OPTIMIZING COMPLEX QUERIES WITH MULTIPLE RELATIONAL INSTANCES

YU CAO

(B.Sc. University of Science and Technology of China)

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

SCHOOL OF COMPUTING

NATIONAL UNIVERSITY OF SINGAPORE

2011

Acknowledgement

I would like to express my very deep appreciation to many people, without whom this thesis would not have happened.

Prof. Tan Kian-Lee and Prof. Chan Chee-Yong are great supervisors and I am mostly indebted to them. During the last few years, they have been patient in guiding and supporting me. I am really grateful that they never push me hard for research achievements and always try their best to relieve my mental stress and convince me that I can graduate. It is their encouragement that drives me to the end. Their insights in database research keep me walking on the right way, and their heuristic guidance in our discussion makes me think and work very independently. They have taught me many things about how to become a good researcher as well as a good person with kindness and wisdom.

Thanks to Gopal Das, Bramandia Ramadhana and Zhou Yongluan, who worked closely with me on various papers. Their participation accelerated the work progress, enriched the technical content and improved the paper presentation. Their help eased the burden on my back to much extent.

Thanks to Prof. Ooi Beng Chin, who provided me the position of research assistant for a whole year.

Thanks to members of my evaluation committees: Prof. Stephane Bressan, Prof. Panos Kalnis, Prof. Pang Hwee Hwa and the anonymous external thesis examiner. They provided me valuable feedback to refine my research work at different stages. I also want to thank other professors in our database group, especially Prof. Ling Tok Wang who invoked my initial interest in database research, and Prof. Anthony Tung, a semiprofessional solo singer well recognized around, who made my Ph.D life more entertaining.

Thanks to many friends I have made during my years at NUS. Because of the memorable friendship between us, my Ph.D life became more enjoyable. They are Bao Zhifeng, Cao Jianneng, Chen Ding, Chen Su, Chen Yueguo, Dai Bingtian, Li Feng, Li Yingguang, Liu Chen, Liu Xuan, Lin Yuting, Lu Meiyu, Lu Peng, Meduri Venkata Vamsikrishna, Shi Lei, Su Shan, Sun Yang, Vo Hoang Tam, Wang Nan, Wang Tao, Wang Xianjun, Wang Xiaoli, Wu Huayu, Wu Ji, Wu Sai, Wu Wei, Xiang Shili, Xu Liang, Xu Linhao, Yang Fei, Yang Xiaoyan, Ying Shanshan, Zhang Dongxiang, Zhang Jingbo, Zhang Zhenjie, Zhao Feng and many others.

My parents always respect my choices and decisions, and never try to impose their belief on me. I am entirely grateful for that. Their love is the most precious treasure I own.

CONTENTS

Abstract

It is not uncommon that analytical database queries contain multiple *instances* of the same (base or derived) relation. Unfortunately, almost all of the conventional relational query processing techniques are oblivious to these instances and instead deal with them as independent relations. As a result, the query evaluation performance would be suboptimal.

This thesis studies the problem of optimizing complex queries with multiple relational instances. Specifically, we investigate three fundamental query execution operations, i.e. table scan, table sorting and table join, to exploit the corresponding optimization opportunities when these operations involve multiple instances. Our contributions are summarized as follows.

First, we present a light-weight multi-instance-aware plan evaluation engine that enables multiple instances of a relation to share one physical table scan. This evaluation engine utilizes a novel interleaved pull iterative execution strategy, which interleaves the query processing between normal processing and resolving blocked shared scans. Our method demonstrates the feasibility and efficiency of a clustered table access strategy for the instances within a single query.

Second, we develop a sort-sharing-aware query processing framework, which consists of a series of useful techniques ranging from query optimization to query execution. It turns out that sorting a table multiple times takes place frequently in many applications, such as building various indexes over the table and business intelligence reporting. With this framework, we are able to maximize the effects of sharing and collaboration during achieving different sorting requirements for multiple instances.

Third, we propose an efficient algorithm for performing self-join operations between two instances and with join predicates involving two distinct instances. This type of self-joins occur often in many traditional as well as recently emerging database applications, such as location-based service (LBS), RFID data management, sensor networks. Our algorithm is generally superior to classical join algorithms like Sort-Merge Join, Hybrid Hash Join and Nested-Loop Join.

Finally, we have implemented our instance-conscious query processing techniques in PostgreSQL, a widely known and deployed open-source object-relational DBMS. Our extensive experimental study shows significant performance improvements over the traditional instance-oblivious evaluation schemes.

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

Relational databases are currently the predominant choice for data storage, such as storing financial records, medical records, manufacturing and logistical information and personnel data. As such, a relational database management system (RDBMS), which manages a set of relational databases, has become a backend component of almost any modern application stack. Consequentially, RDBMS product manufactures such as Oracle, IBM and Microsoft, are all among the largest and most successful software firms around the world, together sharing a multi-billion dollar market.

The huge success of relational databases is significantly attributed to Codd's relational data model [17], which provides a declarative method for specifying data and queries: users directly state what data (in the form of relations) the database stores, manipulate (insert, delete and update) and query the data through a data manipulation language like SQL; the DBMS, managed and tuned by the database administrator, takes care of describing formats for storing the data and retrieval procedures for getting queries answered.

Historically, database systems mainly focused on transactional data processing. Transactions are composed of simple, repetitive and short running action queries. For performance reasons, a DBMS has to interleave the actions of several transactions. Therefore, the major challenge of the DBMS was ensuring the ACID proprieties of transactions to maintain data in the face of concurrent access and system failures. Later on, however, organizations have increasingly emphasized applications in which current and historical data are comprehensively analyzed and explored, identifying useful trends and creating summaries of the data, in order to support high-level decision making. Consequently, two new types of database systems, data warehouses and decision support systems, are being created and maintained to process analytical queries. These queries usually contain many complex query conditions over multiple tables, process large amounts of data and thus run for a long time. Moreover, these queries are often ad-hoc and exploratory, motivated by the desire to find interesting or unexpected trends and patterns in large data sets. As such, the database system faces the challenge of efficiently answering users' complex analytical queries. This challenge has spurred more than thirty years of query processing research, pioneered by Selinger et al. [56] in System R and refined by generations of database researchers and developers. Nowadays, database systems have been tremendously effective in addressing the needs of analytical query processing. However, the existing database techniques are still far from perfect and will doubtless continue to be further improved, with remaining tough research problems (e.g. adaptive query processing [20]), newly emerging research challenges (e.g. database usability [38] and new hardware platforms such as chip multiprocessors and solid state disks), as well as other undiscovered important research areas.

1.1 Thesis Motivation

In this thesis, we investigate the problem of efficient processing of queries with *relational instances*, which are the multiple occurrences of the same (base or derived) relation within a single query.

Consider the TPC-D(ecision)S(upport) benchmark [3] query Q90 below. It contains two sub-queries in the from-list of the main query block, both of which operate on the same set of relations: *web_sales*, *household_demographics*, *time_dim* and *web_page*. Taken as a whole, each distinct relation has two instances in this Q90.

```
SELECT amc/pmc as am_pm_ratio
FROM ( SELECT count(*) as amc
       FROM web_sales, household_demographics, time_dim, web_page
       WHERE ws_sold_time_sk = t_time_sk and ws_ship_hdemo_sk = hd_demo_sk
         and ws_web_page_sk = wp_web_page_sk and t_hour between 8 and 8+1
         and hd_dep_count = 6 and wp_char_count between 5000 and 5200) at,
       ( SELECT count(*) as pmc
       FROM web_sales, household_demographics, time_dim, web_page
       WHERE ws_sold_time_sk = t_time_sk and ws_ship_hdemo_sk = hd_demo_sk
         and ws_web_page_sk = wp_web_page_sk and t_hour between 19 and 19+1
         and hd_dep_count = 6 and wp_char_count between 5000 and 5200) pt;
```

In many database applications, it is not uncommon for a single complex analytical query to contain relations with multiple instances. For instance, among the 99 queries in the TPC-DS benchmark, more than 60% of them contain at least one relation with multiple instances; the maximum number of instances for a relation is 8 (e.g., Q11 and Q88) and the maximum number of relations with multiple instances is 15 (e.g., Q78). The reasons for the prevalence of relational instances are manifold. Complex queries often involve correlated nested subqueries with aggregation functions. Correlation refers

to the use of values from the outer query block to compute the inner subquery. Between a subquery and the outer query and/or between subqueries, a non-empty set of common relations are usually shared. Complex queries (e.g. the above Q90) also frequently contain a lot of common or similar sub-expressions due to the extensive use of relational views. Either materialized or expanded into the query at runtime, the views introduces multiple instances of the materialized results or base tables. As another scenario, relational instances appear in queries representing set operations to establish a relationship between results from several subqueries, such as UNION, INTERSECT and EXCEPT.

Moreover, self-join, a join operation that relates data within a relation by joining the relation with itself, is extensively utilized in many applications. For example, 6 queries in TPC-DS involve self-joins. When RDF data are managed as a triple table in relational DBMS, SPARQL queries are often mapped to relational queries with many self-joins that relate the subjects and objects [5]. Yet another application where self-joins occur frequently is the publication of relational data as XML; here, XML views are defined over the underlying relational data and XML queries (e.g. in XQuery) over the views are translated into self-join queries on the underlying table [57]. Moreover, self-joins occur often in many recently emerging database applications, such as location-based service (LBS), RFID data management, sensor networks, network management.

It is surprising that at least in the public domain there have never been systematic or specialized studies of query processing with relational instances. As a result, despite the frequent relational instances encountered, most of today's relational query engines do not explicitly recognize them within queries during query optimization and/or evaluation. Instead, each instance is treated as a distinct relation.

If a database system is oblivious of multiple instances, a large portion of the total query expense will be wasted when queries contain instances of big relations. The observation lies in the concerns on two components of the query processing cost. On the

one hand, data of a multi-instance relation are repeatedly fetched from disk for each of its instances due to system buffer trashing; later on, many common data are materialized to disk and then retrieved back to memory as intermediate results of query processing by different instances. In terms of each table tuple, it could be manipulated multiple times by different instances. Intuitively, this tuple could serve all its host instances by incurring fewer I/O accesses and thus less I/O cost. On the other hand, CPU-intensive operations are also conducted on the data of multi-instance relations, such as tuple selection and projection and join matching. Among them, many actually derive the same information from the same data, which thereby incurs redundant CPU cost.

In this thesis, we try to recognize scenarios where the diverse ways of treating the existing instances would significantly affect the query evaluation costs. Correspondingly, we want to find the optimal solutions by exploiting novel, elegant and efficient multi-instance conscious techniques.

1.2 Thesis Contributions

This thesis studies in-depth three significant research problems about efficient processing of queries with relational instances, which are outlined in the following subsections.

1.2.1 Shared Table Accesses for Relational Instances

Traditionally, each instance has its own independent access method (sequential or index scan). While there have been some efforts to optimize multiple scans on the same table to minimize disk I/O cost, these works are limited in scope. In [1, 18, 36, 42, 43, 69], scans are coordinated for better buffer reuse (increasing buffer locality). In particular, the data-sharing opportunity arises mainly among scans from different queries running at

the same time. The performance improvement is achieved by exhaustively exploiting the knowledge of query access patterns and carefully scheduling query executions. However, for a single query with multiple relational instances, it is not possible to synchronize the disk access patterns under the pull iterative execution model [31]. As such, the execution of a single multi-instance query do not benefit much from these buffer reuse methods. Works in [19, 66] look at facilitating sharing of a single scan on the base relations at the operator level. However, these works are targeted at pipelining table tuples to consumers in different SQL [19] (OLAP [66]) queries handled by independent threads. Instances within a single query have, as we shall see, certain characteristics that these methods fail to accommodate. Yet another approach is to employ multi-query optimization (MQO) schemes (e.g., [54, 68]) to exploit common subexpressions in queries. However, MQO does not further optimize multiple scans on the materialized views of common subexpressions, which can be considered as base relations with multiple instances. Moreover, these techniques do not handle instances that are not part of the common subexpressions. As such, the performance can be very bad even for an optimal plan especially when the relation with multiple occurrences is a large table.

In this work, we develop MAPLE, a *M*ulti-instance-*A*ware *PL*an *E*valuation engine that enables multiple instances of a relation to share one physical scan (called *SharedScan*) with limited buffer space. During execution, as *SharedScan* pulls a tuple for *any* instance, that tuple is also pushed to the buffers of other instances with matching predicates. To avoid buffer overflow, a novel *interleaved* execution strategy is proposed: whenever an instance's buffer becomes full, the execution is temporarily switched to a *drainer* (an ancestor blocking operator of the instance) to consume all the tuples in the buffer. Thus, the execution is interleaved between normal processing and drainers. We also propose a cost-based approach to generate a plan to maximize the shared scan benefit as well as to avoid interleaved execution deadlocks. MAPLE is light-weight and can

be easily integrated into existing RDBMS executors. This work has been published in SIGMOD 2008 [13].

1.2.2 Collaborative Executions of Sortings of Relational Instances

For complex decision support queries with multiple relational instances, the optimized execution plans may apply various sort operations to different instances of the same relation, usually in the association with sort-merge joins. Besides, it also turns out that such multiple sortings of a table is not uncommon in many other applications. For example, in data warehousing, a fact table typically has two types of columns: those that contain facts and those that are foreign keys to dimension tables. It is often useful to create both 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 in different sort orders. A bank may produce reports ordered by amount deposited/withdrawn/balance, date, branch, and so on. Similarly, examination schedules are usually printed in different orders - order by course number, dates, examiners, and invigilators.

In this work, we study the generalized problem on how to accomplish multiple sortings of a table more efficiently than the straightforward yet wasteful approach of one separate sorting per sort order. 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 extensi-

ble so that it will not miss the truly cheapest execution plan with the sort sharing (post-) optimization turned on. This work has been published in ICDE 2010 [11]. A more comprehensive description is to be published in the VLDB Journal [12].

1.2.3 Optimizing Self-Joins Between Relational Instances

Despite the importance and prevalence of self-joins, there however have been surprisingly few research efforts on optimizing them. On the one hand, existing solutions either employ join indexes [61] or handle the special case where the join attributes are on the same attribute (e.g., $R_1.A = R_2.A$) [16, 27]. As one can see, many emerging queries involve self joins on two distinct attributes. While index-based techniques could be applied to the problem, it is possible that indexes do not exist, especially when the queries are ad-hoc and/or the join attributes are derived ones computed from user defined functions. Even when indexes exist, they may not be used. For example, if the join selectivity is high (i.e. a lot of join results), then indexes, especially the non-clustered ones, are not beneficial. On the other hand, conventional join algorithms, such as Sort-Merge Join (`SMJ`) and Hybrid Hash Join (`HHJ`), treat the two instances of the same relation as distinct relations. As such, they miss the opportunities to enhance the processing performance, particularly in keeping the I/O cost low.

In this work, we present `SCALE` (Sort for Clustered Access with Lazy Evaluation), an efficient general self-join algorithm, which takes advantage of the fact that both inputs of a self-join operation are instances of the same relation. `SCALE` first sorts the relation on one join attribute, say $R.A$. In this way, for every value of the other join attribute, say $R.B$, its matching $R.A$ tuples are essentially clustered. As `SCALE` scans the sorted relation, join results of tuples whose $R.B$ values can be fully or partially matched in memory are produced immediately. For tuples where full-range clustered accesses to their matching tuples are not possible (e.g., matching tuples may not be in memory),

they are buffered (and possibly spilled to disk) and the unfinished part of join processing deferred. Such lazy evaluation minimizes the need for "random" access to the matching tuples. SCALE further optimizes the memory allocation for clustered access and lazy evaluation to keep the processing cost minimal. Our analytical study shows that SCALE degenerates gracefully to a Sort-Merge Join in the worst case.

1.2.4 Prototype System Development

The research on relational database query processing has a long history of over three decades and its academic results have been highly commercialized. Therefore, new academic findings in this field need to be very solid and systematic in order for acceptance and adoption. To this end, in all of the research works, we validate our techniques by integrating them into an open-source database system PostgreSQL [2] and testing their effectiveness using TPC [4] benchmarks. PostgreSQL is a powerful object-relational database system and is widely utilized by organizations and single users. TPC is also well-known in database industry and provides various benchmarks to deliver trusted results to the industry for their new techniques and products. The performance results derived from evaluations at the system level verify that our proposed techniques can practically bring significant performance improvements over the existing approaches.

1.3 Thesis Organization

The rest of the thesis is structured as follows.

Chapter 2 describes MAPLE, the multi-instance-aware plan evaluation engine that enables multiple instances of a relation to share one physical scan. It first presents an overview of MAPLE, which comprises two key components: a *shared scan post-optimizer* (SSPO) and an *interleaved iterative query evaluator* (IIQE). It then explains

how SSPO builds on a query plan by a conventional optimizer to produce an enhanced plan that supports shared scans and interleaved operator executions. It also illustrates how the IIQE can be implemented by making only moderate modifications to the conventional iterator query execution engine.

Chapter 3 elaborates the integration of sort sharing optimization into both query optimization and evaluation. It formally discusses the two sort sharing techniques, result sharing and cooperative sorting, between two instance sortings. It generalizes cooperative sorting to evaluate more than two sort operations, explains how to optimize the evaluation of multiple sortings on a relation, and discusses sort-sharing-aware query optimization.

Chapter 4 discusses the efficient self-join processing with our proposed SCALE algorithm. It presents the technical details of the SCALE algorithm and then presents a thorough analytical study. It also proposes further optimizations and extensions of SCALE.

Along with each individual work, we provide its specific background and related work in the resident chapter. Finally, Chapter 5 concludes the thesis and points out some directions for future work.

CHAPTER 2

Shared Table Scans for Relational Instances

2.1 Introduction

This chapter examines the optimization problem of reducing the total I/O cost incurred by multiple instances in a query in order to access the common underlying table on the disk. While there have been some efforts to optimize multiple scans on the same table, the effects of proposed approaches are limited in our problem context, according to the following analysis.

In [1, 18, 36, 42, 43, 69], scans are coordinated for better buffer reuse (increasing buffer locality). In particular, the data-sharing opportunity arises mainly among scans from different queries running at the same time. The performance improvement is achieved by exhaustively exploiting the knowledge of query access patterns and carefully scheduling query executions. However, for a single query with multiple relational

instances, it is not possible to synchronize the disk access patterns under the pull iterative execution model [31]. As such, single multi-instance queries do not benefit much from these buffer reuse methods. Works in [19, 66] look at facilitating sharing of a single scan on the base relations at the operator level. However, these works are targeted at pipelining table tuples to consumers in different SQL [19] (OLAP [66]) queries handled by independent threads. Instances within a single query have, as we shall see, certain characteristics that these methods fail to accommodate. Yet another approach is to employ multi-query optimization (MQO) schemes (e.g., [54, 68]) to exploit common subexpressions in queries. However, MQO does not further optimize multiple scans on the materialized views of common subexpressions, which can be considered as base relations with multiple instances. Moreover, these techniques do not handle instances that are not part of the common subexpressions.

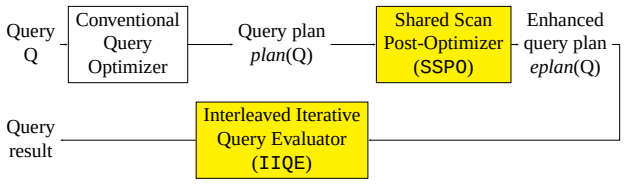


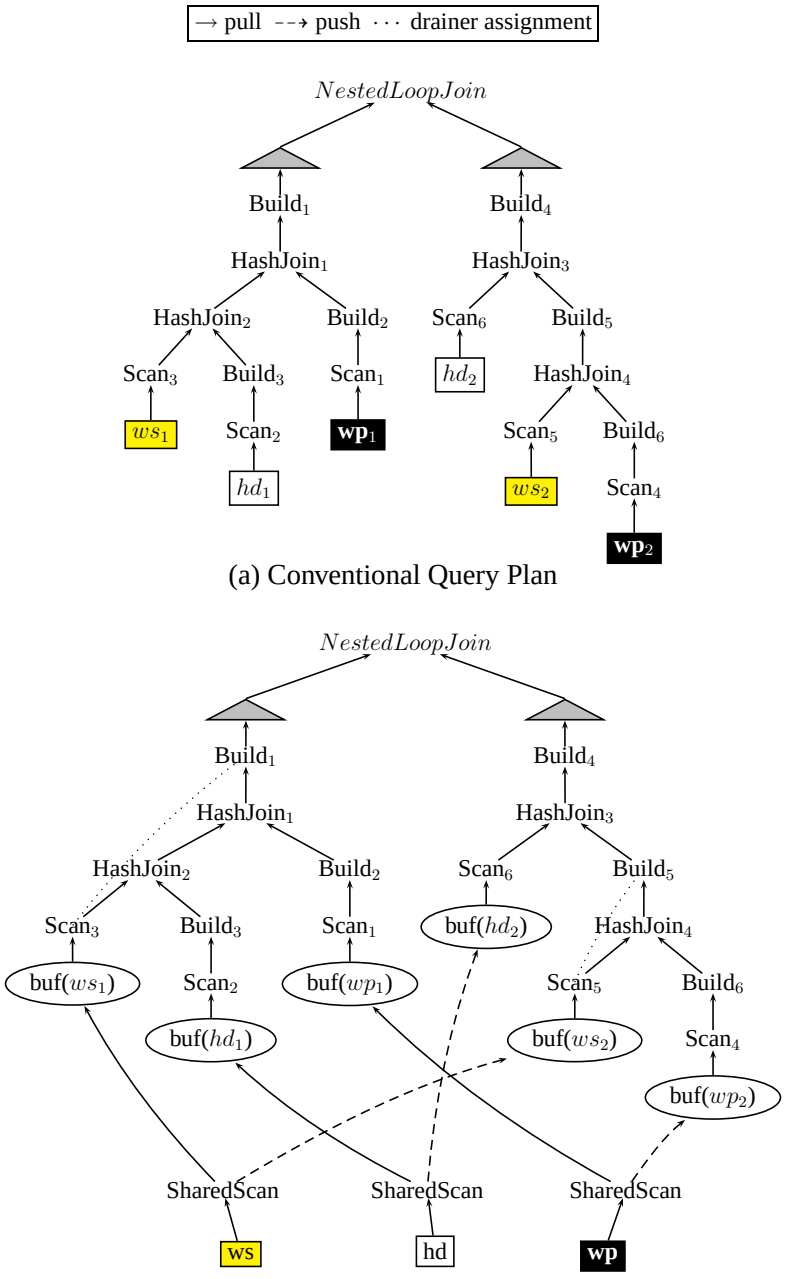
Figure 2.1: Architecture of MAPLE

In this chapter, we present MAPLE, a *M*ulti-instance-*A*ware *PL*an *E*valuation engine that takes advantage of multiple instances in single queries to reduce disk I/O cost. MAPLE comprises two key components (SSPO and IIQE) as shown in Fig. 2.1. First, a *shared scan post-optimizer (*SSPO*)* builds on a query evaluation plan (generated by any existing query optimizer) to produce an enhanced plan as follows. The SSPO opportunistically adds new materialize operators when required and bundles multiple instances of a relation into *share groups* such that instances within a group share one physical table scan (called SharedScan). For each instance of a relation that employs a SharedScan operator, it is allocated a *small* buffer. Moreover, for each instance with buffer overflow risk, an ancestor (blocking) operator in the query plan will be designated as its

drainer. Second, an *interleaved iterative query evaluator* (`IIQE`) is used to execute the enhanced query plan produced by `SSPO`. `IIQE` adopts an *interleaved* pull iterative execution strategy to ensure that each `SharedScan` operator scans the table *only once* (for all instances within the same share group). Essentially, within a share group, as `SharedScan` pulls a tuple for *any* instance, that tuple is also *pushed* to other instances with matching predicates and placed in their buffers for later use. Whenever a buffer becomes full, the corresponding drainer becomes *active*. At this moment, query processing is temporarily switched to this drainer until it consumes all tuples in the buffer. Thus, query processing is interleaved between normal processing and active drainers.

Example 1 Fig. 2.2(a) shows the partial evaluation plan of Q90 in TPC-DS benchmark, generated by PostgreSQL [2]. Q90 contains two instances ws_1 and ws_2 for relation web_sales (denoted by ws), two instances wp_1 and wp_2 for relation web_page (denoted by wp), and two instances hd_1 and hd_2 for relation household_demographics (denoted by hd). Here the hash operator `Build` is used to build hash table in hash join. The plan tree contains one hash subtree in each side of the top nested-loop join and all instances are accessed by table scans.

`MAPLE` generates an enhanced plan, shown in Fig. 2.2(b), with three share groups: $\{ws_1, ws_2\}$, $\{wp_1, wp_2\}$ and $\{hd_1$,$hd_2\}$. No additional materialize operators are introduced. Each relation instance r_i is now associated with a buffer $buf(r_i)$ for storing the tuples pushed by the `SharedScan` operator. Under the iterative model, the execution starts from $\texttt{Build}_1$. Since both wp and hd are small tables, the shared scans on them did not incur buffer overflows in wp_2 and hd_2. However, when ws_1 calls its `SharedScan`, matching tuples pushed to ws_2 will fill up its buffer since ws is a very large table. Now, whenever $buf(ws_2)$ becomes full, the execution temporarily switches to $\texttt{Build}_5$, ws_2's drainer, which consumes all tuples in the buffer to partially construct the hash table, and then switches back to ws_1. The switched execution for ws_2 will complete the nor-



(a) Conventional Query Plan

(b) MAPLE's Enhanced Query Plan

Figure 2.2: Partial Query Evaluation Plans for Query Q90 in TPC-DS Benchmark

mal execution of $\texttt{Build}_6$ using cached tuples in $buf(wp_2)$. Finally, as all three shared scans finish, the remaining execution continues as in the traditional iterative model from $\texttt{Build}_4$ (which completes the execution of $\texttt{Build}_5$ and then conducts the hash join by

probing the hash table with the cached tuples in $buf(hd_2)$).

As illustrated, by using MAPLE, one share group reads the relation only once from the disk. In this example, we save one full scan on each ws, wp and hd. Our experimental results show significant benefit from the saving of one scan of ws since it is huge (1.5GB in 10GB TPC-DS dataset). On the contrary, the CPU overhead of execution switches is negligible. Intermediate results of execution switches are naturally consumed by the Build drainers without incurring additional I/O overhead. □

The key task of SSPO is to generate an enhanced plan that maximizes the benefits of SharedScan. Ideally, all instances of a relation should be grouped within a single share group without introducing any additional blocking operators. However, it turns out that this is not always possible due to several reasons (e.g., interleaved execution deadlocks). In this case, SSPO aims at finding a feasible shareable scan plan with maximum performance benefit.

MAPLE is light-weight and can be easily integrated into existing RDBMSs. We have prototyped our ideas in PostgreSQL. Our extensive performance study on the TPC-DS benchmark shows very significant reduction in execution time of up to 70% for some queries.

The rest of this chapter is organized as follows. In Section 2.2, we present an overview of our MAPLE approach. Section 2.3 describes the shared scan post-optimizer. In Section 2.4, we present how to integrate IIQE into existing query executors. Section 2.5 presents results of an extensive performance study. Section 2.6 reviews related work, and finally, Section 2.7 concludes the chapter.

2.2 Overview of MAPLE

In this section, we present an overview of our light-weight optimization approach named MAPLE.

We use *plan*(Q) to denote a query evaluation plan for Q generated by a conventional query optimizer, and use *eplan*(Q) to denote an enhanced query evaluation plan for Q produced by MAPLE based on *plan*(Q).

A query plan operator is classified as a *blocking operator* if it needs to completely consume its operand(s) before producing any output (e.g., sorting, building hash table, aggregation); otherwise, it is a *non-blocking operator* (e.g., scan, merge-join).

For a multi-instance relation R in Q, we use $G = \{r_1, r_2, \cdots, r_n\}$, $n > 1$ to denote its instances.

2.2.1 Share Groups & Shared Scans

In contrast to the conventional pull-iterative execution engine [31], where the scans of instances of the same relation are performed independently, MAPLE tries to maximize the sharing of relation scans by partitioning the set of instances of a relation into a small number of subsets called *share groups*. Each relation instance r_i in a share group is allocated some small memory space, denoted by $buf(r_i)$, to hold the qualified tuples that satisfied the selection predicates for the scan of r_i. Each share group is associated with a new scan operator called the SharedScan operator[1] that can be invoked by any instance in that group. When a scan of an instance r_i is invoked, MAPLE will first check whether $buf(r_i)$ is empty. If a tuple is available in $buf(r_i)$, the scan of r_i will simply remove this tuple from $buf(r_i)$ and pass it to the scan's parent operator. However, if $buf(r_i)$ is empty, the scan of r_i will invoke the SharedScan operator for its share group. Besides pulling the qualified tuples for r_i into $buf(r_i)$, the SharedScan opera-

[1] Currently, MAPLE considers shared scans only for table scans.

tor will also push qualified tuples for other instances r_j within the share group into their buffers $buf(r_j)$ as well. For space efficiency, the tuples stored in each $buf(r_i)$ only keep the relevant attributes of R for the scan of r_i[2].

In the ideal scenario, the tuples in each $buf(r_i)$ are consumed in a timely manner without causing any buffer overflows. However, in general, a shared scan can become *blocked* when the `SharedScan` operator (invoked by some other instance r_j in the same share group as r_i) tries to push qualified tuples into a full buffer $buf(r_i)$. In this case, we say that r_i is an *overflow instance* and $buf(r_i)$ *overflows*.

A naive approach to fix a blocked shared scan (under the iterative execution model) is to adopt a *drop-out scheme*, where the overflow instance r_i is dropped out of the shared scan of R, and the shared scan of R is allowed to continue among the remaining non-overflow instances of R within the share group. However, this scheme requires a separate partial scan of R to be initiated later to retrieve the remaining non-buffered qualified tuples for the overflow instance r_i, thereby limiting its effectiveness.

Note that if there is only one instance r_i in a group, the scan for r_i is not shared with any other instances of R; therefore, $buf(r_i)$ is not allocated and `SharedScan` is not used for this group.

2.2.2 Interleaved Executions with Drainers

`MAPLE` adopts a more aggressive approach to resolve blocked shared scans. Consider a shared scan invoked by r_i that becomes blocked due to the overflow of $buf(r_j)$. Instead of dropping r_j out of the shared scan of R, `MAPLE` tries to "unblock" the shared scan by suspending the execution of the scan and switching the execution control to another operator, called the *drainer* of r_j, denoted by $drainer(r_j)$. $drainer(r_j)$ is an ancestor

[2] An alternative buffering scheme is to have a single buffer shared among all instances within the share group. But this not only requires storing the entire tuple (in general), but also involves a more elaborate tracking of the tuples that are qualified for each instance scan.

of r_j, whose execution will result in "draining" the tuples from the full buffer $buf(r_j)$. Once all the tuples in $buf(r_j)$ have been consumed (i.e., $buf(r_j)$ becomes empty), the suspended shared scan of R becomes unblocked and can be resumed by r_i. It is possible for nested execution control switches to occur, where the execution of the query subplan under a drainer operator causes another execution control switch to another drainer, and so on. We refer to the enhanced iterative execution model used by MAPLE as *interleaved iterative execution*.

Drainer Operators

When $buf(r_j)$ overflows during a shared scan that is invoked by another instance r_i, MAPLE will try to switch execution to a drainer operator, $drainer(r_j)$, to clear the buffer $buf(r_j)$. Thus, $drainer(r_j)$ must necessarily be an ancestor operator of r_j in the query plan so that the scan of r_j will get evaluated as part of the evaluation of the subquery plan rooted at $drainer(r_j)$.

Consider the scenario where all the ancestor operators of r_j up to and including $drainer(r_j)$ are non-blocking operators. In this case, any tuple produced by the evaluation of $drainer(r_j)$ has to be either cached (possibly incurring disk I/O) or returned to the parent operator of $drainer(r_j)$. The latter option is not possible (under the iterative execution model) since the execution control is passed to $drainer(r_j)$ and not to its parent operator. To avoid incurring unnecessary disk I/O for caching output tuples from $drainer(r_j)$, it makes sense to assign a blocking operator as a drainer. In this way, the evaluation of the blocking drainer will not generate any output tuple until its entire query subplan has been completely evaluated. To minimize the number of operator evaluations for draining $buf(r_j)$, MAPLE chooses the *closest* ancestor blocking operator of r_j as its drainer.

Clearly, a drainer operator does not always exist for an overflow instance. We can

classify an overflow instance as a *drainable instance* if it has an ancestor blocking operator in the query plan; otherwise, the overflow instance is considered to be *non-drainable*.

Since a drainer operator cannot be assigned for a non-drainable instance r_j, it is not possible to drain $buf(r_j)$ (if it becomes full) via an interleaved execution. Thus, non-drainer instances cannot participate in shared scans (i.e, a separate physical scan is necessary for each non-drainable instance). However, a non-drainable instance r_j can be made drainable by inserting an explicit materialize operator op in the query plan such that op becomes an ancestor operator of r_j (i.e., $drainer(r_j) = op$).

Consider the example in Fig. 2.2(b), where ws_1 and ws_2 are assumed to be overflow instances, the drainer assignment for each overflow instance r_j is indicated by a dotted line between $scan(r_j)$ and $drainer(r_j)$.

Deadlock-free Interleaved Execution

To maximize shared scans, an ideal query plan is to have a single share group for each distinct multi-instance relation R that contains all its instances. In this way, only a single physical scan of R is required to scan all its instances. However, this is not always feasible due to two reasons: (1) the existence of non-drainable instances; and (2) the existence of *interleaved execution deadlocks*.

Basically, an interleaved execution deadlock arises whenever an interleaved execution that is triggered to drain a full buffer $buf(r_j)$ eventually leads to more tuples being pushed into $buf(r_j)$. The following example illustrates a simple example of an execution deadlock.

Example 1 Fig. 2.3 shows a self-join between two instances ws_1 and ws_2 of the relation *web_sales* in TPC-DS, where ws_1 is an overflow instance sharing a scan with ws_2. The execution starts with the scan of ws_2. During the scan of ws_2, $buf(ws_1)$ will become full and the execution will be switched to $drainer(ws_1)$, which is the *Sort* operator.

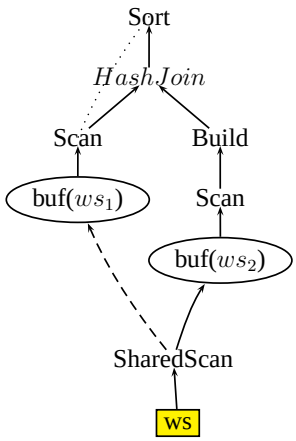


Figure 2.3: Simple Execution Deadlock

However, since the hash table has not been completely constructed yet, before the tuples from ws_1 can be processed, it is necessary to complete the scan of ws_2. But since $buf(ws_1)$ is already full, the execution is deadlocked. □

The following example illustrates a more complex deadlock scenario.

Example 2 Consider again the Q90 query plan in Fig. 2.2(b). Suppose that hd_2 is now an overflow instance. The execution will start with $Build_1$. During the shared scan of hd_1 and hd_2, $buf(hd_2)$ becomes full and the execution switches to $Build_4$, which is the drainer for hd_2. This eventually triggers the execution of the scan of ws_2 and hence a shared scan of ws_1 and ws_2 which results in $buf(ws_1)$ becoming full. Consequently, the execution now switches over to $Build_1$, which is the drainer for ws_1. Here, a deadlock occurs since both $buf(ws_1)$ and $buf(hd_2)$ are full but there are more tuples to be pushed into them. □

To generate a deadlock-free query plan that maximizes shared scans, MAPLE uses a cost-based approach to optimize both the usage of explicit materialize operators as well as the partitioning of share groups. Explicit materialize operators can be used not only to enable non-drainable instances to become drainable (and therefore allowing them to participate in shared scans) but also to avoid deadlock situations.

2.2.3 Architecture of MAPLE

Fig. 2.1 shows the architecture of MAPLE which consists of two components: the shared scan post-optimizer (SSPO) and the interleaved iterative query evaluator (IIQE).

An input query Q is optimized by MAPLE in two steps. First, a conventional query optimizer is used to generate a query evaluation plan (*plan*(Q)). Next, *plan*(Q) is used as input for SSPO to produce an enhanced query plan (*eplan*(Q)). An *eplan*(Q) enhances *plan*(Q) by using share groups, SharedScan operators, and possibly explicit materialize operators.

The generated *eplan*(Q) is then evaluated by the IIQE component which is a variant of the conventional iterative query execution engine enhanced to support shared scans as well as interleaved operator executions.

2.3 Shared Scan Post-Optimizer

In this section, we describe how the *shared scan post-optimizer* (SSPO) component of MAPLE generates an enhanced query plan that supports shared scans and interleaved operator executions.

2.3.1 Overflow Instances

Since SSPO optimizes a query plan statically, it needs to estimate the potential for an instance r_i to overflow and assign a drainer to r_i if necessary. Specifically, for each instance r_i within a share group in the query plan, SSPO uses statistical information on R (to estimate the number of qualified tuples for the scan of r_i) as well as information about the allocated memory space for $buf(r_i)$ to decide whether r_i has the potential to overflow. If the total estimated qualified tuples for r_i cannot fit in $buf(r_i)$, r_i is considered to be an *overflow instance*, and SSPO then assigns $drainer(r_i)$ to be the closest ancestor blocking

operator of r_i if r_i is drainable.

Consider an instance r_i that is determined by SSPO to be a non-overflow instance (i.e., no drainer has been assigned to r_i). If r_i actually overflows at runtime, then MAPLE has no choice but to dynamically materialize the contents of $buf(r_i)$.

2.3.2 Interleaved Execution Deadlocks

In this section, we provide a characterization of interleaved execution deadlocks in terms of *execution dependencies* and *overflow dependencies*.

Execution & Overflow Dependencies

Execution Dependencies. Whenever $buf(r_i)$ overflows during a shared scan and execution control switches to $drainer(r_i)$ which in turn causes the scan of some other relation instance s_j (where s_j is a descendant of $drainer(r_i)$) to be evaluated, we say that there is an *execution dependency* from r_i to s_j (denoted by $r_i \rightarrow s_j$). Here, r_i and s_j can be instances of the same relation or different relations. Note that execution dependencies are transitive: if $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$. Moreover, if $a \rightarrow b$ and $b \rightarrow a$, then both $drainer(a)$ and $drainer(b)$ must be the same.

Overflow Dependencies. Consider two instances r_i and r_j within a share group. If $buf(r_j)$ becomes full during a shared scan invoked by r_i, we say that there is an *overflow dependency* from r_i to r_j (denoted by $r_i \dashrightarrow r_j$).

Instance Dependency Cycles. We can now characterize interleaved execution deadlocks in terms of execution and overflow dependencies. An interleaved execution deadlock occurs when there is an *instance dependency cycle* among a set of relation instances $\{r_1, s_2, t_3, \cdots, z_n\}$, $n > 1$, that consists of an alternating sequence of $\dashrightarrow$ and $\rightarrow$ dependencies of the form $r_1 \dashrightarrow s_2 \rightarrow t_3 \dashrightarrow \cdots \dashrightarrow z_n \rightarrow r_1$.

Observe that in Example 1, there is an instance dependency cycle $ws_2 \dashrightarrow ws_1 \rightarrow$

ws_2; and in Example 2, there is an instance dependency cycle $hd_1 \dashrightarrow hd_2 \rightarrow ws_2 \dashrightarrow ws_1 \rightarrow hd_1$.

Eliminating Dependencies

The above characterization of interleaved execution deadlocks provides two ways to break deadlocks by eliminating overflow or execution dependencies. For an overflow dependency $r_i \dashrightarrow r_j$, which arises when a shared scan for a group containing r_i and r_j causes $buf(r_j)$ to overflow, the overflow dependency can be eliminated by separating r_i and r_j into two different share groups.

For an execution dependency $r_i \rightarrow s_j$, the dependency can be eliminated by introducing a materialize operator *op* into the query plan such that *op* becomes the closest ancestor blocking operator for r_i (i.e., *op* is a descendant of $drainer(r_i)$) and s_j is outside of the query subtree rooted *op*. In this way, $drainer(r_i)$ becomes *op* and the evaluation of this new drainer for r_i will not cause the scan of s_j to be evaluated.

Example 1 Consider once more Example 2 in Fig. 2.2(b), where each distinct relation (i.e., hd, wp, and ws) has a single share group for all its instances, and hd_2 is an overflow instance. There is an execution deadlock in this plan due to the instance dependency cycle $hd_1 \dashrightarrow hd_2 \rightarrow ws_2 \dashrightarrow ws_1 \rightarrow hd_1$. The execution dependency $hd_2 \rightarrow ws_2$ can be eliminated by introducing a materialize operator above $Scan_6$ which will then become the new drainer for hd_2. The overflow dependency $hd_1 \dashrightarrow hd_2$ can be eliminated by separating hd_1 and hd_2 into two separate share groups. □

Deadlock Avoidance

There are two approaches to handle interleaved execution deadlocks. The first is a dynamic approach that detects and breaks instance dependency cycles at run-time to resolve deadlocks. The second is a static approach that avoids deadlocks altogether by

generating and processing only deadlock-free query plans. MAPLE adopts the simpler static approach as it provides a light-weight solution that can be easily integrated into existing query engines. We plan to explore the dynamic approach as part of our future work.

Due to the absence of run-time information on execution and overflow dependencies, the deadlock-free plans generated by a static approach are necessarily more conservative. Specifically, in MAPLE, if a relation instance r_i in a share group G is considered to be an overflow instance, then MAPLE will conservatively assume the following:

- for every other instance r_j in G, there is an overflow dependency $r_j \dashrightarrow r_i$; and
- if r_i is a drainable instance, then for every other instance s_j within the query subtree rooted at $drainer(r_i)$, there is an execution dependency $r_i \rightarrow s_j$.

Given the above conservative assumptions regarding execution and overflow dependencies, we can now generalize the notion of instance execution dependencies to derive a simpler and "higher level" characterization of interleaved execution deadlocks in terms of *group execution dependencies*.

Group Execution Dependencies. Consider two share groups G_1 and G_2. We say that there is a *group execution dependency* from G_1 to G_2, denoted by $G_1 \rightarrow G_2$, if there is an instance x in G_1 and an instance y in G_2 such that $x \rightarrow y$. We refer to x and y as *participants* of the group execution dependency $G_1 \rightarrow G_2$. Note that G_1 and G_2 are not necessarily distinct.

Group Dependency Cycles. We say that there is a *group dependency cycle* among a set of share groups $\{G_1, \cdots, G_n\}$, $n \geq 1$, if there is a cycle of group dependencies $G_1 \rightarrow G_2 \rightarrow \cdots \rightarrow G_n \rightarrow G_1$ such that for each G_i, $i \in [1, n]$, the two participants of the two group execution dependencies involving G_i are distinct.

Example 2 Consider the examples in Fig. 2.4, where instances within the same share

(a) Example 1 (b) Example 2

Figure 2.4: Examples of Group Dependency Cycles

group are boxed and the directed edges between instances represent instance execution dependencies. Fig. 2.4(a) represents the group dependency cycle in Example 1 formed within a single share group G (i.e., $G \rightarrow G$). Fig. 2.4(b) represents the group dependency cycle in Example 2 formed between share groups $G1$ and $G2$. □

Note that each group in a group dependency cycle must be involved in two group execution dependencies. For example, in Fig. 2.4(b), we have $G_1 \rightarrow G_2$ and $G_2 \rightarrow G_1$. Moreover, the two participants in each group must necessarily be distinct; otherwise, it would imply that a shared scan that is invoked by the scan of an instance r_i causes its own buffer $buf(r_i)$ to overflow, which is impossible.

The following results state a useful sufficient condition on deadlock-free interleaved executions based on the absence of group dependency cycles.

Theorem 2.1. *If there are no group dependency cycles in a query plan P, then there are also no instance dependency cycles in P.*

Proof. Based on an instance dependency cycle in P, it is trivial to derive a specific group dependency cycle in P by grouping instances of the same relation in the cycle. □

Corollary 2.2. *If there are no group dependency cycles in a query plan P, then P is free of interleaved execution deadlocks.*

2.3.3 Enhanced Query Plan Optimization

In this section, we describe how SSPO generates an enhanced query plan *eplan*(Q) from the optimal query plan *plan*(Q) produced by a conventional optimizer such that *eplan*(Q) maximizes shared scans without any interleaved execution deadlocks. Specifically, an enhanced plan for *plan*(Q), denoted by *eplan*(Q) = (*plan*(Q), $\mathcal{G}$, $\mathcal{M}$), specifies two additional components:

1. a list of share groups $\mathcal{G} = \{G_1, \cdots, G_k\}$, where each G_i contains a subset of instances from the same relation, $\bigcup_{i=1}^{k} G_i$ is the set of all relation instances in Q, the G_i's in $\mathcal{G}$ are pairwise disjoint. Clearly, $\mathcal{G}$ must contain at least one group for each distinct multi-instance relation in Q, and the maximum number of share groups occurs when each group is a singleton (i.e., without any shared scans).

2. a set (possibly empty) of materialize operators $\mathcal{M} = \{M_1, \cdots, M_n\}$ to be added to *plan*(Q).

Following the discussion in Section 2.3.2, both $\mathcal{G}$ and $\mathcal{M}$ help to eliminate some dependencies, while $\mathcal{M}$ also serves to enable some non-drainable instances to become drainable.

For notational convenience, given an enhanced query plan P, we use $G(P)$ to refer to the share group list component of P, and use $M(P)$ to refer to the materialize operator set component of P.

Cost Model. We now explain the cost model used by SSPO to select an optimal enhanced plan. Let $\mathcal{R} = \{R_1, \cdots R_d\}$ denote the set of distinct multi-instance relations in query Q, and n_i denote the number of instances of R_i. Given the share group list $\mathcal{G}$, let g_i denote the number of groups in $\mathcal{G}$ that have instances of $R_i \in \mathcal{R}$. Thus, each $n_i > 1$ and each $g_i \geq 1$. In Example 1, we have $d = 3$, and $n_i = 2$, $g_i = 1$, $i \in [1, 3]$.

For each $R_i \in \mathcal{R}$, let $scanCost(R_i)$ denote the cost of a single complete scan of R_i. For each $M_i \in \mathcal{M}$, let $matCost(M_i)$ denote the materialization cost of M_i, which includes the cost of writing the intermediate results to disk and the cost of reading them back later. Given $M \subseteq \mathcal{M}$, we define $matCost(M) = \sum_{M_i \in M} matCost(M_i)$.

Let $cost(plan(Q))$ refer to the total cost of scanning each relation instance in *plan*(Q) independently; i.e.,

$$cost(plan(Q)) = \sum_{R_i \in \mathcal{R}} (scanCost(R_i) \times n_i) \tag{2.1}$$

Let $cost(eplan(Q))$ refer to the sum of the total relation scan cost of $\mathcal{G}$ and the total materialization cost of $\mathcal{M}$ incurred by $eplan(Q)$; i.e.,

$$cost(eplan(Q)) = \sum_{R_i \in \mathcal{R}} (scanCost(R_i) \times g_i) + matCost(\mathcal{M}) \tag{2.2}$$

The *benefit* of *eplan*(Q) over *plan*(Q), which measures the savings in the evaluation cost of using *eplan*(Q) instead of *plan*(Q), is given by

$$benefit(eplan(Q)) = cost(plan(Q)) - cost(eplan(Q)) \tag{2.3}$$

Ideal Enhanced Plan. Based on Equations (2.1) to (2.3), the upper bound for $benefit$ is given by $\sum_{R_i \in \mathcal{R}} (scanCost(R_i) \times (n_i - 1))$ which happens when *eplan*(Q) scans each distinct relation exactly once (i.e., there is exactly one share group for each distinct relation), and *eplan*(Q) does not incur any materialization cost (i.e., $\mathcal{M}$ is empty). We refer to such a *eplan*(Q) as an *ideal enhanced query plan*.

We can now state the query optimization problem for SSPO more formally as follows.

Enhanced Plan Optimization Problem. Given an optimal query plan *plan*(Q) produced by a conventional optimizer for a query Q, find an enhanced query plan *eplan*(Q) = (*plan*(Q), $\mathcal{G}$, $\mathcal{M}$) such that *eplan*(Q) is free of interleaved execution deadlocks and benefit(*eplan*(Q)) is maximized.

The above optimization problem is (not surprisingly) a difficult problem as indicated by the following result for a simplified version of the problem.

Theorem 2.3. *Given* plan*(Q) and a set of materialize operators $\mathcal{M}$, the problem of finding a share group list $\mathcal{G}$ such that* eplan*(Q) = (*plan*(Q), $\mathcal{G}$, $\mathcal{M}$) is free of interleaved execution deadlocks and benefit(*eplan*(Q)) is maximized is NP-hard.*

Proof. We first model the problem as an abstract **Vertex Partition Problem** and then prove that it is NP-hard. We construct a directed graph $G = \{V, E\}$. V is the vertex set and $V = \cup_{i=1}^{n} V_i$, where V_i is a subset of vertices and represents all instances of a distinct relation R_i. Each V_i is assigned a positive value b_i representing the value of $scanCost(R_i)$. E is the directed edge set and represents the execution dependencies between instances. As such, a feasible list of share groups $\mathcal{G}$ is equivalent to a valid vertex partitioning $V_i = \cup_{j=1}^{n_i} V_{ij}$, under which there are no directed edge cycles among partitions. The Vertex Partition Problem is to find a valid vertex partitioning which minimizing the score of $B = \sum_{1 \leq i \leq n} b_i(n_i - 1)$.

We show that the Vertex Partition Problem is NP-hard even when every set V_i has exactly 2 vertices. Our proof is based on a polynomial-time reduction from the Vertex Cover Problem (which is an NP-complete problem, see [28]).

Vertex Cover Problem.: Given a undirected graph $H = (U, F)$ and an integer k, can we find a subset $U' \subseteq U$ such that $|U'| \leq k$ and for every $(u, v) \in F$ at least one of u and v belongs to U'?

First, we describe the reduction from the Vertex Cover Problem to the Vertex Partition Problem. Given H, we create a directed graph $G = (V, E)$ as follows. First, $V =$

$\cup_{u \in U} V_u$ where $V_u = \{u, u'\}$ and let $b_u = 1$ for all $u \in U$. Second, $E = \{(u, v'), (v, u') \mid (u, v) \in F\}$. Note that G is an acyclic graph.

We claim that there exists a set cover of H of size k if and only if there exists a valid vertex partitioning of score k.

$\Rightarrow$ **Let U' be a vertex cover of size k, we have a valid vertex partitioning for G of score k.**

For $u \in U'$, we partition V_u into two sets $\{u\}$ and $\{u'\}$. For $u \notin U'$, we retain V_u as one set. It is easy to check that the score of the partition is k. Observe that for $u \in U'$, $\{u\}$ has indegree 0 and $\{u'\}$ has outdegree 0. Hence, there is no cycle passing through $\{u\}$ and $\{u'\}$. For $u \notin U'$, in $\{u, u'\}$ any outgoing edge goes to a $\{v'\}$ with outdegree 0, any incoming edge comes from a $\{w\}$ with indegree 0. Therefore, all paths passing through $\{u, u'\}$ cannot form a directed cycle. Hence, this is a valid partitioning of score k.

$\Leftarrow$ **Given a valid vertex partitioning for G of score k, we can construct a vertex cover of size k.**

Since the score of the valid vertex partition is k, there exists $u_1, \ldots, u_k$ such that V_{u_i} are partitioned into two sets $\{u_i\}$ and $\{u_i'\}$. Define $U' = \{u_1, \ldots, u_k\}$. It can be checked that U' is a vertex cover. □

2.3.4 Optimization Algorithm

Given the hardness of the enhanced plan optimization problem, SSPO uses a heuristic approach that is shown in Algorithm 1.

Consider a query Q consisting of d distinct multi-instance relations $R_1, \cdots, R_d$ with a query plan *plan*(Q). For each instance r_j of each R_i, SSPO first estimates whether r_j is an overflow instance and initializes the drainer for each drainable relation instance, $drainer(r_j)$, to be the closest ancestor blocking operator of r_j (steps 1 to 4).

Next, SSPO checks whether a deadlock-free ideal enhanced query plan exists for *plan*(Q) (steps 5 to 9). Recall that an ideal enhanced query plan has an "ideal" enhancement with an empty set of materialize operators and a share group list given by $\mathcal{G} = \{G_1, \cdots, G_d\}$, where each share group G_i contains all the instances of R_i except for non-drainable instances. If the set of group dependency cycles in $\mathcal{G}$, specified by C, is empty and all the overflow instances in *plan*(Q) are drainable, then the constructed plan P_{opt} is indeed a deadlock-free ideal enhanced plan, in which case SSPO returns P_{opt} and terminates.

If the constructed enhanced plan P_{opt} is not a deadlock-free ideal enhanced plan, SSPO then optimizes P_{opt} by refining its share group list $\mathcal{G}$ and/or adding materialize operators using a two-phases approach. In the first phase (steps 10 to 17), SSPO generates a collection of candidate materialize operator sets. In the second phase (steps 18 to 30), SSPO takes each candidate materialize operator set $\mathcal{M}$ to create a deadlock-free candidate enhanced plan P with $M(P) = \mathcal{M}$ and $G(P) = \mathcal{G}_{opt}$, where $\mathcal{G}_{opt}$ is an optimized refinement of $\mathcal{G}$ (w.r.t. $\mathcal{M}$). Among all the candidate enhanced plans generated, SSPO returns the plan with the maximum benefit as the optimized enhanced query plan.

The details of the two phases are presented in the rest of this section.

Generating Materialize Operator Sets

Useful Materialized Operator Sets. Let M_{all} denote the set of all possible materialize operators that can be inserted into *plan*(Q). Instead of generating all possible subsets of M_{all}, SSPO considers only candidate materialize operator sets that are *useful*. Intuitively, a set of materialize operators $\mathcal{M} \subseteq M_{all}$ is considered to be *useless* (or not useful) if there exists a deadlock-free enhanced query plan P' with $M(P') \neq \mathcal{M}$ such that for every deadlock-free enhanced query plan P'' with $M(P'') = \mathcal{M}$, $cost(P') < cost(P'')$. Thus, a useless set of materialize operators can be safely ignored without affecting the

Algorithm 1: `Post-Optimizer`

Input: optimal plan *plan*(Q) for query Q
Output: enhanced query plan $eplan(Q)$

let $\mathcal{R}_{multi} = \{R_1, \cdots, R_d\}$ be the set of distinct multi-instance relations in Q
for each $R_i \in \mathcal{R}_{multi}$ **do**
  for each overflow instance r_j of R_i **do**
    initialize $drainer(r_j)$ if r_j is drainable
let $\mathcal{G} = \{G_1, \cdots, G_d\}$, where each share group G_i contains all instances of R_i except for non-drainable instances
let P_{opt} = (*plan*(Q), $\mathcal{G}$, $\emptyset$)
let C be the set of group dependency cycles in $\mathcal{G}$
if ($C = \emptyset$) **and** (every overflow instance is drainable) **then**
  return P_{opt}
let M_{all} be the set of all possible materialize operators that can be inserted into *plan*(Q)
let $M_{drain} = \{M_i \in M_{all} \mid drainSet(M_i) \neq \emptyset\}$
let $\mathcal{S}_{drain}$ be the collection of all useful subsets of M_{drain}
let $M_{cycle} = \{M_i \in M_{all} \mid cycleSet^{-}(M_i, C) \neq \emptyset\}$
for each $\mathcal{M}_{drain} \in \mathcal{S}_{drain}$ **do**
  let $C' = C \cup cycleSet^{+}(\mathcal{M}_{drain}, C)$
  let $\mathcal{S}_{cycle}(\mathcal{M}_{drain})$ be the collection of all useful subsets of M_{cycle} w.r.t C'
let $S = \{(\mathcal{M}_{drain}, \mathcal{M}_{cycle}) \mid \mathcal{M}_{drain} \in \mathcal{S}_{drain},\ \mathcal{M}_{cycle} \in \mathcal{S}_{cycle}(\mathcal{M}_{drain})\}$
initialize $P_{best} = (plan(Q), \emptyset, \emptyset)$
for each $(\mathcal{M}_{drain}, \mathcal{M}_{cycle}) \in S$ **do**
  for each instance $r_j \in drainSet(\mathcal{M}_{drain})$ **do**
    $drainer(r_j)$ = the closest ancestor operator of r_j from $\mathcal{M}_{drain} \cup \mathcal{M}_{cycle}$
  let $\mathcal{G}' = \{\{r_i\} \mid r_i$ is a non-drainable instance $\}$
  let $\mathcal{G}_{new} = \{G_1, \cdots, G_d\}$, where each share group G_i contains all instances of R_i except for non-drainable instances
  if ($\mathcal{R}_{multi} = \{R_1\}$) **and** (no two drainable instances in R_1 have the same drainer) **then**
    $\mathcal{G}_{new}$ = `OptimalGrouping`(G_1)
  else
    $\mathcal{G}_{new}$ = `HeuristicGrouping`($\mathcal{G}_{new}$)
  $P = (plan(Q), \mathcal{G}_{opt}, \mathcal{M}_{drain} \cup \mathcal{M}_{cycle})$, where $\mathcal{G}_{opt} = \mathcal{G}_{new} \cup \mathcal{G}'$
  if ($cost(P) < cost(P_{best})$) **then**
    $P_{best} = P$
return P_{best}

optimality of the enhanced query plan.

We now provide a more concrete characterization of the notion of a useful set of materialize operators. Recall that adding a materialize operator M to *plan*(Q) can help enhance its performance in two ways. First, M can enable a non-drainable instance r_i

to become drainable thereby allowing r_i to participate in a shared scan. Second, M can eliminate some execution dependencies thereby enabling a plan to become deadlock-free (i.e., $C = \emptyset$). These two benefits of M can be formalized in terms of its *drain set* and *remove-cycle set* defined as follows.

The *drain set* of M, denoted by $drainSet(M)$, is defined to be the set of non-drainable instances in *plan*(Q) that become drainable if M is added to *plan*(Q). Thus, M becomes the drainer operator for each of the instances in $drainSet(M)$.

The *remove-cycle set* of M (w.r.t. C), denoted by $cycleSet^{-}(M, C)$, is defined to be the subset of group dependency cycles in C that are eliminated by the addition of M to *plan*(Q).

The following result states a useful relationship between $drainSet(M)$ and $cycleSet^{-}(M, C)$.

Lemma 2.4. *At most one of $drainSet(M)$ and $cycleSet^{-}(M, C)$ can be non-empty.*

Lemma 2.4 follows from the observation that if $drainSet(M) \neq \emptyset$ (i.e., M becomes a drainer for some non-drainable instance r_i), then r_i cannot have any ancestor drainer operator prior to the addition of M, which implies that there are no instance execution dependencies (and hence group execution dependencies) that M can eliminate. Hence $cycleSet^{-}(M, C) = \emptyset$. Conversely, if $cycleSet^{-}(M, C) \neq \emptyset$, then M is able to eliminate some group dependency cycle (via the elimination of some instance execution dependency) which implies that there must exist some drainer operator that is an ancestor of M. Hence, there cannot be any non-drainable instances within the query subtree rooted at M (i.e., $drainSet(M) = \emptyset$).

Based on Lemma 2.4, the useful materialize operators (w.r.t. C) can be partitioned into two disjoint sets M_{drain} and M_{cycle} defined as follows:

$$M_{drain} = \{M \in M_{all} \mid drainSet(M) \neq \emptyset\}$$

$$M_{cycle} = \{M \in M_{all} \mid cycleSet^{-}(M, C) \neq \emptyset\}$$

A materialize operator that is not contained in $M_{drain} \cup M_{cycle}$ is useless.

However, adding a materialize operator M to *plan*(Q) not only incurs a processing cost (i.e., $matCost(M)$) but could also introduce additional group dependency cycles. We characterize the latter cost for M as follows. The *add-cycle set* of M (w.r.t. C), denoted by $cycleSet^{+}(M, C)$, is defined to be the set of new group dependency cycles (i.e., not contained in C) that are introduced by the addition of M to *plan*(Q).

The following result states that adding a materialize operator from M_{cycle} to *plan*(Q) does not create any new group dependency cycles.

Lemma 2.5. $cycleSet^{+}(M, C) = \emptyset$ *for each* $M \in M_{cycle}$.

Lemma 2.5 can be established by contradiction. Suppose $cycleSet^{+}(M, C) \neq \emptyset$. Then the addition of M must have introduced a new instance execution dependency $r_i \rightarrow s_j$ (that contributed to a new group dependency cycle), where both r_i and s_j are within the query subtree rooted at M. However, $M \in M_{cycle}$ implies that there must be a drainer operator that is an ancestor of M in the query plan which contradicts the fact that $r_i \rightarrow s_j$ is a new dependency.

The definitions of $drainSet(M)$, $cycleSet^{+}(M, C)$, and $cycleSet^{-}(M, C)$ can be generalized naturally for a set of materialize operators $\mathcal{M} \subseteq M_{all}$ (e.g., $drainSet(\mathcal{M}) = \bigcup_{M \in \mathcal{M}} drainSet(M)$).

By Lemmas 2.4 and 2.5, we can define a useful materialize operator set in terms of its two disjoint subsets: a useful subset of M_{drain} and a useful subset of M_{cycle} as follows.

We say that $\mathcal{M} \subseteq M_{drain}$ is *useful* (w.r.t. C) if there does not exist another $\mathcal{M}' \subseteq M_{drain}$ such that all the following four conditions hold: (1) $drainSet(\mathcal{M}) \subseteq drainSet(\mathcal{M}')$, (2) $cycleSet^{+}(\mathcal{M}, C) \supseteq cycleSet^{+}(\mathcal{M}', C)$, (3) $matCost(\mathcal{M}) \geq matCost(\mathcal{M}')$, and (4) at least one of the three previous conditions is strict.

Similarly, we say that $\mathcal{M} \subseteq M_{cycle}$ is *useful* (w.r.t. C) if there does not exist another $\mathcal{M}' \subseteq M_{cycle}$ such that all the following three conditions hold: (1) $cycleSet^{-}(\mathcal{M}, C) \subseteq$

$cycleSet^{-}(\mathcal{M}', C)$, (2) $matCost(\mathcal{M}) \geq matCost(\mathcal{M}')$, and (3) at least one of the two previous conditions is strict.

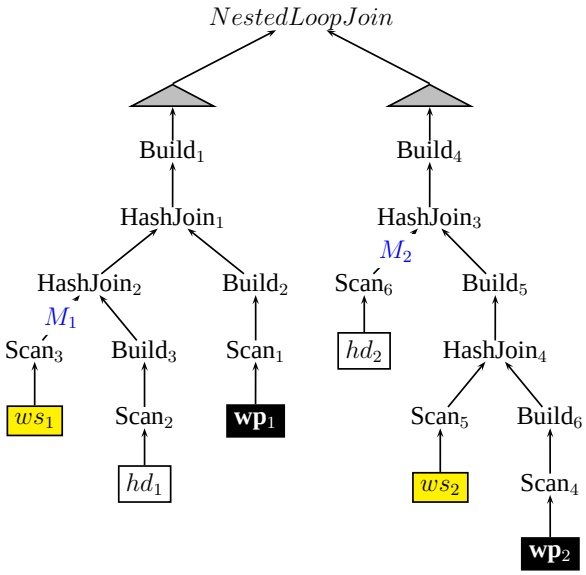
Finally, consider a set $\mathcal{M} \subseteq M_{all}$, where $\mathcal{M} = \mathcal{M}_{drain} \cup \mathcal{M}_{cycle}$, $\mathcal{M}_{drain} \subseteq M_{drain}$, and $\mathcal{M}_{cycle} \subseteq M_{cycle}$. We say that $\mathcal{M}$ is *useful* (w.r.t. C) if $\mathcal{M}_{drain}$ is useful (w.r.t. C) and $\mathcal{M}_{cycle}$ is useful (w.r.t. $C \cup cycleSet^{+}(\mathcal{M}_{drain}, C)$).

A materialize operator set that is not useful cannot form an optimal enhanced query plan.

Algorithm. SSPO (steps 10 to 17 in Algorithm 1) generates a collection of useful candidate materialize operator sets (denoted by S) as follows. First, SSPO generates $\mathcal{S}_{drain}$, the collection of all useful subsets of M_{drain}. Next, SSPO takes each $\mathcal{M}_{drain} \in \mathcal{S}_{drain}$ to generate $\mathcal{S}_{cycle}(\mathcal{M}_{drain})$, the collection of all useful subsets of M_{cycle} w.r.t. C', where $C' = C \cup cycleSet^{+}(\mathcal{M}_{drain}, C)$. The final collection S is given by $\{\mathcal{M}_{drain} \cup \mathcal{M}_{cycle} \mid \mathcal{M}_{drain} \in \mathcal{S}_{drain},\ \mathcal{M}_{cycle} \in \mathcal{S}_{cycle}(\mathcal{M}_{drain})\}$. Note that as the empty set is contained in both $\mathcal{S}_{drain}$ and $\mathcal{S}_{cycle}(\mathcal{M}_{drain})$, an empty set of materialize operators is also generated by SSPO.

Although the time complexity of the procedure above is exponential in the number of materialize operators in M_{drain} and M_{cycle}, this number is reasonably small in practice. Alternatively, some heuristic can be applied to generate smaller M_{drain} and M_{cycle} so as to reduce the running time, in the cost of missing some useful candidate materialize operator sets in S.

Example 3 Fig. 2.5(a) shows two useful materialize operators, M_1 and M_2, that can be used to break the execution dependency cycle $hd_1 \dashrightarrow hd_2 \rightarrow ws_2 \dashrightarrow ws_1 \rightarrow hd_1$ for the query plan of Q90 in Example 2. M_1 breaks the cycle by eliminating $ws_1 \rightarrow hd_1$ while M_2 breaks the cycle by eliminating $hd_2 \rightarrow ws_2$. Ignoring the $matCost(.)$ component, there are three useful sets of materialize operators: $\emptyset$, $\{M_1\}$ and $\{M_2\}$. $\square$



(a) Candidate Materialize Operators

Plan	M	Share Groups
P_1	$\emptyset$	$\{ws_1, ws_2\}$, $\{hd_1\}$, $\{hd_2\}$, $\{wp_1, wp_2\}$
P_2	$\{M_1\}$	$\{ws_1, ws_2\}$, $\{hd_1, hd_2\}$, $\{wp_1, wp_2\}$
P_3	$\{M_2\}$	

(b) Candidate Enhanced Query Plans

Figure 2.5: Enhanced Query Plans for Example 2

Optimizing Share Group List

Given a candidate set of materialize operators $\mathcal{M}$, the second phase of SSPO (steps 18 to 30 in Algorithm 1) computes an optimized share group list $\mathcal{G}$ to produce a deadlock-free enhanced plan P with $M(P) = \mathcal{M}$ and $G(P) = \mathcal{G}$. SSPO has two algorithms for this computation: an optimal algorithm (that can compute an optimal share group list) is used if *plan*(Q) meets certain conditions; otherwise, a greedy heuristic algorithm is used.

Optimal Grouping. An optimal share group list can be computed using Algorithm 2 when *plan*(Q) satisfies two conditions:

(C1) there is exactly one multi-instance relation R_1 in *plan*(Q); and

(C2) the drainers for all the drainable-instances of R_1 are all distinct.

Algorithm 2 takes a single share group G_1 as input, where G_1 contains all the instances of R_1 except for non-drainable instances. The algorithm first constructs a directed graph G, where the nodes in G are instances in G_1, and the edges represent execution dependencies among the instances in G_1. By condition (C2), G must be a directed acyclic graph. The algorithm then iteratively refines G_1 into a collection of share groups $G'_1, \cdots, G'_n$ such that $G'_1 \rightarrow G'_2 \cdots \rightarrow G'_n$. The time complexity of Algorithm 2 is $O(m^2)$, where m is the number of instances in G_1.

Heuristic Grouping. Algorithm 3 is a greedy heuristic approach to optimize an input share group list $\{G_1, \cdots, G_d\}$, where each G_i contains all the instances of relation R_i excluding the non-drainable instances. The share groups are ordered such that $scanCost(R_1) \leq \cdots \leq scanCost(R_d)$. The heuristic refines each share group G_i by splitting it into a collection of smaller groups S_i in the order $G_1, \cdots, G_d$. The intuition behind processing the share groups in non-descending order of the scan cost of the associated relations is to minimize the total scan cost of the refined share groups. For each group G_i, the heuristic tries to split G_i into the smallest number of groups by iteratively removing from G_i, the instance that is involved in the largest number of cycles. The removed instance is inserted into an existing split group of G_i whenever possible; otherwise, it is inserted into a new split group. The insertions into split groups are performed such that no new group dependency cycles are formed. The time complexity of of Algorithm 3 is $O(n^2)$, where $n = \sum_{i=1}^{d} |G_i|$.

Example 4 Continuing with Example 3, Fig. 2.5(b) shows the optimized share group lists computed for each candidate materialize operator set using the heuristic algorithm in Algorithm 3. □

Algorithm 2: `OptimalGrouping`

Input: a single share group G_1 containing the instances of R_1 excluding non-drainable instances

Output: an optimal list of share groups

1: let $G = (V, E)$, where
   V = G_1 and $E = \{(a, b) \mid a, b \in V, a \rightarrow b\}$
2: initialize $n = 0$
3: **repeat**
4: $n = n + 1$; $G'_n = \{v \in V \mid$ v has in-degree of 0 in $G\}$
5: remove each $v \in G'_n$ from G and its incident edges
6: **until** $V = \emptyset$
7: **return** $\{G'_1, \cdots, G'_n\}$

Algorithm 3: `HeuristicGrouping`

Input: $\{G_1, \cdots, G_d\}$, where each G_i is a share group containing all instances of relation R_i excluding non-drainable instances such that *scanCost*(R_1) $\leq \cdots \leq$ *scanCost*(R_d)

Output: an optimized list of share groups

1: let C = set of group dependency cycles among $G_1, \cdots, G_d$
2: **for** $i = 1$ to d **do**
3: initialize $S_i = \{G_i\}$
4: **while** (C contains a cycle involving G_i) **do**
5: let r_j be the instance in G_i that participates in the largest number of cycles in C
6: **if** r_j can be added into some $G_k \in S_i$, $k \neq i$, without introducing any new group dependency cycles **then**
7: add r_j into G_k
8: **else**
9: create a new group $G' = \{r_j\}$
10: add G' into S_i
11: remove r_j from G_i
12: remove cycles in C that involved r_j
13: **return** $S_1 \cup \cdots \cup S_d$

2.4 Interleaved Iterative Execution

In this section, we explain how the `IIQE` component of `MAPLE` can be implemented by making only moderate modifications to the conventional iterative query execution engine; thus, demonstrating that `MAPLE` is indeed a light-weight approach to optimize complex queries with multiple relation instances.

For each relation instance r_i in *eplan*(Q), `IIQE` maintains the following static infor-

mation: (1) a boolean flag, denoted by *switchEnabled*(r_i), which has a `true` value if and only if r_i is estimated by `SSPO` to be an overflow instance; and (2) $drainer(r_i)$ if if r_i is a drainable instance. In addition to the above information, which remains unchanged during the execution of the query, `IIQE` also maintains some global runtime information that is updated dynamically as the query execution progresses. Specifically, each relation instance r_i is associated with a status variable for its drainer, denoted by *drainerStatus*(r_i), which has three possible values: *inactive*, *active*, and *successful*, indicating, respectively, that the drainer is not active, the drainer is active and the draining is in progress, and the drainer is active and the draining has completed. The value of *drainerStatus*(r_i) is initialized to *inactive* for each relation instance r_i before the execution of *eplan*(Q). Whenever $buf(r_j)$ becomes full during the shared scan of some other instance of r (say r_i) and `IIQE` decides to switch execution to $drainer(r_j)$, the value of *drainerStatus*(r_j) is updated to *active*. Subsequently, when all the tuples in the full buffer $buf(r_j)$ have been consumed, the scan operator for r_j will update the value of *drainerStatus*(r_j) from *active* to *successful*. When the execution control is returned from $drainer(r_j)$, the value of *drainerStatus*(r_j) is reset to *inactive*.

Algorithm 4: `Scan`

Input: r_i, the instance being scanned

```
1:  let Op be the parent operator of Scan(r_i) in query plan
2:  if (buf(r_i) is not empty) then
3:     let t be the first tuple in buf(r_i)
4:     deliver t to Op
5:     remove t from buf(r_i)
6:  else
7:     if (drainerStatus(r_i) = active) then
8:        drainerStatus(r_i) = successful
9:        deliver a dummy-null tuple to Op
10:    else
11:       SharedScan (r_i)
```

Recall that in the iterative execution model, each operator is specified in terms of

Algorithm 5: `SharedScan`

Input: r_i, the instance being scanned

```
1:  let G_x be the share group that r_i belongs to
2:  initialize continueScan = true
3:  while (continueScan) do
4:      let t be the next tuple from relation r
5:      if (t is null) or (t qualifies for Scan(r_i)) then
6:          deliver t to the parent operator of Scan(r_i)
7:          continueScan = false
8:      for each (r_j ∈ G_x, r_j ≠ r_i) do
9:          if (t is null) or (t qualifies for Scan(r_j)) then
10:             if (buf(r_j) is full) and switchEnabled(r_j) then
11:                 SwitchExecution (r_j)
12:             append t into buf(r_j)
```

Algorithm 6: `SwitchExecution`

Input: r_i, an overflow instance

```
1:  drainerStatus(r_i) = active
2:  transfer execution control to drainer(r_i) operator
3:  drainerStatus(r_i) = inactive
```

three functions: *open*, *getNext*, and *close*. Algorithm 4 highlights the modifications required for the *getNext* procedure of the table scan operator (referred to as `Scan`). Given a relation instance r_i, `Scan` first checks whether its associated buffer $buf(r_i)$ is empty: if it is not empty, the first tuple in $buf(r_i)$ will be returned to the parent operator of $Scan(r_i)$ and then removed from $buf(r_i)$. The key modification for the `Scan` algorithm occurs when the buffer is empty, where there are two cases to consider. In the first case (steps 8-9), if the scan of r_i has been initiated to drain its buffer (i.e., *drainerStatus*(r_i) = *active*), then it means that the draining process has completed successfully. The value of *drainerStatus*(r_i) is then updated to *successful*, and a `dummy-null` tuple is returned to the parent operator of $Scan(r_i)$. The use of `dummy-null` tuples is important to distinguish a successful draining process (i.e., all the tuples in a full buffer have been consumed) from a completed relation scan event (i.e., there are no more tuples to be

placed in the buffer). In the latter case, an actual `null` tuple is returned. In this way, whenever an operator op receives a `dummy-null` tuple from its child operator, op will know that the tuple is due to a completed draining process and will therefore pass the `dummy-null` tuple up to its parent operator, and so on. The upward propagation of the `dummy-null` tuple in the query plan tree continues until the tuple is received by a successful drainer operator op (i.e., $op = drainer(r_k)$ and *drainerStatus*(r_k) = *successful*). Thus, the drainer operator op then returns execution control to the interrupted relation scan that initiated op. The value of *drainerStatus*(r_k) is also reset to *inactive*. In the second case (step 11), where the scan of r_i is a "normal" scan (i.e., not initiated for buffer draining), the `SharedScan` of r_i will be invoked.

The details of `SharedScan` are shown in Algorithm 5. Essentially, `SharedScan` continues scanning r for the next tuple that qualifies for r_i; i.e., satisfies the selection predicate conditions associated with the scan of r_i. For each scanned tuple t, `SharedScan` also checks if t qualifies for other instances r_j that are in the same share group as r_i; the qualified tuples are pushed into the appropriate buffers. If some buffer $buf(r_j)$ becomes full, then there are two cases to consider. If `SSPO` has correctly estimated that r_j is an overflow instance (i.e., *switchEnabled*(r_j) has a `true` value), a drainer operator $drainer(r_j)$ would have been assigned by `SSPO` and the execution control then switches to this drainer (step 11) by invoking the `SwitchExecution` function. Otherwise, if the overflow of $buf(r_j)$ has not been anticipated by `SSPO`, the full buffer $buf(r_j)$ will not be drained and it will instead be implicitly materialized; i.e., in `IIQE`, whenever a tuple is added to a full buffer $buf(r_j)$, the buffer contents will be materialized.

In general, the `SharedScan` of r_i could lead to full buffers for multiple instances in the same share group as r_i. When this happens, there is the issue of the execution order of the multiple drainers. The current implementation of `MAPLE` simply picks an

arbitrary sequence; possible optimization of this ordering is part of our future work.

The `SwitchExecution` function (shown in Algorithm 6) is invoked to switch execution to $drainer(r_i)$ for an overflow instance r_i. The function needs to update the *activeDrainer* status of r_i to *active* before transferring control to the drainer and reset the *activeDrainer* status to *inactive* upon its return.

In summary, implementing the `IIQE` component of `MAPLE` requires only moderate modifications to the traditional iterative execution evaluation engine used by most RDBMSs. Specifically, the main changes include: two new functions `SharedScan` and `SwitchExecution`; and minor modifications to operator code to distinguish between `null` and `dummy-null` tuples.

2.5 Performance Study

We validated our techniques using an experimental prototype built on PostgreSQL 8.1.3. All experiments were performed on a Dell workstation with a Quad-Core Intel Xeon 2.33GHz processor, 3GB of memory, one 160GB SATA disk and another 750GB SATA disk, running Linux 2.6.20. Both the operating system and PostgreSQL system are built on the 160GB disk, while the databases of PostgreSQL are stored on the 750GB disk.

Since PostgreSQL 8.1.3 does not support the WITH clause, we replaced those WITH procedures in queries with VIEW definitions. In this way, PostgreSQL applies view unfolding to replace the views by their definitions during optimization.

As default, the initial system buffer pool in PostgreSQL is set to 1,000 8K-pages. We also tested with larger buffer pool sizes. The results were similar and thus omitted.

2.5.1 Test Queries

As mentioned, more than 60% (61 out of total 99) of the TPC-DS queries contain multiple instances. We have conducted experiments on many of these queries. We present here a representative set that offers some interesting insights. A query is chosen if it satisfies *all* the following criteria: (a) It contains multiple instances that are eligible for scan sharing, i.e., apply sequential scan on the same table. (b) It contains multiple instances of at least one of the three big relations: store_sales (ss), catalog_sales (cs) and web_sales (ws). (c) It is executable by PostgreSQL. Some operators in the queries are not recognized/supported by PostgreSQL. (d) It can be optimized by PostgreSQL's *dynamic programming* (DP) optimizer. For queries that are too complex to optimize using the DP method, PostgreSQL provides another *genetic optimizer(geqo)*. However, since geqo does not guarantee to generate consistent plans for the same query, we cannot use it. (e) It is not a *batch* query which contains a batch of separate queries that run in parallel. We have to exclude batch queries because our current implementation only supports single queries, although our techniques can be easily extended to support batch queries. (f) Its execution time is affordable for us. Some queries, like Q74 and Q95, require super long-time executions. Table 2.1 presents a summary of the 49 queries excluded according to the criteria above.

criterion	a	b	c	d	e	f
# of queries	5	28	8	2	4	2

Table 2.1: Queries Filtered by Each Criterion

Finally we are left with 12 queries listed in Table 2.2, along with the instance number of ss, cs and ws inside. However, all other instances within a chosen query, irrespective of their sizes, were also considered by MAPLE.

Since TPC-DS queries are all very complex, we cannot afford to draw full queries/plans.

	rel $*$ inst#		**rel $*$ inst#**
Q2	cs $*$ 2, ws $*$ 2	**Q61**	ss $*$ 2
Q4	ss $*$ 4, cs $*$ 4	**Q65**	ss $*$ 2
Q11	ss $*$ 4, ws $*$ 4	**Q72**	ss $*$ 2, cs $*$ 2
Q31	ss $*$ 3, ws $*$ 3	**Q88**	ss $*$ 8
Q51	ss $*$ 2	**Q90**	ws $*$ 2
Q59	ss $*$ 2	**Q97**	ws $*$ 2

Table 2.2: Test Queries in Experiments

2.5.2 Experiment Design

In our implementation, MAPLE is integrated into the original system. By setting a flag, we can switch between the *original mode* and MAPLE *mode*. In the original mode, the original execution engine will be used; in the MAPLE mode, the MAPLE engine will be used. Both engines share the same query optimizer.

In each experiment below, we ran the same test query in both the original mode and the MAPLE mode to compare the execution time difference. When a test query was running, no other queries was running in parallel. Between queries we restarted the operating system to clear caches.

In PostgreSQL, *each* sorting and hashing operation has a dedicated *operator memory*. In MAPLE, besides the operator memories, each overflow instance uses additional buffer memory, which we shall refer to as *instance-buffer*. For a fair comparison, in each experiment we distributed the total amount of instance-buffer used in MAPLE mode evenly to each operator memory in original mode.

We studied the effect of three experiment parameters: operator memory (operator_mem), instance-buffer size (buffer) and the *dataset* size. For the latter, we used both 10 GB and 100 GB TPC-DS datasets.

The TPC-DS datasets are imported into PostgreSQL's databases, which are stored on the 750GB disk. In the experiments, the same disk was used to store the temporary files generated during query execution.

The default settings that we used for our experiments are 1 MB (instance) buffer, 10 MB operator memory and a 10 GB dataset.

In following subsections, we shall refer to the system under original mode as PostgreSQL.

2.5.3 Optimization Overhead

	psql	MAPLE		**psql**	MAPLE
Q2	90	125	**Q61**	366	434
Q4	113	126	**Q65**	311	351
Q11	117	133	**Q72**	137570	137789
Q31	104	115	**Q88**	397	413
Q51	427	502	**Q90**	346	354
Q59	88	119	**Q97**	420	473

Table 2.3: Optimization times (in *microsecond*) with Default Settings

It is desirable to measure the optimization overhead of MAPLE, which is incurred mainly by SSPO. Therefore, we compared the actual optimization times of PostgreSQL and MAPLE with default parameter settings. In order to eliminate any first-level instruction cache effect in query optimization, we restarted the operating system between optimizations. The optimization times of PostgreSQL and MAPLE can be found in Table 2.3. It is very clear that the optimization overhead of MAPLE is low, and as we shall see shortly, it is also negligible compared to the query execution time.

2.5.4 Operator Memory

In this experiment, we study the effect of operator_mem. We use three different sizes: 5 MB, 10 MB and 20 MB.

Fig. 2.6 shows the performance improvements (in %) of MAPLE over the PostgreSQL; and Fig. 2.7 shows the corresponding query execution times in MAPLE and PostgreSQL. In Fig. 2.7, the execution times of PostgreSQL can be computed by adding

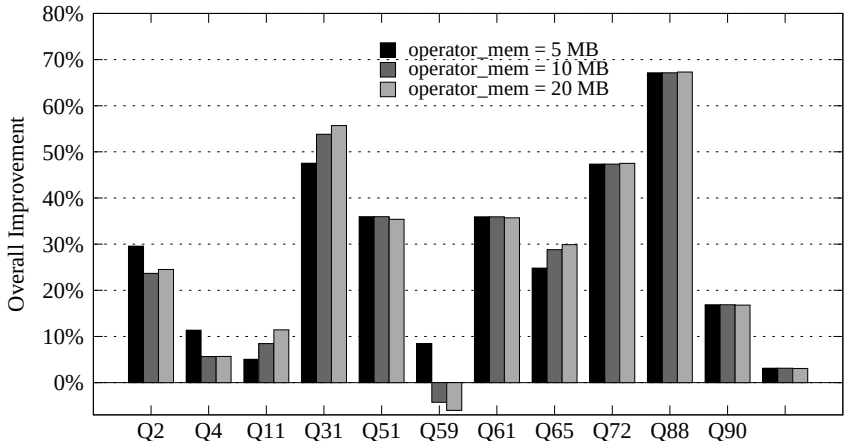


Figure 2.6: Performance Improvements By MAPLE

the execution time of MAPLE with the time MAPLE saved. Fig. 2.8 depicts the *expected saving* and the *actual saving* for all queries with 5 MB operator_mem. The expected saving refers to the time MAPLE is expected to save over PostgreSQL. The actual saving is the saving of MAPLE over PostgreSQL for the actual total query execution time. We shall not present detail query-by-query analysis. Instead, we will summarize the more interesting findings here.

First, as shown, MAPLE offers significant performance improvement in almost all queries (except Q59, for which we will explain shortly). The average improvement is around 30% and the highest improvement is 67% achieved by Q88. In terms of absolute time, the savings range from a few seconds to 700 seconds. These results are expected as MAPLE requires only one scan of multiple instances of a relation. Second, we also observe that MAPLE remains superior as we vary operator_mem.

Second, we note that, for some queries (Q2, Q4, Q11, Q31, Q59 and Q65), the execution times of both MAPLE and PostgreSQL vary with different operator_mem. There are two main reasons for this:(a) The query plan generated by PostgreSQL (and hence MAPLE) may be different under different operator_mem size. In the experiment, the plans for Q4 and Q65 are different when we change operator_mem from 5 MB to 10/20 MB;

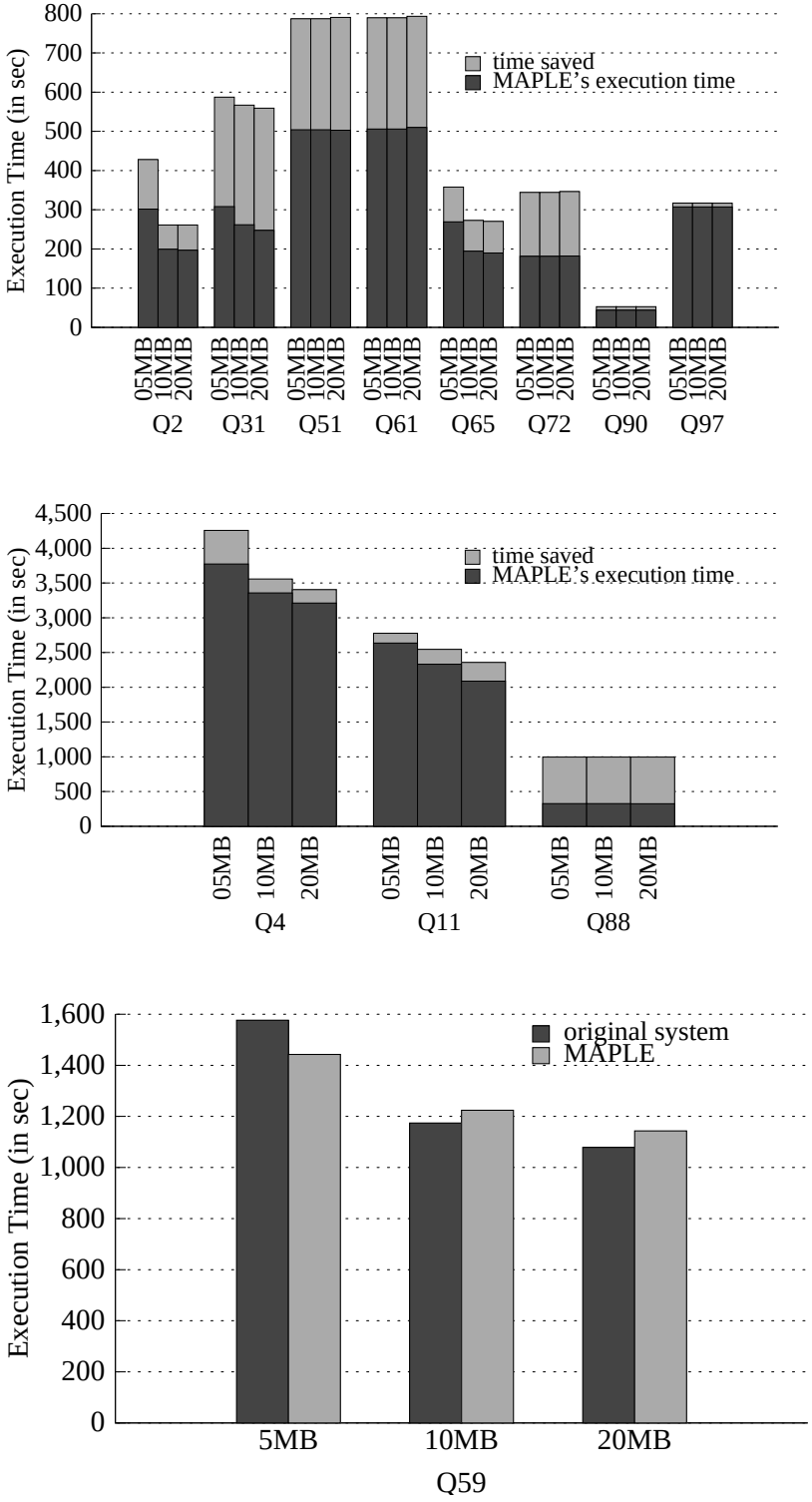


Figure 2.7: Query Execution Times

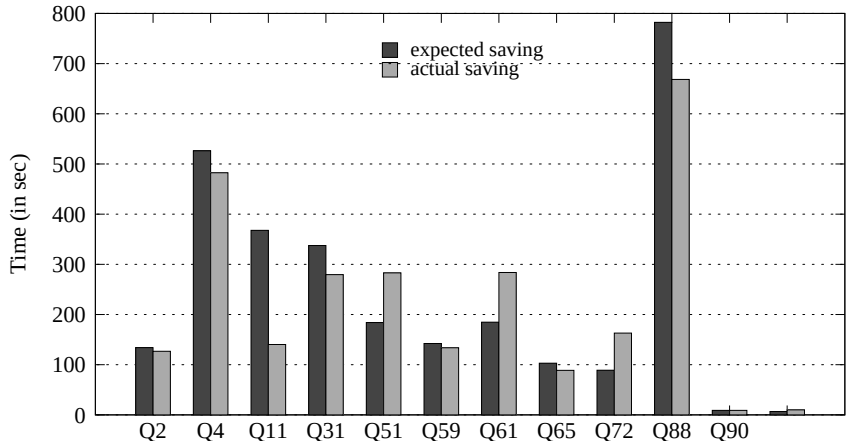


Figure 2.8: Expected Saving and Actual Saving With 5MB operator_mem

for Q11, there are three different plans for the three operator_mem sizes; for the other queries, their plans remain the same for the three operator_mem sizes. (b) A larger operator_mem may reduce the I/O cost, e.g., for sorting, the number of runs may be reduced, and for hybrid hash join, the amount of data in the partitions to be written out and re-read will also be smaller. This reduces the execution times. With a reduced execution time, the savings for MAPLE over PostgreSQL may correspondingly reduce.

Third, from Fig. 2.8, we find that the actual savings in MAPLE are close to the expected savings for most queries. The difference is mainly due to the additional overhead (like the cost of copying tuples to buffers) incurred by interleaved execution. However, for Q11, the actual saving is significantly lower than expected, whereas for Q51, Q61 and Q72, the actual savings are much higher than expected! Our investigation shows these are contributed by several effects of the interleaved execution: (a) *FragmentedReadWrite effect.* Under the interleaved execution model, the processing of one drainer may trigger other drainers to become active. As a result, when the processing of these drainers involve disk accesses (e.g., sorting), the intermediate results written (and subsequently read) are more fragmented across the disk (than it would be had there been only one single drainer running, as in PostgreSQL). (b) *BufferHit effect.* This effect arises when

both an active drainer and an interrupted drainer share some cache content. As a result, when the active drainer requires some data, it finds it in the buffer, and when the suspended drainer resumes processing, it also finds its required data in the buffer. Clearly, the FragmentedReadWrite is a negative effect while the BufferHit is a positive effect.

For Q51, Q61 and Q72, we observe the BufferHit effect For example, in Q51, there are some common index scans in subtrees of two drainers: while execution switches between these two drainers, the index pages fetched from disk can be shared via the system buffer. As such, besides the expected savings from using SharedScan, the sharing of these indexed pages also contributes to the actual savings.

On the other hand, for Q11, it turns out that the drainers that are processed in an interleaved fashion need to write out large amount of intermediate results, resulting in the FragmentReadWrite effect that reduces the savings.

For Q90 and Q97, little improvement opportunity was left to MAPLE due to the *OS CacheHit* effect. This is because in these two queries ws is the only large table for which a large part is cached by the OS in the 3GB RAM.

Finally, for Q59, the plan involves a sort operator on a large intermediate result produced by a hash join operator. When the operator_mem is small (5 MB), the buffer is not sufficient to hold the entire hash table, and the sort operation incurs more disk I/O cost. As such, although there is a FragmentReadWrite effect in MAPLE, this is relatively small and hence MAPLE outperforms PostgreSQL. However, when the operator_mem increases to 10/20 MB, the hash join can be processed in memory, and the sort operator incurs lesser I/O cost. As a result, PostgreSQL's execution time reduces significantly. On the contrary, the FragmenReadWrite effect remains in MAPLE. It turns out that this effect far outweighs the benefits of SharedScan, resulting in its poorer performance than PostgreSQL. We note that we can statically determine the number of switches (which gives a hint on how fragmented the drainer's output will be). If the value is above a

certain threshold, we will not post-optimize the PostgreSQL plan. We plan to explore this further.

2.5.5 Instance-buffer Size

We next study the effect of instance-buffer size. In this experiment, we use two different instance-buffer sizes: 100 KB, 1 MB. Recall that for PostgreSQL, the total amount of instance-buffer sizes used for MAPLE goes to its operator memories. The results for this experiment are shown in Fig. 2.9.

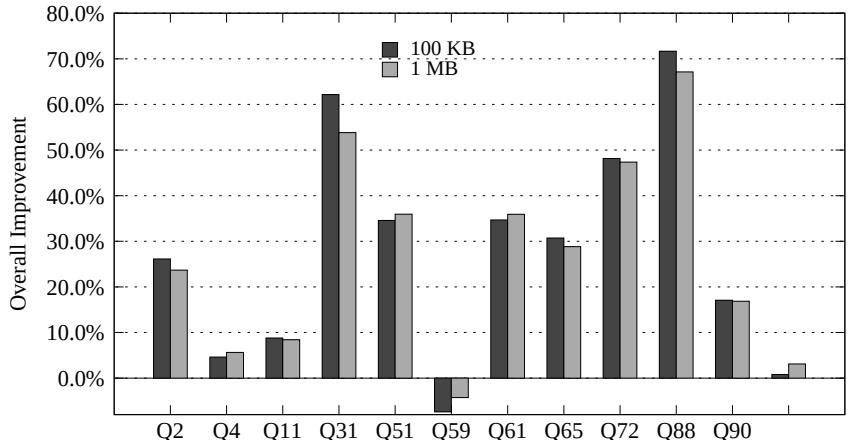


Figure 2.9: MAPLE Effect of Changing Instance-buffer Size

Generally, the performance of MAPLE under different buffer sizes are more or less the same. We see two different effects of the buffer size. For MAPLE a larger instance-buffer reduces the number of interleaved executions, and hence less FragmentReadWrite effect. On the other hand, for PostgresSQL, a larger operator memory (recall that the instance-buffer of MAPLE are distributed to the operator memory) reduces the I/O cost of sort and hash operators. For some queries (e.g., Q4, Q11, Q51, Q59, Q61, Q97), the improvement over PostgreSQL increases with larger instance-buffer. However, for some queries, like Q72 and Q88, the performance improvement over PostgreSQL degraded marginally with increased buffer sizes.

2.5.6 Dataset

We also conducted an experiment with a 100 GB dataset to study the scalability of MAPLE. Here, we use 10 MB operator_mem and 1 MB buffer. Fig. 2.10 shows both the results of 100GB dataset and the results of 10GB dataset with the same parameter settings.

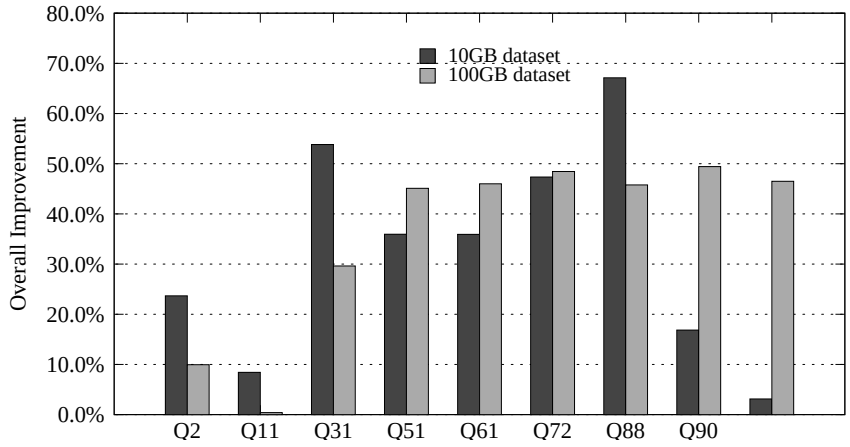


Figure 2.10: MAPLE Effect in 100GB Dataset

Since the execution times of queries on 100GB dataset were very long, we did not finish all 12 queries. From the figure, we see that MAPLE still performs well with a larger dataset.

For some queries like Q2, Q31 and Q88, the improvements over PostgreSQL (in %) with the 100 GB dataset is lower than that with the 10 GB dataset. There are two main reasons for this behavior: a) while (big) relation sizes have increased ten fold from 10 GB to 100 GB dataset, their scan times have not increased proportionally. For example, the scan time of web_sales in Q2 increased from 20 seconds to 90 seconds in 100 GB dataset. b) With the 100 GB dataset, in PostgreSQL the ratio of the total table scan time of instances to the total execution time is reduced compared to that of 10 GB dataset. For example, the total table scan time of instances of Q2 took around 49% and

39% of the total execution time in 10GB and 100GB dataset, respectively.

On the other hand, for Q90 and Q97, MAPLE performs much better in the 100 GB dataset. This is because the OS CacheHit effect present in the 10 GB dataset disappeared in 100 GB case, and the gain of shared scan becomes more significant.

2.5.7 Two Disks

So far, in all the experiments, we use the same 750GB disk to store both PostgreSQL's databases and the temporary files generated during query execution. In this experiment, we use another 160GB disk to separately store the temporary files. In this way, the reading relational data during execution was not interrupted by the disk access actions of temporary files. We used 10 GB dataset, 20 MB operator_mem and 1 MB buffer. Fig. 2.11 shows both the results of two disks and the results of one disk with the same parameter settings.

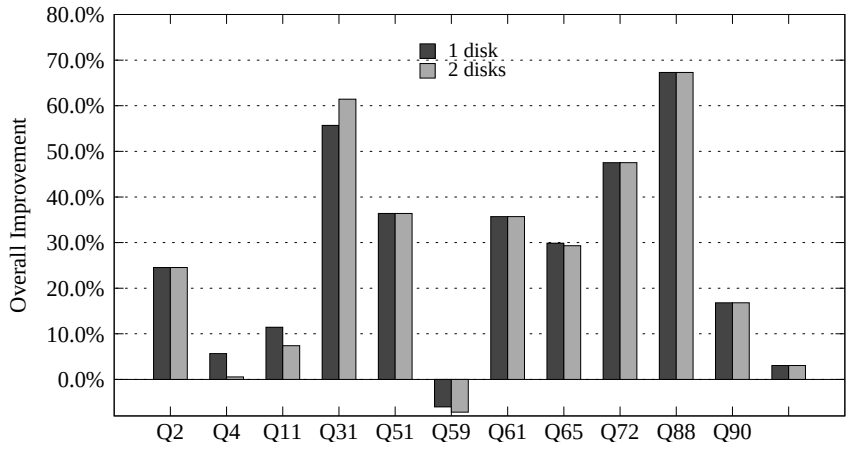


Figure 2.11: MAPLE Effect of Using Two Disks

For Q4, Q11 and Q59, using two disks actually favoured the original system more since the interleaved disk accesses for relations and temporary files disappeared and thus the reading of relational data was more sequential. In MAPLE, due to interleaved

executions, the reading of relational data still remained a bit random and the execution time did not reduce in proportion compared to PostgreSQL. Therefore, for these queries the improvements achieved by MAPLE are lower than the situation of one disk.

For other queries, their executions generated little and small temporary files. Therefore, using two disks made little difference.

We also find that MAPLE may help to achieve higher improvement in some cases, e.g. Q31.

2.6 Related Work

The need to efficiently coordinate multiple disk scans on the same table to exploit data-sharing has long been recognized. Early work focused on designing buffer replacement algorithms (e.g., LRU-K [50]) to maximize buffer locality. However, these works do not explicitly optimize data sharing. Moreover, their effectiveness is limited especially for large tables that do not fit in the cache. Several commercial Database systems have implemented various forms of *circular scans* on database relations (Teradata [8], RedBrick [18] and Microsoft SQL Server [1]). The basic idea is to let a newly starting scan attach to an ongoing scan to reuse buffer pages brought by the ongoing scan. In QPipe [36], Harizopoulos et al. propose to maintain one scan thread that keeps scanning a table while table scan operators can attach to and detach from this thread in order to share the scanned buffer pages. However, the degree of sharing the buffer pool provided in these methods is extremely sensitive to the speed diversity of scans. Recently, a modified circular scan has been proposed in IBM DB2 system [42, 43] by adding explicit group control and allowing throttling of faster scans. Zukowski et al. [69] introduce an enhanced buffer manager that dynamically schedules disk reads of scan operators such that multiple concurrent scans reuse the same buffer pages.

Sharing scan of base relations and pipelining of common subexpression results reduce disk access costs on the level of query processing operators. Zhao et al. [66] consider sharing scans and pipelining subexpressions among OLAP queries (aggregation on a join of fact table with dimension tables). Nilesh et al. [19] discussed the feasibility of pipelining in multi-query optimization. They aim to pipeline results of a common subexpression or tuples of a base relation to consumers in different SQL queries. Our work can also be extended to handle common subexpressions and multiple queries.

In [19], the determination of a valid pipeline schedule has a similar motivation as our deadlock avoidance method in Section 2.3.2. However, the two are actually different. As an example, consider Q90 in Fig. 2.2. The schedule of sharing the scan of all three relations will be considered valid by [19] as each cycle has two opposite materialized edges (the build edges of hash join). However, as we discussed in the chapter, whether it is a valid sharing scan schema depends on whether hd_2 is an overflow instance or not (see example 1 and example 2). In fact, the interleaved execution deadlock described in this chapter is different from the deadlock situation in [19, 36].

In multi-query optimization(MQO) [54, 68], exploiting common subexpressions in (multi-instance) queries indirectly leads to avoiding multiple scans on the same relation table. However, the materialized results of a common subexpression need be separately read by different consumers, just like the independent scans of relation instances. Moreover, MQO is not able to optimize scans of instances that are outside the common subexpression. Therefore, for multi-instance queries, MAPLE can be either applied independently or ultilized as the next optimization step after MQO.

The philosophy under our interleaved execution strategy is that when event a is blocked, process event b to continue a. The *query scrambling* [60] technique follows another similar but different philosophy: when event a is blocked, process event b until a resume itself. Used in distributed query processing, query scrambling reacts to unex-

pected delays in obtaining initial requested tuples from remote sources by performing other useful work which would normally be scheduled for a later point in the execution.

We also note that Graefe has hinted on the idea of switched execution in [31]. However, there is no discussion on how to realize it. We are the first to investigate the interleaved execution model and demonstrate its practical effectiveness.

2.7 Summary

In this chapter, we have presented MAPLE, a *M*ulti-instance-*A*ware *PL*an *E*valuation engine. MAPLE enables multiple instances of a relation in single queries to share one physical scan with limited buffer space. MAPLE is light-weight and can be easily integrated into existing RDBMS executors. We have developed a prototype in PostgreSQL, and our experimental study using the TPC-DS benchmark showed that MAPLE can significantly reduce the execution time (compared to the original plans produced by PostgreSQL).

CHAPTER 3

Collaborative Sort Executions for Relational Instances

3.1 Introduction

The previous chapter discusses how to efficiently retrieve the tuples of different instances from the common table resident on the disk, with a minimum total I/O cost. In this and the next chapter, we consider the optimization scenarios after tuples of instances have flowed up to query execution components above the storage engine.

For complex decision support queries with multiple instances, the optimized execution plans may apply various sort operations to different instances of the same table, usually in the association with sort-merge joins. Moreover, we find that the demand of sorting a table multiple times also arises in many other scenarios. 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.

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 chapter.

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 [6, 35].

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 situation, 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 chapter is organized as follows. In Section 3.2, we present some preliminaries. In Section 3.3, 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 3.4. In Section 3.5, 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 3.6. Our experimental study presented in Section 3.7 validates the effectiveness of our proposed techniques. We discuss relevant work in Section 3.8 and finally conclude in Section 3.9.

3.2 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 chapter we utilize the following main notations, some of which are borrowed from [35]:

- $s_i = sort(T, o_i)$: 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.

[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* [41].

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

$$cost(s) = 2 \times B(T) \times (\lceil log_F(\frac{B(T)}{2M}) \rceil + 1) \tag{3.1}$$

3.3 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 3.5. 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 3.6.

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 chapter, 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 [6, 35]. 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 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 [35], 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 3.1. Following [35], $cost(s_2) = \sum_{i=1}^{D(T,o')} cost(sort(se_i, o'_2))$, 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 3.1. 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 chapter. 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.

3.4 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 $|o_{22}| \geq 0$.

3.4.1 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 1 Consider the relation $T(a, b)$ in Fig. 3.1, 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. □

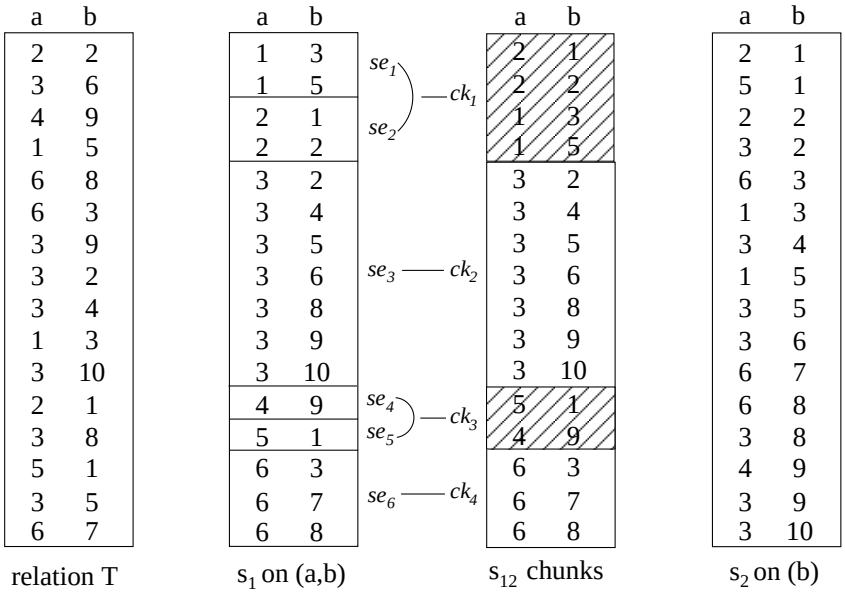


Figure 3.1: Cooperative Sorting Example: $M = 4$ and $F = 2$

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 intermedi-

ate 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 3.4.2 and 3.4.3. 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 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 2 Look at the running example in Fig. 3.1. 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 3.4.4 and the experimental results in Section 3.7, 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.

3.4.2 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 [14, 30]. 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 3 Fig. 3.2 illustrates the two initial s_1 runs ordered by $o_1 = (a, b)$ and generated from the relation T in Fig. 3.1. 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. □

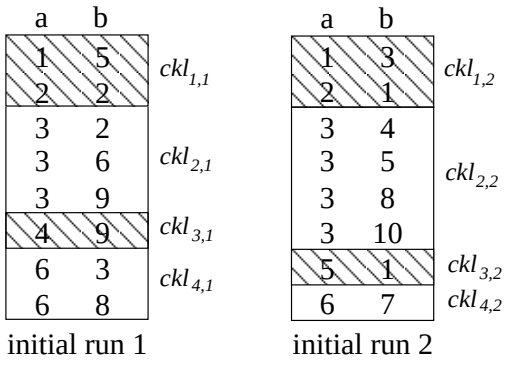


Figure 3.2: Initial s_1 Runs for Relation T in Example of Fig. 3.1

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 3.4.3.

Example 4 When merging the two initial s_1 runs in Fig. 3.2, 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. 3.1. □

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.

3.4.3 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 3.4.2, 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

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

read:

$$B(RP_i^m) + B(RP_i^d) + \sum_{k>i} B(RP_k^m) \leq M \tag{3.2}$$

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 3.2 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 3.2 can lead to "deadlock" situations. For example, consider the two initial s_1 runs shown in Fig. 3.2 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 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. 3.2 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 3.2.

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.

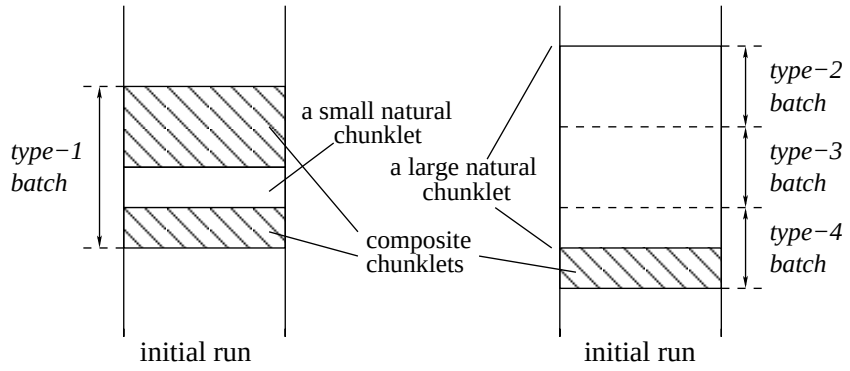


Figure 3.3: Illustration of Four Types of Tuple Batches in Initial S_1 runs

Fig. 3.3 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 7 generates the complete set of tuple batches and records them in an array TB. Algorithm 7 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 8, 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 9 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.

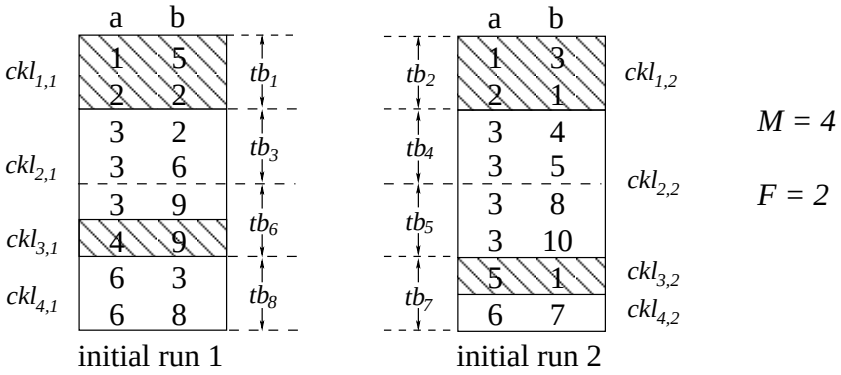


Figure 3.4: Tuple Batches of the Two Initial S_1 Runs in Fig. 3.2

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 3.1: The Entries in TB for Example in Fig. 3.4

Example 5 Fig. 3.4 shows the eight tuple batches (tb_1 to tb_8) comprising the two initial s_1 runs in Fig. 3.2. Table 3.1 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. □

Algorithm 7: `ComputeTB`

Output: a tuple batch array TB according to the to be merged F initial s_1 runs

```
1:  idx ← 1
2:  for i ← 1 to N do // N is the total number of s12 chunks
3:      if the i^th s12 chunk is composite then
4:          for j ← 1 to F do
5:              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 s1 run that corresponds to the i^th s12 chunk
6:                  TB[idx] ← TupleBatch(ckl_{i,j}) // form a type-1 tuple batch starting from the
                    head of ckl_{i,j}
7:                  idx ← idx + 1
8:      else
9:          for j ← 1 to F do
10:             if ckl_{i,j} is non-empty and has not been assigned to tuple batches yet then
11:                 if size(ckl_{i,j}) > ⌊M/F⌋ then
12:                     TB[idx] ← a type-2 tuple batch starting from the head of ckl_{i,j}
13:                     idx ← idx + 1
14:                     TB[idx] ← TupleBatch(ckl_{i,j}) // form the corresponding type-4 tuple batch
15:                     idx ← idx + 1
16:                 else
17:                     TB[idx] ← TupleBatch(ckl_{i,j}) // form a type-1 tuple batch starting from the
                        head of ckl_{i,j}
18:                     idx ← idx + 1
```

3.4.4 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 con-

Algorithm 8: `TupleBatch`

Input: $ckl_{i,j}$
Output: a type-1 (or type-4) tuple batch tb starting from the head (or interior) of $ckl_{i,j}$

1: initialize a type-1 (or type-4) tb including the whole (or part of) $ckl_{i,j}$
2: $k \leftarrow i + 1$
3: **while** *true* **do** // *check whether* $ckl_{k,j}$ *can be included by* tb
4: **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 3.2 for the l^{th} $(i \leq l < k)$ s_{12} chunk which is composite **then**
5: **break**
6: include $ckl_{k,j}$ in tb
7: $k \leftarrow k + 1$

Algorithm 9: `ScheduleReadingOfTupleBatches`

Input: TB
Output: the order on which tuple batches in TB will be read during the actual run merging

1: initialize an empty tuple batch pool P
2: $i \leftarrow 1$
3: **while** $i \leq length(TB)$ **do**
4: read $TB[i]$ whenever enough memory space is available
5: $tb \leftarrow TB[i]$ // *mark this tuple batch for later reference*
6: **if** $TB[i]$ is a type-1 tuple batch **then** // *otherwise it must be type-2*
7: $i \leftarrow i + 1$
8: **else**
9: 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 $TB[i+1]$
10: $i \leftarrow i + 2$
11: **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**
12: **if** P is non-empty **then**
13: driven by the merge progress, read on a specific order all the type-3 and type-4 tuple batches in P
14: restore P to be empty

sists 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 3.1) plus the cost C_{is} of performing internal sortings on composite chunklets within initial s_1 runs; (2) the cost

$C_{s_{12}\rightarrow s_1}$ of deriving s_1 which is equal to the total cost of performing internal sortings for all the composite s_{12} chunks; (3) the cost $C_{s_{12}\rightarrow s_2}$ 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.

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

$$N = \lceil D(T, o_{11})/k \rceil \tag{3.3}$$

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

$$C_{is} = \frac{B(T)}{2M} \times N \times cpu\_cost(2M/N) \tag{3.4}$$

$$C_{s_{12}\rightarrow s_1} = N \times cpu\_cost(k \times B(T)/D(T, o_{11})) \tag{3.5}$$

Case 2: $B(T)/D(T, o_{11}) > 0.5M$.

In this case, all s_{12} chunks are natural, and

$$N = D(T, o_{11}) \tag{3.6}$$

$$C_{is} = C_{s_{12}\rightarrow s_1} = 0 \tag{3.7}$$

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.

3.4.5 Extensions

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

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.

Adapting to Other Merge Patterns

Our description of cooperative sorting in Section 3.4.2 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* [40]. 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.

3.5 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.

3.5.1 K-way Cooperative Sorting

In Section 3.4, 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 $o'_{pi} = ((o_{p1} \cdot o_{p2}) \cdot o_{p3}) \cdot ... \cdot o_{pi}$ $(1 < i \leq k)$ 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 1 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 3.4.4 shows that it is not necessary to consider k-way cooperative sorting for $k > 2$.

Theorem 3.1. *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.1.

3.5.2 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 3.1, 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*.

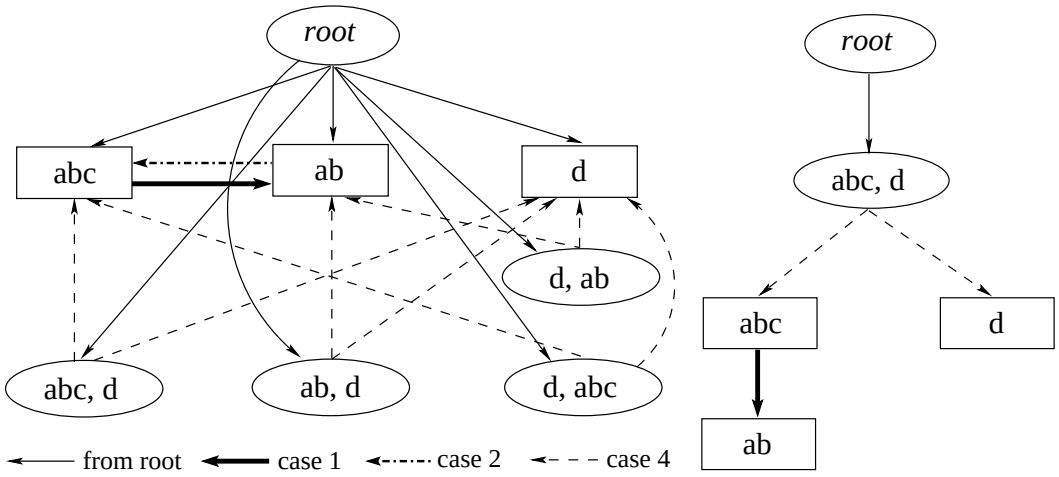
Each sort node $u \in V_a$ is associated with a sort order, denoted by $order(u)$. For each sort operation $s = sort(T, o) \in S$, we create a sort node $u \in V_a$ with $order(u)$ o. Each directed edge (u, v) from sort node u to sort node v is associated with $cost(u, v)$ equal to the cost of sorting 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 3.3.

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 3.4. $cost(w, v)$ 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.

Example 2 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



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

Figure 3.5: An Example of Multiple Sorting Optimization

Fig. 3.5, where the sort (resp. cooperative sort) nodes are represented by rectangles (resp. ellipses). The computed Steiner tree for this graph is shown in Fig. 3.5. 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 [39], 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 [29]. 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 [15, 37] 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.

3.5.3 Sort-sharing-aware Query Optimization

The optimization techniques in Section 3.5.2 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.

We first discuss how to extend the system-R [56] style query optimizer, which is also adopted by PostgreSQL. We have modified the PostgreSQL optimizer for our experiments. After that, we discuss how to extend the Volcano [34] style query optimizer.

System-R 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 [56].

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 3.5.2 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.

Volcano Style Query Optimizer

The Volcano method is based on an AND-OR DAG representation [53], [34] 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 AN(op) to denote an AND-node according to an operation op; use 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 AN(s) and a new OR-node ON($T, \{o\}$). AN(s) corresponds to the physical sort operation s, and ON($T, \{o\}$) corresponds to the sorted T with order o. Suppose the caller AND-node of s is AN(c), then ON($T, \{\}$) is originally one child of AN(c). Now, this chain AN(c) $\rightarrow$ ON($T, \{\}$) in the DAG is replaced with a new chain AN(c) $\rightarrow$ ON($T, \{o\}$) $\rightarrow$ AN(s) $\rightarrow$ 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 AN(ds) to form a new chain ON($T, \{o_2\}$) $\rightarrow$ AN(ds) $\rightarrow$ 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 AN(ps) to form a new chain ON($T, \{o_2\}$) $\rightarrow$ AN(ps) $\rightarrow$ 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, AN(s_{12}), AN($s_{12} \twoheadrightarrow s_1$) and AN($s_{12} \twoheadrightarrow s_2$), as well as a new OR-node ON($T, \{o_1 \uplus o_2\}$), to form two new chains ON($T, \{o_1\}$) $\rightarrow$ AN($s_{12} \twoheadrightarrow s_1$) $\rightarrow$ ON($T, \{o_1 \uplus o_2\}$) $\rightarrow$ AN(s_{12}) $\rightarrow$ ON($T, \{\}$) and ON($T, \{o_2\}$) $\rightarrow$ AN($s_{12} \twoheadrightarrow s_2$) $\rightarrow$ ON($T, \{o_1 \uplus o_2\}$) $\rightarrow$ AN(s_{12}) $\rightarrow$ 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., [54]) can be utilized to find the optimal sort-sharing-aware execution plan.

3.6 Discussions

In this section, we discuss the incorporation of ascending and descending orders into the sort sharing techniques (Section 3.6.1). We present a dynamic way (Section 3.6.2) 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 3.6.3). Finally, we briefly discuss the impact of functional dependency and attribute correlation on sort sharing optimization (Section 3.6.4).

3.6.1 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.

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.

3.6.3 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 [63], 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.

3.6.4 Functional Dependency and Attribute Correlation

The functional dependencies existing among relational attributes have been exploited for the purpose of *sort order reduction* [58], 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.

3.7 Performance Study

We validated our ideas using a prototype built in PostgreSQL 8.3.5 [2]. 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 3.5.3. 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.

3.7.1 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

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

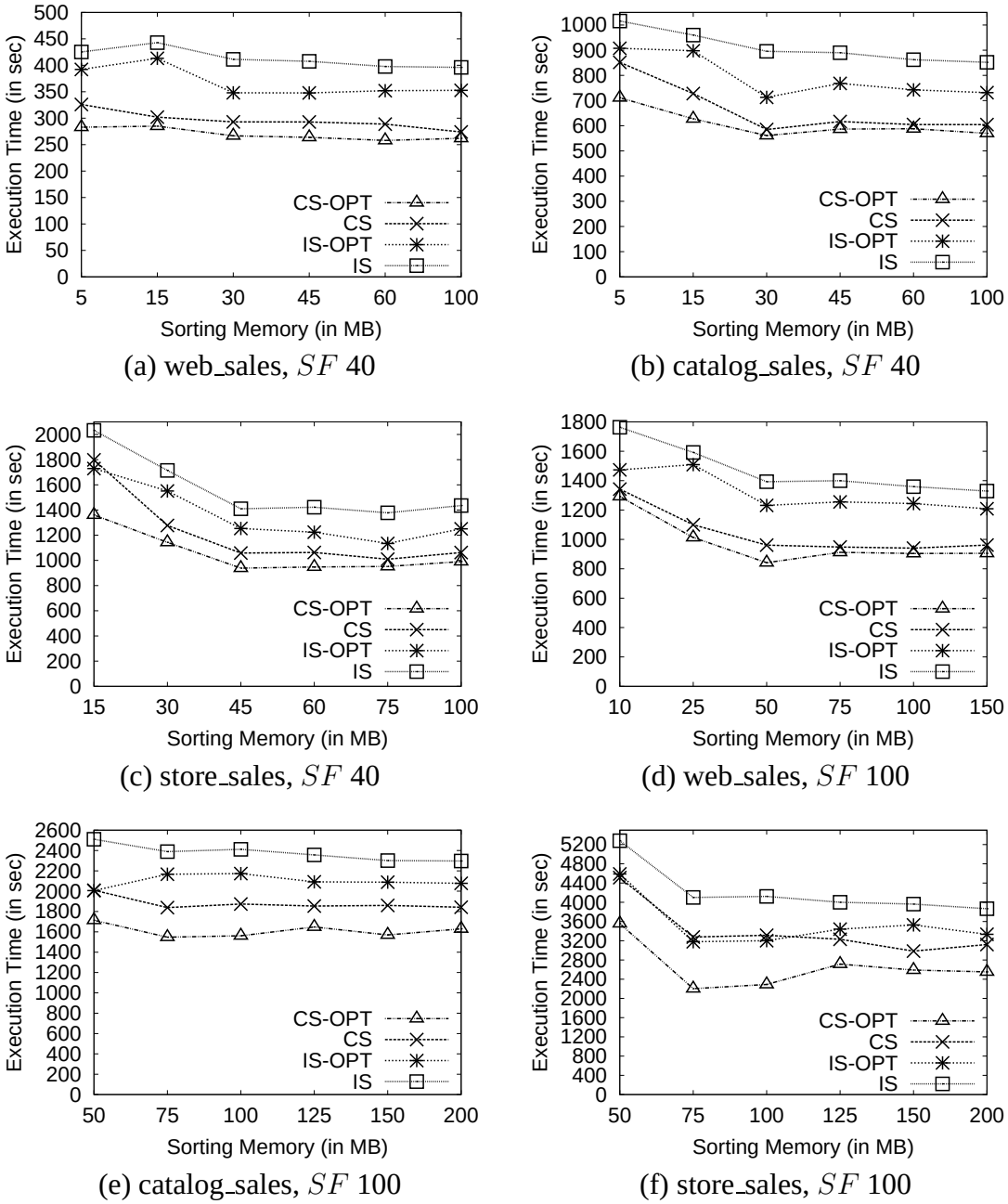
Table 3.2: Tested TPC-DS Dataset

and thus satisfy case 3.

We generate six concrete queries with the above query template by using three different relations from the TPC-DS [3] 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 3.2. 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.



(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

Figure 3.6: Performance Comparison on TPC-DS Dataset

General Results

Fig. 3.6 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.

	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 3.3: Component Costs of CS and IS

The detailed breakdown of the various cost components for CS and IS are shown in Table A.1 in Appendix A.2. The meanings of these cost components are given in Table 3.3.

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. 3.6(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 A.1, 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 3.3.

Effect of Result Sharing

As discussed at the beginning of Section 3.4.1, 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. 3.7 compares the performance of the query on web_sales with SF 40; the comparison for other queries have similar trends and are omitted.

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. 3.6).

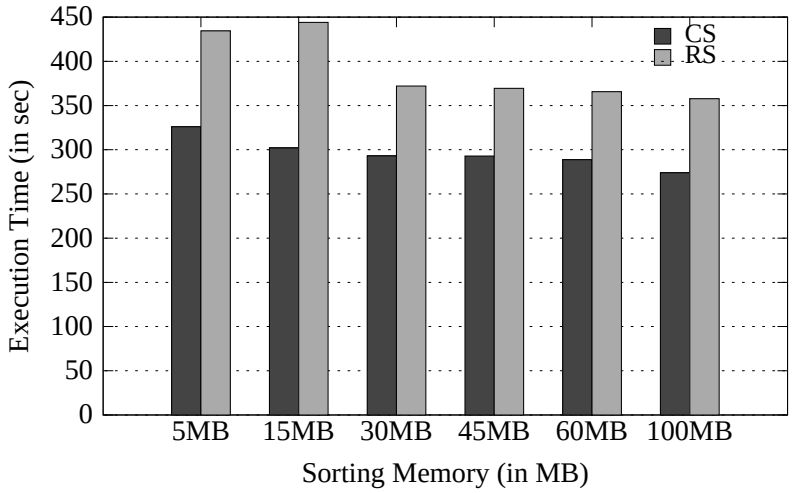


Figure 3.7: Comparison of CS with RS on web_sales, SF 40

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 3.7.1. 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).

Only the results of web_sales with SF 40 are shown in Fig. 3.8. 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.

3.7.2 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

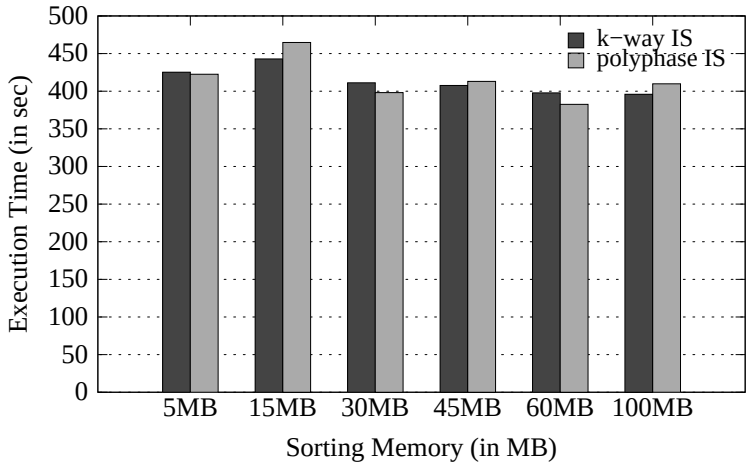


Figure 3.8: Comparison of K-way IS with Polyphase IS on web_sales, SF 40

previous section on the synthetic tables to compare the performance of CS and IS.

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.

The experimental result is shown in Fig. 3.9. 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

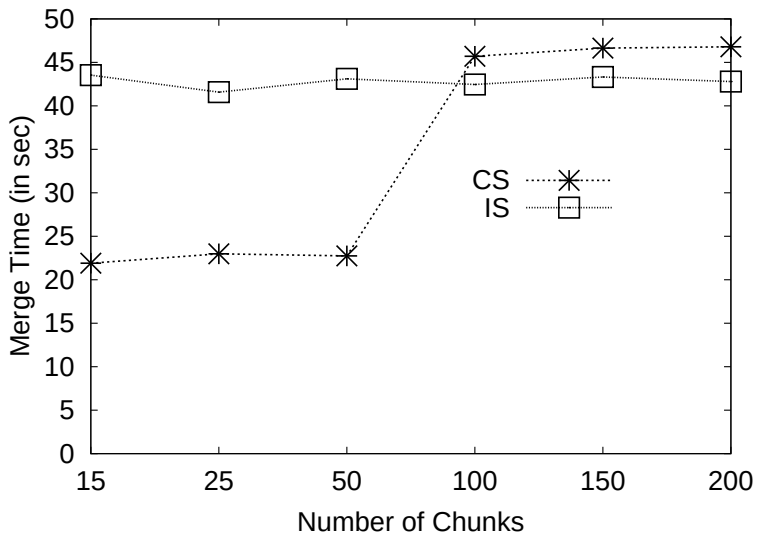


Figure 3.9: Varying Total Number of s_{12} Chunks

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.

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. 3.10 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.

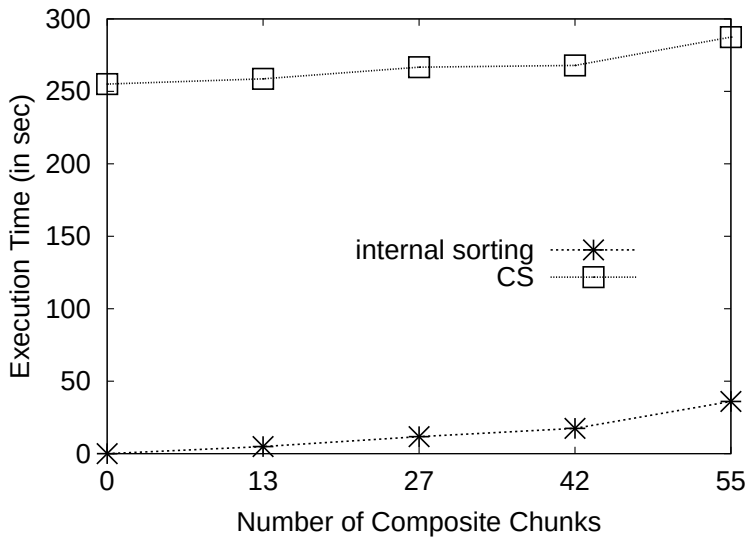


Figure 3.10: Varying Number of Composite S_{12} Chunks

3.7.3 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 $key1$ and $key2$ satisfy case 4.

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 3.4: TPC-DS Dataset for Comparing Performance of Index Construction

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 3.4. The scale factor SF values used are 40 and 100. Another experimental parameter that we varied is the available sorting memory dedicated to each

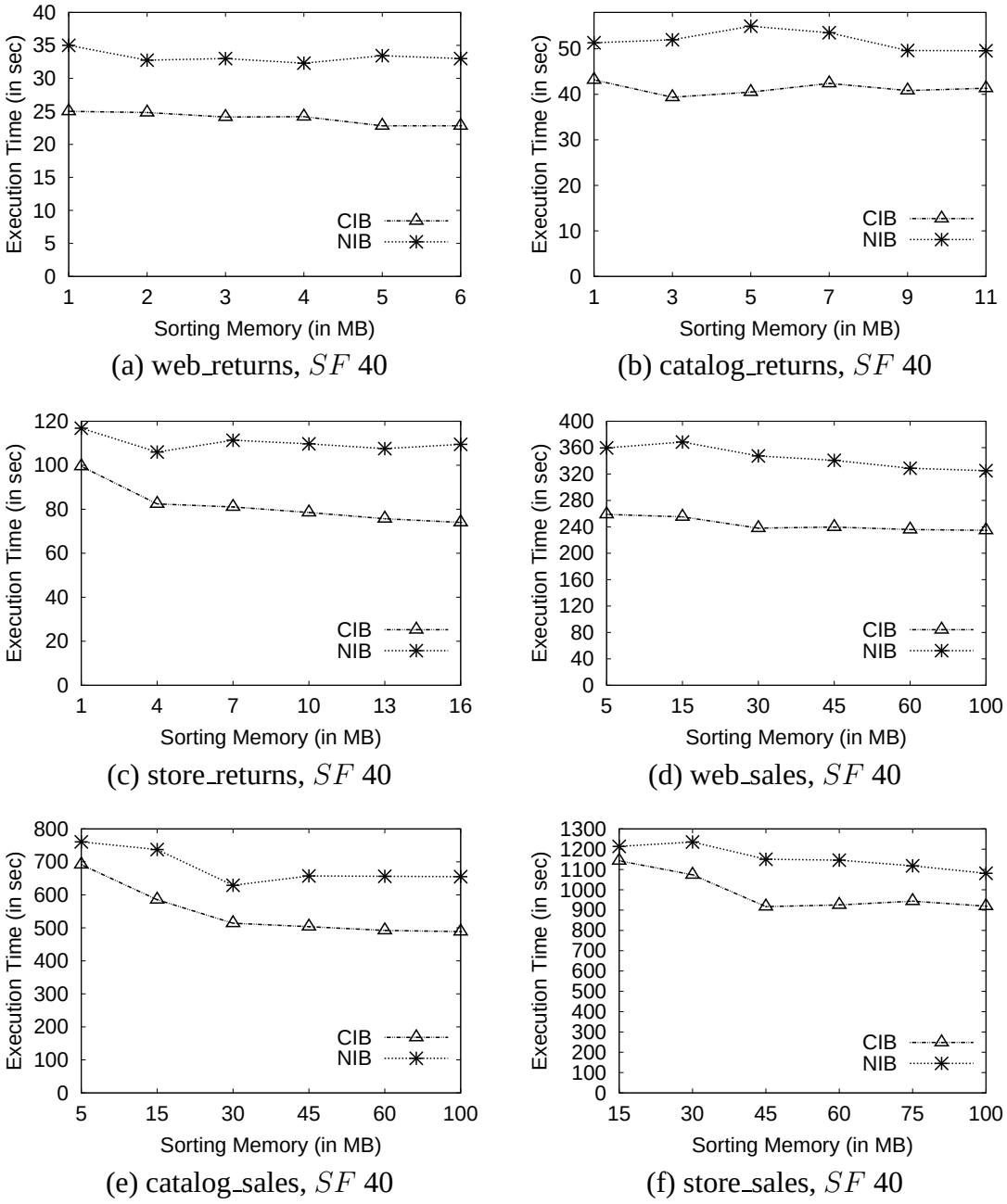
	notation	description
CIB	$RF_{cs}(s_{12})$	initial run formation cost for s_{12} (i.e., creating initial s_1 sorted runs)
	$RM_{cs}(s_{12})$	run merge cost for s_{12} (i.e., creating s_{12} chunks)
	$RM_{cs}(s_2)$	run merge cost for s_2 (i.e., merging s_{12} chunks to derive s_2)
	$SC_{cs}(s_{12})$	cost 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 3.5: Component Costs of CIB and NIB

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

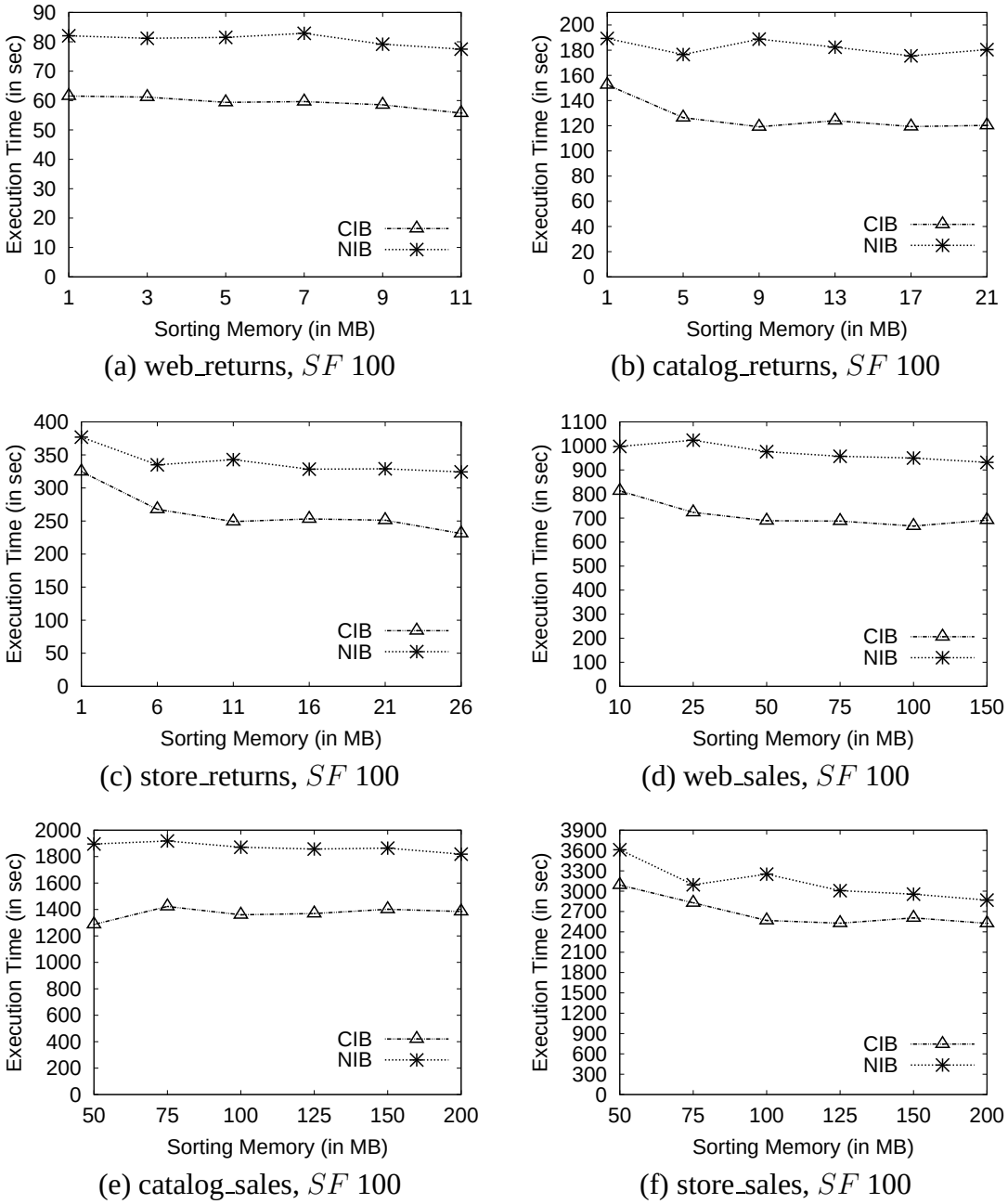
(d) web_sales, SF 40

(e) catalog_sales, SF 40

(f) store_sales, SF 40

Figure 3.11: Performance Comparison on TPC-DS Dataset, with SF 40

Figs. 3.11 and 3.12 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 A.2 and A.3 in the Appendix A.2. The meanings of these cost components are given



(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

Figure 3.12: Performance Comparison on TPC-DS Dataset, with SF 100

in Table 3.5. 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 A.2 and A.3 is marked as zero. Note that for every row in Tables A.2 and A.3, $RM_{is}(s_2)$ is always zero and is thus omitted. $SC_{cs}(s_1)$ is not separately listed but merged into $LD_{cs}(s_1)$. For each row in

Tables A.2 and A.3, 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. 3.12. In terms of relative improvement, the average percentage 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. 3.12. 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 3.7.1). 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.

3.7.4 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 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 `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)
```

With 50MB sorting memory, the optimal plans generated by the original PostgreSQL optimizer for these two queries are shown in Fig. 3.13.

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

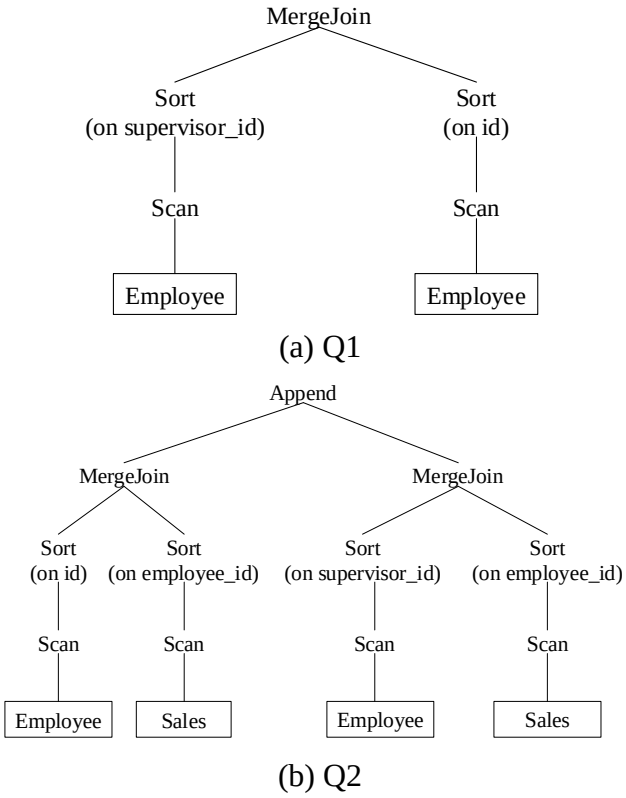


Figure 3.13: The Optimal Plans for Q1 and Q2 by the Original PostgreSQL Optimizer

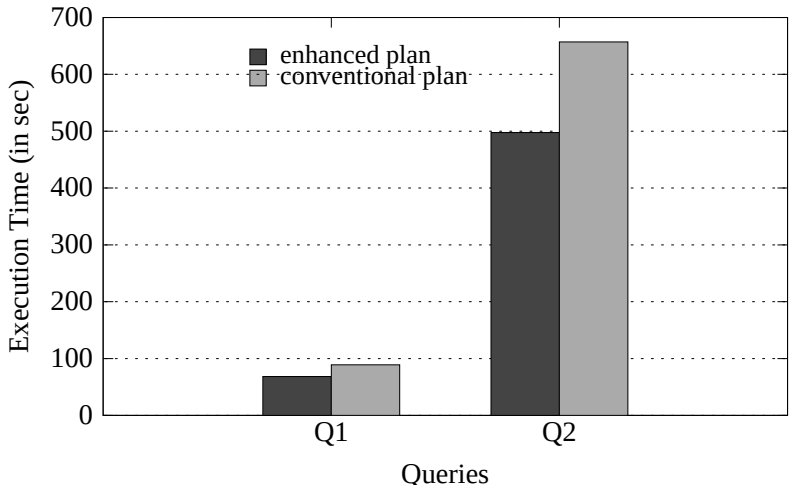


Figure 3.14: Query Execution Times of Q1 and Q2

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. 3.14. 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).

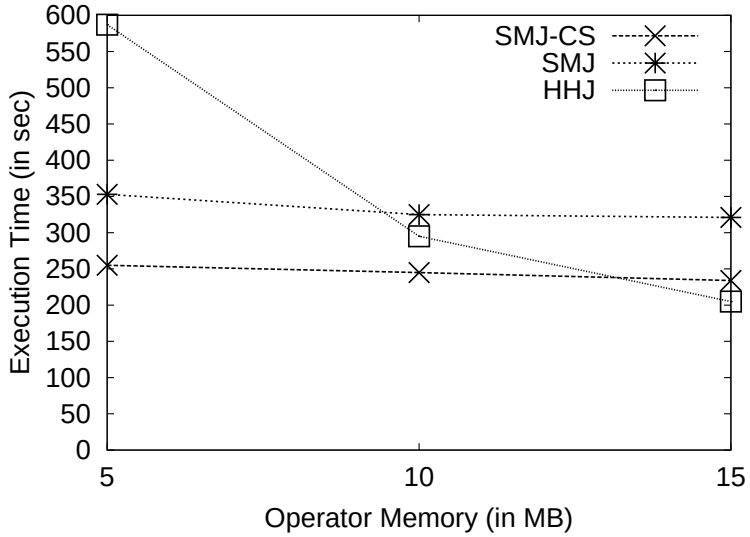


Figure 3.15: Plans Considered During Query Optimization for Q1

Fig. 3.15 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.

3.8 Related Work

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 ([40, 51, 55, 64, 65, 67]). 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. Knuth's classical text [40] 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 ([23, 25, 59]). Larson [44] introduced a cache-aware replacement selection that works for various length keys. There are also many techniques to speed up the run merge phase ([64, 65, 67]), 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 [40], read-ahead [65], etc. Much research has been done on adaptive sorting [24] exploiting near-sortedness. The survey [32] by Graefe discussed how sorting is implemented in database systems with many tricks and optimizations. Specifically, [32] 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 3.4.1. However, neither analytical nor experimental study on the effectiveness of the proposed

approach were conducted. Moreover, [32] did not generalize this special instance to the general case 3.

Simmen et al. [58] 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 [49, 62, 48], which are all independent and complementary to our work.

In [35], 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, [35] and our work are complementary to each other. A similar idea to partial sorting was considered previously in [6] 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` [13] and *cooperative scan* [69].

3.9 Summary

In this chapter, 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 partic-

ular 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.

CHAPTER 4

Self-Join Processing for Relational Instances

4.1 Introduction

In Chapter 2 and Chapter 3, we have studied the optimizations for relational instances that are related to the table scan operation and the table sort operation respectively. In this chapter, we examine the self-join operation, which is a join operation that relates data within a relation by joining the relation with itself. In other words, a self-join involves two instances of the same relation. More specifically, we consider self-joins with the join predicates involving two distinct attributes (i.e., $R_1.A\ op\ R_2.B$, where R_1 and R_2 are instances of relation R). We observe that this type of self joins occur frequently in many recently emerging database applications, such as location-based service (LBS), RFID data management, sensor networks, network management etc. Below are two examples:

Example 1: *Consider a database of moving objects with a user-defined view, TRAJECTORY, storing the line segments of the trajectories of many moving objects. The schema of TRAJECTORY is (objID, location1, location2, time1, time2). Each record describes the movement of an object from location1 at time1 to location2 at time2. An example LBS query is: for each location L, return all the pairs of objects O1 and O2, such that O1 arrived at L shortly, say within t time units, before O2 moved out of L. This query involves a self equi-join condition:*

```
SELECT *
FROM TRAJECTORY O1, TRAJECTORY O2
WHERE O1.location2 = O2.location1
      AND O1.time2 > O2.time1 − t
      AND O1.time2 < O2.time1
```

Example 2: *Consider a table storing the readings from a network of temperature sensors with schema: (sensorId, temperature, timestamp). A view STATS is created on top of the table with schema: (sensorId, hourId, avgTemp, maxTemp, minTemp). Each row in STATS provides the average, maximum and minimum temperatures reported by a sensor over a particular hour of the day. A user may be interested in finding out the pairs of sensors, S1 and S2, such that the average temperature reported by S1 is equal to or lower by at most 2 degrees than the minimum temperature reported by S2, within a certain time proximity, say one hour. This query involves two self band-join [22, 46] conditions*[1]*:*

```
SELECT *
FROM STATS S1, STATS S2
WHERE S1.avgTemp <= S2.minTemp
```

[1]A general band-join condition has the form $R.A - c_1 \leq S.B \leq R.A + c_2$, where c_1 and c_2 are constants but can not both be zero.

AND $S1.avgTemp >= S2.minTemp - 2$

AND $S1.hourId <= S2.hourId + 1$

AND $S1.hourId >= S2.hourId - 1$

AND $S1.sensorId <> S2.sensorId$

Moreover, self-join is also common in RDF data management where self-join is used to relate the subjects and objects of a triple table, and in the publication of relational data as XML where XML queries (e.g. XQuery) over XML views are translated into self-joins of base relational tables.

Despite the importance and prevalence of self-joins, there however have been surprisingly few research efforts on optimizing them.

On the one hand, existing solutions either employ join indexes [45] or handle the special case where the join attributes are on the same attribute (e.g., $R_1.A = R_2.A$) [16, 27]. As one can see from the examples, many emerging queries involve self joins on two distinct attributes. While index-based techniques could be applied to the problem, it is possible that indexes do not exist, especially when the queries are ad-hoc and/or the join attributes are derived and computed from user defined functions as shown in Example 2. Even when indexes exist, they may not be useful. For example, if the join selectivity is high (i.e. a lot of join results), then indexes, especially the non-clustered ones, are not beneficial.

On the other hand, conventional join algorithms, such as Sort-Merge Join (`SMJ`) and Hybrid Hash Join (`HHJ`), treat the two instances of the same relation as distinct relations. As such, they miss the opportunities to enhance the processing performance, particularly in keeping the I/O cost low.

To improve performance, we need a scheme that can take advantage of the fact that the two inputs of a self-join operator are instances of the same underlying relation. Towards this end, we propose a novel and efficient self equi-join algorithm called `SCALE`

(Sort for Clustered Access with Lazy Evaluation), which is also easily extendible to handle self band-joins [22, 46]. SCALE first sorts the relation, say R, on one of the join attributes, say A, to produce a sorted sequence, denoted as $\mathcal{S}_A(R)$ (when R has already been ordered by some join attribute, this sorting step can be avoided). Now, given a tuple t with $t.B = x$, where B is the other join attribute, all the matching tuples whose $R.A$ value is x are clustered within $\mathcal{S}_A(R)$. As such, by scanning $\mathcal{S}_A(R)$, we have two possible cases. First, for each tuple t with $t.B = x$, we have a clustered access of all matching tuples if t co-exists with them in memory. In this case, we can generate and produce the join results at no extra cost since this is within a single scan of $\mathcal{S}_A(R)$. Second, for a tuple t with $t.B = x$ for which such clustered access is not possible (e.g., matching $R.A$ tuples may not co-exist with t in memory), it is buffered and possibly spilled to disk to defer the join processing to a later time. Such lazy evaluation minimizes the need for "random" accesses to the matching tuples.

To support clustered access and lazy evaluation, the memory space has to be effectively allocated between these two tasks. To optimize performance, SCALE adopts a cost-based approach to manage the memory allocation for clustered access and lazy evaluation. SCALE is also able to handle the situation where the two joining instances of the same relation are associated with different tuple selection and projection predicates. Moreover, we can improve SCALE with sideways information passing techniques to further reduce the cost when many tuples have no join matchings.

Our analytical study shows that SCALE degenerates gracefully to Sort-Merge Join (SMJ) in the worst case. We have also implemented SCALE in PostgreSQL [2], and the results of our extensive experimental study confirms our analytical results. Moreover, it shows that SCALE outperforms both Sort-Merge Join and Hybrid Hash Join by 20% - 40% in (almost) all cases.

The rest of the chapter is organized as follows. Section 4.2 surveys the related work.

Section 4.3 discusses the technical details of the SCALE algorithm. We then present a thorough analytical study of SCALE in Section 4.4. An extensive experimental study is presented in Section 4.5. In Section 4.6, extensions to SCALE supporting self band-join and side-ways information passing are proposed. We conclude the chapter in Section 4.7.

4.2 Related Work

The join operation is one of the most time-consuming and data-intensive operations. Therefore, it is critical to implement joins in the most efficient way possible. The join operation has been studied and discussed extensively in the literature (see [31] for a comprehensive survey). Common ad-hoc join techniques can be classified into three broad categories: nested-loop join, sort-merge join [9] and hash-based join [10, 21, 26, 47]. Recently, Goetz proposed *g-join* [33], a generalized join algorithm, intending to replace the above join techniques. The major advantage of *g-join* over sort-merge join and hash-based join lies in its robustness rather than its potential performance gain. This implies that *g-join*'s costs should be comparable with those of sort-merge join and hybird hash join. Therefore, in our experiments we did not directly compare with *g-join*.

Nevertheless, to the best of our knowledge, only a few works specifically focus on self-join processing. In [45], Lei and Ross proposed the *Stripe* algorithm for performing a join with a *join index* [61], which maintains pairs of identifiers of tuples that would match in case of a join between two relations. Stripe join was designed for general join processing but is particularly efficient for self-joins. However, the applicability of Stripe join is highly dependent on the availability of the suitable join index, which must have been materialized and maintained by the database systems before the join execution. In contrast, our algorithm is more useful in application contexts identified in Section 4.1,

e.g. the join attribute is a derived one that is not indexed. The problem of self-join size estimation has been tackled in both centralized [7] and large-scale distributed [52] database systems.

In this chapter, we mainly discuss the situation that the self-join predicate involves two distinct attributes. In the special case where the join predicate is an equi-join over the same attribute (e.g., $R_1.A = R_2.A$), the join may be evaluated by partitioning the base relation according to the attributes involved in the equality join predicates and then performing a simplified join operation separately on each partition. The identification and execution of such a special case of self-join were addressed previously in [16, 27]. Our algorithm is actually applicable to such special cases and essentially behaves like the partition-based evaluation strategies in [16, 27]. Therefore, the performances are expected to be comparable for such special cases.

4.3 The SCALE Algorithm

In this section, we consider a self equi-join $R_1.A = R_2.B$. Both R_1 and R_2 are instances of the same relation R, where A and B are two (single or composite) attributes.

In the self-join, each tuple t in R is associated with two sets of matching tuples referred to as its *left-matching* and *right-matching* tuples. A tuple t' is a left-matching (resp. right-matching) tuple of t if $t.A = t'.B$ (resp. $t'.A = t.B$). We define the *left-join* (resp. *right-join*) of t to be the result of joining t with its left-matching (resp. right-matching) tuples. Thus, the self-join of R can be computed as the union of the left-join of each tuple in R or symmetrically as the union of the right-join of each tuple in R.

4.3.1 Overview

Without loss of generality, we present our self-join evaluation algorithm SCALE in terms of the union of the right-join of each tuple. The choice between a left-join or right-join evaluation of R can be decided in a cost-based manner using the cost model in Section 4.4.1.

To evaluate the self-join of R in terms of right-joins, SCALE first sorts R in ascending order of A to produce the sorted table $\mathcal{S}_A(R)$ (sorting is unnecessary if R has already been ordered by A due to the existence of clustered indices). As such, for each tuple in R, all its right-matching tuples are clustered together in $\mathcal{S}_A(R)$. SCALE then processes $\mathcal{S}_A(R)$ in at most two passes as follows. In the first pass, SCALE maintains three main-memory buffers, namely, *main*, *hold*, and *defer* buffers. SCALE sequentially scans the tuples in $\mathcal{S}_A(R)$ into the main buffer and computes the right-joins between the newly scanned tuple and the existing tuples in the main and hold buffers. The main buffer is managed using a replacement policy to evict tuples when the buffer becomes full. Due to tuple evictions, the right-join of a tuple t could be partially processed when t is evicted out of the main buffer. Thus, at the same time that a tuple t is being evicted from the main buffer, we can classify t into one of three possible states (*complete*, *prefix-complete*, or *incomplete*) as follows. If the right-join of t has been completed, t is classified as complete; otherwise, if the right-join between t and all its right-matching tuples that precede t in $\mathcal{S}_A(R)$ has been completed, t is classified as prefix-complete; otherwise, t is classified as incomplete.

Before evicting a tuple t from the main buffer, SCALE first determines the state of the right-join of t. If t is complete, then t is simply evicted from the main buffer; otherwise, t needs to be buffered elsewhere (either in the hold or in the defer buffer) for subsequent processing of its remaining right-join. Specifically, if t is incomplete, then t is transferred to the defer buffer; otherwise, t must be prefix-complete and it is transferred to the hold

buffer. The tuples in the hold buffer will "wait" for their unread right-matching tuples in $\mathcal{S}_A(R)$ to be scanned into the main buffer to complete their right-join processing during the first pass. The tuples in the defer buffer will complete their right-join computation in the second pass.

At the end of the first pass, if the defer buffer is empty, this means the self-join of R has been completely evaluated and SCALE will therefore terminate; otherwise, SCALE will proceed with the second pass to process the tuples in the defer buffer.

In the second pass, SCALE computes the remaining right-join of each tuple in the defer buffer by performing a merge join of the tuples in $\mathcal{S}_A(R)$ and the defer buffer. The tuples in $\mathcal{S}_A(R)$ (which are already sorted on A) will be scanned sequentially to merge with the tuples in the defer buffer which will be retrieved in ascending order of B.

To facilitate the right-join processing, both the hold and defer buffers are organized as min-heaps ordered on the B values. When the hold/defer buffer overflows, SCALE flushes a sorted run from the appropriate heap to the disk. Thus, SCALE will subsequently need to load back the disk-based sorted-runs during processing the right-joins, and an additional buffer, referred to as the *run* buffer, is used for this purpose.

It is important that a join computation that has been performed in the first pass is not computed again during the second pass. SCALE ensures this by simply recording for each tuple t spilled to the defer buffer, the ranks of the first and last tuples in $\mathcal{S}_A(R)$ (denoted by *first(t)* and *last(t)* respectively) that right-join with t during the first pass. During the second pass, for each tuple t in the defer buffer, SCALE computes the right-join of t with a right-matching tuple t' in $\mathcal{S}_A(R)$ if and only if the rank of t' either precedes *first(t)* or succeeds *last(t)*. For correctness, the tuple eviction policy of the main buffer ensures that if two tuples t and t' in the main buffer have the same A values and t is evicted before t', then t must precede t' in $\mathcal{S}_A(R)$. Thus, SCALE correctly computes the self-join of R without missing out any join result and without computing the same join more than once.

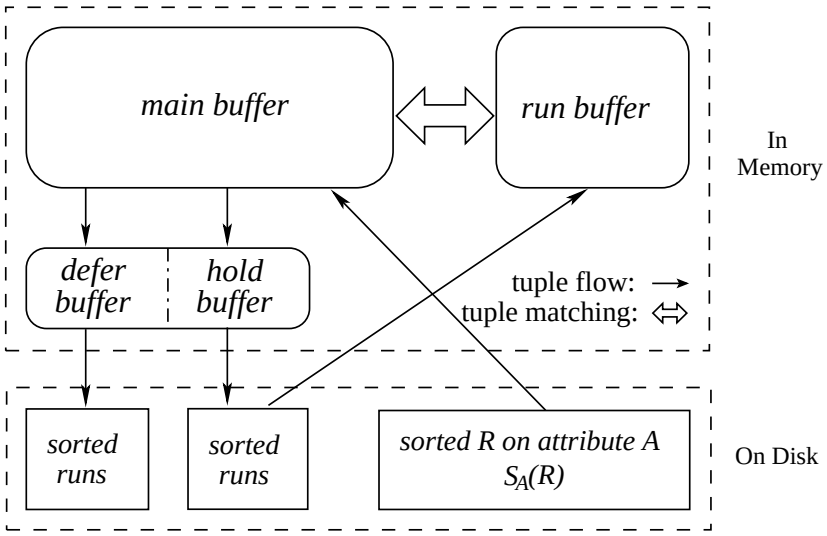


Figure 4.1: SCALE execution during the first pass of processing $\mathcal{S}_A(R)$

Fig. 4.1 shows the execution of SCALE during the first pass of processing $\mathcal{S}_A(R)$. The functions of each component, the tuple flow and the tuple matching procedure will be elaborated in the following subsection.

4.3.2 Algorithm Details

For a tuple t, we denote the set of its right-matching tuples in $\mathcal{S}_A(R)$ by *RM(t)*. During the first pass of processing $\mathcal{S}_A(R)$, tuples in *RM(t)* can be divided into three subsets:

- *$RM_1(t)$*: the tuples in *RM(t)* that have left the main buffer when t is read into the main buffer.
- *$RM_2(t)$*: the tuples in *RM(t)* that will meet and join with t in the main buffer.
- *$RM_3(t)$*: the tuples in *RM(t)* that will only be read into the main buffer after t has left the main buffer.

For a tuple t currently in the main buffer, if it is evicted, there will be four possible cases according to the distribution of *RM(t)* tuples within *RM$_1$(t)* and *RM$_3$(t)*, as illustrated in Table 4.1.

case #	*RM$_1$(t)*	*RM$_3$(t)*	right-join state of t
1	empty	empty	complete
2	non-empty	empty	incomplete
3	non-empty	non-empty	incomplete
4	empty	non-empty	prefix-complete

Table 4.1: The possible distribution of *RM(t)* tuples within *RM$_1$(t)* and *RM$_3$(t)*, along with the corresponding right-join state of t

Tuple Eviction Policy of the Main Buffer

Whenever the main buffer becomes full, each tuple inside is classified into one of the above four cases and is assigned with an eviction priority. Tuples with higher priorities will get evicted first. The destination of an evicted tuple is dependent on its right-join state, as discussed in Section 4.3.1. The ultimate goal of the tuple eviction policy is to improve tuples' clustered access to their right-matching tuples and thus maximize the total number of tuples reaching a complete right-join state. Besides, a secondary goal is to produce early join results at high rates. We describe our tuple eviction policy by comparing the eviction priorities of two arbitrary tuples t and t'.

First of all, when $t.A = t'.A$, if t precedes t' in $\mathcal{S}_A(R)$, then t has a higher eviction priority than t'. This rule ensures the correctness of `SCALE` as discussed at the end of Section 4.3.1. When $t.A \neq t'.A$, the following heuristic rules are applied:

• When t is in cases 1 or 2 and t' is in cases 3 or 4, t has a higher eviction priority than t', since by staying in the main buffer t' could continue joining with more *RM$_3$(t')* tuples being or to be read from $\mathcal{S}_A(R)$ and thus is likely to be classified into cases 1 or 2 when it is evicted in the future.

• When both t and t' are in one of cases 1 and 2, they have equal eviction priorities.

• When t is in case 3 and t' is in case 4, t has a higher eviction priority than t', as by staying in the main buffer t' still has the chance to reach the complete state later.

• When both t and t' are in case 3, they have equal eviction priorities.

• When both t and t' are in case 4, t has a higher eviction priority than t' if $t.B > t'.B$, since as such $RM_3(t')$ tuples will be read earlier than $RM_3(t)$ tuples and thus by staying in the main buffer t' has a bigger chance to reach the complete status later.

It is obvious that the above rules will cover all scenarios between t and t', and thus can produce a global ranking of the eviction priorities of all tuples in the main buffer.

Processing Tuples in the Hold Buffer

For each tuple t transferred from the main buffer to the hold buffer, $RM_1(t)$ is empty and $RM_3(t)$ is non-empty. During the first pass of processing $\mathcal{S}_A(R)$, t will wait in the hold buffer until the $RM_3(t)$ tuples are sequentially scanned into the main buffer, and then t will be moved into the run buffer so as to complete its remaining right-join processing with the $RM_3(t)$ tuples[2].

Since the $\mathcal{S}_A(R)$ tuples are sorted on A, tuples in the hold buffer need to be retrieved and processed in the order of their B values. To achieve this, the hold buffer is organized as a min-heap ordered on the B values and is used to generate disk-based sorted tuple runs when buffer overflows. Once some tuples in the hold buffer need to join with their remaining right-matching tuples newly read into the main buffer, they are retrieved into the run buffer by progressively reading and (recursively) merging the sorted runs, while the min-heap may be simultaneously dumping and appending tuples to some existing or new runs on the disk. Fig. 4.2 shows the procedure of flushing hold buffer tuples from

[2]In the hold buffer, multiple tuples could share the same attribute B value. If their total size is larger than the size of the run buffer, then the join processing for them degrades to a nested loop join, and each tuple has to be moved into the run buffer more than once.

the min-heap to the sorted runs and then reading them back to the run buffer.

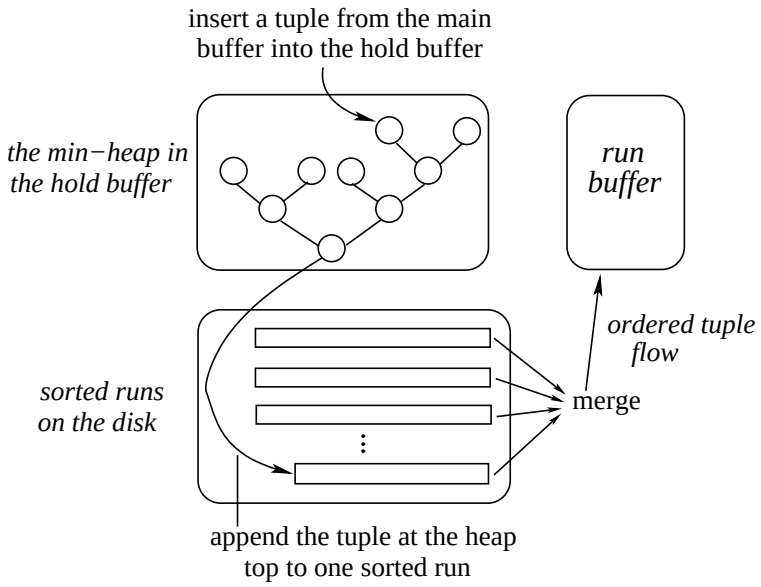


Figure 4.2: Insert tuples to the hold buffer as well as read them into the run buffer

Processing Tuples in the Defer Buffer

For each tuple t transferred from the main buffer to the defer buffer, at least $RM_1(t)$ is non-empty. Since $RM_1(t)$ tuples precede t in $\mathcal{S}_A(R)$, t is not able to join with $RM_1(t)$ tuples during the first pass of processing $\mathcal{S}_A(R)$. As a result, t will only complete its right-join with tuples in $RM_1(t)$ and $RM_3(t)$ after the first pass of processing $\mathcal{S}_A(R)$ ends, via a merge join of the tuples in $\mathcal{S}_A(R)$ and the tuples in the defer buffer. For the purpose of merge join, $\mathcal{S}_A(R)$ tuples will be sequentially scanned once again into the main buffer, and defer buffer tuples will be read into the run buffer in the order of their B values. Similarly to the hold buffer, we organize the defer buffer as a min-heap ordered on the B values and generate disk-based sorted tuple runs when buffer overflows. However, although generated during the first pass of processing $\mathcal{S}_A(R)$, these sorted runs are (recursively) merged in the run buffer during the second pass.

Note that actually tuples in the hold buffer can be processed together with tuples in the defer buffer via the merge join after the first pass of processing $\mathcal{S}_A(R)$. However, processing tuples in the hold buffer during the first pass can incur less I/O cost as analyzed in Section 4.4.1, and can generate join results earlier.

Memory Allocation for Buffers

In the second pass of processing $\mathcal{S}_A(R)$, `SCALE` only maintains the main buffer and the run buffer to conduct the merge join. Both buffers dynamically share all available memory space. As such, the run buffer may be able to grab enough memory to conduct a single-step merge of sorted runs of defer buffer tuples.

In the first pass, both the main buffer and the run buffer will have predetermined sizes, while the hold buffer and the defer buffer will dynamically share the remaining memory space so as to maximize the average lengths of (and minimize the total number of) generated sorted runs. We thereby develop a cost-based heuristic to optimize the memory allocation for buffers in this pass.

Suppose the size of R, in terms of the number of pages, is N and the total available join memory is M pages from R. The sizes of tuples in the hold buffer and in the defer buffer are N_1 and N_2 respectively. The memory allocation scenario would affect the I/O costs incurred by tuples in the hold and defer buffers most significantly. In Section 4.4.1, we theoretically analyze that N_1 would be small and thus the memory allocation of the run buffer is not critical to the performance. Thus, we will experientially predetermine a small size for the run buffer. In so doing, the rest of our work is simplified to finding a good memory distribution between the main buffer (with a size M_s), and the hold and defer buffers (with a total size M_f). We use M' to denote the amount of memory available for M_s and M_f, i.e. $M_s + M_f = M'$.

From the analysis in Section 4.4.1, we can see that the values of N_1 and N_2 depend

on M_s. Basically, a higher M_s can produce smaller N_1 and N_2. On the other hand, M_f affects the number of run merging steps for the defer buffer. In general, a smaller M_f may increase the number of run merging steps for the defer buffer.

The cost model derived in Section 4.4.1 estimates N_1 and N_2 based on M_s and then calculates the total I/O cost of `SCALE` as

$$2N(\lceil \log_M \lceil \frac{N}{2M} \rceil \rceil + 2) + 2N_1 + 2N_2(\lceil \log_M \lceil \frac{N_2}{2M_f} \rceil \rceil + 1) \tag{4.1}$$

The objective of memory allocation is to minimize the total cost shown in Eqn. (4.1). To simplify the problem, we assume a linear relationship between M_s and the values of N_1 and N_2. With this assumption, we could estimate N_1 and N_2 with different M_s values as follows. First, we use the cost model to estimate N_1 (or N_2) under the two situations of $M_s = M'$ and $M_s = 0$. The values are denoted as N_{11} (or N_{21}) and N_{12} (or N_{22}) respectively. Then given any $M_s \leq M'$, we can estimate N_1 and N_2 as

$$N_1 = (M' - M_s)(N_{12} - N_{11})/M' + N_{11} \tag{4.2}$$

$$N_2 = (M' - M_s)(N_{22} - N_{21})/M' + N_{21} \tag{4.3}$$

The memory allocation algorithm is shown in Algorithm 10, which is based on the following observation of Eqn. (4.1): the dominant impact that the increase of M_f is probably able to make is reducing the value of the term $f(M_f) = \lceil \log_M \lceil N_2/(2M_f) \rceil \rceil$. The algorithm starts by setting the M_f to a minimum value, say 1. Then it iteratively attempts to set M_f to a higher number such that the value of $f(M_f)$ decreases by 1 in each iteration. If this results in a lower total cost of Eqn. (4.1), then the attempts will be continued. Otherwise the loop will stop. The loop will also stop if $f(M_f)$ already reaches 1. As in practice the value of $f(M_f)$ tend to be very small due to the logarithmic effect, the loop will stop after several iterations.

Algorithm 10: `Memory Allocation for Buffers`

Output: the value of M_f

1: $M_f \leftarrow$ the minimum value
2: $minCost \leftarrow$ infinite
3: **while** *true* **do**
4: $curCost \leftarrow$ cost of Eqn. (4.1)
5: **if** $curCost < minCost$ **then**
6: $minCost \leftarrow curCost$
7: $r \leftarrow M_f$
8: **else if** $curCost \geq minCost$ **then**
9: **break**
10: **if** $f(M_f) = 1$ **then**
11: **break**
12: $p \leftarrow f(M_f)$
13: set M_f to the smallest value such that $f(M_f) = p - 1$
14: **return** r

4.3.3 Integration with Tuple Selection and Projection Pushdown

For a self-join between two instances R_1 and R_2 of relation R, it is possible that each instance additionally involves distinct tuple filtering and projection conditions. In the conventional query processing, tuple selection and projection operations are usually *pushed down* onto the lowest feasible levels within the query execution tree. As a result, the physical join implementation of the self-join has to deal with two input sets of tuples, each of which is (horizontally and/or vertically) a subset of R. Suppose the tuple selection and projection attached to $R_i (i \in \{1, 2\})$ are σ_i and Π_i respectively, and suppose the tuple projection attached to the self-join operation is Π, the algebra expression of the self-join is hereby $\Pi((\Pi_1(\sigma_1(R_1))) \bowtie (\Pi_2(\sigma_2(R_2))))$[3].

According to σ_1 and σ_2, the tuples of R that are relevant to the self-join can be classified into three categories: satisfying σ_1 only, satisfying σ_2 only and satisfying both σ_1 and σ_2, which are denoted with C_{σ_1}, C_{σ_2} and $C_{\sigma_1 \cap \sigma_2}$ respectively. The filtered R_1 consists of C_{σ_1} and $C_{\sigma_1 \cap \sigma_2}$ tuples, while the filtered R_2 consists of C_{σ_2} and $C_{\sigma_1 \cap \sigma_2}$ tuples.

[3]Note that the query optimizer will ensure Π_1 contains A and Π_2 contains B, even if Π does not contain A and/or B.

Clearly, the (un-projected) self-join result can be represented by $(C_{\sigma_1} \bowtie C_{\sigma_2}) \cup (C_{\sigma_1} \bowtie C_{\sigma_1 \cap \sigma_2}) \cup (C_{\sigma_1 \cap \sigma_2} \bowtie C_{\sigma_2}) \cup (C_{\sigma_1 \cap \sigma_2} \bowtie C_{\sigma_1 \cap \sigma_2})$.

In order to evaluate a self-join integrated with tuple selection and projection push-down, a straightforward extension to SCALE works as follows. As the first step, we sort R on A to obtain a $\mathcal{S}_A((\sigma_1 \cup \sigma_2)(R))$, which contains a sorted sequence of all above three categories of tuples. In addition, in $\mathcal{S}_A((\sigma_1 \cup \sigma_2)(R))$, the C_{σ_1}, C_{σ_2} and $C_{\sigma_1 \cap \sigma_2}$ tuples are projected by Π_1, Π_2 and $\Pi_1 \cup \Pi_2$ respectively. We then run the rest of the algorithm as usual, with the mere modifications that C_{σ_1} (resp. C_{σ_2}) tuples are distinguished to act only as the left-hand (resp. right-hand) side of the self-join, and that C_{σ_1} tuples behave as if they had no right-matching tuples when they are considered for tuple eviction in the main buffer. However, there are several potential problems with such a straightforward extension. First of all, intuitively it incurs wasteful I/O and CPU costs to sort C_{σ_2} tuples on A, as C_{σ_2} tuples are not a part of right-matching tuples. Moreover, during the first pass of processing $\mathcal{S}_A((\sigma_1 \cup \sigma_2)(R))$, C_{σ_2} tuples will keep C_{σ_1} and $C_{\sigma_1 \cap \sigma_2}$ tuples farther away from their right-matching tuples. As such, more C_{σ_1} and $C_{\sigma_1 \cap \sigma_2}$ tuples would be forced to enter the hold buffer and the defer buffer and incur additional I/O overhead.

There is another more efficient approach. The rough idea is to split those three categories of tuples in R into two parts: one part R^+ contains all C_{σ_1} and $C_{\sigma_1 \cap \sigma_2}$ tuples, and the other part R^- contains the rest C_{σ_2} tuples. Similarly, the C_{σ_1} tuples in R^+ are projected by Π_1, the $C_{\sigma_1 \cap \sigma_2}$ tuples in R^+ are projected by $\Pi_1 \cup \Pi_2$ and the C_{σ_2} tuples in R^- are projected by Π_2. Given the self-join condition $R_1.A = R_2.B$, we need to sort R^+ on A into $\mathcal{S}_A(R^+)$ and sort R^- on B into $\mathcal{S}_B(R^-)$. We then sequentially read both $\mathcal{S}_A(R^+)$ and $\mathcal{S}_B(R^-)$ into memory to merge join them, which generates the projected result tuples of $(C_{\sigma_1} \bowtie C_{\sigma_2}) \cup (C_{\sigma_1 \cap \sigma_2} \bowtie C_{\sigma_2})$. In the meantime, we also apply the self-join techniques of SCALE to $\mathcal{S}_A(R^+)$ to produce the projected result tuples of $(C_{\sigma_1} \bowtie C_{\sigma_1 \cap \sigma_2}) \cup (C_{\sigma_1 \cap \sigma_2} \bowtie C_{\sigma_1 \cap \sigma_2})$.

4.4 Analytical Study

In this section, we first analyze the cost of the SCALE algorithm and then analytically prove that the performance of SCALE is at least as good as Sort-Merge join.

4.4.1 Cost Model

Our SCALE algorithm on join condition $R.A = R.B$ is symmetric: it could choose to sort on either A or B during the first-step external sorting. However, this choice may affect the final total join cost and thus should be be decided in a cost-based way. Moreover, the query optimizer also needs a cost model to estimate the self-join subplan costs when doing join enumeration and pruning.

Notation	Definition
N	the size of R in terms of pages
M	the total number of buffer pages available for join
M_s	the total number of pages occupied by main buffer
$\mathbb{M}$	the number of R tuples that can be held by main buffer
M_f	the number of pages occupied by hold and defer buffers
N_1	the total size of tuples transferred to hold buffer
N_2	the total size of tuples transferred to defer buffer
$t(x, y)$	a tuple such that $t.A = x$ and $t.B = y$
$\mathbb{N}(x, y)$	the number of tuples in R that have values x and y on the attributes A and B respectively
$\mathbb{N}_A(x)$	the number of tuples in R that have value x on A
$\mathbb{N}_B(y)$	the number of tuples in R that have value y on B
$\mathbb{P}_t$	the position of tuple t in $\mathcal{S}_A(R)$
$\mathbb{P}^{1st}_{RM(t)}$	the position of the first tuple of *RM(t)* in $\mathcal{S}_A(R)$
$\mathbb{P}^{last}_{RM(t)}$	the position of the last tuple of *RM(t)* in $\mathcal{S}_A(R)$

Table 4.2: Notations used in the analytical study of SCALE

Without loss of generality, in the following discussions, we assume that SCALE sorts R on attribute A during the first-step external sorting. Table 4.2 summarizes the notations used throughout the analytical study of SCALE. Generally, the I/O cost of SCALE consists

of the following components:

(a) The cost of externally sorting R into $\mathcal{S}_A(R)$.

(b) The cost of sequentially scanning $\mathcal{S}_A(R)$ during the first pass of processing $\mathcal{S}_A(R)$.

(c) The cost of inserting tuples into the hold buffer and the defer buffer (i.e. generating sorted runs) during the first pass of processing $\mathcal{S}_A(R)$.

(d) The cost of reading and merging sorted runs of hold buffer tuples during the first pass of processing $\mathcal{S}_A(R)$.

(e) The cost of the merge join of defer buffer tuples and $\mathcal{S}_A(R)$ tuples during the second pass of processing $\mathcal{S}_A(R)$.

In the ideal situation, the I/O cost of `SCALE` consists of only (a)–(b). Suppose the size of R, in terms of the number of pages, is N and the total available join memory is M pages from R. Then cost (a)–(b) can be calculated as $2N(\lceil \log_M \lceil N/2M \rceil \rceil + 1) + N$.

To calculate (c)–(e), we first assume that we can estimate the total sizes of tuples in the hold buffer and the defer buffer, denoted as N_1 and N_2 respectively. We will show later how to estimate them.

Cost incurred by the hold buffer. The tuples spilled into the hold buffer are stored in a number of disk-based run files sorted on attribute B, which have to be merged on-the-fly using the run buffer during the first pass. Theoretically, we can calculate the number of merging steps based on the size of the run buffer and the number of sorted runs in the hold buffer. However this will result in an overestimation of the actual merging steps due to the following reasons:

- During the first pass of processing $\mathcal{S}_A(R)$, as tuples are read into the main buffer in the order of A, roughly the B values of the case 4 tuples being spilled to the hold buffer should be in a nearly sorted order. This is because for such a case 4

tuple with $B = b$, it then must have some matching tuples with $A = b$ that have not been read into the main buffer. Consequently, a few (but large) run files will be generated.

- The runs in the hold buffer are merged progressively while the tuples are being spilled. In other words, at any moment during the process, we are only merging up to the tuples that have been spilled so far.

- As one will see in the later analysis, compared with the FIFO tuple eviction policy for the main buffer, the number of tuples spilled to the hold buffer is significantly reduced by our prioritized tuple eviction policy described in Section 4.3.2.

Therefore, in our cost model, we assume that the tuples in the hold buffer are written to disk and read into memory only once. Our experiment results validate this assumption. Furthermore, this assumption simplifies the cost estimation and saves the optimization cost.

Cost incurred by the defer buffer. The tuples in the defer buffer are also stored as disk-based run files sorted on B and need to be merged during the second pass. The size of each run depends on the size of the total memory, denoted as M_f, that is dynamically shared by the hold buffer and the defer buffer. As mentioned above, the tuples are spilled to the hold buffer in a nearly sorted order and thus require only a small hold buffer. Therefore, we can assume that almost the whole M_f is allocated to the defer buffer. Given the size of M_f, the expected number of runs will be $\lceil N_2/2M_f \rceil$. Furthermore, we can use nearly all the available join memory to merge defer buffer tuples during the second pass. Then the number of steps of run merging defer buffer tuples is $\lceil \log_M \lceil N_2/2M_f \rceil \rceil$. Finally, we also need to count the cost of writing the sorted runs in the defer buffer to the disk before the merging.

In summary, the cost components (c)–(e) can be calculated as follows:

$$2N_1 + N_2(2\lceil \log_M \lceil \frac{N_2}{2M_f} \rceil \rceil + 1) + N_2 + N \tag{4.4}$$

and hence the total cost of our algorithm is

$$2N(\lceil \log_M \lceil \frac{N}{2M} \rceil \rceil + 2) + 2N_1 + 2N_2(\lceil \log_M \lceil \frac{N_2}{2M_f} \rceil \rceil + 1) \tag{4.5}$$

Cost with FIFO Tuple Eviction Policy

Below we discuss how to estimate the values of N_1 and N_2, i.e. the sizes of tuples spilled to the hold buffer and the defer buffer respectively. Due to the dynamic behavior of our algorithm, the exact estimation is quite complicated and costly to perform. Hence, we perform a simplified analysis by assuming that the main buffer applies the FIFO tuple eviction policy. Moreover, we assume a tuple $t(x, y)$ is randomly located within the segment of tuples with $A = x$ in $\mathcal{S}_A(R)$. As such, there are three scenarios under which tuples have to be spilled into the hold buffer and the defer buffer.

(I) If a tuple $t(x, x)$ belongs to case 2, 3 or 4, then it has to be spilled to either the hold buffer or the defer buffer. As t in this case is located inside its own *RM(t)* in $\mathcal{S}_A(R)$, we have $\mathbb{P}^{1st}_{RM(t)} \leq \mathbb{P}_t \leq \mathbb{P}^{last}_{RM(t)}$. Hence, the probability that $t(x, x)$ falls into case 2 or 3 and thus is spilled to the defer buffer can be calculated as follows:

$$P(t(x,y) \text{ is spilled to the defer buffer} \mid x = y) = \begin{cases} 0 & \mathbb{N}_A(x) \leq \mathbb{M} \\ 1 - \frac{\mathbb{M}}{\mathbb{N}_A(x)} & \text{otherwise.} \end{cases} \tag{4.6}$$

Furthermore, the probability that $t(x, x)$ belongs to case 4 and thus is spilled to the hold

buffer is:

$$P(t(x,y) \text{ is spilled to the hold buffer} \mid x = y) = \begin{cases} 0 & \mathbb{N}_A(x) \leq \mathbb{M} \\ \frac{\mathbb{M}}{\mathbb{N}_A(x)} & \mathbb{N}_A(x) \geq 2\mathbb{M} \\ 1 - \frac{\mathbb{M}}{\mathbb{N}_A(x)} & \text{otherwise.} \end{cases} \tag{4.7}$$

(II) With $x > y$, a tuple $t(x, y)$ cannot be in cases 3 and 4. Thus, $t(x, y)$ will not be spilled to the hold buffer, and $P(t(x, y)$ is spilled to the hold buffer $\mid x > y) = 0$.
If $t(x, y)$ with $x > y$ is in case 2, then it has to be spilled to the defer buffer. Note that, in this case, $\mathbb{P}_t \leq \mathbb{P}^{1st}_{RM(t)}$ due to the fact that $x > y$. Therefore, the probability that $t(x, y)$ falls into this case is:

$$P(t(x,y) \text{ is spilled to the defer buffer} \mid x > y) = \begin{cases} 0 & \mathbb{N}_A(x) \leq \mathbb{M} - \sum_{i=y}^{x-1} \mathbb{N}_A(i) \\ 1 & \sum_{i=y}^{x-1} \mathbb{N}_A(i) \geq \mathbb{M} \\ 1 - \frac{\mathbb{M} - \sum_{i=y}^{x-1} \mathbb{N}_A(i)}{\mathbb{N}_A(x)} & \text{otherwise.} \end{cases} \tag{4.8}$$

(III) With $x < y$, a tuple $t(x, y)$ cannot be in cases 2 and 3. Thus, $t(x, y)$ will not be spilled to the defer buffer, and $P(t(x, y)$ is spilled to the defer buffer $\mid x < y) = 0$.
If $t(x, y)$ with $x < y$ is in case 4, then it has to be spilled to the hold buffer. Here, we have $\mathbb{P}_t \geq \mathbb{P}^{last}_{RM(t)}$. Similar to the previous case, the probability of $t(x, y)$ being in this

case can be derived as follows:

$$P(t(x,y) \text{ is spilled to the hold buffer} \mid x < y) = \begin{cases} 0 & \mathbb{N}_A(x) \leq \mathbb{M} - \sum_{i=x+1}^{y} \mathbb{N}_A(i) \\ 1 & \sum_{i=x+1}^{y} \mathbb{N}_A(i) \geq \mathbb{M} \\ 1 - \frac{\mathbb{M} - \sum_{i=x+1}^{y} \mathbb{N}_A(i)}{\mathbb{N}_A(x)} & \text{otherwise.} \end{cases} \tag{4.9}$$

Now we can derive the values of N_1 and N_2 as follows:

$$N_1 = \frac{\sum_{x,y} \mathbb{N}(x,y) P(t(x,y) \text{ is spilled to the hold buffer})}{\rho} \tag{4.10}$$

$$N_2 = \frac{\sum_{x,y} \mathbb{N}(x,y) P(t(x,y) \text{ is spilled to the defer buffer})}{\rho} \tag{4.11}$$

where ρ is the number of R tuples that can be stored in each page.

Cost with Prioritized Tuple Eviction Policy

The above analysis on the values of N_1 and N_2 does not consider tuple eviction priorities defined in Section 4.3.2, and hence may not reflect the real cost of our algorithm correctly. Below we try to measure some effects of our prioritized tuple eviction policy.

(I) A tuple $t(x,x)$ in case 4 now is more likely to be kept in the main buffer until all its right-matching tuples have been scanned. Therefore, $t(x,x)$ can meet with all of its matching tuples in the main buffer and can be directly discarded afterwards. As such,

the probability of $t(x,x)$ being spilled into the hold buffer becomes:

$$P(t(x,y) \text{ is spilled to the hold buffer} \mid x = y) = \begin{cases} 0 & \mathbb{N}_A(x) \leq \mathbb{M} \\ \frac{1}{\mathbb{N}_A(x)} & \text{otherwise.} \end{cases} \tag{4.12}$$

By comparing with Eqn. (4.7), one can see that the adoption of tuple eviction priorities significantly reduces the probability of spilling $t(x,x)$ to the hold buffer. The number of tuples spilled to the defer buffer in this case is unchanged.

(II) Similarly, a tuple $t(x,y)$ with $x < y$ that belongs to case 4 is more likely to be kept in the main buffer until its right-matching tuples are fully scanned. Consider the set S of tuples in $\mathcal{S}_A(R)$ whose attribute A values fall in the range $(x,y]$. Within S, those tuples in cases 1, 2 and 3 all have higher eviction priorities than $t(x,y)$. By assuming that $t(x,y)$ has a higher eviction priority than any case 4 tuple in S and that the total number of case 4 tuples in S is maximum, we can derive an upper bound of the probability that $t(x,y)$ is spilled to the hold buffer, which can serve as an approximation of the actual probability:

$$P(t(x,y) \text{ is spilled to the hold buffer} \mid x < y) \leq \begin{cases} 0 & \mathbb{N}_A(x) \leq \mathbb{M} - k(x,y) \\ 1 & k(x,y) \geq \mathbb{M} \\ 1 - \frac{\mathbb{M}-k(x,y)}{\mathbb{N}_A(x)} & \text{otherwise.} \end{cases} \tag{4.13}$$

where

$$k(x,y) = \sum_{\substack{x<i\leq y \\ i\leq j}} \mathbb{N}(i,j) \tag{4.14}$$

Again, by comparing with Eqn. (4.9), we can see that the prioritized tuple eviction policy can significantly reduces the probability of spilling $t(x,y)$ to the hold buffer.

Practical Considerations

In DBMS systems, we could utilize a two-dimensional histogram to summarize the joint distribution function $\mathbb{N}(x, y)$ and hence we can use the above cost model to estimate the cost of the join algorithm. Note that the sum aggregates in Eqns. (4.8) and (4.9) can be efficiently calculated with the widely adopted one dimensional equi-depth histogram and cumulative histogram. However, Eqn. (4.14) would be expensive to calculate. Therefore, we will adopt Eqn. (4.9) in our cost model for algorithm implementation, which provides an upper bound of the algorithm's cost.

When the two-dimensional histograms are unavailable, we will use the one-dimensional statistics to estimate $\mathbb{N}(x, y)$ as follows. Suppose we only have the one functions $\mathbb{N}_A(x)$ and $\mathbb{N}_B(y)$. By assuming that the attributes A and B are statistically independent of each other, we can derive the function $\mathbb{N}(x, y)$ as follows ($|R|$ is the total number of tuples in R): $\mathbb{N}(x, y) = \frac{\mathbb{N}_A(x) \cdot \mathbb{N}_B(y)}{|R|}$.

4.4.2 Comparison with Sort-Merge Join

Now we try to compare the cost of our `SCALE` algorithm with that of Sort-Merge Join (`SMJ`). The I/O cost of `SMJ` consists of: (a) the cost of externally sorting R into $\mathcal{S}_A(R)$, (b) the cost of externally sorting R into $\mathcal{S}_B(R)$ and (c) the cost of merge join of $\mathcal{S}_A(R)$ and $\mathcal{S}_B(R)$. Hence the total cost can be estimated as follows:

$$4N\lceil \log_M \lceil \frac{N}{2M} \rceil \rceil + 6N \tag{4.15}$$

In the worst case of `SCALE`, all the tuples will be spilled to the defer buffer during the first pass of processing $\mathcal{S}_A(R)$. That is, in this case, $N_2 = N$ and $N_1 = 0$. By

substituting them into Eqn. (4.5), we have

$$2N\lceil\log_M\lceil\frac{N}{2M}\rceil\rceil + 2N\lceil\log_M\lceil\frac{N}{2M_f}\rceil\rceil + 6N$$

By comparing this with Eqn. (4.15), one can see the cost of SCALE would be the same as that of SMJ if $\lceil\log_M\lceil N/2M_f\rceil\rceil = \lceil\log_M\lceil N/2M\rceil\rceil$.

As described in Section 4.3.2, generally speaking, the larger the portion of tuples that are spilled to the defer buffer, the more memory is allocated to the hold and defer buffers. In the adverse case, M_f will be set to a value close to M such that $\lceil\log_M\lceil N/2M_f\rceil\rceil = \lceil\log_M\lceil N/2M\rceil\rceil$. In other words, SCALE degenerates to SMJ in the adverse case.

4.5 Performance Study

We have integrated our proposed SCALE algorithm into PostgreSQL 8.4.4 [2] as a standard join operation. We enabled the query optimizer to additionally include SCALE in its plan search space, based on the cost model provided in Section 4.4. We used the default system settings without any tuning. We empirically compared SCALE with the native join operations of PostgreSQL: Sort-Merge Join (SMJ), Hybrid Hash Join (HHJ) and Nested-Loop Join (NLJ). However, the performance of NLJ was always significantly worse than the other three join operations in all experiments. Thus, we will not report the experimental results of NLJ here.

We conducted all experiments on a Dell workstation which is equipped with a Quad-Core Intel Xeon 2.66Hz CPU, 4GB DRAM and two SATA disks with storage capacities of 500GB and 750GB. Both the operating system, Ubuntu 7.10 with Linux 2.6.22 kernel, and the PostgreSQL system run on the 500GB disk, while the databases as well as intermediate results of PostgreSQL are stored on the 750GB disk.

4.5.1 Synthetic Dataset Generation

We generated numerous synthetic tables with different properties in order to comprehensively and extensively evaluate the performance of SCALE. In general, every synthetic table consists of two join attributes, A and B, along with another 23 padding attributes[4]. All attributes are of the (4-byte) integer data type and thus each tuple has a fixed size of 100 bytes. The attribute A, on which a synthetic table R will be externally sorted by SCALE to generate $\mathcal{S}_A(R)$, has the value domain $[1, 10^6]$.

As noted, the performance of SCALE is dependent on the overall distance (*nearness*) between tuples and their corresponding right-matching tuples in $\mathcal{S}_A(R)$. A shorter average distance means to us a stronger correlation between A and B[5] and hence better performance of SCALE. We expect to test SCALE on synthetic tables with tunable correlation extents. To this end, we have four essential configurable parameters when generating a table R:

- AD: the statistical distribution of A values, uniform or Zipf.
- MD: the maximum absolute difference between the A value and the B value of a single tuple t, i.e., $|t.A - t.B| \leq \mathsf{MD}$. MD is used to model the correlation between R.A and R.B. A small MD value means that R.A and R.B are more correlated. Hence, more tuples will be able to complete their right-joins before they are evicted from the main buffer. This is the situation where SCALE is expected to perform well. On the other hand, a large MD means that it is less likely for tuples to find matching tuples in the main buffer. Such cases are not favorable to SCALE.
- DD: the statistical distribution of $(t.A - t.B + \mathsf{MD})$ values, either uniform or Zipf.

[4]Note that with fewer padding attributes SCALE could perform even better, as the same amount of join memory now can hold more tuples and thus the sizes of the hold and defer buffers may decrease.

[5]Note here the meaning of correlation is a bit different from its traditional definition, which measures the relationship between the A and B values within the same tuple.

- DV: the number of distinct values on A, which are uniformly distributed over $[1, 10^6]$.

The Zipf distribution has a parameter θ, which affects the skewness of the data distribution: the greater the value of θ, the greater the skewness. We also varied the θ values. In the following presentation, we shall use "Zipf x" to represent "Zipf with $\theta = x$".

4.5.2 Experiment Design

On each synthetic table R, we executed self-join queries of the following basic form:

```
SELECT *
FROM R AS R1, R AS R2
WHERE R1.A = R2.B
```

and compared the total query execution times of SCALE, SMJ and HHJ. In certain queries, we also applied extra tuple selection conditions with different selectivities on both R_1 and R_2. No clustered indices on A or B were available and thus both SCALE and SMJ were forced to explicitly sort R.

The experiments conducted consist of four parts. The first part is a micro-benchmark test, which enumerated different combinations of the above four parameters (AD, MD, DD and DV), as well as the total available join memory MEM on a set of synthetic tables with fixed sizes. The second part tested the scalability of SCALE by varying the table sizes. The third part measured and verified the effectiveness of our memory allocation scheme presented in Section 4.3.2. The final part focused on the performance of SCALE when combined with tuple selection and projection, as described in Section 4.3.3.

Note that in all experiments, during the first pass of SCALE, the size of the run buffer was set to MEM/10. Except for those experiments that studied the memory allocation scheme, for all other experiments, we completely relied on our memory allocation scheme to divide the remainder of MEM between the main buffer, and the hold and de-

fer buffers. Between queries, we clear the operating system cache by using the Linux command *"echo 3 > /proc/sys/vm/drop_caches"*.

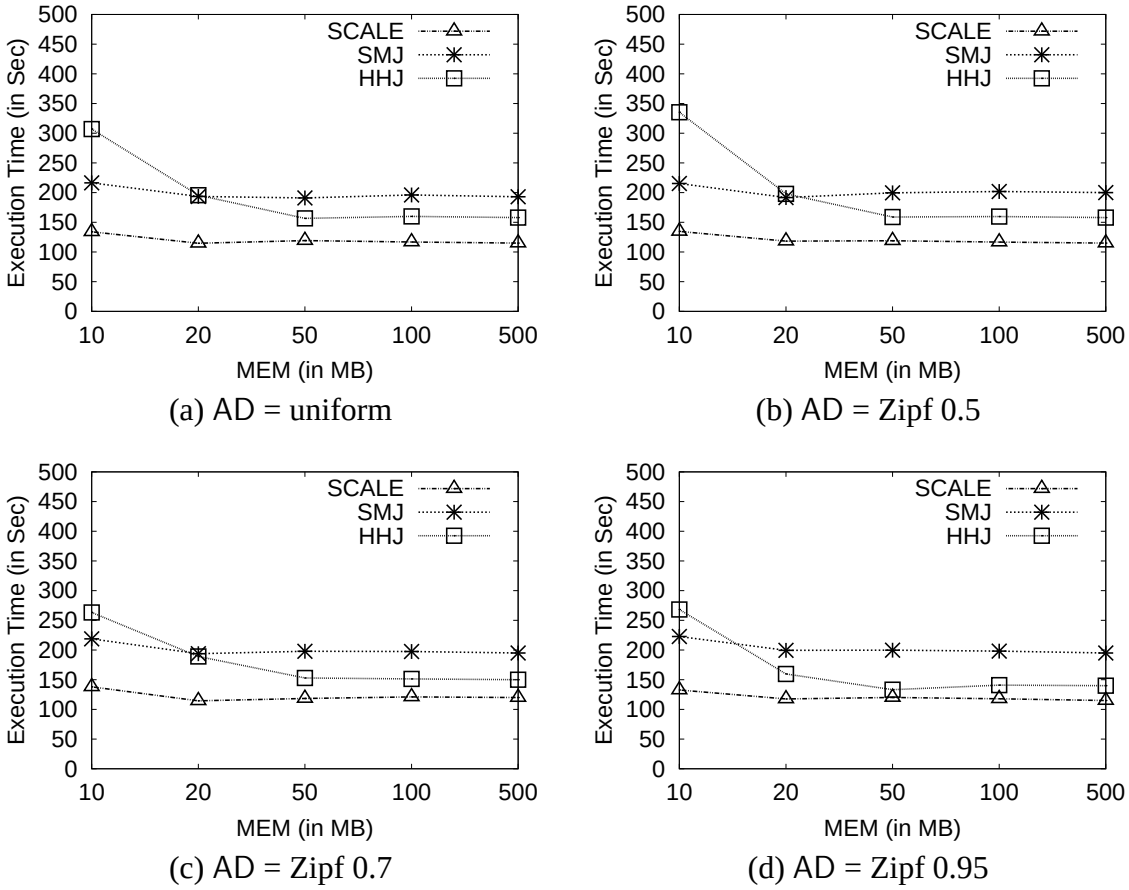
We tested SCALE under a wide range of extents of correlation between A and B. Throughout our experiments (Fig. 4.3 to 4.10 below), we utilized three DV values, i.e. 10^5 (the most common), 5×10^5 and 9×10^5, and two MD values, i.e. 10^5 (the most common) and 5×10^5. In Fig. 4.3, 4.5, 4.8 and 4.10, $\mathsf{DV} = \mathsf{MD} = 10^5$ so that A and B were not obviously correlated; in Fig. 4.4, $\mathsf{MD} = 5 \times 10^5$ and $\mathsf{DV} = 10^5$ so that A and B were much uncorrelated; in Fig. 4.6, 4.7, 4.8, 4.9 and 4.10, A and B were (a bit or very) correlated.

4.5.3 Experimental Results

Micro-Benchmark Test

All synthetic tables in this test have 10 million tuples, with a total size of 1GB. The experimental results are depicted by Fig. 4.3 – 4.7, from which we can clearly see that SCALE significantly outperformed both SMJ and HHJ in all situations. The performance gain of SCALE over the winner between SMJ and HHJ was between 20% to 45%. In all figures, we observe that the execution times of SCALE were quite stable for a wide range of join memory MEM. The execution times of SMJ and HHJ also stabilized when the join memory MEM increased to 100MB. We will not show the statistical details about tuple distribution over the six cases as well as the sizes of hold buffer tuples and defer buffer tuples in SCALE. Generally speaking, most tuples fell into cases 1, 2 and 4 as expected, among which case 1 tuples occupied a very significant portion. As a result, compared to the size of R, the total size of tuples in hold and defer buffers was usually small. The above observations explain the superiority of SCALE. We then briefly analyze the behaviour of SCALE according to the figures.

In both Fig. 4.3 and Fig. 4.4, when MEM was fixed, the execution times of SCALE

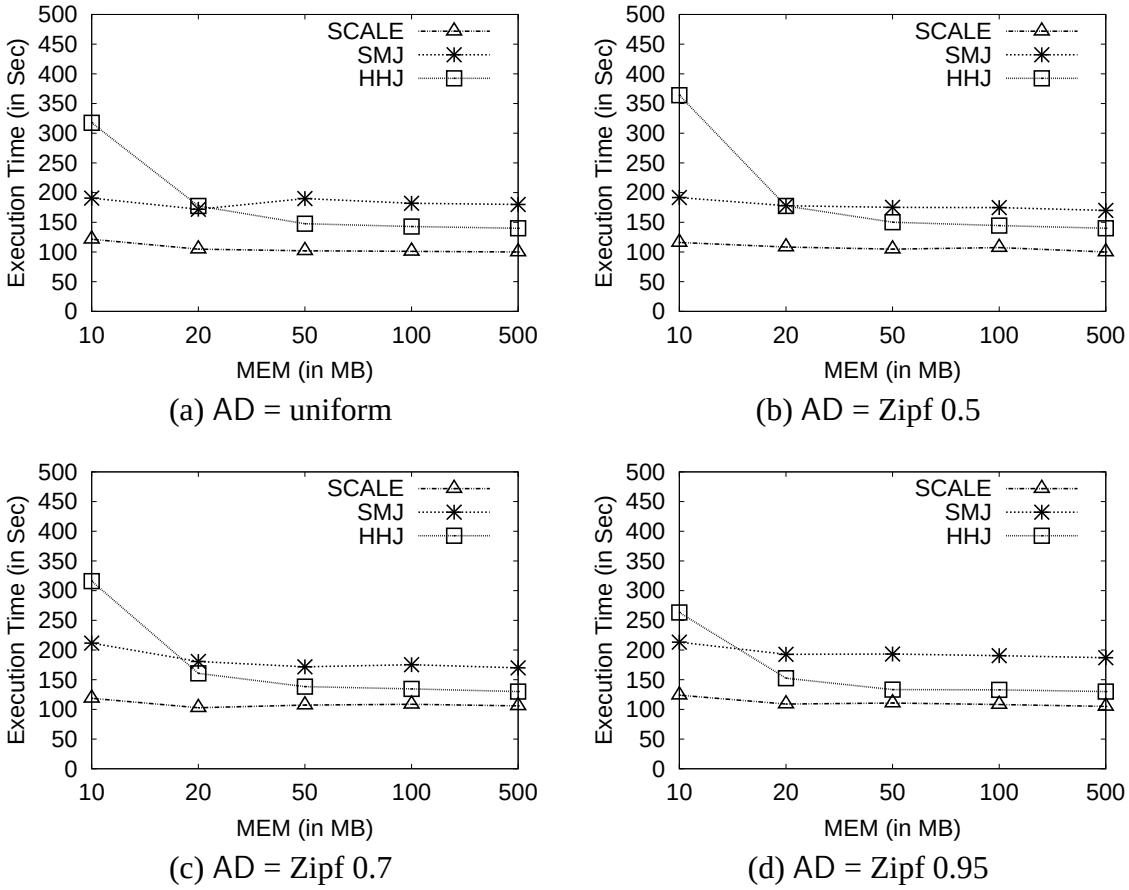


(a) AD = uniform (b) AD = Zipf 0.5

(c) AD = Zipf 0.7 (d) AD = Zipf 0.95

Figure 4.3: Benchmark test, 1GB tables with 10 million tuples, AD varies, MD = 10^5, DD = uniform, DV = 1×10^5

remained nearly unchanged as AD varied. The underlying reason lies in that different AD settings resulted in more or less the same numbers of join result tuples, as well as the similar tuple distributions over the six cases. On the other hand, with a specific AD, when MEM increased, the number of tuples in case 2 (so is the size of defer buffer tuples) decreased slowly, the number of tuples in case 4 also decreased but the the hold buffer was always empty. Therefore, the difference between the execution times of SCALE highly depended on how effectively the tuple runs in the defer buffer were merged. The fact is that multiple merge passes were required only when MEM $= 10$MB, which led to an execution time notably higher than those with larger MEM values. Comparing Fig. 4.3 with Fig. 4.4, the only parameter setting difference was the value of MD. With



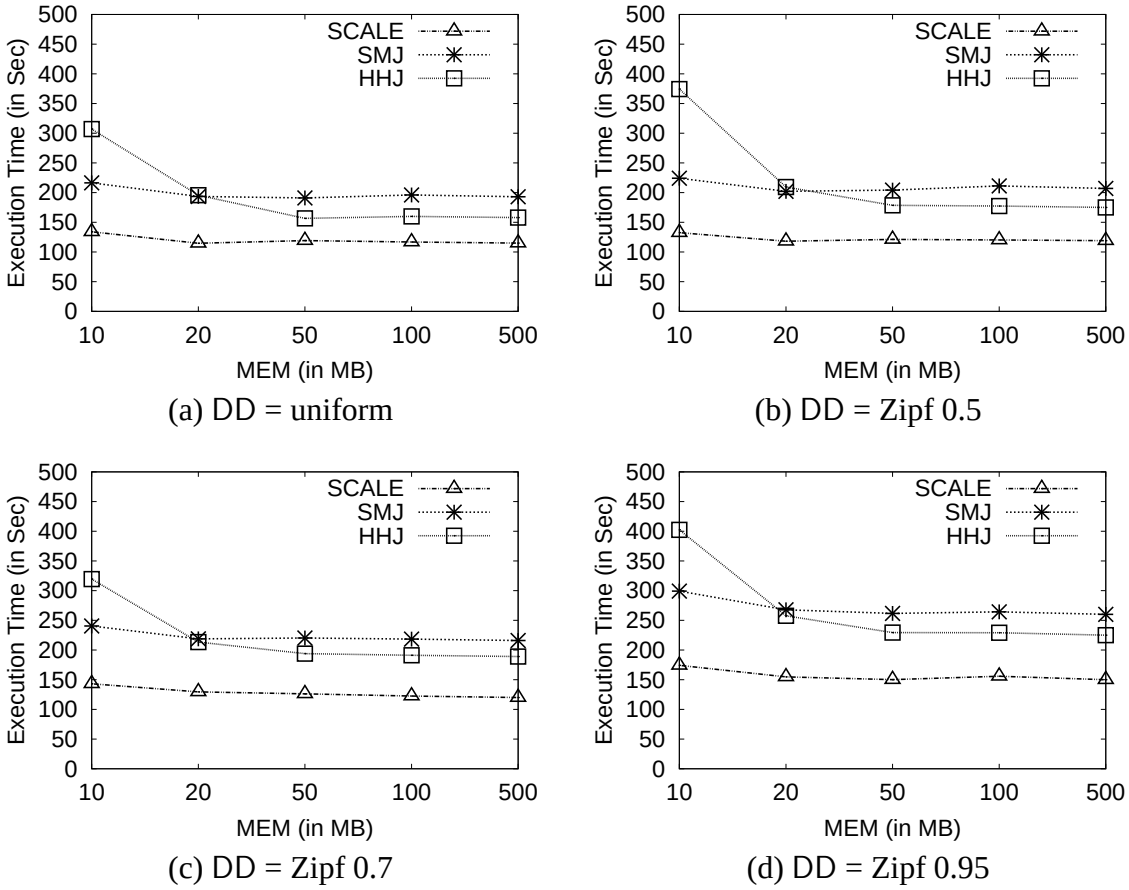
(a) AD = uniform (b) AD = Zipf 0.5

(c) AD = Zipf 0.7 (d) AD = Zipf 0.95

Figure 4.4: Benchmark test, 1GB tables with 10 million tuples, AD varies, MD = 5×10^5, DD = uniform, DV = 1×10^5

a greater MD, although the sizes of hold buffer tuples and defer buffer tuples increased, the number of join result tuples was reduced dramatically and thus much less CPU cost was incurred. Consequently, the execution times of SCALE in Fig. 4.4 were lower than their counterparts in Fig. 4.3.

In Fig. 4.5, the DD setting was varied. With the same MEM value, from Fig. 4.5(a) to Fig. 4.5, the sizes of hold buffer tuples and defer buffer tuples dropped gradually but the number of join result tuples rose quickly, and therefore the execution times increased correspondingly. Note that Fig. 4.5(a) is actually the same as Fig. 4.3(a). Within each of Fig. 4.5(b) – 4.5(d) having the Zipf DD, when the MEM increased, the size of defer buffer tuples decreased slightly while the hold buffer was always empty. Besides, multiple



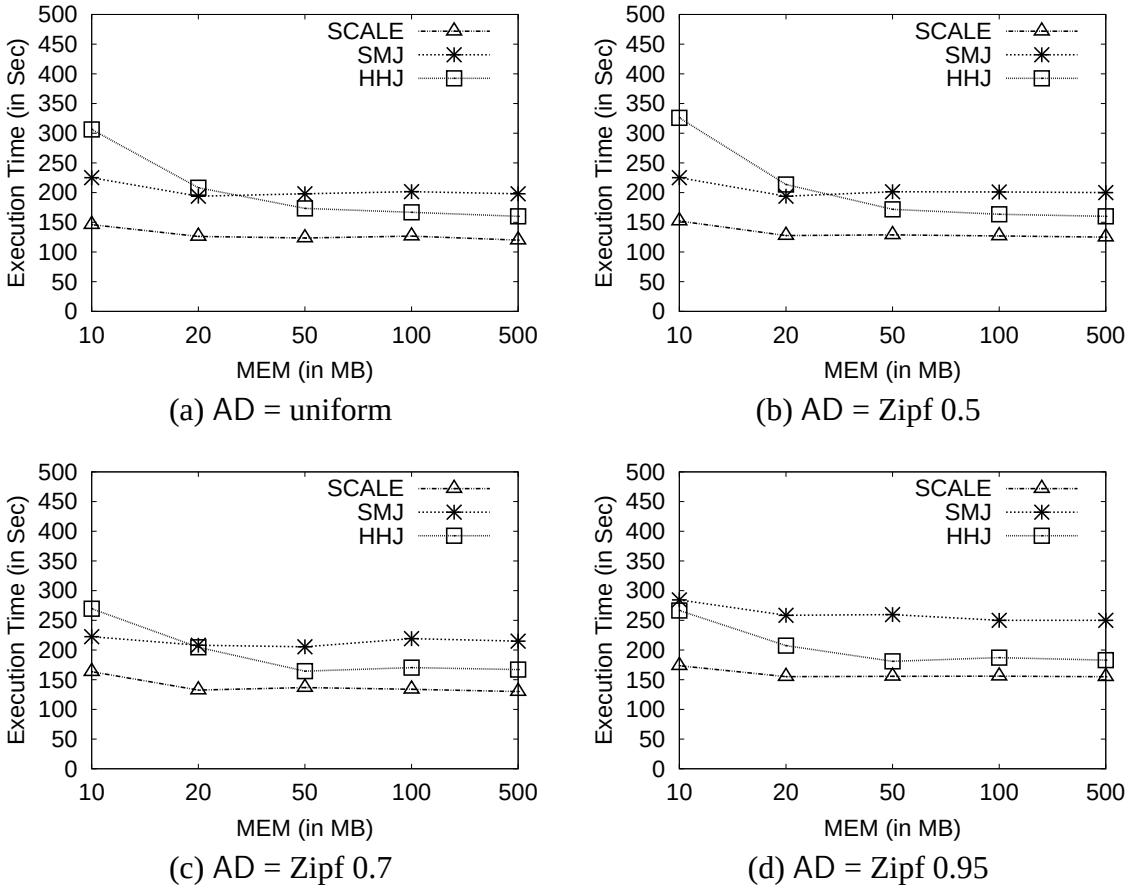
(a) DD = uniform (b) DD = Zipf 0.5

(c) DD = Zipf 0.7 (d) DD = Zipf 0.95

Figure 4.5: Benchmark test, 1GB tables with 10 million tuples, AD = uniform, MD = 10^5, DD varies, DV = 1×10^5

merge passes for tuples in the defer buffer were required only when MEM $= 10$MB. Therefore, in all four subfigures of Fig. 4.5, the execution times of SCALE with 10MB MEM were much higher than those with larger MEM values.

In both Fig. 4.6 and 4.7, when MEM was fixed and AD changed from uniform to Zipf (θ increasing from 0.5 to 0.95), both hold buffer tuples and defer buffer tuples shrank in sizes. However, in the meantime, the number of join result tuples increased, which incurred much more CPU time as well as a higher total execution time. On the other hand, with a specific AD, when MEM increased, the number of tuples in case 2 (so is the size of tuples in the defer buffer) decreased slowly, but the relatively small number of tuples in case 4 (so is the size of tuples in the defer buffer) decreased fast.

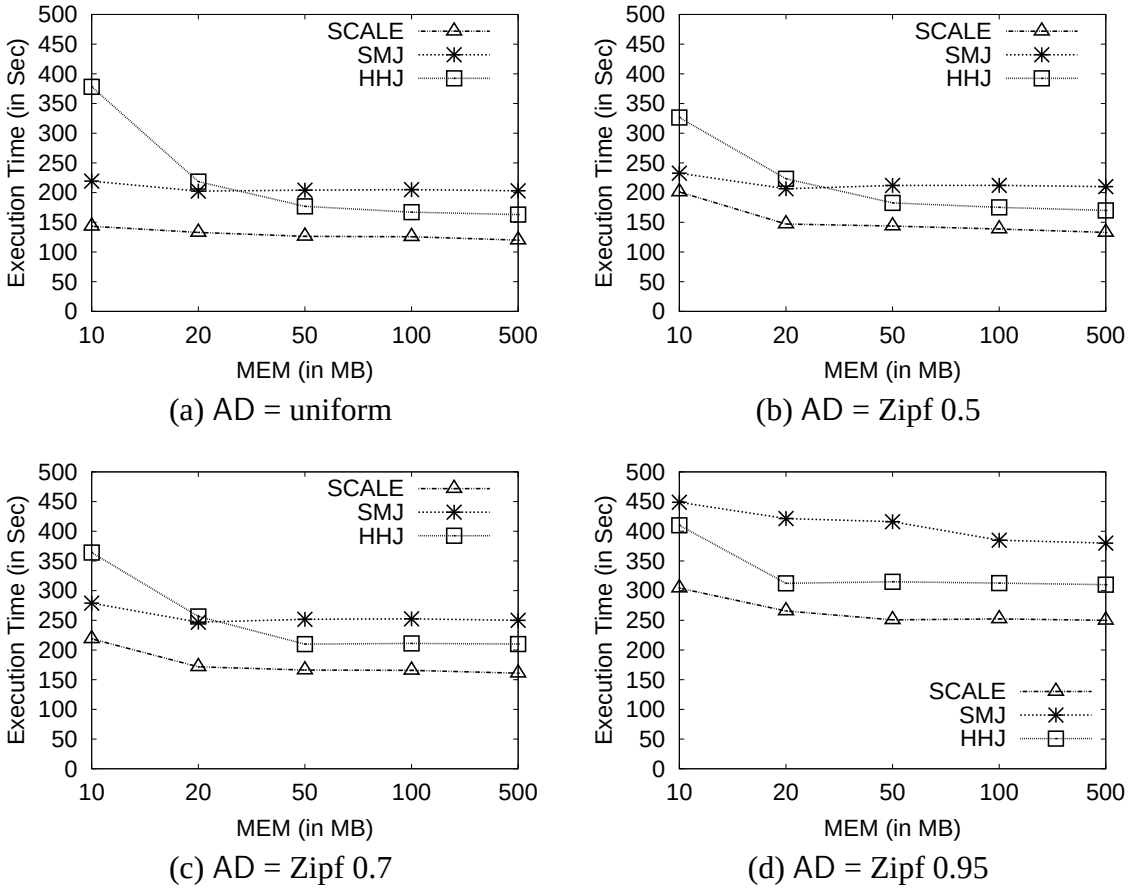


(a) AD = uniform (b) AD = Zipf 0.5

(c) AD = Zipf 0.7 (d) AD = Zipf 0.95

Figure 4.6: Benchmark test, 1GB tables with 10 million tuples, AD varies, MD = 10^5, DD = uniform, DV = 5×10^5

As a whole, similar to the scenarios in Fig. 4.3 and 4.4, the execution time differences of SCALE were determined by the number of merge passes when merging tuples in the defer buffer. Still, for the 10MB MEM, SCALE generated multiple merge passes and thus resulted in a higher execution time than others with larger MEM values. Among Fig. 4.3, Fig. 4.6 and Fig. 4.7, their parameter settings differed only on the value of DV. With a smaller DV, the sizes of hold buffer tuples and defer buffer tuples and the number of join result tuples all rose a bit. Consequently, the corresponding execution times of SCALE in Fig. 4.3, Fig. 4.6 and Fig. 4.7 were in an ascending order.



(a) AD = uniform (b) AD = Zipf 0.5

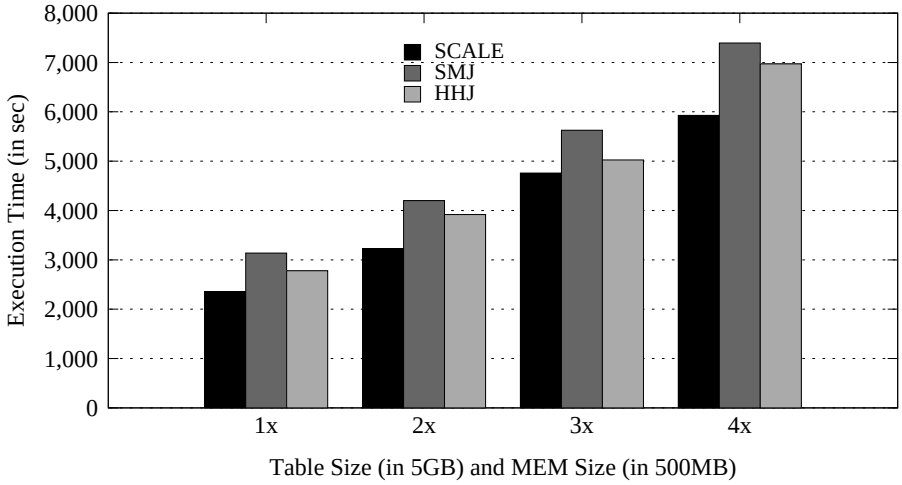
(c) AD = Zipf 0.7 (d) AD = Zipf 0.95

Figure 4.7: Benchmark test, 1GB tables with 10 million tuples, AD varies, MD = 10^5, DD = uniform, DV = 9×10^5

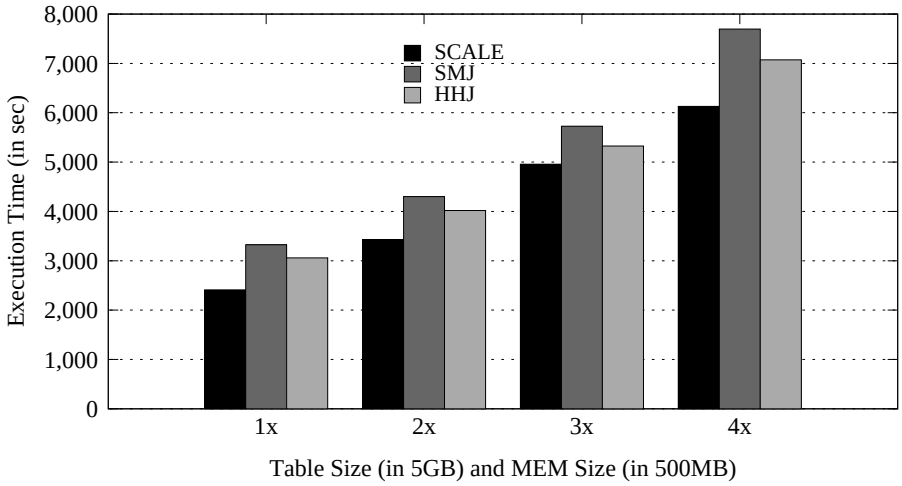
Scalability Test

In this test, we investigated how the relative performance of SCALE compared to SMJ and HHJ will change with respect to the synthetic table sizes. We fixed AD (uniform), MD (10^5) and DD (uniform), and then generated two groups of tables, each according to a different DV value (either 9×10^5 or 10^5). Each group contains four tables of 50 million (5GB), 100 million (10GB), 150 million (15GB) and 200 million (20GB) tuples, on which self-joins were conducted with join memory MEMs of 500MB, 1000MB, 1500MB and 2000MB respectively. The experimental results are plotted in Fig. 4.8.

As shown, SCALE kept gaining significant performance improvement over both SMJ



(a) DV = 1×10^5



(b) DV = 9×10^5

Figure 4.8: Scalability test, with varying table sizes and join memory sizes, AD = uniform, MD = 10^5, DD = uniform

and HHJ as the table sizes increased. Moreover, all execution times of SCALE with DV $= 9 \times 10^5$ in Fig. 4.8 were higher than their counterparts with DV $= 1 \times 10^5$ in Fig. 4.8, which is consistent with our observations from Fig. 4.3, Fig. 4.6 and Fig. 4.7. As such, it would be convincing to claim that similar benchmark tests with different table sizes will bring the same conclusions on SCALE as those presented in the above micro-benchmark test.

Verification of Memory Allocation Scheme

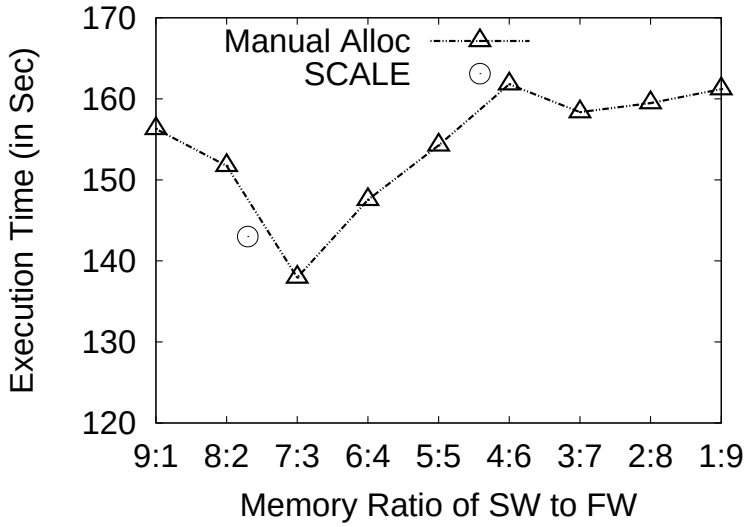


Figure 4.9: Verify the effect of memory allocation scheme, 1GB table with 10 million tuples, MEM = 10MB, AD = uniform, MD = 10^5, DD = uniform, DV $= 9 \times 10^5$

In order to verify the effectiveness of our memory allocation scheme proposed in Section 4.3.2, we conducted an experiment with the synthetic table in Fig. 4.7. We fixed MEM to 10MB and then ran `SCALE` with nine different memory ratios of the main buffer (denoted by SW) to the hold and defer buffers (denoted by FW). The experimental results are shown in Fig. 4.9.

It is obvious that the curve in Fig. 4.9 contains a trough whose lowest point corresponds to the ratio of 7:3 with the minimum execution time of 138 seconds. The small circle in Fig. 4.9 represents the chosen ratio, 77:23, by our automatic memory allocation scheme, with the actual execution time of 143 seconds. It turns out that our decision on the memory allocation is quite near to the optimal scenario in the exploited space.

Effect of Integration with Tuple Selection and Projection

It is desirable to see how the tuple selection and projection conditions that are pushed down to the joining instances of a self-join will affect the effectiveness and efficiency of

SCALE. We therefore designed an experiment to investigate this.

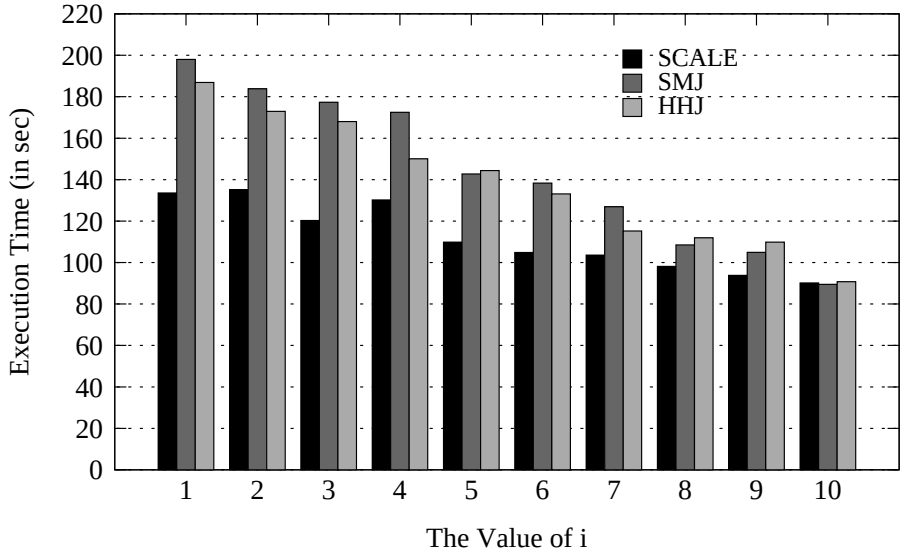
SCALE incorporated the first approach in Section 4.3.3 to enable tuple selection and projection pushdown, which requires much less implementation effort but has obviously worse performance than the second approach. We defined a refined self-join query template:

SELECT *
FROM R AS R_1, R AS R_2
WHERE $R_1.A = R_2.B$
  AND $R_1.C \geq i \times 5 \times 10^4$
  AND $R_2.C \leq 10^6 - i \times 5 \times 10^4$

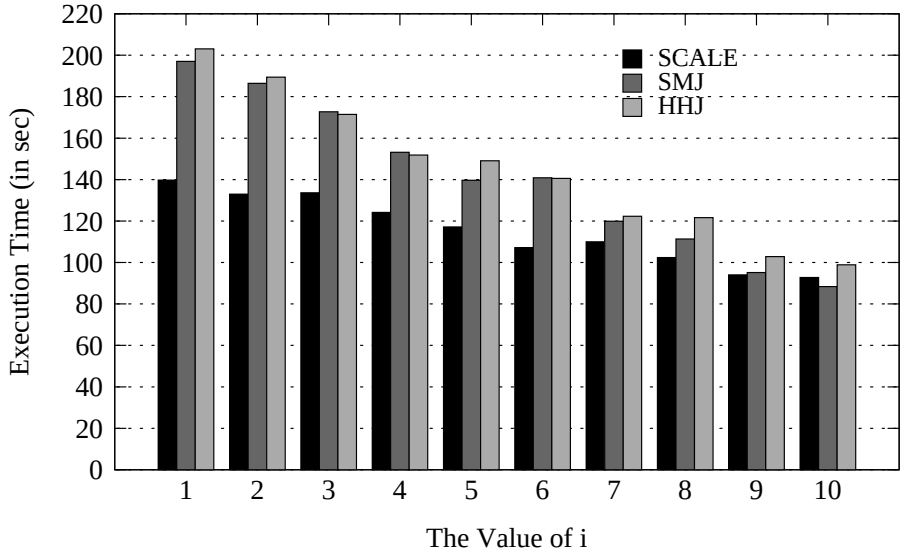
where C is a third integer attribute of R whose values are uniformly distributed over $[1, 10^6]$ and i is an integer parameter ranging from 1 to 10. By varying the value of i, we can easily and accurately control the tuple selection selectivities of R_1 and R_2, as well as the number of overlapped tuples between these two instances. For simplicity we did not introduce tuple projection into the queries.

We tested the above refined self-join queries against two synthetic tables with fixed MEM (10MB), AD (uniform), MD (10^5) and DD (uniform) but two different DV values (1×10^5 and 5×10^5). The experimental results are shown in Fig. 4.10. In both SMJ and HHJ, the selection conditions in the queries were pushed down to the level of scanning R_1 and R_2.

As can be seen, SCALE still performed better than SMJ and HHJ in almost all scenarios. As i increased, the benefit of SCALE disappeared gradually. However, this trend is expected because the benefit of SCALE mainly originates from the overlap between R_1 and R_2. When i became 10, which was actually the worst case as R_1 and R_2 are totally disjoint, the execution times of these three approaches are more or less the same. This phenomenon, however, is surprisingly positive. Note that regardless of the i value, in



(a) DV = 1×10^5



(b) DV = 5×10^5

Figure 4.10: Test on integration with selection condition $R_1.C \geq i \times 5 \times 10^4$ and $R_2.C \leq 10^6 - i \times 5 \times 10^4$, 1GB tables with 10 million tuples, MEM = 10MB, AD = uniform, MD = 10^5, DD = uniform

this test SCALE was always sorted on the original R and then two full sequential scans of $\mathcal{S}_A(R)$ were conducted. Furthermore, as mentioned above, our implementation of SCALE chose the worse one of two candidate solutions for the purpose of combining tuple selection and projection. Therefore, we can optimistically conclude that a fully optimized

SCALE will be superior to both SMJ and HHJ when dealing with general self-join queries.

4.6 Extensions to SCALE

In this section, we propose two extensions to SCALE. The first extension can improve SCALE's performance, while the second extension generalizes SCALE for it to be utilized by more applications.

4.6.1 Sideways Information Passing

A tuple t in R plays roles as both the left-hand side (LHS) and the right-hand side (RHS) in the self-join condition $R_1.A = R_2.B$. Let $\mathbb{N}_A(a)$ and $\mathbb{N}_B(b)$ denote the total number of tuples in R that have the attribute value $A = a$ and $B = b$ respectively. Suppose SCALE sorts R on A into $\mathcal{S}_A(R)$.

If $\mathbb{N}_A(t.B)$ (resp. $\mathbb{N}_B(t.A)$) is zero, t will not be able to find corresponding right- (resp. left-) matching tuples in R. If both $\mathbb{N}_A(t.B)$ and $\mathbb{N}_B(t.A)$ are zero, then t is totally irrelevant to the self-join. Therefore, it would be beneficial to prune such irrelevant tuples from R as early as possible. To achieve this, we collect the value distribution information not only for attribute A, but also for attribute B, during the initial run formation phase of externally sorting R into $\mathcal{S}_A(R)$. We can then discard those irrelevant tuples on-the-fly when merging the initial sorted runs during the subsequent run merge phase.

During the first pass of processing $\mathcal{S}_A(R)$, it is safe and beneficial to remove a tuple t from the main buffer once t can no longer left-join or right-join with any other existing or incoming tuples in the main buffer. This is called *eager tuple pruning strategy*.

At the moment when t becomes eligible for the early pruning, $RM_3(t)$ must be empty. In the meantime, all the left-matching tuples of t must also have been read into the main buffer. This situation can be easily determined by counting the tuples whose attribute B

values are equal to $t.A$ and so far have been read into the main buffer, and comparing the number with $\mathbb{N}_B(t.A)$ which will be collected during the external sorting of R as described above.

4.6.2 Self Band–Join

A band-join [22, 46] between two relations R and S on attributes $R.A$ and $S.B$ has the join condition of the form $R.A - c_1 \leq S.B \leq R.A + c_2$, where c_1 and c_2 are constants that may be equal, and either one of them, but not both, may be zero. Band joins are common in queries that require joins over continuous domains such as time and distance. A self band-join involves two instances R_1 and R_2 of relation R with a join condition $R_1.A - c_1 \leq R_2.B \leq R_1.A + c_2$.

Extending `SCALE` to the self band-join is straightforward. For a tuple t with $t.B = b$, its right-matching tuples *RM(t)* becomes the set of consecutive tuple segments in $\mathcal{S}_A(R)$ with their A values falling into the range $[b - c_2, b + c_1]$. Besides, there are no other modifications required to enable `SCALE` to handle self band-joins.

The two schemes of sideway information passing discussed in Section 4.6.1 are also extendible according to self band-join. For a tuple t in R, when $\sum_{i=t.B-c_2}^{t.B+c_1} \mathbb{N}_A(i) = 0$, t will not be able to find corresponding right-matching tuples in R; similarly, when $\sum_{i=t.A-c_1}^{t.A+c_2} \mathbb{N}_B(i) = 0$, t will not be able to find corresponding left-matching tuples in R. Therefore, if $\sum_{i=t.B-c_2}^{t.B+c_1} \mathbb{N}_A(i) = 0$ and $\sum_{i=t.A-c_1}^{t.A+c_2} \mathbb{N}_B(i) = 0$, then t is irrelevant to the self band-join and can be pruned during the external sorting of R into $\mathcal{S}_A(R)$.

The principle of the eager tuple pruning for the main buffer also applies to the self band-join. However, in order to determine if all the left-matching tuples of a tuple t have been read into the main buffer, it requires counting all the tuples whose attribute B values fall in the range $[t.A - c_1, t.A + c_2]$.

It is also obvious that the self band-join can be integrated with tuple selection and

projection pushdown as described in Section 4.3.3, since the merge join between $\mathcal{S}_A(R^+)$ and $\mathcal{S}_B(R^-)$ is also well adaptive to band-join.

4.7 Summary

In this chapter, we have proposed SCALE, a self-join algorithm that efficiently deals with self equi-joins. SCALE can benefit from but does not rely on indices, is compatible with tuple selection and projection pushdown, and is easily extendable to handle self band-joins. Our analytical study showed that SCALE, in the ideal situation is simply one sequential scan of the mutli-instance relation referred by the join, and in the worst case degenerates to Sort-Merge Join. Our extensive performance evaluation showed that that SCALE is generally superior to conventional join algorithms like Sort-Merge Join, Hybrid Hash Join and Nested-Loop Join.

CHAPTER 5

Conclusion

Multiple instances of the same relation frequently exist within complex analytical queries. However, the existence of relational instances has not received much research attention in the past. While it is feasible to treat these instances as distinct relations during query processing, such obliviousness will result in sub-optimal query performance. In contrast, distinguishing the instances with well customized query evaluation techniques can bring substantial performance improvement. This thesis is the first systematic study on optimizing complex queries with multiple relational instances.

This chapter concludes the thesis by summarizing our work, discussing the contributions and presenting some interesting directions or concrete problems that are relevant to the thesis topic and worthy of exploration in the future.

5.1 Contributions

This thesis revisited three traditional research problems, i.e. accessing tables resident on the disk, external table sorting and relational join processing, in a new light of relational instances. We figured out several optimization opportunities that are brought by relational instances and the accompanying solutions for further performance enhancement.

In Chapter 2, we demonstrated that it is both beneficial and feasible to allow multiple instances of the same base relation to share a single physical table scanner. We developed MAPLE, a *M*ulti-instance-*A*ware *PL*an *E*valuation engine to execute queries with multiple instances. The major encountered obstacle was the fact that, under the conventional pull iterator execution model and with limited buffer space, a shared scan would be slowed down or even totally blocked if the tuples of different instances are consumed at dramatically diverse speeds. To resolve blocked shared scans and in the meantime to minimize the query cost, MAPLE makes use of an interleaved iterative query evaluator. The query plans to execute originate from a traditional query optimizer, but are enhanced in a cost-based manner with additional materialized operators and explicit share groups. Instances within each share group share one physical scan. We implemented MAPLE in PostgreSQL, and our experimental study on the TPC-DS benchmark showed significant reduction in execution time.

In Chapter 3, we showed that better query performance can be reached by enabling the sharing and collaboration when carrying out different sortings on instances. The optimization opportunities arise when the relationship between two sort orders falls into one of the four general cases that we identified, and the actual enhancement is achieved by applying various sort sharing techniques that we investigated, including the novel cooperative sorting idea. For a query containing a set of instances, we are able to generate its sort-sharing equipped execution plan with either two-phase or single-phase query

optimization. In the two-phase method, we post-optimize the plan resulted from a conventional query optimizer to determine the best cost-efficient way to applying our sort sharing techniques. In the single-phase method, the query optimizer directly generates an optimal sort-sharing-aware execution plan. In comparison with the single-phase method, the two-phase method is more light-weight but cannot guarantee the global optimality of the execution plan. We demonstrated the efficiency of our ideas with a prototype built in PostgreSQL and evaluated the performance using both TPC-DS benchmark and synthetic data. Our experimental results showed significant performance improvement over the traditional scheme.

In Chapter 4, we addressed the problem of self-join with the join predicates involving two distinct attributes, by proposing an efficient SCALE (Sort for Clustered Access with Lazy Evaluation) join algorithm. SCALE improves tuples' clustered accesses to their joining counterparts, and tries to maximize the overall chance of full-range clustered access, where a tuple needs to be read into memory only once to join with all of its matching tuples. SCALE handles tuples for which a full-range clustered access is still not possible, by adopting a lazy evaluation strategy to defer their remaining join processing to a later time. Such lazy evaluation minimizes the need for "random" accesses to the matching tuples. For supporting clustered access and lazy evaluation, the memory space has to be effectively allocated between these two tasks. SCALE applies a cost-based approach to address this problem. SCALE is also able to handle the situation where the two joining instances of the same relation are associated with different selection and projection predicates. Moreover, SCALE is further strengthened with side-ways information passing techniques and is extendible to handle self band-joins. Our analytical study showed that SCALE degenerates gracefully to a Sort-Merge Join in the worst case. We also implemented SCALE in PostgreSQL, and results of our extensive experimental study showed that it outperforms Nested-Loop Join, Sort-Merge Join and Hybrid Hash Join by a wide

margin in (almost) all cases.

Note that our techniques in the three chapters are fundamental and general, from which some research problems in research fields other than relational query processing could also benefit. For example, the shared table scan idea in MAPLE can be applied to the context of distributed or cloud data management. The cooperative sorting idea can be used for top-k query processing and MapReduce framework as both involve a lot of sortings on massive data. The SCALE algorithm is also suitable for self-joins arising within spatial/temporal query processing and string similarity search.

5.2 Future Work

There are several future research directions to investigate how relational instances can be better handled during relational query processing.

5.2.1 Refining Invented Techniques

Techniques presented in this thesis have not been fully exploited yet, and there still remains great potential for further enhancement in terms of both completeness and applicability.

The MAPLE system. First, MAPLE is easily extendible to support common subexpressions within a single query (instead of just table scans) as well as across multiple queries. The result of a common subexpression, either pipelined or materialized, can be treated as a virtual table and shared "scanned" by all instances. For multiple queries, common subexpressions or tables across multiple queries can be shared in a similar manner. In addition, these queries can be processed simultaneously without any execution dependency between them. Second, as shown in our experimental study, interleaved execution of operators may impact performance of a query in a negative way, i.e., the FragmentRead-

Write effect. It remains a challenge to explore how we can extend MAPLE to consider these factors. In particular, we need to ensure that a MAPLE-enhanced plan must not be inferior to the corresponding PostgreSQL plan. Finally, MAPLE is a post-optimization strategy. As such, it only enhances a single plan generated by the optimizer. It would be interesting to explore an integrated strategy, i.e., to extend the search space of a query optimizer to support instance-awareness as a plan is built. In this way, the generated plan is expected to be superior over MAPLE's plan.

Self-join processing. First, in SCALE, the join result tuples are not delivered in a sorted order. However, in some cases, it would be desirable and beneficial to produce a sorted join output. It remains a challenge to revise the algorithm so as to generate sorted results without incurring too much overhead. Second, the memory allocation among buffers of SCALE is done statically based on the cost estimation before the first scan of the sorted relation. As the cost estimation might not be accurate due to inaccurate statistics, it would be interesting to design a dynamic allocation algorithm to adjust the memory allocation among all buffers at runtime. Third, SCALE does not exploit the opportunity of outputting join results during the external sorting of the relation at the beginning. Intuitively, interleaving the sorting with the tuple matching procedure would further improve the execution time. Fourth, our current techniques work well for a binary self-join. When executing a query with multi-way self-joins, there might be a more efficient way than simply applying a binary tree of SCALE operations.

A uniform framework. In this thesis, we have studied the three research problems in isolation. It would be promising to develop a uniform optimization framework which combines all the techniques proposed for the three problems. In so doing, we may be able to exploit additional opportunities for optimization.

5.2.2 Developing New Techniques

Hash sharing. For many database matching tasks such as join and aggregation, hashing is a competitor to sorting. As a result, frequently there are also multiple demands of hashing the same relation based on different hash keys. It is natural to think over the notion of *hash sharing* which is parallel to the sort sharing idea in Chapter 3. Similarly, the initial study should focus on the relationships between hash keys to derive feasible and efficient *partition sharing* and *cooperative hashing* techniques. One straightforward example of partition sharing is when the attributes of a hash key hk_1 is a subset of the attributes of another hash key hk_2, partitions on hk_2 can be produced in a pipelined manner from the partitions on hk_1. As for cooperative hashing, an immediate idea is that given two hash keys hk_1 and hk_2 whose attributes do not overlap, perform a two-dimensional hashing on $\{hk_1, hk_2\}$ to generate a partition matrix, in which a row of partitions together form a partition on hk_1 and a column of partitions together form a partition on hk_2. To achieve good performance, the chosen hash functions should avoid resulting in too many partitions in the matrix, and the reading of partitions should be carefully scheduled so as to incur as few random I/O activities as possible.

Minimizing non-I/O costs. Till now, our optimization goal is to minimize the total I/O cost, which has been the major performance bottleneck in traditional disk-based database systems for a long period. Recently, the abundance of main memory makes it more and more likely for a moderate enterprise database system to completely reside in the memory. As such, database applications are becoming increasingly compute and memory intensive. Along with this trend, two major scenarios arise for optimizing relational instances in a main-memory DBMS.

- When the memory bandwidth cannot keep pace with the increasing capability of CPU(s), accessing memory becomes the performance bottleneck. In this case, it

would be meaningful to look closely at the *in-memory* versions of disk-based query processing problems about instances, including those already covered in this thesis (i.e., shared scan, sort sharing and self-join) as well as the newly mentioned hash sharing problem above.

- When there is sufficient memory bandwidth, the majority of query evaluation time goes to CPU computation and thus the system is CPU-bound. In this case, we should seek the opportunities of sharing in the CPU cache the common computations (e.g., tuple filtering, attribute projection and aggregation) among tuples of instances to minimize the total CPU cycles consumed.

BIBLIOGRAPHY

[1] Microsoft SQL Server Library. `http://msdn2.microsoft.com/en-us/library/bb545450.aspx`.

[2] Postgresql Offical Website. `http://www.postgresql.org/`.

[3] TPC BENCHMARK Decision Support. `http://www.tpc.org/tpcds/`.

[4] Transaction Processing Performance Council. `http://www.tpc.org/`.

[5] Daniel J. Abadi, Adam Marcus, Samuel R. Madden, and Kate Hollenbach. Scalable semantic web data management using vertical partitioning. In *VLDB*, pages 411–422, 2007.

[6] Sameet Agarwal, Rakesh Agrawal, Prasad Deshpande, Ashish Gupta, Jeffrey F. Naughton, Raghu Ramakrishnan, and Sunita Sarawagi. On the computation of multidimensional aggregates. In *VLDB*, pages 506–521, 1996.

[7] Noga Alon, Phillip B. Gibbons, Yossi Matias, and Mario Szegedy. Tracking join and self-join sizes in limited storage. In *PODS*, pages 10–20, 1999.

[8] Ramesh Bhashyam. TPC-D: the challenges, issues and results. *SIGMOD Rec.*, 25(4):89–93, 1996.

[9] M.W. Blasgen and K.P. Eswaran. On the evaluation of queries in a relational database system. *IBM Research Report*, RJ 1745, 1976.

[10] Kjell Bratbergsengen. Hashing methods and relational algebra operations. In *VLDB*, pages 323–333, 1984.

[11] Yu Cao, Ramadhana Bramandia, Chee-Yong Chan, and Kian-Lee Tan. Optimized query evaluation using cooperative sorts. In *ICDE*, pages 601–612, 2010.

[12] Yu Cao, Ramadhana Bramandia, Chee-Yong Chan, and Kian-Lee Tan. Sort-sharing-aware query processing. *The VLDB Journal*, 2011.

[13] Yu Cao, Gopal C. Das, Chee-Yong Chan, and Kian-Lee Tan. Optimizing complex queries with multiple relation instances. In *SIGMOD*, pages 525–538, 2008.

[14] Moses Charikar, Surajit Chaudhuri, Rajeev Motwani, and Vivek Narasayya. Towards estimation error guarantees for distinct values. In *PODS*, pages 268–279, 2000.

[15] Moses Charikar, Chandra Chekuri, Zuo Dai, Ashish Goel, Sudipto Guha, and Ming Li. Approximation algorithms for directed steiner problems. In *Journal of Algorithms*, pages 73–91, 1999.

[16] Damianos Chatziantoniou and Kenneth A. Ross. Groupwise processing of relational queries. In *VLDB*, pages 476–485, 1997.

[17] Edgar F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 26(1):64–69, 1983.

[18] Latha S. Colby, Richard L. Cole, Edward Haslam, Nasi Jazayeri, Galt Johnson, William J. McKenna, Lee Schumacher, and David Wilhite. Redbrick vista: Aggregate computation and management. In *ICDE*, pages 174–177, 1998.

[19] Nilesh N. Dalvi, Sumit K. Sanghai, Prasan Roy, and S. Sudarshan. Pipelining in multi-query optimization. *Journal of Computer and System Sciences*, 66(4):728–762, 2003.

[20] Amol Deshpande, Zachary Ives, and Vijayshankar Raman. Adaptive query processing. *Found. Trends databases*, 1:1–140, January 2007.

[21] David J. DeWitt, Randy H Katz, Frank Olken, Leonard D Shapiro, Michael R Stonebraker, and David A. Wood. Implementation techniques for main memory database systems. *SIGMOD Rec.*, 14(2):1–8, 1984.

[22] David J. DeWitt, Jeffrey F. Naughton, and Donovan A. Schneider. An evaluation of non-equijoin algorithms. In *VLDB*, pages 443–452, 1991.

[23] R.J. Dinsmore. Longer strings from sorting. *Comm. ACM*, 8(1):48, 1965.

[24] Vladmir Estivill-Castro and Derick Wood. A survey of adaptive sorting algorithms. *ACM Computing Surveys (CSUR)*, 24(4):441–476, 1992.

[25] W. Donald Frazer and Chi Kuen Wong. Sorting by natural selection. *Communications of the ACM*, 15(10):910–913, 1972.

[26] Shinya Fushimi, Masaru Kitsuregawa, and Hidehiko Tanaka. An overview of the system software of a parallel relational database machine GRACE. In *VLDB*, pages 209–219, 1986.

[27] Cesar A. Galindo-legaria, Goetz Graefe, Milind M. Joshi, and Ross T. Bunker. System and method for segmented evaluation of database queries. *US Patent No. 7,599,953*, 29 Nov. 2004.

[28] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. 1990.

[29] Leonidas Georgiadis. Arborescence optimization problems solvable by edmonds' algorithm. *Theor. Comput. Sci.*, 301(1-3):427–437, 2003.

[30] Phillip B. Gibbons. Distinct sampling for highly-accurate answers to distinct values queries and event reports. In *VLDB*, pages 541–550, 2001.

[31] Goetz Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

[32] Goetz Graefe. Implementing sorting in database systems. *ACM Computing Surveys*, 38(3), 2006.

[33] Goetz Graefe. A generalized join algorithm. In *BTW*, pages 267–286, 2011.

[34] Goetz Graefe and William J. McKenna. The volcano optimizer generator: Extensibility and efficient search. In *ICDE*, pages 209–218, 1993.

[35] Ravindra Guravannavar and S. Sudarshan. Reducing order enforcement cost in complex query plans. In *ICDE*, pages 856–865, 2007.

[36] Stavros Harizopoulos, Vladislav Shkapenyuk, and Anastassia Ailamaki. Qpipe: a simultaneously pipelined relational query engine. In *SIGMOD*, pages 383–394, 2005.

[37] Ming-I Hsieh, Eric Hsiao-Kuang Wu, and Meng-Feng Tsai. Fasterdsp: A faster approximation algorithm for directed steiner tree problem. *J. Inf. Sci. Eng.*, 22(6):1409–1425, 2006.

[38] H. V. Jagadish, Adriane Chapman, Aaron Elkiss, Magesh Jayapandian, Yunyao Li, Arnab Nandi, and Cong Yu. Making database systems usable. In *SIGMOD*, pages 13–24, 2007.

[39] Richard M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[40] Donald E. Knuth. *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. Addison-Wesley, 1998.

[41] Robert Philip Kooi. *The optimization of queries in relational databases*. PhD thesis, Case Western Reserve University, 1980.

[42] Christian A. Lang, Bishwaranjan Bhattacharjee, Tim Malkemus, Sriram Padmanabhan, and Kwai Wong. Increasing buffer-locality for multiple relational table scans through grouping and throttling. In *ICDE*, pages 1136–1145, 2007.

[43] Christian A. Lang, Bishwaranjan Bhattacharjee, Tim Malkemus, and Kwai Wong. Increasing buffer-locality for multiple index based scans through intelligent placement and index scan speed control. In *VLDB*, pages 1298–1309, 2007.

[44] Per-Ake Larson. External sorting: Run formation revisited. *IEEE Transactions on Knowledge and Data Engineering*, 15(4):961–972, 2003.

[45] Hui Lei and Kenneth A. Ross. Faster joins, self-joins and multi-way joins using join indices. *Data Knowl. Eng.*, 29(2):179–200, 1999.

[46] Hongjun Lu and Kian-Lee Tan. On sort-merge algorithm for band joins. *IEEE Transactions on Knowledge and Data Engineering*, 7(3):508–510, 1995.

[47] Masaya Nakayama, Masaru Kitsuregawa, and Mikio Takagi. Hash-partitioned join method using dynamic destaging strategy. In *VLDB*, pages 468–478, 1988.

[48] Thomas Neumann and Guido Moerkotte. An efficient framework for order optimization. In *ICDE*, pages 461–472.

[49] Thomas Neumann and Guido Moerkotte. A combined framework for grouping and order optimization. In *VLDB*, pages 960–971, 2004.

[50] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The lru-k page replacement algorithm for database disk buffering. In *SIGMOD*, pages 297–306, 1993.

[51] Vinay S. Pai and Peter J. Varman. Prefetching with multiple disks for external mergesort: simulation and analysis. In *ICDE*, pages 273–282, 1992.

[52] Theoni Pitoura and Peter Triantafillou. Self-join size estimation in large-scale distributed data systems. In *ICDE*, pages 764–773, 2008.

[53] Nicholas Roussopoulos. View indexing in relational databases. *ACM Trans. Database Syst.*, 7(2):258–290, 1982.

[54] Prasan Roy, S. Seshadri, S. Sudarshan, and Siddhesh Bhobe. Efficient and extensible algorithms for multi query optimization. In *SIGMOD*, pages 249–260, 2000.

[55] Betty Salzberg. Merging sorted runs using large main memory. *Acta Informatica*, 27(3):195–215, 1989.

[56] Patricia Griffiths Selinger, Morton M Astrahan, Donald Dean Chamberlin, Raymond A. Lorie, and Thomas Gordon Price. Access path selection in a relational database management system. In *SIGMOD*, pages 23–34, 1979.

[57] Jayavel Shanmugasundaram, Jerry Kiernan, Eugene J. Shekita, Catalina Fan, and John Funderburk. Querying xml views of relational data. In *VLDB*, pages 261–270, 2001.

[58] David Simmen, Eugene Shekita, and Timothy Malkemus. Fundamental techniques for order optimization. In *SIGMOD*, pages 57–67, 1996.

[59] T.C. Ting and Y.W. Wang. Multiway replacement selection sort with dynamic reservoir. *The Computer Journal*, 20(4):298–301, 1977.

[60] Tolga Urhan, Michael J. Franklin, and Laurent Amsaleg. Cost-based query scrambling for initial delays. *SIGMOD Rec.*, 27(2):130–141, 1998.

[61] Patrick Valduriez. Join indices. *ACM Trans. Database Syst.*, 12(2):218–246, 1987.

[62] Xiaoyu Wang and Mitch Cherniack. Avoiding sorting and grouping in processing queries. In *VLDB*, pages 826–837, 2003.

[63] Andreas Weininger. Efficient execution of joins in a star schema. In *SIGMOD*, pages 542–545, 2002.

[64] Weiye Zhang and Per-Ake Larson. Dynamic memory adjustment for external mergesort. In *VLDB*, pages 376–385, 1997.

[65] Weiye Zhang and Per-Ake Larson. Buffering and read-ahead strategies for external mergesort. In *VLDB*, pages 523–533, 1998.

[66] Yihong Zhao, Prasad M. Deshpande, Jeffrey F. Naughton, and Amit Shukla. Simultaneous optimization and evaluation of multiple dimensional queries. In *SIGMOD*, pages 271–282, 1998.

[67] Luoquan Zheng and Per-Ake Larson. Speeding up external mergesort. *IEEE Transactions on Knowledge and Data Engineering*, 8(2):322–332, 1996.

[68] Jingren Zhou, Per-Ake Larson, Johann-Christoph Freytag, and Wolfgang Lehner. Efficient exploitation of similar subexpressions for query processing. In *SIGMOD*, pages 533–544, 2007.

[69] Marcin Zukowski, Sándor Héman, Niels Nes, and Peter Boncz. Cooperative scans: dynamic bandwidth sharing in a DBMS. In *VLDB*, pages 723–734, 2007.

APPENDIX A

Supplementary Materials for Chapter 3

A.1 The Proof of Theorem 3.1

In this section, we provide the proof of Theorem 3.1 in Section 3.5.1. 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 $((o_1 \cdot o_2) \cdot o_3) \cdot ... \cdot o_i$.

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.

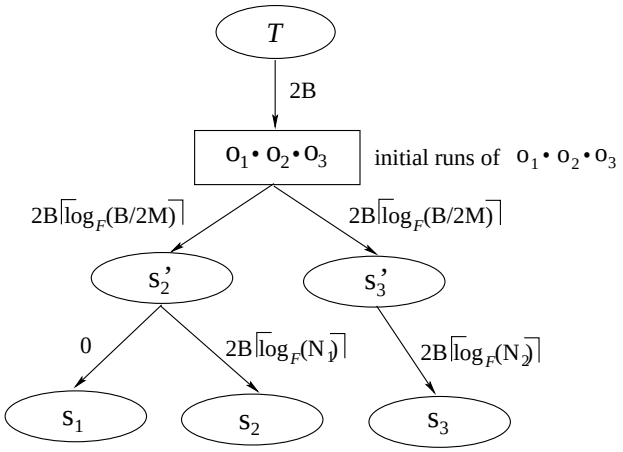


Figure A.1: The Execution Plan of 3-way Cooperative Sorting

Analysis of 3-way cooperative sorting. Fig. A.1 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.

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 $C_{s'_2 \rightarrow s_1}$ of performing internal sortings for all the composite chunks of s'_2, and s_2 can be produced by the chunk merging procedure (Section 3.4.1) 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

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

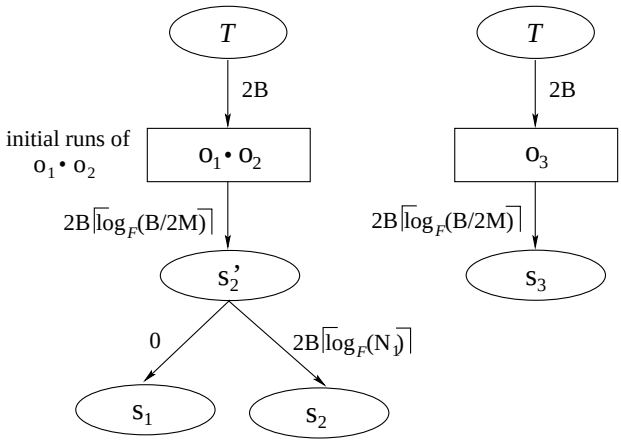


Figure A.2: The Alternative Execution Plan of 2-way Cooperative Sorting

We compare this execution plan with another plan that is based on 2-way cooperative sorting depicted in Fig. A.2, 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

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

The difference obtained by subtracting Eqn. A.2 from Eqn. A.1 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.

The execution plan of 4-way cooperative sorting is shown in Fig. A.3. The table T

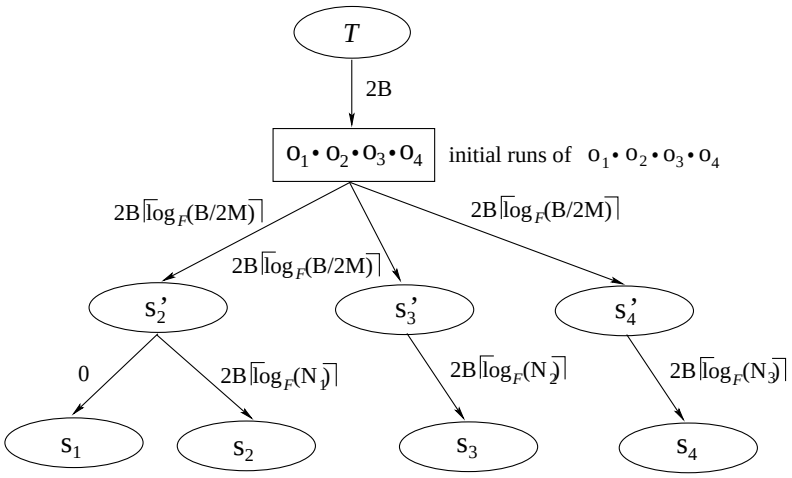


Figure A.3: The Execution Plan of 4-way Cooperative Sorting

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

$$\begin{aligned} 2 \times B \times (1 + 3 \times \lceil log_F \frac{B}{2M} \rceil + \lceil log_F N_1 \rceil + \lceil log_F N_2 \rceil \\ + \lceil log_F N_3 \rceil) + 3 \times C_{is} + C_{s'_2 \rightarrow s_1} \end{aligned} \tag{A.3}$$

The alternative execution plan that utilizes binary cooperative sorting is depicted in Fig. A.4. 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

$$\begin{aligned} 2 \times B \times (2 + 2 \times \lceil log_F \frac{B}{2M} \rceil + \lceil log_F N_1 \rceil + \lceil log_F N_4 \rceil) \\ + 2 \times C_{is} + C_{s'_2 \rightarrow s_1} + C_{s'_a \rightarrow s_3} \end{aligned} \tag{A.4}$$

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

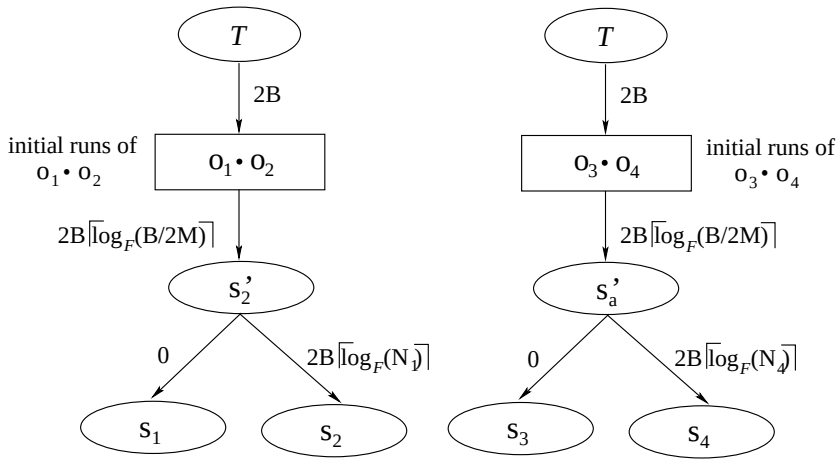


Figure A.4: The Alternative Execution Plan of 2-way Cooperative Sorting

The difference obtained by subtracting Eqn. A.4 from Eqn. A.3 is

$$2 \times B \times (\lceil log_F \frac{B}{2M} \rceil + \lceil log_F N_2 \rceil + \lceil log_F N_3 \rceil - \lceil log_F N_4 \rceil - 1) + C_{is} - C_{s'_a \rightarrow s_3} \quad \text{(A.5)}$$

First of all, we assume that the value of $|C_{is} - C_{s'_a \rightarrow s_3}|$ 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. A.5 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. A.5 and Fig. A.6, respectively. In Fig. A.6, sa_i is the intermediate sort operation for the cooperative sorting between s_i and s_{i+1}.

As shown, the plan in Fig. A.5 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. A.6 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.

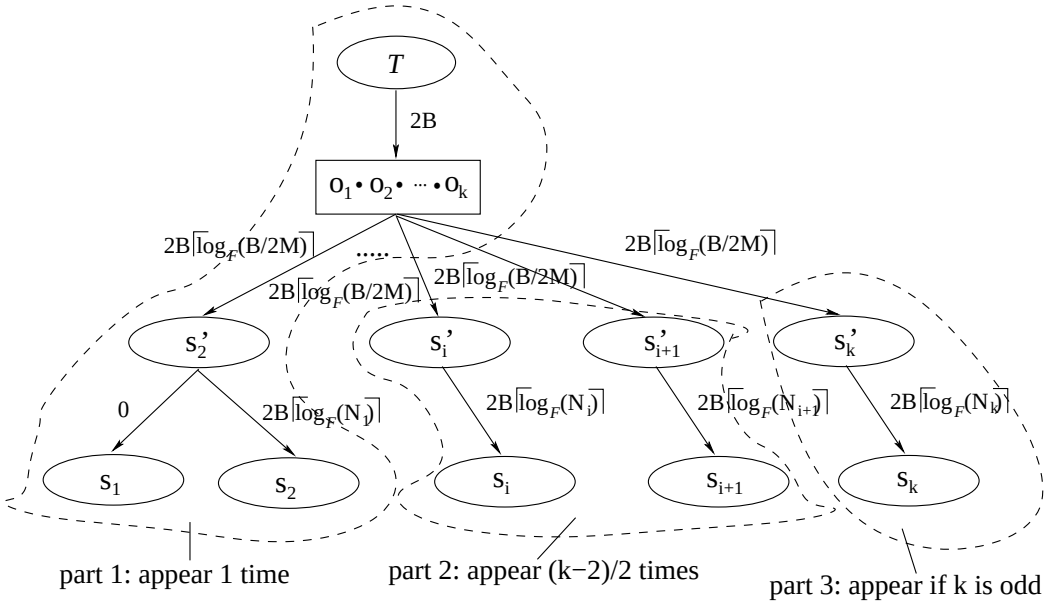


Figure A.5: The Execution Plan of k-way Cooperative Sorting

First of all, the cost of part 1 in both figures are equal. Note that the cost difference between part 3 in Fig. A.5 and in Fig. A.6 is exactly the same as the difference between

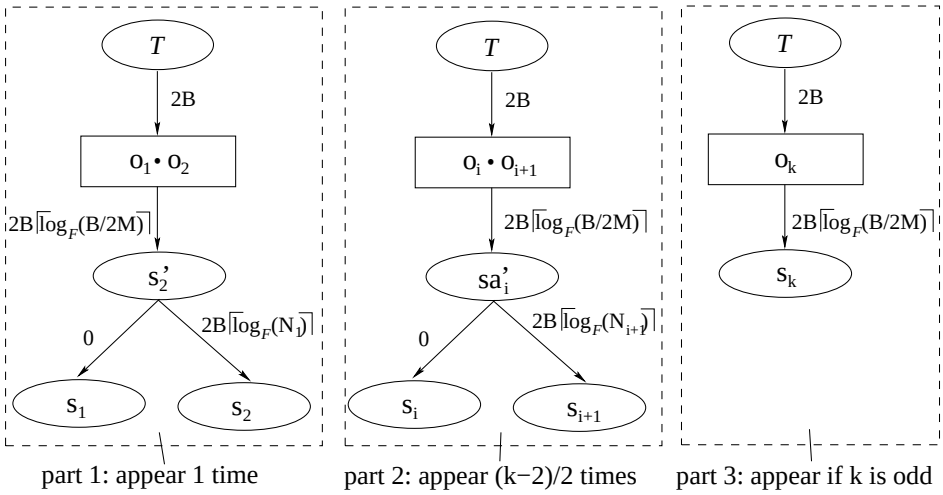


Figure A.6: The Alternative Execution Plan of 2-way Cooperative Sorting

Eqn. A.1 and Eqn. A.2 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. A.5 and Fig. A.6, the cost difference of deriving them between the former figure and the latter is actually the same as the difference between Eqn. A.3 and Eqn. A.4, i.e. Eqn. A.5, 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. A.6 is no higher than part i in Fig. A.5. 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.

A.2 Component Costs of Sorting Results in Performance Study

In this section, we list the component costs of sortings in Section 3.7.1 and Section 3.7.3. The meanings of cost components in Section 3.7.1 are given in Table 3.3. The meanings of cost components in Section 3.7.3 are given in Table 3.5.

	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 A.1: Component Costs of Sortings in the Micro-benchmark Test of Section 3.7.1 (in seconds)

	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 A.2: Component Costs of CIB and NIB with SF 40 in Section 3.7.3 (in seconds)

	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 A.3: Component Costs of CIB and NIB with SF 100 in Section 3.7.3 (in seconds)