Output: a type-1 (or type-4) tuple batch $tb$ starting from the head (or interior) of $ckl_{i,j}$
initialize a type-1 (or type-4) $tb$ including the whole (or part of) $ckl_{i,j}$
$k \leftarrow i + 1$
while true do // check whether $ckl_{k,j}$ can be included by $tb$
    if ($ckl_{k,j}$ is natural && $size(ckl_{k,j}) > \lfloor M/F \rfloor$) ||
    including $ckl_{k,j}$ in $tb$ violates the size restrictions in the rules ||
    including $ckl_{k,j}$ in $tb$ violates the Inequation 2 for the $l^{th}$ ($i \leq l < k$) $s_{12}$ chunk which is composite then
        break
    include $ckl_{k,j}$ in $tb$
    $k \leftarrow k + 1$
```

Case 1: $B(T)/D(T, o_{11}) \leq 0.5M$.

In this case, all s_{12} chunks are composite, and the number of s_{12} chunks is given by

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

where $k = \left\lfloor \frac{M \times D(T, o_{11})}{B(T)} \right\rfloor$ is the number of o_{11}-*segments* in each composite chunk.

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

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

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

Algorithm 3 `ScheduleReadingOfTupleBatches`

```
Input: $TB$
Output: the order on which tuple batches in $TB$ will be read during the actual run merging
initialize an empty tuple batch pool $P$
$i \leftarrow 1$
while $i \leq length(TB)$ do
    read $TB[i]$ whenever enough memory space is available
    $tb \leftarrow TB[i]$ // mark this tuple batch for later reference
    if $TB[i]$ is a type-1 tuple batch then // otherwise it must be type-2
        $i \leftarrow i + 1$
    else
        add into $P$ the corresponding type-4 tuple batch $TB[i+1]$ along with the set of type-3 tuple batches between $TB[i]$ and $TB[i+1]$
        $i \leftarrow i + 2$
    if after reading $tb$, the merging of chunklets for a natural $s_{12}$ chunk has just be initialized, i.e. all the type-1 and type-2 tuple batches containing tuples of this $s_{12}$ chunk have been read but none of the corresponding type-3 and type-4 tuple batches (recorded in $P$) have been read then
        if $P$ is non-empty then
            driven by the merge progress, read on a specific order all the type-3 and type-4 tuple batches in $P$
            restore $P$ to be empty
```

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

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

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

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.

We conduct a brief analytical comparison between resulting sharing and cooperative sorting as follows. When applying result sharing technique to directly merge o_{11}*-segments* in the output of s_1 to derive the output of s_2, the total cost consists of C_{s_1} as well as the cost incurred by $\lceil log_F D(T, o_{11}) \rceil$ merge passes (i.e., $2 \times B(T) \times \lceil log_F D(T, o_{11}) \rceil$). In case 1, the $\lceil log_F N \rceil$ component of the $C_{s_{12} \rightarrow s_2}$ (i.e., $2 \times B(T) \times \lceil log_F N \rceil$) is at most equal to and often less by at least 1 than $\lceil log_F D(T, o_{11}) \rceil$. As a result, considering the relatively minor CPU costs C_{is} and $C_{s_{12} \rightarrow s_1}$, the total cost of cooperative sorting is often cheaper than that of result sharing. In case 2, the total cost of cooperative sorting is exactly the same as that of applying result sharing.

4.5 Extensions

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

4.5.1 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.

When the final merge optimization is enabled, the intermediate sort operation s_{12} of cooperative sorting will end up with N $(1 < N \leq F)$ s_{12} runs. The output of s_1 is derived by merging these N s_{12} runs on-the-fly, with the *batched tuple reading* strategy being used to sort the tuples in composite chunklets on o_1 before the merging. As for s_2, each chunklet within the s_{12} runs is treated as an initial sorted 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. In case many of the chunklets within these initial s_1 runs are composite, it could be overall cheaper to simply ignore the final merge optimization and directly form a single s_{12} run.

4.5.2 Adapting to Other Merge Patterns

Our description of cooperative sorting in Section 4.2 has assumed that the sorted runs are merged by 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* [17]. In the general case, the collection of the sorted runs to be merged could consist of a combination of initial s_1 runs and s_{12} runs. The *batched tuple reading* strategy can be easily modified so that the 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.

5 Optimization of Multiple Sortings

In this section, we first consider the extension of cooperative sorting to handle more than two sort orders. We then consider post-processing the query execution plans resulted from a conventional query optimizer, so as to further optimize the evaluation of multiple sortings on a relation appearing within these plans. Specifically, we 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. Finally, we describe how to enable the query optimizer to take into account the impact of sort sharing and directly generate the optimal sort-sharing-aware query execution plans.

5.1 K-way Cooperative Sorting

In Section 4, we develop cooperative sorting to evaluate two sort operations s_1 and s_2. In this section, we consider whether it is feasible and makes sense to generalize the binary (2-way) cooperative sorting to a k-way version so that all k sort operations can be simultaneously and efficiently evaluated.

Given two sort orders o_i and o_j, let $o_i \cdot o_j$ denote the sort order $o_i + (o_j - attrs(o_i))$. The *k-way cooperative sorting* is applicable to the k sort operations in S if there exists some permutation of S, $(s_{p1}, s_{p2}, \cdots, s_{pk})$ $(1 \leq pi \leq k)$, such that for each pair of sort orders $o'_{pi} = ((o_{p1} \cdot o_{p2}) \cdot o_{p3}) \cdot \ldots \cdot o_{pi}$ $(1 < i \leq k)$ and o_{pi}, the latter is a subset-prefix of the former. k-way cooperative sorting works as follows: it generates $k-1$ intermediate sort operations $\{s'_2, s'_3, \cdots, s'_k\}$ from a single collection of initial runs that are sorted on o'_{pk}. Each s'_i corresponds to the pair of sort orders o'_{pi} and o_{pi}. s_{p1} is derived from any s'_j $(1 < j \leq k)$ following the way how s_1 is derived from s_{12} in the 2-way cooperative sorting, and each s_{pi} $(1 < i \leq k)$ is derived from s'_i following the way how s_2 is derived from s_{12} in the 2-way cooperative sorting.

Example 7 Consider three sort operations $s_1 = sort(T, (a))$, $s_2 = sort(T, (b))$ and $s_3 = sort(T, (c))$, where a, b, and c are attributes of T. Any permutation of s_1, s_2 and s_3 is qualified for 3-way cooperative sorting. For one such permutation (s_1, s_2, s_3), initial runs sorted on (a, b, c) are generated for two intermediate sort operations s'_2 (w.r.t sort order pair $\{(a, b)$, $(b)\}$) and s'_3 (w.r.t sort order pair $\{(a, b, c)$, $(c)\}$). s_1 and s_2 are then derived from s'_2, while s_3 is derived from s'_3. □

However, the following analytical result based on our cost model in Section 4.4 shows that it is not necessary to consider k-way cooperative sorting for $k > 2$.

Theorem 1 *For each query plan P that involves k-way cooperative sorting, $k > 2$, there exists another equivalent query plan P' that uses only 2-way cooperative sorting such that the cost of P' is no higher than the cost of P.*

The proof of this theorem is given in Appendix A.

5.2 Multiple Sorting Optimization

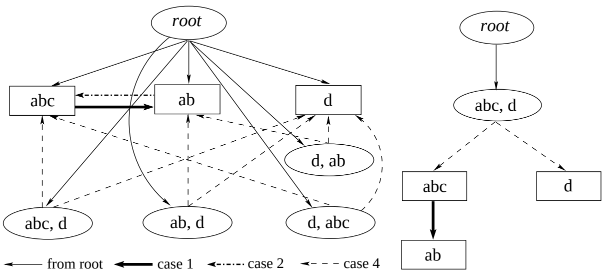
Given a collection S of k sort operations, there are many ways in which these operations can be ordered to exploit sort sharing. In this section, we model this optimization problem as a graph problem. Based on Theorem 1, we consider only the binary cooperative sorting in subsequent discussions. Given S, we construct a directed 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 $u \in V_a$ with $order(u)$ 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 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 3.

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 $w \in V_b$. This node represents a potential cooperative sorting 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 4. $cost(w, v)$ may additionally include the cost of sorting tuple segments for $order(v)$.

Finally, an artificial node $root \in V_a$ 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 sort operation.

Once the graph has been constructed, the optimal solution is obtained by computing the minimum directed Steiner tree spanning G. The sort nodes in V_a are the exact set of vertices's that the Steiner tree aims to interconnect.



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

Fig. 5 An Example of Multiple Sorting Optimization

Example 8 Consider three sortings $sort(T, (a, b))$, $sort(T,$ $(a, b, c))$ and $sort(T, (d))$, where a, b, c and d are attributes of T. The graph for these three sortings 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 graph is shown in Fig. 5(b). Based on the Steiner tree, a feasible evaluation plan is as follows: first evaluate $sort(T, (a, b, c))$ and $sort(T, (d))$ with cooperative sorting, and then derive $sort(T, (a, b))$ from $sort(T, (a, b, c))$. □

Although finding the minimum directed Steiner tree is an NP-hard problem [16], applying a brute-force algorithm is actually acceptable 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 [10]. 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 [6,15] can be applied here.

Execution order of sortings. Each sorting corresponds to a node in the Steiner tree. When an unfinished sorting is triggered by the query execution, in the path from root to this node, all unfinished sortings will be conducted one after another to complete the target sorting. If this target sorting is an internal node of the tree, it is marked after the sorted result is utilized; otherwise, it is deleted from the tree along with the deletion of temporary sorting files. As old leaf nodes are deleted, some internal nodes become new leaves and those marked ones will be repeatedly deleted until all leaves are unmarked yet.

5.3 Sort-sharing-aware Query Optimization

The optimization techniques in Section 5.2 can be encapsulated into a post-optimizer, which receives an execution plan from the original query optimizer, exploits sharing and cooperation opportunities between the sortings in a cost-based manner and, whenever possible, generates a cheaper plan enhanced with the sort sharing techniques. While this two-phase optimization procedure will be very effective and efficient, it cannot guarantee that the refined plan still remains optimal with additional sort sharing consideration. For example, the original optimizer may choose hash join over sort-merge join for a pair of relations, even if the latter may turn out to be cheaper after applying the sort sharing post-optimization on the sortings it involves.

In the rest of this section, we discuss how to equip the standard query optimizer with the ability of sort sharing optimization. As such, the whole search space will be enlarged by the sort sharing extension and an optimal sort-sharing enhanced execution plan will be generated via the single-phase query optimization.

Here we restrict our focus to the system-R [26] style query optimizer, which is also adopted by PostgreSQL. We have modified the PostgreSQL optimizer for our experiments. In Appendix B, we discuss how to extend the Volcano [13] style query optimizer.

The core of the System-R method is its join enumeration algorithm, whose input is a connected join graph $G = (V,E)$ where V represents the set of relations to be joined, and each edge in E represents a join predicate between two relations. During join enumeration, a set of *interesting properties* are defined for subplan pruning. The frequent interesting properties include the total execution cost and interesting orders [26].

Our approach to acquire an optimal sort-sharing-aware plan for V works as follows. We add a new interesting property ip_{ss}. For each subset V' of V, its candidate subplan set P' are generated with the updated set of interesting properties. Generally speaking, ip_{ss} is used to ensure that a previously dominated subplan sp will now remain in P' if it could finally be part of the optimal global sort-sharing-aware plan. Once the plan set P for V are available, we apply the post-optimization described in Section 5.2 to each plan p in P to get a sort-sharing enhanced plan p^+. Finally, the cheapest p^+ is chosen as the final optimal plan for V.

The modeling of ip_{ss} can be various and here we describe one possible modeling. For single table access plan p, let $ip_{ss}(p) = 0$. A sort operation $s = sort(T,o)$ is called as *interesting sorting* if T is a multi-instance relation in V. For a join plan $p_{12} = sp_1 \bowtie sp_2$, let $ip_{ss}(p_{12}) = cost(\bowtie) + ip_{ss}(sp_1) + ip_{ss}(sp_2) - cost_s(\bowtie)$, where $cost(\bowtie)$ is the cost of the join algorithm evaluation and $cost_s(\bowtie)$ is the total cost of the interesting sortings introduced by the join algorithm (e.g., sort-merge join). In other words, $ip_{ss}(p_{12})$ is the reduced plan cost of p_{12} after subtracting the costs of all interesting sortings within the plan tree of p_{12}. For two plans p_{12} and p'_{12}, if $cost(p_{12}) < ip_{ss}(p'_{12})$, then p_{12} is superior to p'_{12} in terms of ip_{ss}. The intuition behind this modeling is that, even if all interesting sortings within p'_{12}'s plan tree can finally be waived via sort-sharing post-optimization owe to the case 1 and thus incur no cost, p_{12} is still cheaper even without any sort sharing optimization. Such an ip_{ss} modeling is conservative but can guarantee the optimality of the resultant plan.

The additional optimization overhead incurred by ip_{ss} is highly dependent on the number, the distribution and the physical properties of the relational instances existing in the join graph G. On the one hand, when there are few instances in G, we expect the optimization overhead will be negligible, as not many extra subplans will be reserved during plan pruning. On the other hand, more instances imply a greater potential to generate a cheaper sort-sharing-aware execution plan, and the cost saving in terms of query execution can easily offset the relatively small cost increase of the query optimization.

6 Discussions

In this section, we discuss the incorporation of ascending and descending orders into the sort sharing techniques (Section 6.1). We present a dynamic way (Section 6.2) to choose at runtime the smartest solution for sortings in cases 3 and 4, instead of the static estimation depending on historical (and thus possibly inaccurate) statistics. We also study how to apply cooperative sorting to simultaneously build multiple indices on a table (Section 6.3). Finally, we briefly discuss the impact of functional dependency and attribute correlation on sort sharing optimization (Section 6.4).

6.1 Ascending/Descending Ordering

Our proposed techniques can be extended 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. Our discussion is based on the case into which the relationship between o_1 and o_2 falls if all the sort 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 it to case 3, the discussion for it is similar to case 3.

6.2 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.

6.3 Cooperative Index Building

In data-intensive applications, such as decision support and data warehousing, an important component of physical database design is selecting the right set of indexes for a given workload. The chosen indices are then created in a batched manner. Sometimes it would be beneficial to create multiple indices on the same table. For example, consider a fact table in a star schema, which contains foreign keys pointing to the other dimension tables. Each dimension table contains a key which corresponds to a foreign key of the fact table and is used for joining with the fact table. As pointed out in [30], the existence of indices on the foreign keys of the fact table enables the *index push-down* optimization, which effectively improves the execution of join queries on the star schema.

Sorting is widely utilized in DBMSs to speed up index creation. The procedure of building an index $Idx(T,k)$ for a table T with key k is as follows. First, sequentially scan T's tuples and extract a list L of *index tuples* where each index tuple consists of a key value and the tuple identifier. Second, externally or internally sort L on the sort order k. Finally, create the index via bulk loading the index tuples of the sorted L and each tuple becomes an entry in the index leaf page.

It is straightforward to exploit cooperative sorting to reduce the total index building cost. For two indices $Idx(T,k_1)$ and $Idx(T,k_2)$, where k_1 and k_2 satisfy case 3 or case 4, we make use of cooperative sorting to generate sorted L_1 and L_2, which are then bulk loaded separately. We call such a procedure *cooperative index building*.

We use s_1 (resp. s_2) to represent the independent sorting on order k_1 (resp. k_2) and use s_{12} to represent the cooperative sorting. After completing s_{12}, the generated s_{12} chunks consist of index tuples containing redundant attributes for k_1 and/or k_2. Therefore, we need to conduct a step of attribute projection when scanning and merging these s_{12} chunks. Depending on which case k_1 and k_2 satisfy, the details of attribute projection are slightly different.

For case 3, $attrs(k_2) \subset attrs(k_1)$. The index tuples in initial s_1 runs will contain attributes $attrs(k_1)$. Therefore, when merging resulted s_{12} chunks to derive the output of s_2 (i.e., the sorted L_2), we remove the redundant attributes $attrs(k_1 - attrs(k_2))$.

For case 4, the initial s_1 runs generated by s_{12} will contain attributes $attrs(k_1) \cup attrs(k_2)$. As a result, it requires an attribute projection to remove redundant attributes $attrs(k_2 - attrs(k_1))$ from index tuples when deriving the output of s_1 (i.e., the sorted L_1); it also requires another attribute projection to remove redundant attributes $attrs(k_1 - attrs(k_2))$ when scanning and merging the generated s_{12} chunks.

6.4 Functional Dependency and Attribute Correlation

The functional dependencies existing among relational attributes have been exploited for the purpose of *sort order reduction* [27], which rewrites the order specification of a sort operation in a simple canonical form by eliminating redundant sort attributes. As such, some sort operations within the query execution plan become unnecessary and thus can be removed. Sort order reduction is complementary to sort sharing optimization, and can be applied separately before sort sharing optimization.

However, during sort sharing optimization, it would be beneficial to take functional dependencies into account when classifying the relationship between two specific sort orders o_1 and o_2. For example, suppose $o_1 = (a,b,d)$ and $o_2 = (b,c)$. Normally, o_1 and o_2 would be judged to satisfy case 4 where $o'_1 = (a,b,d)$ and $o'_2 = (b)$. However, if there is a functional dependency $\{a\} \rightarrow \{c\}$, which means that for any two tuples with the same attribute a values, their attribute c values are also the same, then o_1 can be equivalently treated as (a,c,b,d). As such, o_1 and o_2 actually satisfy case 3, and thus can avoid the additional step of sorting *b-segments* on (c) introduced by case 4 for o_2.

The correlation among attributes could also contribute to sort sharing optimization. For example, consider two sort orders $o_1 = (a)$ and $o_2 = (b)$. Attributes a and b are highly correlated so that for any two tuples t_1 and t_2, if t_1 has a smaller attribute a value than that of t_2, then it is very probable (but not guaranteed) that t_1 also has a smaller attribute b value than that of t_2. As a result, after a relation T has been sorted on o_1, T can be viewed as *nearly* sorted on o_2. Therefore, we can derive the sort output on o_2 by directly sorting the sort output on o_1 and hopefully generating longer and fewer initial sorted runs, which in turn lead to much cheaper run merge cost.

7 Performance Study

We validated our ideas using a prototype built in PostgreSQL 8.3.5 [1]. 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 are stored on the 750GB disk.

This performance study focused on the effect of cooperative sorting. In our implementation, the cooperative sorting 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. Moreover, we modified the PostgreSQL's optimizer to implement the optimization techniques in Section 5.3. By switching between the original and the new optimizer, we can easily compare the cost of processing a query under the cooperative sorting operation against that of the conventional approach based on two independent sort operations.

7.1 Micro-benchmark Test with TPC-DS Dataset

In this section, we use a micro-benchmark test to compare the performance of cooperative sorting 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 a result union (without duplicate removal) of two sortings, 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 generate six concrete queries with the above query template by using three different relations from the TPC-DS [2] benchmark for T and two different scale factors (denoted by SF) to vary the size of T. The statistical information about the three relations, along with their sort attributes, are shown in Table 2. The scale factor SF values used are 40 and 100. Another experimental parameter that we varied is the available sorting memory dedicated to each sort 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 sortings* (denoted by IS) and our proposed *cooperative sorting* (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.

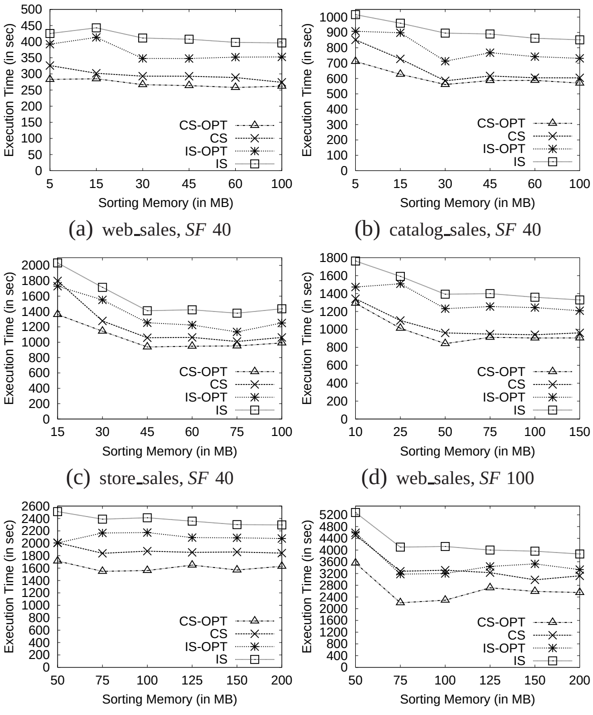
Each total execution time reported refers to the total query evaluation time including the I/O cost of reading the sorted outputs of s_1 and s_2. Each query timing is measured with the query running alone in the database system; and the operating system is restarted between queries to clear the system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

7.1.1 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

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 2 Tested TPC-DS Dataset



(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

	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)

Table 3 Component Costs of CS and IS

IS are shown in Table 4. The meanings of these cost components are given in Table 3.

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 many 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(f). In terms of relative 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, in Table 4, for all the 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 tuple reading* strategy.

Third, for all the six queries, $RF_{cs}(s_{12})$, $RF_{is}(s_1)$ and $RF_{is}(s_2)$ are 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 the 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 the six relations, $SC_{cs}(s_{12})$ 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, it is cheaper to independently sort many smaller composite chunks/chunklets than independently sort fewer larger composite chunks/chunklets, as shown by the analysis of case 2 in Section 3.

7.1.2 Effect of Result Sharing

As discussed at the beginning of Section 4.1, 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 per-

	CS					IS			
Memory	$RF_{cs}(s_{12})$	$RM_{cs}(s_{12})$	$RM_{cs}(s_2)$	$SC_{cs}(s_{12})$	$SC_{cs}(s_1)$	$RF_{is}(s_1)$	$RM_{is}(s_1)$	$RF_{is}(s_2)$	$RM_{is}(s_2)$
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

web_sales, *SF* 40 TPC-DS Dataset

	CS					IS			
Memory	$RF_{cs}(s_{12})$	$RM_{cs}(s_{12})$	$RM_{cs}(s_2)$	$SC_{cs}(s_{12})$	$SC_{cs}(s_1)$	$RF_{is}(s_1)$	$RM_{is}(s_1)$	$RF_{is}(s_2)$	$RM_{is}(s_2)$
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

catalog_sales, *SF* 40 TPC-DS Dataset

	CS					IS			
Memory	$RF_{cs}(s_{12})$	$RM_{cs}(s_{12})$	$RM_{cs}(s_2)$	$SC_{cs}(s_{12})$	$SC_{cs}(s_1)$	$RF_{is}(s_1)$	$RM_{is}(s_1)$	$RF_{is}(s_2)$	$RM_{is}(s_2)$
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

store_sales, *SF* 40 TPC-DS Dataset

	CS					IS			
Memory	$RF_{cs}(s_{12})$	$RM_{cs}(s_{12})$	$RM_{cs}(s_2)$	$SC_{cs}(s_{12})$	$SC_{cs}(s_1)$	$RF_{is}(s_1)$	$RM_{is}(s_1)$	$RF_{is}(s_2)$	$RM_{is}(s_2)$
10MB	491.59	335.56	312.58	12.46	67.81	497.00	354.79	496.74	290.32
25MB	482.22	235.16	181.37	20.17	68.40	482.43	278.31	478.44	242.93
50MB	476.58	219.54	60.73	32.56	70.54	477.87	187.49	466.16	154.08
75MB	483.23	164.60	76.37	46.81	72.64	487.61	177.98	481.98	143.24
100MB	476.82	165.38	70.67	51.36	73.02	477.90	162.51	467.08	143.35
150MB	478.52	133.11	95.14	63.77	78.61	481.79	141.69	472.28	121.95

web_sales, *SF* 100 TPC-DS Dataset

	CS					IS			
Memory	$RF_{cs}(s_{12})$	$RM_{cs}(s_{12})$	$RM_{cs}(s_2)$	$SC_{cs}(s_{12})$	$SC_{cs}(s_1)$	$RF_{is}(s_1)$	$RM_{is}(s_1)$	$RF_{is}(s_2)$	$RM_{is}(s_2)$
50MB	705.38	338.82	565.95	54.38	112.06	703.48	457.27	693.35	419.20
75MB	711.31	398.54	304.16	69.84	119.20	714.88	394.66	694.66	342.41
100MB	715.49	385.83	329.92	77.85	127.05	716.09	395.31	706.47	352.02
125MB	720.86	334.10	330.60	89.33	135.23	723.51	337.24	721.25	319.61
150MB	753.44	312.17	300.76	104.36	147.99	726.86	339.33	703.37	285.73
200MB	726.80	310.83	306.59	107.31	147.15	722.88	335.29	707.18	282.62

catalog_sales, *SF* 100 TPC-DS Dataset

	CS					IS			
Memory	$RF_{cs}(s_{12})$	$RM_{cs}(s_{12})$	$RM_{cs}(s_2)$	$SC_{cs}(s_{12})$	$SC_{cs}(s_1)$	$RF_{is}(s_1)$	$RM_{is}(s_1)$	$RF_{is}(s_2)$	$RM_{is}(s_2)$
50MB	1051.42	1513.20	1204.94	64.11	158.26	1044.71	1474.01	1022.83	1245.96
75MB	999.68	893.91	694.19	78.83	165.50	1052.69	799.03	1048.09	707.88
100MB	976.48	967.27	677.03	91.12	165.33	1026.37	832.70	1047.79	724.80
125MB	1045.31	752.22	689.59	108.00	178.09	1038.55	791.06	1034.95	656.65
150MB	1002.96	614.47	600.49	130.60	187.06	1075.51	739.58	1061.20	601.23
200MB	1079.92	648.63	590.58	152.59	194.64	1043.65	703.66	1048.80	595.02

store_sales, *SF* 100 TPC-DS Dataset

Table 4 Component Costs of Sortings in the Micro-benchmark Test (in seconds)

formance of the query on web_sales with *SF* 40; the comparison for other queries have similar trends and are omitted.

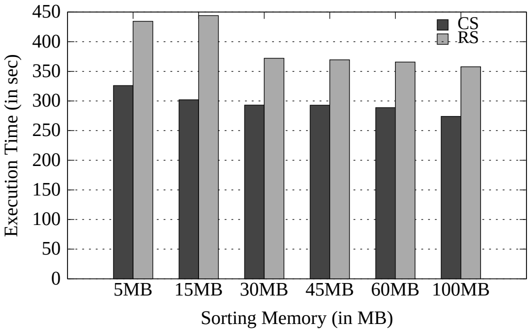


Fig. 7 Comparison of CS with RS on web_sales, *SF* 40

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)).

7.1.3 *Effect of Polyphase Merge Pattern*

The original sort operation in PostgreSQL adopts the polyphase run merge pattern, while we implemented a k-way version sort operation for performance comparison with cooperative sorting. It is natural to ask whether changing the merge pattern will affect the conclusions obtained in Section 7.1.1. In this experiment, we evaluate Q against the 6 tables with the original sort operation (polyphase IS) and compare the execution times with our sort operation (k-way IS).

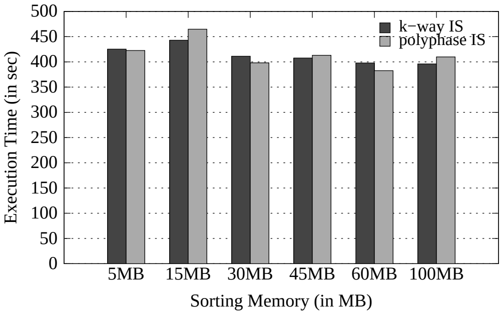


Fig. 8 Comparison of K-way IS with Polyphase IS on web_sales, *SF* 40

Only the results of web_sales with *SF* 40 are shown in Fig. 8. We observe that the performances of polyphase IS and k-way IS are more or less the same, which demonstrates that our results hold independent of the merge pattern.

7.2 Micro-benchmark Test with Synthetic Dataset

We also utilize synthetic data to investigate the sensitivity of CS. We generate synthetic tables following the schema of 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.

7.2.1 *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 sorting 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 a total of n natural chunks.

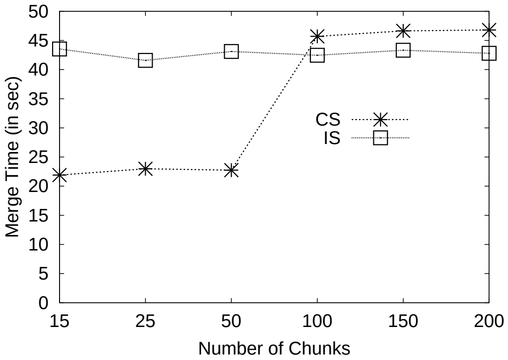


Fig. 9 Varying Total Number of s_{12} Chunks

The experimental result is shown in Fig. 9. The y-axis denotes the run merge time for s_2. With 20MB sorting 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 when varying n, the merge costs of CS remain more or less unchanged as long as the numbers of merge passes required stay the same. We also 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.

7.2.2 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 sortings 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 sorting memory to 50 MB and m to 55. Five values of n are used: 0, 13, 27, 42 and 55.

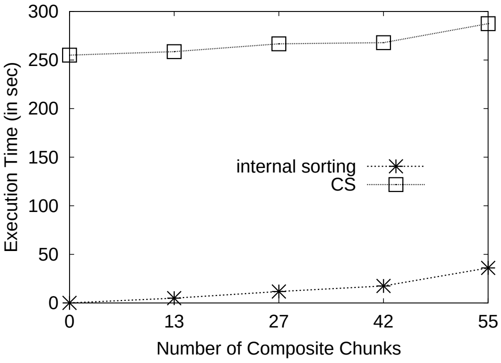


Fig. 10 Varying Number of Composite s_{12} Chunks

Fig. 10 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 20% of the total CS cost.

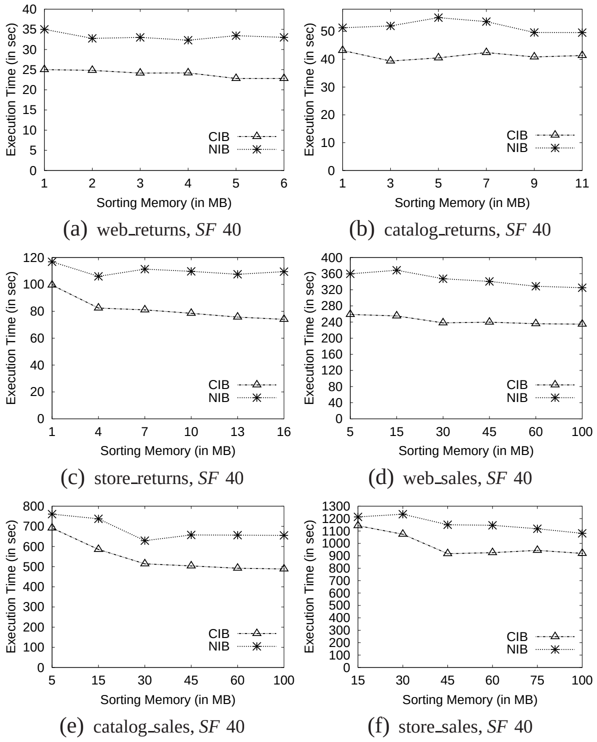
7.3 Performance of Cooperative Index Building

We run another test to compare the performance of cooperative sorting against two independent sort operations for index creation. To achieve this, we create a primary key index $idx_1 = Idx(T, key1)$ as well as a foreign key index $idx_2 = Idx(T, key2)$ on a table T. The index keys *key*1 and *key*2 satisfy case 4.

We generated twelve concrete queries by using six different relations from the TPC-DS benchmark for T and two different scale factors (denoted by *SF*) to vary the size of T. The statistical information about the six relations, along with the index keys, are shown in Table 5. The scale factor *SF* values used are 40 and 100. Another experimental parameter that we varied is the available sorting memory dedicated to each sort operation (denoted by *M*) with values ranging from 1 MB to 200 MB.

We compare the performance of *normal index building* using two independent sortings (denoted by NIB) and our proposed *cooperative index building* using cooperative sorting (denoted by CIB). We always enable the final merge optimization as it is desirable during index creation. However, for the cooperative sorting s_{12}, when the number of initial s_1 runs generated is no more than the merge order F, we disable the final merge optimization for a cheaper cost.

Each total execution time reported refers to the total query evaluation time including the cost of bulk loading the sorted outputs of s_1 and s_2.



(a) web_returns, *SF* 40

(b) catalog_returns, *SF* 40

(c) store_returns, *SF* 40

(d) web_sales, *SF* 40

(e) catalog_sales, *SF* 40

(f) store_sales, *SF* 40

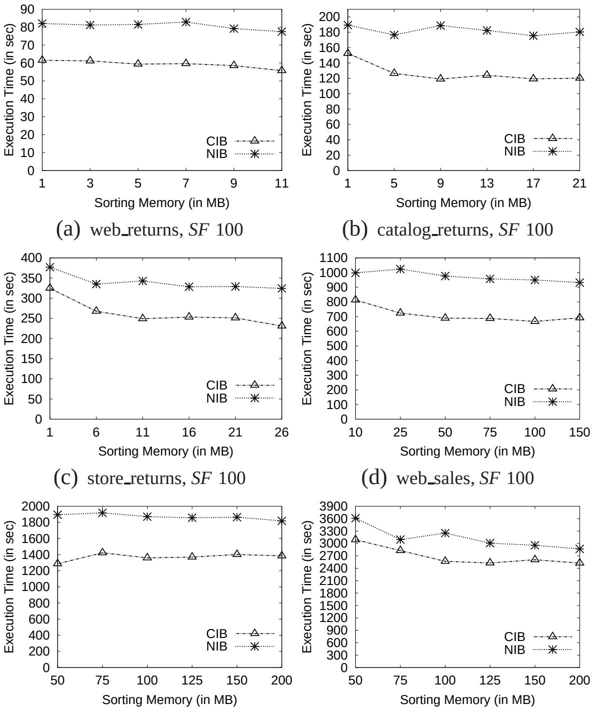
Fig. 11 Performance Comparison on TPC-DS Dataset, with *SF* 40

Figs. 11 and 12 compare the performance of CIB and NIB 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 CIB and NIB are shown in Tables 7 and 8. The meanings of these cost components are given in Table 6. If a specific merge step is skipped because of final merge optimization, the corresponding entry value ($RM_{cs}(s_2)$ or $RM_{is}(s_1)$) in Tables 7 and 8 is marked as zero. Note that for every row in Tables 7 and 8, $RM_{is}(s_2)$ is always zero and is thus omitted. Due to space limitation, $SC_{cs}(s_1)$ is not separately listed but merged into $LD_{cs}(s_1)$. For each row in Tables 7 and 8, if $RM_{is}(s_1)$ is zero, it means that there is only one merge level for the initial s_1 runs and s_{12} does not apply final merge optimization as stated above.

First, even though sorting is just a part of the index building procedure, CIB still offers significant performance improvement over NIB for most queries. The savings range from a few seconds to 683 seconds which is achieved for the query on *store_sales* with $M = 100$ and $SF = 100$ in Fig. 12. In terms of relative improvement, the average percentage

relation	*key1*	*key2*	number of tuples (in million)	tuple size (in byte)
web_returns	`(wr_item_sk, wr_order_number)`	`wr_returned_time_sk`	$0.072 \times SF$	150
catalog_returns	`(cr_item_sk, cr_order_number)`	`cr_returned_time_sk`	$0.144 \times SF$	162
store_returns	`(sr_item_sk, sr_ticket_number)`	`sr_returned_time_sk`	$0.288 \times SF$	134
web_sales	`(ws_item_sk, ws_order_number)`	`ws_sold_time_sk`	$0.72 \times SF$	226
catalog_sales	`(cs_item_sk, cs_order_number)`	`cs_sold_time_sk`	$1.44 \times SF$	226
store_sales	`(ss_item_sk, ss_ticket_number)`	`ss_sold_time_sk`	$2.88 \times SF$	164

Table 5 TPC-DS Dataset for Comparing Performance of Index Construction



(a) web_returns, *SF* 100

(b) catalog_returns, *SF* 100

(c) store_returns, *SF* 100

(d) web_sales, *SF* 100

(e) catalog_sales, *SF* 100

(f) store_sales, *SF* 100

Fig. 12 Performance Comparison on TPC-DS Dataset, with *SF* 100

	notation	description
CIB	$RF_{cs}(s_{12})$	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}
	$LD_{cs}(s_1)$	cost of deriving and bulk-loading output of s_1 to build idx_1
	$LD_{cs}(s_2)$	cost of deriving and bulk-loading output of s_2 to build idx_2
NIB	$RF_{is}(s_1)$	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)
	$LD_{is}(s_1)$	cost of deriving and bulk-loading output of s_1 to build idx_1
	$LD_{is}(s_2)$	cost of deriving and bulk-loading output of s_2 to build idx_2

Table 6 Component Costs of CIB and NIB

improvement is around 24% and the highest improvement is 37% achieved for the query on *catalog_returns* with $M = 9$ and $SF = 100$ in Fig. 12. The main reason for such performance gain is due to the fact that the sorting time is always much higher than the subsequent bulk loading time.

Second, there are some trends similar to those in the previous micro-benchmark test (Section 7.1). For all 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. For all tables, $SC_{cs}(s_{12})$ and $SC_{cs}(s_1)$ increase along with the size of sorting memory.

Third, for all queries that require more than one merge level for the initial s_1 runs (i.e., $RM_{is}(s_1) \neq 0$), $RM_{cs}(s_{12})$ (resp. $LD_{cs}(s_1) - SC_{cs}(s_1)$) is close to or even less than the corresponding $RM_{is}(s_1)$ (resp. $LD_{is}(s_1)$). This is due to the I/O effectiveness and efficiency of our *batched reading* strategy. Note that $RM_{cs}(s_{12})$ does not include the internal sorting cost $SC_{cs}(s_{12})$.

Fourth, for most tables, in terms of the total cost of run merge plus bulk loading for s_2, CIB's cost is higher than NIB's cost when the sorting memory size is small, i.e., $RM_{cs}(s_2)$ $+ LD_{cs}(s_2) > RM_{is}(s_2) + LD_{is}(s_2)$. This is expected as CIB is operating on a larger set of s_2 data and generates more initial s_2 runs to merge than NIB. However, when the sorting memory increases, the difference between these two costs decreases, and eventually the cost in CIB is even cheaper than the cost in NIB.

7.4 Query Processing with Sort Sharing

So far, we have evaluated cooperative sorting for the basic scenario of processing two sort operations on different orders. In this section, we evaluate the effectiveness of sort sharing techniques and the enhanced sort-sharing-aware query optimizer when executing queries. We generate a synthetic

	CIB						NIB				
Memory	$RF_{cs}(s_{12})$	$RM_{cs}(s_{12})$	$RM_{cs}(s_2)$	$SC_{cs}(s_{12})$	$LD_{cs}(s_1)$	$LD_{cs}(s_2)$	$RF_{is}(s_1)$	$RM_{is}(s_1)$	$RF_{is}(s_2)$	$LD_{is}(s_1)$	$LD_{is}(s_2)$
1MB	12.17	1.67	3.16	0.38	4.90	2.58	12.35	1.84	12.56	5.26	2.31
2MB	12.15	1.08	4.74	0.65	3.46	2.42	12.40	1.21	12.46	3.81	2.14
3MB	12.12	1.33	2.34	0.77	4.80	2.43	12.66	1.56	11.92	3.80	2.47
4MB	12.53	2.05	0.88	0.91	4.76	2.89	12.04	1.66	12.00	3.16	2.79
5MB	12.18	1.52	0.97	0.90	4.71	2.16	12.61	1.84	12.18	3.79	2.33
6MB	12.01	1.65	0.92	1.04	4.41	2.46	13.09	0.0	12.34	4.30	2.45

web_returns, *SF* 40 TPC-DS Dataset

	CIB						NIB				
Memory	$RF_{cs}(s_{12})$	$RM_{cs}(s_{12})$	$RM_{cs}(s_2)$	$SC_{cs}(s_{12})$	$LD_{cs}(s_1)$	$LD_{cs}(s_2)$	$RF_{is}(s_1)$	$RM_{is}(s_1)$	$RF_{is}(s_2)$	$LD_{is}(s_1)$	$LD_{is}(s_2)$
1MB	19.07	3.85	10.53	0.66	4.77	3.73	19.39	3.74	17.88	5.98	3.74
3MB	18.96	2.67	3.69	1.11	8.02	4.65	18.13	3.00	18.02	7.54	4.15
5MB	20.31	3.61	1.90	1.52	7.94	4.79	19.28	4.09	19.50	6.37	4.98
7MB	19.40	3.84	2.01	1.93	10.28	4.55	18.75	4.85	18.99	6.01	4.29
9MB	19.05	4.03	1.91	2.01	8.77	4.69	20.02	0.0	19.15	6.11	3.44
11MB	19.85	3.72	0.0	2.04	9.26	6.14	19.14	0.0	19.85	6.00	3.68

catalog_returns, *SF* 40 TPC-DS Dataset

	CIB						NIB				
Memory	$RF_{cs}(s_{12})$	$RM_{cs}(s_{12})$	$RM_{cs}(s_2)$	$SC_{cs}(s_{12})$	$LD_{cs}(s_1)$	$LD_{cs}(s_2)$	$RF_{is}(s_1)$	$RM_{is}(s_1)$	$RF_{is}(s_2)$	$LD_{is}(s_1)$	$LD_{is}(s_2)$
1MB	39.76	15.53	21.52	1.37	11.62	8.89	39.96	15.41	42.29	12.54	5.76
4MB	42.90	5.96	9.94	2.33	13.23	7.69	40.42	6.76	41.27	10.68	5.96
7MB	40.07	7.65	4.00	2.80	18.80	7.34	42.76	10.17	39.85	10.60	7.14
10MB	39.28	8.64	4.44	3.04	15.45	7.34	39.27	10.79	39.87	11.93	6.82
13MB	40.49	8.44	5.00	3.36	10.81	7.17	42.56	0.0	39.61	14.34	9.93
16MB	41.20	8.21	0.0	3.81	12.25	8.29	40.13	0.0	42.36	16.55	9.10

store_returns, *SF* 40 TPC-DS Dataset

	CIB						NIB				
Memory	$RF_{cs}(s_{12})$	$RM_{cs}(s_{12})$	$RM_{cs}(s_2)$	$SC_{cs}(s_{12})$	$LD_{cs}(s_1)$	$LD_{cs}(s_2)$	$RF_{is}(s_1)$	$RM_{is}(s_1)$	$RF_{is}(s_2)$	$LD_{is}(s_1)$	$LD_{is}(s_2)$
5MB	114.82	42.92	21.38	3.75	36.72	37.79	122.94	50.88	121.70	31.09	31.65
15MB	118.25	37.73	12.19	8.37	48.49	28.39	122.58	61.13	122.55	29.07	31.81
30MB	114.79	46.11	0.0	11.92	38.00	26.29	124.16	0.0	123.13	65.71	31.46
45MB	121.06	46.90	0.0	13.42	34.97	22.50	123.30	0.0	129.82	56.49	27.42
60MB	121.00	40.29	0.0	16.16	31.37	22.16	119.95	0.0	121.78	50.88	31.87
100MB	120.71	38.80	0.0	22.60	30.42	21.39	120.30	0.0	122.86	46.29	29.64

web_sales, *SF* 40 TPC-DS Dataset

	CIB						NIB				
Memory	$RF_{cs}(s_{12})$	$RM_{cs}(s_{12})$	$RM_{cs}(s_2)$	$SC_{cs}(s_{12})$	$LD_{cs}(s_1)$	$LD_{cs}(s_2)$	$RF_{is}(s_1)$	$RM_{is}(s_1)$	$RF_{is}(s_2)$	$LD_{is}(s_1)$	$LD_{is}(s_2)$
5MB	242.68	132.64	137.28	6.94	86.53	80.35	251.62	144.24	250.50	47.85	63.00
15MB	248.14	122.63	32.24	16.41	88.32	73.82	244.54	125.28	243.98	57.23	63.78
30MB	243.26	81.99	26.56	23.51	69.05	66.49	227.69	0.0	229.18	112.56	55.81
45MB	243.86	110.16	0.0	28.42	68.74	50.27	226.68	0.0	227.72	141.63	56.25
60MB	244.23	97.42	0.0	34.17	63.00	50.71	244.42	0.0	242.53	104.58	59.26
100MB	245.17	85.09	0.0	48.15	61.91	45.87	243.59	0.0	244.63	98.61	60.58

catalog_sales, *SF* 40 TPC-DS Dataset

	CIB						NIB				
Memory	$RF_{cs}(s_{12})$	$RM_{cs}(s_{12})$	$RM_{cs}(s_2)$	$SC_{cs}(s_{12})$	$LD_{cs}(s_1)$	$LD_{cs}(s_2)$	$RF_{is}(s_1)$	$RM_{is}(s_1)$	$RF_{is}(s_2)$	$LD_{is}(s_1)$	$LD_{is}(s_2)$
15MB	357.17	376.00	56.68	18.42	172.56	156.89	362.11	229.02	354.70	150.13	114.50
30MB	394.37	225.36	64.21	29.86	201.09	151.49	353.83	257.54	356.87	140.07	123.38
45MB	364.07	172.20	69.11	33.98	140.53	130.42	384.02	0.0	367.14	263.73	128.57
60MB	389.42	240.03	0.0	47.61	136.47	107.93	384.39	0.0	353.24	279.82	121.23
75MB	391.61	240.29	0.0	58.01	141.05	108.65	359.72	0.0	354.20	277.25	120.84
100MB	390.97	200.70	0.0	70.37	144.21	108.10	363.89	0.0	351.61	245.71	111.46

store_sales, *SF* 40 TPC-DS Dataset

Table 7 Component Costs of CIB and NIB with *SF* 40 (in seconds)

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 salespersons and has 10 million 32-byte tuples, `Sales` records the sale transactions and has 50 million 12-byte tuples and

	CIB						NIB				
Memory	$RF_{cs}(s_{12})$	$RM_{cs}(s_{12})$	$RM_{cs}(s_2)$	$SC_{cs}(s_{12})$	$LD_{cs}(s_1)$	$LD_{cs}(s_2)$	$RF_{is}(s_1)$	$RM_{is}(s_1)$	$RF_{is}(s_2)$	$LD_{is}(s_1)$	$LD_{is}(s_2)$
1MB	31.48	7.51	11.25	1.04	5.26	4.45	31.91	6.59	32.04	6.28	4.50
3MB	32.91	3.30	11.17	1.55	6.17	5.77	31.43	3.29	32.39	8.56	4.77
5MB	32.21	3.98	5.22	1.84	9.63	6.04	32.04	4.72	31.58	7.13	5.18
7MB	32.08	4.62	2.65	2.50	11.14	6.07	31.73	5.71	32.06	7.63	4.89
9MB	32.09	4.79	2.57	2.65	9.69	6.37	33.86	0.0	32.97	7.84	3.16
11MB	33.32	4.98	2.78	2.84	6.95	4.56	32.52	0.0	32.94	7.77	3.21

web_returns, *SF* 100 TPC-DS Dataset

	CIB						NIB				
Memory	$RF_{cs}(s_{12})$	$RM_{cs}(s_{12})$	$RM_{cs}(s_2)$	$SC_{cs}(s_{12})$	$LD_{cs}(s_1)$	$LD_{cs}(s_2)$	$RF_{is}(s_1)$	$RM_{is}(s_1)$	$RF_{is}(s_2)$	$LD_{is}(s_1)$	$LD_{is}(s_2)$
1MB	72.09	24.75	25.88	1.83	15.78	10.64	70.06	24.38	71.50	14.13	8.33
5MB	69.96	13.01	10.10	2.85	17.78	12.37	71.73	12.77	70.36	12.35	7.59
9MB	64.80	14.71	5.33	3.77	18.95	11.31	72.77	18.21	70.64	13.81	12.19
13MB	69.73	14.96	5.73	4.99	17.60	10.68	71.93	0.0	72.94	23.74	12.14
17MB	69.61	14.04	0.0	5.47	16.76	12.98	71.41	0.0	68.80	21.22	11.99
21MB	72.55	13.79	0.0	6.00	17.24	10.28	70.43	0.0	72.77	22.40	13.04

catalog_returns, *SF* 100 TPC-DS Dataset

	CIB						NIB				
Memory	$RF_{cs}(s_{12})$	$RM_{cs}(s_{12})$	$RM_{cs}(s_2)$	$SC_{cs}(s_{12})$	$LD_{cs}(s_1)$	$LD_{cs}(s_2)$	$RF_{is}(s_1)$	$RM_{is}(s_1)$	$RF_{is}(s_2)$	$LD_{is}(s_1)$	$LD_{is}(s_2)$
1MB	119.64	65.80	70.51	2.84	38.58	27.45	122.96	77.21	119.67	27.94	27.37
6MB	122.45	51.49	27.87	3.89	35.98	24.98	119.46	36.36	121.23	29.65	26.42
11MB	119.78	45.89	16.56	5.65	36.27	23.23	120.42	44.74	122.27	27.82	26.01
16MB	123.37	45.95	12.20	7.63	37.71	24.77	122.79	0.0	121.99	57.61	23.74
21MB	120.31	45.31	12.08	8.54	34.67	28.62	122.44	0.0	119.80	55.89	28.44
26MB	121.33	47.55	0.0	9.05	30.58	21.90	120.69	0.0	118.96	59.74	22.27

store_returns, *SF* 100 TPC-DS Dataset

	CIB						NIB				
Memory	$RF_{cs}(s_{12})$	$RM_{cs}(s_{12})$	$RM_{cs}(s_2)$	$SC_{cs}(s_{12})$	$LD_{cs}(s_1)$	$LD_{cs}(s_2)$	$RF_{is}(s_1)$	$RM_{is}(s_1)$	$RF_{is}(s_2)$	$LD_{is}(s_1)$	$LD_{is}(s_2)$
10MB	367.37	185.29	64.86	28.33	115.41	72.73	367.88	130.74	353.44	76.76	67.55
25MB	366.31	101.06	35.17	36.52	147.14	63.19	364.42	140.74	367.71	74.39	74.51
50MB	366.29	134.47	0.0	49.96	109.81	62.53	367.67	0.0	367.48	161.26	75.56
75MB	353.53	107.97	0.0	62.23	119.55	68.41	366.20	0.0	365.19	145.66	74.17
100MB	355.73	98.76	0.0	87.97	105.77	62.54	367.09	0.0	368.00	131.01	75.88
150MB	367.23	95.03	0.0	110.87	113.11	64.53	365.35	0.0	364.66	117.61	75.23

web_sales, *SF* 100 TPC-DS Dataset

	CIB						NIB				
Memory	$RF_{cs}(s_{12})$	$RM_{cs}(s_{12})$	$RM_{cs}(s_2)$	$SC_{cs}(s_{12})$	$LD_{cs}(s_1)$	$LD_{cs}(s_2)$	$RF_{is}(s_1)$	$RM_{is}(s_1)$	$RF_{is}(s_2)$	$LD_{is}(s_1)$	$LD_{is}(s_2)$
50MB	684.11	286.07	0.0	54.49	194.25	139.52	700.05	0.0	701.99	318.83	166.82
75MB	699.00	306.33	0.0	73.87	191.15	146.53	698.60	0.0	698.12	346.95	167.25
100MB	692.16	255.92	0.0	83.69	191.43	131.78	696.26	0.0	698.25	300.61	166.82
125MB	691.51	239.11	0.0	100.66	192.40	141.00	696.13	0.0	698.65	284.29	166.21
150MB	697.67	252.29	0.0	112.42	196.01	137.42	698.40	0.0	701.03	284.75	167.67
200MB	698.56	214.58	0.0	121.02	209.85	133.78	702.50	0.0	700.29	250.96	149.78

catalog_sales, *SF* 100 TPC-DS Dataset

	CIB						NIB				
Memory	$RF_{cs}(s_{12})$	$RM_{cs}(s_{12})$	$RM_{cs}(s_2)$	$SC_{cs}(s_{12})$	$LD_{cs}(s_1)$	$LD_{cs}(s_2)$	$RF_{is}(s_1)$	$RM_{is}(s_1)$	$RF_{is}(s_2)$	$LD_{is}(s_1)$	$LD_{is}(s_2)$
50MB	1021.32	616.76	353.91	72.12	635.09	359.47	1025.74	832.74	1033.73	359.41	343.69
75MB	1022.98	448.50	361.05	85.49	457.38	430.03	1025.55	0.0	1026.94	717.00	308.64
100MB	959.97	629.48	0.0	103.33	453.39	408.62	973.20	0.0	1024.31	933.09	305.94
125MB	991.77	820.71	0.0	115.40	430.16	413.82	978.07	0.0	983.09	714.52	316.50
150MB	977.85	588.76	0.0	145.80	460.39	417.66	1025.25	0.0	951.52	696.81	266.49
200MB	1000.33	502.42	0.0	164.53	431.94	413.46	972.06	0.0	942.63	634.19	300.10

store_sales, *SF* 100 TPC-DS Dataset

Table 8 Component Costs of CIB and NIB with *SF* 100 (in seconds)

`Item` records the products in transactions and has 10 million 24-byte tuples.

We evaluate two queries on this database:

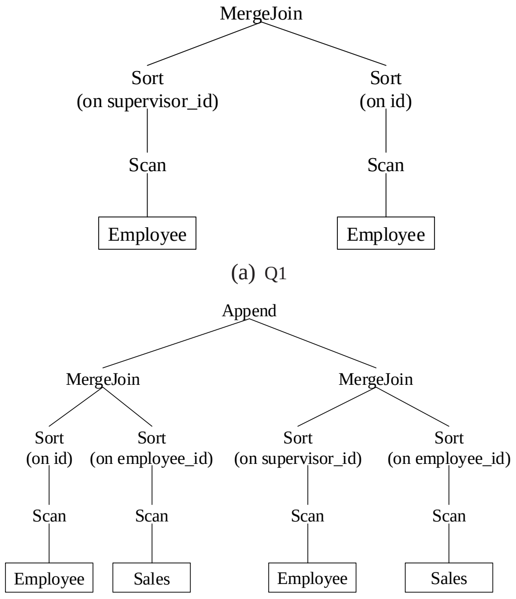
Q1: Find the name of each salesperson and its supervisor.

`select` *A.id, A.name, B.id, B.name*

`from` *Employee A, Employee B*
`where` *B.id = A.supervisor_id*

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

(`select` *A.id, A.name* `from` *Employee A, Sales B*
`where` *A.id = B.employee_id* `and` *B.quantity > 1000*)
`union all`
(`select` *A.id, A.name* `from` *Employee A, Sales B*
`where` *A.supervisor_id = B.employee_id*
`and` *B.quantity > 1000*)



(a) Q1

(b) Q2

Fig. 13 The Optimal Plans for Q1 and Q2 by the Original PostgreSQL Optimizer

With 50MB sorting memory, the optimal plans generated by the original PostgreSQL optimizer for these two queries are shown in Fig. 13.

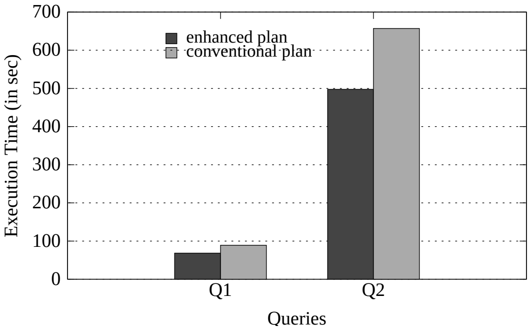


Fig. 14 Query Execution Times of Q1 and Q2

We also optimize Q1 and Q2 with our enhanced PostgreSQL optimizer. The resultant optimal plans enable the cooperative sorting between two instances of `Employee` in both Q1 and Q2. For Q2, the plan also skips one redundant sort on `Sales` via result sharing for case 1 and thus saves about another 90 seconds' time. The comparison of the overall query execution times are shown in Fig. 14. The results clearly show that both queries can be processed in lesser time with sort sharing techniques.

We then study the potential benefit of enriching the optimizer search space with sort sharing. 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).

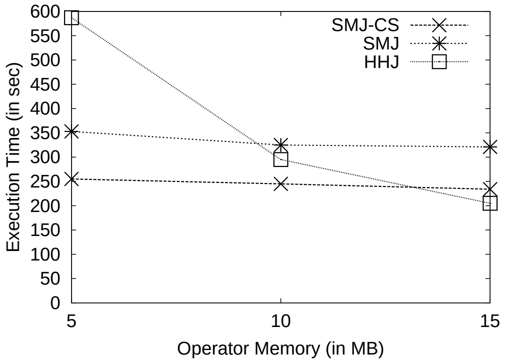


Fig. 15 Plans Considered During Query Optimization for Q1

Fig. 15 shows the candidate plans considered during optimizing Q1, along with their actual execution times (we force the execution of a non-optimal plan). Besides the SMJ and HHJ that are enumerated by the original PostgreSQL optimizer, our enhanced PostgreSQL optimizer also measures SMJ-CS. When the operator memory is 15MB, both the original optimizer and our enhanced optimizer generate the same optimal HHJ plan. However, when the operator memory is 5MB or 10MB, the SMJ-CS is recognized by our enhanced optimizer as the optimal plan, instead of the SMJ or HHJ recognized by the original optimizer.

8 Related Work

Sorting is one of the most extensively studied problems in computing. Knuth's classical text [17] provides extensive coverage of the fundamentals of sorting, including both replacement selection for run formation and run merge patterns.

Standard replacement selection produces runs twice the size of memory on average. There have been several efforts to increase the run length further ([7, 9, 28]). Larson [19] introduced a cache-aware replacement selection that works for various length keys. There are also many techniques to speed up the run merge phase ([31–33]), 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 tuple reading strategy, which relies more on the pre-collected knowledge about input data distribution. Our current implementation only applies simple forecasting technique to read the type-3 tuple batches. But it is possible to incorporate other optimization techniques like double buffering [17], read-ahead [32], etc. Much research has been done on adaptive sorting [8] exploiting near-sortedness. The survey [12] by Graefe discussed how sorting is implemented in database systems with many tricks and optimizations. Specifically, [12] identified a special instance of case 3, where $o_1 = (a, b)$ and $o_2 = (b)$, and pointed out that the sorting on o_2 can be evaluated by directly merging the output of the sorting on o_1, which is exactly the same as we discuss at the beginning of Section 4.1. However, neither analytical nor experimental study on the effectiveness of the proposed approach were conducted. Moreover, [12] did not generalize this special instance to the general case 3.

Simmen et al. [27] described how to determine the ordering propagation from the inputs to the outputs of joins, based on functional dependencies and selection conditions. As such, some sort operations within the query execution plan become redundant and thus can be removed. Their work was followed and extended by [20, 29, 21], which are all independent and complementary to our work.

In [14], 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 sort sharing between two distinct sort operations. To some extent, [14] and our work are complementary to each other. A similar idea to partial sorting was considered previously in [3] 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 work on optimizing multiple scans on the same table, such as `MAPLE` [4] and *cooperative scan* [34].

9 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 sorting technique to efficiently handle sorting of a table on two orders. We also proposed optimization techniques to exploit sort sharing in a traditional query evaluation plan. We have implemented our techniques in PostgreSQL, and our extensive performance study indicated a significant performance gain over the naive strategy of processing each sorting independently.

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

References

1. Postgresql Offical Website. `http://www.postgresql.org/`.
2. TPC BENCHMARK Decision Support. `http://www.tpc.org/tpcds/`.
3. 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.
4. Y. Cao, G. C. Das, C.-Y. Chan, and K.-L. Tan. Optimizing complex queries with multiple relation instances. In *SIGMOD*, 2008.
5. M. Charikar, S. Chaudhuri, R. Motwani, and V. Narasayya. Towards estimation error guarantees for distinct values. In *PODS*, 2000.
6. M. Charikar, C. Chekuri, Z. Dai, A. Goel, S. Guha, and M. Li. Approximation algorithms for directed steiner problems. In *Journal of Algorithms*, pages 73–91, 1999.
7. R. Dinsmore. Longer strings for sorting. *Comm. ACM*, 8(1):48, 1965.
8. V. Estivill-Castro and D. Wood. A survey of adaptive sorting algorithms. *ACM Computing Surveys (CSUR)*, 24(4):441–476, 1992.
9. W. Frazer and C. Wong. Sorting by natural selection. *Communications of the ACM*, 15(10):910–913, 1972.
10. L. Georgiadis. Arborescence optimization problems solvable by edmonds' algorithm. *Theor. Comput. Sci.*, 301(1-3):427–437, 2003.
11. P. B. Gibbons. Distinct sampling for highly-accurate answers to distinct values queries and event reports. In *VLDB*, 2001.
12. G. Graefe. Implementing sorting in database systems. *ACM Comput. Surv.*, 38(3):10, 2006.
13. G. Graefe and W. J. McKenna. The volcano optimizer generator: Extensibility and efficient search. In *ICDE*, 1993.
14. R. Guravannavar and S. Sudarshan. Reducing order enforcement cost in complex query plans. In *ICDE*, 2007.
15. 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.*, 22(6):1409–1425, 2006.
16. R. M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.
17. D. E. Knuth. *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. Addison-Wesley, 1998.
18. R. P. Kooi. *The optimization of queries in relational databases*. PhD thesis, Case Western Reserve University, 1980.

19. P. Larson. External sorting: Run formation revisited. *IEEE Transactions on Knowledge and Data Engineering*, 15(4):961–972, 2003.
20. T. Neumann and G. Moerkotte. A combined framework for grouping and order optimization. In *VLDB*, 2004.
21. T. Neumann and G. Moerkotte. An efficient framework for order optimization. In *ICDE*, 2004.
22. V. Pai and P. Varman. Prefetching with multiple disks for external mergesort: simulation and analysis. In *ICDE*, 1992.
23. N. Roussopoulos. View indexing in relational databases. *ACM Trans. Database Syst.*, 7(2):258–290, 1982.
24. P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. In *SIGMOD*, 2000.
25. B. Salzberg. Merging sorted runs using large main memory. *Acta Informatica*, 27(3):195–215, 1989.
26. P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD*, 1979.
27. D. Simmen, E. Shekita, and T. Malkemus. Fundamental techniques for order optimization. In *SIGMOD*, 1996.
28. T. Ting and Y. Wang. Multiway replacement selection sort with dynamic reservoir. *The Computer Journal*, 20(4):298–301, 1977.
29. X. Wang and M. Cherniack. Avoiding sorting and grouping in processing queries. In *VLDB*, 2003.
30. A. Weininger. Efficient execution of joins in a star schema. In *SIGMOD*, 2002.
31. W. Zhang and P. Larson. Dynamic memory adjustment for external mergesort. In *VLDB*, 1997.
32. W. Zhang and P.-A. Larson. Buffering and read-ahead strategies for external mergesort. In *VLDB*, 1998.
33. L. Zheng and P. Larson. Speeding up external mergesort. *IEEE Transactions on Knowledge and Data Engineering*, 8(2):322–332, 1996.
34. M. Zukowski, S. Héman, N. Nes, and P. A. Boncz. Cooperative scans: Dynamic bandwidth sharing in a DBMS. In *VLDB*, 2007.

APPENDIX

A The Proof of Theorem 1

In this section, we provide the proof of Theorem 1 in Section 5.1. The proof is based on induction. We first analyze the performance of 3-way and 4-way cooperative sorting and compare them with the alternative realizations using 2-way cooperative sorting. Subsequently, we generalize the analysis to k-way cooperative sorting for $k \geq 3$. For simplicity, we assume the permutation of S is $s_1 s_2 \cdots s_k$ and let o'_i denote $((o_1 \cdot o_2) \cdot o_3) \cdot \ldots \cdot o_i$.

The figures below represent the execution plans of different cooperative sortings. Each node represents the set of tuples in relation T associated with a specific tuple arrangement. Each directed edge represents an operation which reorganize the tuples of one node to derive another node. The edges are annotated with the I/O costs of operations. Besides the I/O costs, we also explicitly count in two types of non-trivial CPU costs incurred by cooperative sortings, i.e. the cost of internally sorting the composite chunklets within initial sorted runs during the intermediate sort operation s_{12} and the cost of internally sorting the composite chunks of s_{12} to derive s_1. We assume that CPU costs of the same type are universally equal.

Analysis of 3-way cooperative sorting. Fig. 16 shows an execution plan of 3-way cooperative sorting. The table T is first sorted into initial runs on $o'_3 = o_1 \cdot o_2 \cdot o_3$, which are then separately fed into the two intermediate sort operations s'_2 and s'_3. Finally, s_1 and s_2 are derived from s'_2, while s_3 is derived from s'_3.

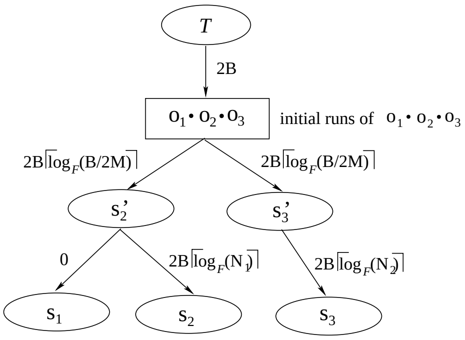


Fig. 16 The Execution Plan of 3-way Cooperative Sorting

The cost of generating initial sorted runs on o'_3 is $2 \times B$, where B is the total number of blocks of tuples in T (i.e., $B = B(T)$). The costs of s'_2 and s'_3 are both $2 \times B \times \lceil log_F \frac{B}{2M} \rceil$ plus C_{is}, which is the cost of performing internal sortings on composite chunklets within the initial runs. s_1 can be derived from s'_2 with the cost $C_{s'_2 \to s_1}$ of performing internal sortings for all the composite chunks of s'_2, and s_2 can be produced by the chunk merging procedure (Section 4.1) from s'_2 with a cost $2 \times B \times \lceil log_F N_1 \rceil$, where N_1 is the number of chunks of s'_2. s_3 is computed by a chunk merge procedure from s'_3 with a cost $2 \times B \times \lceil log_F N_2 \rceil$, where N_2 is the number of chunks of s'_3. Hence, the total cost of 3-way cooperative sorting is

$$2 \times B \times (1 + 2 \times \lceil log_F \frac{B}{2M} \rceil + \lceil log_F N_1 \rceil + \lceil log_F N_2 \rceil) \\ + 2 \times C_{is} + C_{s'_2 \to s_1} \tag{8}$$

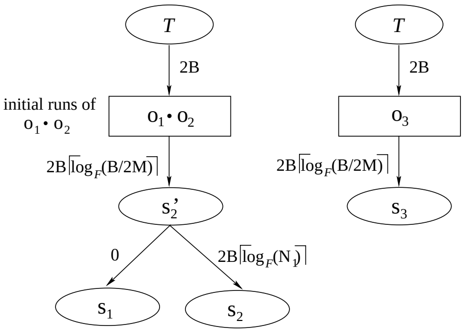


Fig. 17 The Alternative Execution Plan of 2-way Cooperative Sorting

We compare this execution plan with another plan that is based on 2-way cooperative sorting depicted in Fig. 17, where s_1 and s_2 are derived from the intermediate sort operation s'_2 of a 2-way cooperative sorting, and s_3 is a normal external sorting. The total cost of this plan is

$$2 \times B \times (2 + 2 \times \lceil log_F \frac{B}{2M} \rceil + \lceil log_F N_1 \rceil) + C_{is} + C_{s'_2 \to s_1} \tag{9}$$

The difference obtained by subtracting Eqn. 9 from Eqn. 8 is: $2 \times B \times (\lceil log_F N_2 \rceil - 1) + C_{is}$, which is always non-negative. Hence, 3-way cooperative sorting is no cheaper than its alternative realizations using 2-way cooperative sorting.

Analysis of 4-way cooperative sorting. A similar analysis can be derived to compare the performance of 4-way cooperative sorting with 2-way cooperative sorting.

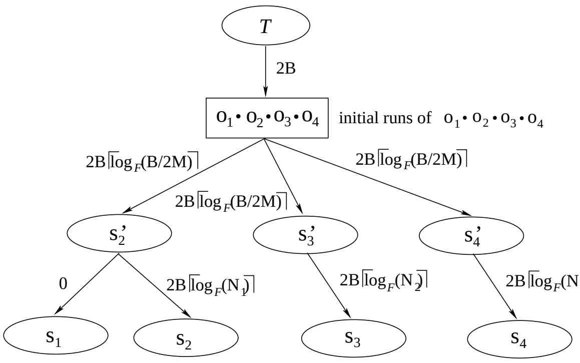


Fig. 18 The Execution Plan of 4-way Cooperative Sorting

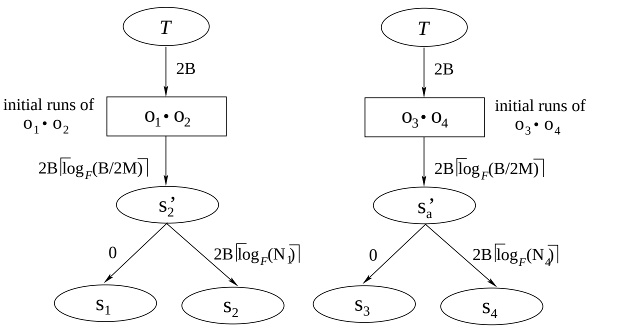


Fig. 19 The Alternative Execution Plan of 2-way Cooperative Sorting

The execution plan of 4-way cooperative sorting is shown in Fig. 18. The table T is first sorted into initial runs on $o_4' = o_1 \cdot o_2 \cdot o_3 \cdot o_4$, which are then separately fed into the three intermediate sort operations s_2', s_3' and s_4'. Finally, s_1 and s_2 are derived from s_2', s_3 is derived from s_3' and s_4 is derived from s_4'. N_i $(i \in \{1,2,3\})$ is the number of chunks of s_i'. The total cost of this execution plan is

$$2 \times B \times (1 + 3 \times \lceil log_F \frac{B}{2M} \rceil + \lceil log_F N_1 \rceil + \lceil log_F N_2 \rceil + \lceil log_F N_3 \rceil) + 3 \times C_{is} + C_{s_2' \to s_1} \tag{10}$$

The alternative execution plan that utilizes binary cooperative sorting is depicted in Fig. 19. In this plan, s_a' is the intermediate sort operation for the cooperative sorting between s_3 and s_4 where N_4 is the number of chunks of s_a'. s_1 and s_2 are still derived from the intermediate sort operation s_2'. The total cost of this plan is

$$2 \times B \times (2 + 2 \times \lceil log_F \frac{B}{2M} \rceil + \lceil log_F N_1 \rceil + \lceil log_F N_4 \rceil) + 2 \times C_{is} + C_{s_2' \to s_1} + C_{s_a' \to s_3} \tag{11}$$

where $C_{s_a' \to s_3}$ is the cost of internally sorting composite chunks of s_a' to derive s_3.

The difference obtained by subtracting Eqn. 11 from Eqn. 10 is

$$2 \times B \times (\lceil log_F \frac{B}{2M} \rceil + \lceil log_F N_2 \rceil + \lceil log_F N_3 \rceil - \lceil log_F N_4 \rceil - 1) + C_{is} - C_{s_a' \to s_3} \tag{12}$$

First of all, we assume that the value of $|C_{is} - C_{s_a' \to s_3}|$ is negligible compared to the dominant I/O cost.

Note that each o_{31}-*segment* of s_a' consists of one or multiple o_{41}'-*segments* of s_4'. With this constraint, the maximum possible value of N_4/N_3 is achieved when all chunks of s_a' and s_4' are composite. In this case, $N_4 = \frac{2*B}{M}$ (the upper bound of total number of chunks possible) and $N_3 = \frac{B}{M}$ (the lower bound of the total number of chunks possible). Since the merge order F is at least 2, $\lceil log_F N_4 \rceil - \lceil log_F N_3 \rceil \leq 1$.

Therefore, the minimum value of Eqn. 12 is $2 \times B \times (\lceil log_F \frac{B}{2M} \rceil + \lceil log_F N_2 \rceil - 2)$, which is always non-negative. This means that 4-way cooperative sorting is no cheaper than its alternative realizations using 2-way cooperative sorting.

Analysis of ***k*****-way cooperative sorting**. The generalized execution plan of k-way cooperative sorting as well as the alternative plan with cooperative sorting are depicted in Fig. 20 and Fig. 21, respectively. In Fig. 21, sa_i is the intermediate sort operation for the cooperative sorting between s_i and s_{i+1}.

As shown, the plan in Fig. 20 is composed of three parts: part 1 represents equivalently a 2-way cooperative sorting between s_1 and s_2; part 2 is the derivation of s_3 to s_{k-1} (or s_k, if k is even) from their corresponding intermediate sort operations; part 3 contains the derivation of s_k if k is odd. Both part 2 and part 3 are probably but always exclusively empty.

Similarly, the plan in Fig. 21 also consists of three parts: part 1 is a 2-way cooperative sorting between s_1 and s_2; part 2 contains $(k-2)/2$ 2-way cooerpative sortings to derive s_3 to s_{k-1} (or s_k, if k is even), each of which is between s_i and s_{i+1}; part 3 is a normal external sorting s_k if k is odd. Both part 2 and part 3 are probably but always exclusively empty.

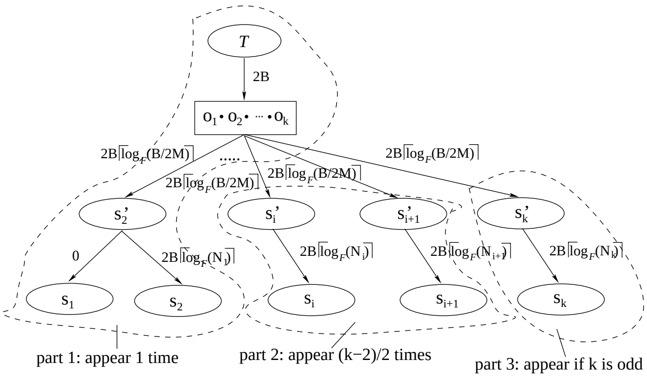


Fig. 20 The Execution Plan of k-way Cooperative Sorting

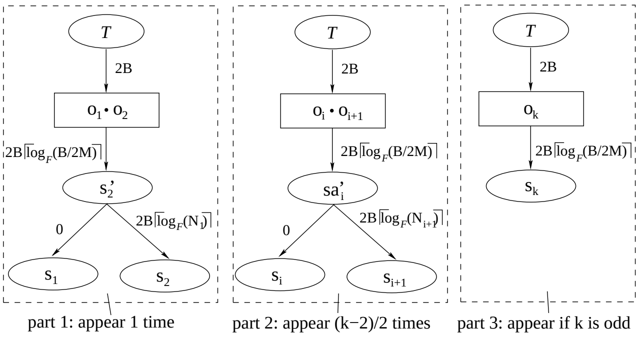


Fig. 21 The Alternative Execution Plan of 2-way Cooperative Sorting

First of all, the cost of part 1 in both figures are equal. Note that the cost difference between part 3 in Fig. 20 and in Fig. 21 is exactly the same as the difference between Eqn. 8 and Eqn. 9 in the analysis of 3-way cooperative sorting, which is always non-negative. Also observe that for each pair of s_i and s_{i+1} that are generated in part 2 of both Fig. 20 and Fig. 21, the cost difference of deriving them between the former figure and the latter is actually the same as the difference between Eqn. 10 and Eqn. 11, i.e. Eqn. 12, in the analysis of 4-way cooperative sorting, which is always non-negative. As a result, the cost of part i ($i \in \{1,2,3\}$) in Fig. 21 is no higher than part i in Fig. 20. Therefore, it is easy to deduce that in general, k-way cooperative sorting ($k \geq 3$) is not more efficient compared to their equivalent realizations using 2-way cooperative sorting.

B Incorporate Sort Sharing Optimization In Volcano-style Optimizers

In Section 5.3, we have discussed how to integrate the sort sharing optimization into a system-R style query optimizer. In this section, we describe how to make a Volcano [13] style query optimizer to be sort-sharing aware.

The Volcano method is based on an AND-OR DAG representation [23], [13] to compactly represent alternative query plans. The optimizer traverses the DAG expanded by applying all possible algebraic transformation rules on every node to search for the cheapest plan. In the AND-OR DAG, we use $\texttt{AN}(op)$ to denote an AND-node according to an operation op; use $\texttt{ON}(e,P)$ to denote an OR-node according to a logical expression e and an optional interesting physical property set P. Normally, the *enforcer* operations (e.g., hashing and sorting) are implicitly represented by their caller AND-nodes.

Given a query, we generate with the traditional method the fully expanded AND-OR DAG, on which we subsequently apply modifications.

First of all, we treat sorting as if it is a logical algebraic operation. As a result, in the DAG, for each enforcer sort operation $s = sort(T,o)$, we add a new AND-node $\texttt{AN}(s)$ and a new OR-node $\texttt{ON}(T,\{o\})$. $\texttt{AN}(s)$ corresponds to the physical sort operation s, and $\texttt{ON}(T,\{o\})$ corresponds to the sorted T with order o. Suppose the caller AND-node of s is $\texttt{AN}(c)$, then $\texttt{ON}(T,\{\})$ is originally one child of $\texttt{AN}(c)$. Now, this chain $\texttt{AN}(c) \rightarrow \texttt{ON}(T,\{\})$ in the DAG is replaced with a new chain $\texttt{AN}(c) \rightarrow \texttt{ON}(T,\{o\}) \rightarrow \texttt{AN}(s) \rightarrow \texttt{ON}(T,\{\})$.

We then model the sort sharing between two sortings $s_1 = sort(T,o_1)$ and $s_2 = sort(T,o_2)$ in above partially modified DAG. For case 1, we add a new AND-node $\texttt{AN}(ds)$ to form a new chain $\texttt{ON}(T,\{o_2\}) \rightarrow \texttt{AN}(ds) \rightarrow \texttt{ON}(T,\{o_1\})$, where ds represents the dummy operation of deriving s_2 from s_1. For case 2, we add a new AND-node $\texttt{AN}(ps)$ to form a new chain $\texttt{ON}(T,\{o_2\}) \rightarrow \texttt{AN}(ps) \rightarrow \texttt{ON}(T,\{o_1\})$, where ps represents the partial sort operation of deriving s_2 from s_1.

For case 3, we add three new AND-nodes, $\texttt{AN}(s_{12})$, $\texttt{AN}(s_{12} \twoheadrightarrow s_1)$ and $\texttt{AN}(s_{12} \twoheadrightarrow s_2)$, as well as a new OR-node $\texttt{ON}(T,\{o_1 \uplus o_2\})$, to form two new chains $\texttt{ON}(T,\{o_1\}) \rightarrow \texttt{AN}(s_{12} \twoheadrightarrow s_1) \rightarrow \texttt{ON}(T,\{o_1 \uplus o_2\}) \rightarrow \texttt{AN}(s_{12}) \rightarrow \texttt{ON}(T,\{\})$ and $\texttt{ON}(T,\{o_2\}) \rightarrow \texttt{AN}(s_{12} \twoheadrightarrow s_2) \rightarrow \texttt{ON}(T,\{o_1 \uplus o_2\}) \rightarrow \texttt{AN}(s_{12}) \rightarrow \texttt{ON}(T,\{\})$. Here s_{12} is the cooperative sorting based on (o_1, o_2); $s_{12} \twoheadrightarrow s_1$ is the operation of deriving s_1 from s_{12}; $s_{12} \twoheadrightarrow s_2$ is the operation of deriving s_2 from s_{12}; $o_1 \uplus o_2$ denotes the hybrid output format of a cooperative sorting s_{12} for s_1 and s_2. The processings for case 4 are straightforward extensions of case 3 and thus omitted.

Till now, we get a completely modified AND-OR DAG. In this DAG, it is possible that a sorting or cooperative sorting OR-node may have more than one parent AND-node. Such OR-nodes can be viewed as unified common subexpressions, and their sort results are materialized and reusable. Therefore, the multiple query optimization (MQO) techniques (e.g., [24]) can be utilized to find the optimal sort-sharing-aware execution plan.