OPTIMIZATION TECHNIQUES FOR COMPLEX MULTI-QUERY APPLICATIONS

Wang Guoping

NATIONAL UNIVERSITY OF SINGAPORE

2014

NATIONAL UNIVERSITY OF SINGAPORE

DOCTORAL THESIS

OPTIMIZATION TECHNIQUES FOR COMPLEX MULTI-QUERY APPLICATIONS

Author:
Wang Guoping

Supervisor:
Prof. Chan Chee Yong

A thesis submitted
for the degree of Doctor of Philosophy

in the

Department of Computer Science
School of Computing

2014

DECLARATION

I hereby declare that this thesis is my original work and it has been written by me in its entirety.

I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.

Wang Guoping

January, 2014

ACKNOWLEDGEMENT

I would like to express the deepest appreciation to my supervisor, Prof. Chan Chee Yong. Without his guidance and persist help, my thesis would not have been finished. During the last few years, he has spent countless time to patiently guide me to build interesting ideas, strengthen the algorithms and improve the writings. As a supervisor, he shows his wisdom, insights, wide knowledge and conscientious attitude. All of these set me a good example to be a good researcher. In addition to my research, He also helps me a lot on my personal life. After my scholarship terminated, He hired me as a research assistant and gave me the GSR support under his research grant so that I can concentrate on my research without worrying about the financial problems. During my job hunting, he gave me many valuable suggestions and comments. I am really grateful to have him as my supervisor in my Ph.D. life.

I would like to thank my thesis committee, Prof. Tan Kian Lee and Prof. Stephane Bressan for their valuable comments on my thesis as well as recommendation letters for my research assistant position as well as job hunting.

I would like to thank all my friends in the database group who have made my Ph.D. life more colorful. They are Bao Zhifeng, Li Lu, Li Hao, Zeng Zhong, Kang Wei, Zhou Jingbo, Tang Ruiming, Song Yi, Zeng Yong, Xiao Qian and many others. Special thanks to the church events organized by Prof. Tan Kian Lee and Dr. Wang Zhengkui every year which bring us together as a family.

Finally, I would like to thank my parents for their silent support and trust for every decision I made during my Ph.D. life.

CONTENTS

SUMMARY

Many applications often involve complex multiple queries which share a lot of common subexpressions (CSEs). Identifying and exploiting the CSEs to improve query performance is essential in these applications. Multiple query optimization (MQO), which aims to identify and exploit the CSEs among queries in order to reduce the overall query evaluation cost, has been extensively studied for over two decades and demonstrated to be an effective technique in both RDBMS and MapReduce contexts by existing works. In this thesis, we study the following three novel MQO problems.

First, we study the problem of efficient processing of enumerative set-based queries (SQs) in RDBMS. Enumerative SQs aim to find all the sets of entities of interest to meet certain constraints. In this work, we present a novel approach to evaluate enumerative SQs as a collection of cross-product queries (CPQs) and propose efficient and scalable MQO heuristics to optimize the evaluation of a collection of CPQs. Our experimental results demonstrate that our proposed approach is significantly more efficient than conventional RDBMS methods. To the best of our knowledge, that is the first work that addresses the efficient evaluation of a collection of CPQs.

Second, we study multi-query/job optimization techniques and algorithms in the MapReduce framework. In this work, we first propose two new multi-job optimization techniques to share map input scan and map output in the MapReduce paradigm. We then propose a new optimization algorithm that, given an input batch of jobs, produces an optimal plan by a judicious partitioning of the jobs into groups and an optimal assignment of the processing technique to each group. Our experimental results on Hadoop demonstrate

the efficiency and effectiveness of our proposed techniques and algorithms by comparing with the state-of-the-art techniques and algorithms.

Finally, we examine the optimal join enumeration (OJE) problem, which is a fundamental query optimization task for SQL-like queries, in the MapReduce framework. In this work, we study both the single-query and multi-query OJE problems and propose efficient join enumeration algorithms for these problems. The study of the single-query OJE problem serves as a foundation for the study on the multi-query OJE problem. Our experimental results demonstrate the efficiency of our proposed join enumeration algorithms. To the best of our knowledge, this work presents the first systematic study of the OJE problem in the MapReduce paradigm.

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

In this chapter, we first present some background on multiple query optimization. We then state the research problems and contributions of this thesis. Finally, we discuss the organization of this thesis.

1.1 Multiple Query Optimization

Many applications often involve complex multiple queries which share many common subexpressions (CSEs) [54, 51, 14, 74, 44]. In the presence of multiple queries, either produced by complex applications or batched by some systems like database and MapReduce systems, a simplistic solution to answer these queries is to evaluate them one by one, ignoring the CSEs among them. However, this solution is suboptimal since the CSEs are redundantly evaluated. An optimal solution should be able to evaluate the CSEs once and reuse the results of the CSEs for subsequent queries to improve the overall query performance. Since complex multiple queries usually take a long time to evaluate due to the inherent complexity of the queries, there could be considerable performance saving by sharing the computation of the CSEs among the queries. As a result, identifying

and exploiting the CSEs to improve the query performance is essential in these complex multi-query applications.

To share the computation of the CSEs among multiple queries, a well known technique is multiple query optimization (MQO). MQO, which aims to identify the CSEs among queries and exploit them to reduce the query evaluation cost, has been extensively studied for over two decades. MQO is originally proposed in the RDBMS context and existing works [12, 27, 54, 49, 51, 73, 14, 74] in the RDBMS context have already shown that substantial performance saving can be obtained by applying MQO techniques. For example, the experimental results from [74] indicate that their proposed MQO techniques can outperform the simplistic solution by up to 3 times.

In addition to the MQO techniques in the RDBMS context, there are also some preliminary studies [46, 44, 40] on the MQO techniques in the MapReduce context. The MapReduce framework, proposed by Google [15], has recently emerged as a new paradigm for large-scale data analysis and been widely embraced by Amazon, Google, Facebook, Yahoo!, and many other companies. There are two key reasons for its popular adoption. First, the framework can scale to thousands of commodity machines in a fault-tolerant manner and thus is able to use more machines to support parallel computing. Second, the framework has a simple yet expressive programming model through which users can parallelize their programs without being concerned about issues like fault-tolerance and execution strategy.

To simplify the expression of MapReduce programs, some high-level languages, such as Hive [58, 59], Pig [47, 26] and MRQL [20], have recently been proposed for the MapReduce framework. The declarative property of these languages also opens up new opportunities for automatic optimization in the framework [44, 18, 40]. Since different queries/jobs often perform similar work, there are many opportunities to exploit the shared processing among the queries/jobs to optimize performance. As noted and demonstrated by several works [46, 44], it is useful to apply the MQO techniques to optimize the processing of multiple queries/jobs by avoiding redundant computation in the MapReduce framework.

In summary, existing works have already shown that MQO techniques can significantly improve query/job performance in the contexts of both RDBMS and MapReduce framework. In this thesis, we study three novel MQO problems (one in RDBMS context and two in MapReduce context), namely, efficient processing of enumerative set-based queries, multi-query optimization in MapReduce framework and optimal join enumeration in MapReduce framework, and present novel MQO techniques for these problems.

While MQO techniques [12, 27, 54, 49, 51, 73, 14, 74] have been extensively studied in the RDBMS context, they mainly focus on optimizing a handful of SQL (join) queries. Our MQO problem in the RDBMS context is different from these works since we focus on optimizing a large collection (hundreds or thousands) of cross product queries produced by the applications of enumerative set-based queries. Furthermore, existing MQO techniques [44, 40] in the MapReduce framework are very limited and do not fully exploit the sharing opportunities among multiple queries/jobs. Thus, our two MQO problems in the MapReduce context present a more comprehensive study of MQO techniques to further exploit the sharing opportunities among multiple queries/jobs. In the following section, we describe the three MQO problems.

1.2 Research Problems

In this thesis, we study three novel MQO problems, namely, efficient processing of enumerative set-based queries, multi-query optimization in MapReduce framework and optimal join enumeration in MapReduce framework.

1.2.1 Efficient Processing of Enumerative Set-based Queries

Many applications, such as online shopping and recommender systems, often require finding sets of entities of interest that meet certain constraints [69, 39, 60, 29, 7, 70]. Such set-based queries (SQs) can be broadly classified into two types: *optimization SQs* that involve some optimization constraint and *enumerative SQs* that do not have any optimization constraint. For example, consider a relation *R(id,type,city,price,duration,rating)* shown in Table 1.1 that stores information about various places of interest (POI), where *type* refers to the category of the POI (e.g., museum, park), *duration* refers to the recommended duration to spend at the POI and *rating* refers to the average visitors' rating of the POI. Suppose that a tourist is interested to find all tour trips near Shanghai consisting of POIs that meet the following constraints: the trip must include both Shanghai (S.H.) and Suzhou (S.Z.) cities, the trip must include POIs of type museum and park, and the total duration of the trip should be between 6 and 10 hours. There are two packages that satisfy the above query: $\{t_1, t_2\}$ and $\{t_1, t_2, t_3\}$. The above is an example of an enumerative SQ to find all sets of POIs that satisfy the given constraints. If the query had an additional constraint to minimize the total cost of the tour package, it would become an optimization SQ.

Table 1.1: An example relation R

id	type	city	price	duration	rating
t_1	museum	S.H.	50	4	7
t_2	park	S.Z.	70	3	5
t_3	museum	H.Z.	60	3	8
t_4	shopping	S.H.	80	5	7

As another example, suppose that an employer is looking to hire a team of language translators for a project that meet the following constraints: each team member must know English; the team collectively must be knowledgeable in French, Russian, and Spanish; the team consists of at least two translators; and the total monthly salary of the team is no more than $\$50K$. Consider a relation *Translator(id,location,salary,english,french,russian,spanish)* that stores information about language translators available for hire, where the four binary valued attributes *english*, *french*, *russian*, and *spannish* indicate whether a translator is knowledgeable in the specific languages, *location* represents the translator's living place, and *salary* represents the translator's expected monthly salary. To browse through all the possible teams for hiring, the employer executes an enumerative SQ on the *Translator* relation.

Another application of enumerative SQs is in the area of set preference queries [17, 9, 71], which computes all sets of entities of interest that satisfy some preference function. Consider again our example on hiring translators. In addition to the previously discussed constraints, the employer could prefer to hire a team where (a) the team members are located close to one another and (b) their total salary is low. Thus, this set preference query is essentially a skyline set-query to retrieve non-dominated teams where the members have close proximity and low total salary. The most general approach to evaluate skyline set-queries is to first enumerate all the candidate sets followed by pruning away the dominated sets. Although there has been recent work to integrate these two steps [71], such optimization is applicable only for restricted cases (e.g., when the sets are of fixed cardinality and the preference function satisfies certain properties); and is not applicable for queries such as our example query. Therefore, efficient algorithms to evaluate enumerative SQs are essential for the efficient processing of set preference queries.

There has been much research on evaluating optimization SQs where the focus is on heuristic techniques to compute approximately optimal or incomplete query results (e.g., [29, 7, 60, 70, 69, 71, 39]). However, to the best of our knowledge, there has not been any prior work on the evaluation of enumerative SQs. Enumerative SQs are essentially a generalization of conventional selection queries to retrieve a collection of sets of tuples

(instead of a collection of tuples), and they represent the most fundamental fragment of set-based queries.

In this thesis, we address the problem of evaluating enumerative SQs using RDBMS. We present a novel approach to evaluate an enumerative SQ as a collection of cross-product queries (CPQs). However, applying existing multiple query optimization (MQO) techniques for this evaluation problem is not effective for two reasons. First, the scale of the problem could be very large involving hundreds of CPQ evaluations. Existing MQO heuristics, which are mainly designed for optimizing a handful of queries, are not scalable for our problem. Second, as the queries here are CPQs (and not join queries), existing MQO techniques, which are based on materializing and reusing the results of common subexpressions, is not effective as the cost of materialization exceeds the cost of recomputation. Thus, in this work, we study specialized MQO heuristics to optimize the evaluation of a collection of CPQs.

1.2.2 Multi-Query Optimization in MapReduce Framework

The MapReduce framework has recently emerged as a powerful parallel computation paradigm for large scale data analysis. The declarative property of the recently proposed high-level languages for the framework, such as Hive [58, 59] and Pig [47, 26], opens up new opportunities for automatic optimization in the framework [44, 18, 40]. Since different jobs (specified or translated from some high-level query languages) often perform similar work (e.g., jobs scanning the same input file or producing some shared map output), there are many opportunities to exploit the shared processing among the jobs to optimize performance.

The state-of-the-art work in this direction is MRShare [44], which proposed two sharing techniques for a batch of jobs. The *share map input scan* technique aims to share the scan of the input file among jobs, while the *share map output* technique aims to reduce the communication cost for map output tuples by generating only one copy of each shared map output tuple. The key idea behind MRShare is a *grouping technique* to merge multiple jobs that can benefit from the sharing opportunities into a single job.

While MRShare's grouping technique is able to share map input scan and map output for certain jobs, it has not fully exploited the sharing opportunities (i.e., share map input scan and map output techniques) among multiple jobs. For example, consider the two MapReduce jobs that are expressed in SQL queries over the relation $T(a, b, c)$ as follows:

J_1: **select** a, sum(c) **from** T **where** a $\leq$ 10 **group by** a
J_2: **select** a, b, sum(c) **from** T **where** a $\geq$ 5 **group by** a, b

MRShare's grouping technique can only share map input scan for the two jobs since it considers that the two jobs produce totally different map output that cannot be shared. However, the map output of J_2 for $5 \leq a \leq 10$ indeed can be reused to derive the partial map output of J_1. Thus, MRShare's grouping technique is very limited in exploiting the sharing opportunities among multiple jobs.

In this thesis, we present a more comprehensive study of multi-query/job optimization techniques to share map input scan and map output and algorithms to choose an evaluation plan for a batch of jobs in the MapReduce context.

1.2.3 Optimal Join Enumeration in MapReduce Framework

The MapReduce framework has been widely adopted by modern enterprises, such as Facebook [59], Greenplum [3] and Aster [2], to process complex analytical queries on large data warehouse systems due to its high scalability, fine-grained fault tolerance and easy programming model for large-scale data analysis. Given the long execution times for such complex queries, it makes sense to spend more time to optimize such queries to reduce the overall query processing time.

In this thesis, we examine the optimal join enumeration (OJE) problem, which is a fundamental query optimization task for SQL-like queries, in the MapReduce framework. Specifically, we study both the single-query and multi-query OJE (denoted as SOJE and MOJE respectively) problems where the study of the SOJE problem serves as a foundation for our study on the MOJE problem.

While the OJE problem has attracted much recent attention in the conventional RDBMS context [48, 41, 42, 16, 21, 24, 22, 23, 51, 14, 74], the solutions developed there are not applicable to the MapReduce context due to the differences in the query evaluation framework and algorithms.

There are two major differences between the OJE problem in MapReduce and that in RDBMS. First, both binary and multi-way joins are implemented in MapReduce while only binary joins are implemented in RDBMS. Specifically, given a join query, RDBMS will evaluate it as a sequence of binary joins while MapReduce will evaluate it as a sequence of

binary or multi-way joins. As a result, the SOJE problem in MapReduce has a larger join enumeration space than that in RDBMS due to presence of multi-way joins. While there has been much recent works in the RDBMS context on the study of the complexity [48] of the SOJE problem and its join enumeration algorithms [41, 42, 16, 21, 24, 22, 23], to the best of our knowledge, there has not been any prior work on the study of these problems in the presence of multi-way joins in the MapReduce context.

Second, intermediate results in MapReduce are always materialized instead of being pipelined/materialized as in RDBMS which simplifies the MOJE problem in MapReduce in two ways. First, the MOJE problem in RDBMS may incur deadlock due to the pipelining framework [14] while that in MapReduce does not have the deadlock problem due to the materialization framework. Second, materializing and reusing the results of the CSEs in RDBMS may incur additional materialization and reading cost due to the pipelining framework. However, since intermediate results are always materialized in the MapReduce framework, there is no additional overhead incurred with the materialization technique in MapReduce. Although the MOJE problem in RDBMS has been shown to be a very hard problem with a search space that is doubly exponential in the size of the queries [51, 14, 74], due to the simplification in MapReduce, we are able to propose efficient join enumeration algorithms for the MOJE problem in MapReduce based on our comprehensive study of the SOJE problem.

To the best of our knowledge, our work presents the first systematic study of the OJE problem in the MapReduce paradigm and proposes efficient join enumeration algorithms for the problem.

1.3 Thesis Contributions

In this thesis, we make the following contributions.

Efficient processing of enumerative set-based queries. In this work, we first present a baseline-SQL solution to evaluate enumerative SQs. While enumerative SQs can be expressed using SQL, our experimental results on PostgreSQL demonstrate that existing relational engines, unfortunately, are not able to efficiently optimize and evaluate such queries due to their complexity.

We then propose a novel two-phase evaluation approach for enumerative SQs. In the first phase, we partition the input table based on the different combinations of constraints

satisfied by the tuples. In the second phase, we compute the answer sets by appropriate combinations of the partitions which essentially are a collection of cross-product queries (CPQs). To efficiently evaluate a collection of CPQs, we propose novel MQO techniques which works for both in-memory and large disk-based data.

Finally, we implemented our approach on PostgreSQL 8.4.4 and conducted a comprehensive experimental study to show the efficiency of our approach. Our experimental results demonstrate that our proposed approach is significantly more efficient than conventional RDBMS methods by up to three orders of magnitude.

Multi-query optimization in MapReduce framework. In this work, we first present two new multi-job optimization techniques. The first technique is a *generalized grouping technique (GGT)* that relaxes MRShare's requirement for sharing map output. The second technique is a *materialization technique (MT)* that partially materializes the map output of jobs (in the map and/or reduce phase) which provides another alternative means for jobs to share both map input scan and map output.

We then propose a novel two-phase optimization algorithm to choose an evaluation plan for a batch of jobs. In the first phase, we choose the map output key for each job to maximize the sharing. In the second phase, we partition the batch of jobs into multiple groups and choose the processing technique for each group to minimize the evaluation cost.

Finally, we conducted a comprehensive performance evaluation of the multi-job optimization techniques using Hadoop. Our experimental results show that our proposed techniques are scalable for a large number of queries and significantly outperform MRShare's techniques by up to 107%.

This work has been published in VLDB 2014 [65].

Optimal join enumeration in MapReduce framework. In this work, we first present a comprehensive study of the SOJE problem which serves as a foundation for our study on the MOJE problem. Specifically, we first study the complexity of the SOJE problem in the MapReduce framework in the presence of multi-way joins for chain, cycle, star and clique queries. We then propose both bottom-up and top-down join enumeration algorithms for the SOJE problem with an optimal complexity w.r.t. the query graph based on a proposal of an efficient and easy-to-implement plan enumeration algorithm.

We then propose an efficient multi-query join enumeration algorithm for the MOJE problem. The main idea is to first apply the single-query join enumeration algorithm for each query to generate all the interesting plans and then stitch the interesting plans for the queries into a global optimal plan. A query plan is interesting if it is either the optimal plan or produces some output that can be reused for other queries.

Finally, we conducted a comprehensive experimental study to demonstrate the efficiency of our proposed algorithms. Our experimental results show that our proposed single query join enumeration algorithm significantly outperforms the baseline algorithms by up to 473%, and our proposed multi-query join enumeration algorithm is able to scale up to 25 queries where the number of relations in the queries ranges from 1 to 10.

1.4 Thesis Organization

The rest of the thesis is structured as follows.

- Chapter 2 presents a comprehensive literature review of the three problems that we have studied.
- Chapter 3 studies the evaluation problem for enumerative SQs and proposes efficient evaluation techniques for enumerative SQs.
- Chapter 4 studies the multi-query/job optimization problem and proposes efficient and effective multi-job optimization techniques and algorithms in the MapReduce framework.
- Chapter 5 studies the OJE problem and proposes efficient join enumeration algorithms for the problem in the MapReduce context.
- Chapter 6 concludes our thesis and points out some directions for future work.

CHAPTER 2

RELATED WORK

In this chapter, we present a comprehensive literature review of studies related to the three works we have done. Accordingly, this review is classified in terms of the three works we have done. Specifically, Section 2.1 presents the background of MapReduce framework. Section 2.2 presents the related work of our work on efficient processing of enumerative set-based queries. Section 2.3 presents the related work of our work on multi-query optimization in MapReduce framework. Section 2.4 presents the related work of our work on optimal join enumeration in MapReduce framework.

2.1 Preliminaries on MapReduce

MapReduce, proposed by Google [15], has emerged as a new paradigm for parallel computation due to its high scalability, fine-grained fault tolerance and easy programming model. Since its emergence, it has been widely embraced by enterprises to process complex large-scale data analysis such as online analytical processing, data mining and machine learning.

MapReduce adopts a master/slave architecture where a master node manages and monitors map/reduce tasks and slave nodes [1] process map/reduce tasks assigned by the master node, and uses a distributed file system (DFS) to manage the input and output files. The input files are partitioned into fix-sized splits when they are first loaded into the DFS. Each split is processed by a map task and thus the number of map tasks for a job is equal to the number of its input splits. Therefore, the number of map tasks for a job is determined by the input file size and split size. However, the number of reduce tasks for a job is a configurable parameter.

A job is specified by a pair of map and reduce functions, and its execution consists of a map phase and a reduce phase. In the map phase, each map task first parses its corresponding input split into a set of input key-value pairs. Then it applies the map function on each input key-value pair and produces a set of intermediate key-value pairs which are sorted and partitioned into r partitions, where r is the number of configured reduce tasks. Note that both the sorting and partitioning functions are customizable. An optional combine function can be applied on the intermediate map output to reduce its size and hence the communication cost to transfer the map output to the reducers. In the reduce phase, each reduce task first gets its corresponding map output partitions from the map tasks and merges them. Then for each key, the reducer applies the reduce function on the values associated with that key and outputs a set of final key-value pairs.

MapReduce uses job schedulers to manage all submitted jobs. The default job scheduler in Hadoop [2] is FIFO which maintains a job queue for all submitted jobs according to their submission times and priorities. FIFO allows a job to take all the slots within the cluster and picks the first pending job for execution when there are available slots or a job releases its slots. Other alternative schedulers include Yahoo!'s capacity scheduler and Facebook's fair scheduler [36]. The main idea of these schedulers is to maintain multiple job queues for submitted jobs (one for each user or each organization) and allocate certain resources for each queue. The main advantage of these schedulers is to allow jobs belonging to different users or organizations to be concurrently executed. Among all the schedulers, FIFO has been shown to have the minimum batch response time [36], and thus is used as the job scheduler for our experiments in Chapter 4.

[1] Each slave node has fixed number of map/reduce slots which are configurable parameters

[2] We use Hadoop's scheduler as a representative of MapReduce scheduling mechanisms

2.2 Efficient Processing of Enumerative Set-based Queries

To the best of our knowledge, this is the first work that addresses the problem of efficient evaluation of enumerative set-based queries. We present a novel approach to evaluate enumerative set-based queries as a collection of cross product queries (CPQs) and propose novel MQO techniques to optimize the evaluation of a collection of CPQs. As a result, there are two main areas related to this work: set-based queries (SQs) and multi-query optimization (MQO). In the following, we separately discuss them and position our work.

Set-based queries. Set-based queries aim to find sets of entities of interest to meet certain constraints. There are several works on evaluation of set-based queries: OPAC queries for business optimization problems [29], composite items construction in online shopping applications [7], composite recommendation in recommender systems [70, 69], team formation in social networks [39], set-based preference queries [71] and set-based queries with aggregation constraints [60]. However, the focus of all these works is on optimization SQs whereas our focus is on enumerative SQs. Moreover, as most of these works deal with NP-hard optimization problems, their algorithms are mostly approximate or produce incomplete solutions; in contrast, our algorithm is exact and complete. Finally, our work is focused on optimizing query evaluation at the database engine level, whereas these works is focused on middleware-level solution with mostly main-memory resident data.

Multi-query optimization (MQO). MQO aims to find evaluation plans that share computation of common subexpressions (CSEs) for a batch of queries. Most of existing works [31, 27, 13, 12, 53, 49, 51, 54, 57, 74] focus on materializing and reusing the results of CSEs. The works in [49, 54] describe exhaustive search algorithms and heuristic search pruning techniques to find a global optimal query plan by searching all the plan space. However, the exhaustive search of the plan space incurs high optimization overhead which make these works impractical. To reduce the high optimization cost, the works in [51, 74] propose several cost-based greedy heuristics to find a global query plan. However, all these works are not useful for our context since materializing and reusing the results of CPQs is extremely costly. Thus, our approach for evaluating CPQs does not employ the materialization technique; instead, we evaluate them by pipelining the results of CSEs to CPQs.

There are several works [14, 73] that exploit pipelining for MQO. The work in [73] considers specialized MQO techniques to pipeline the results of CSEs for OLAP queries. Their work addresses star join queries where all the dimension tables are assumed to be main-memory resident (i.e., only the fact table is disk-based). In contrast, our MQO techniques

are proposed for general CPQs without any strong assumption about the main-memory residency of the relations.

The work in [14] addresses the MQO problem with pipelining and follows a two-phase optimization strategy which is different from our proposed two-phase approach. The first phase uses existing techniques (such as [51, 74]) to generate a global plan for a set of queries which is represented as a plan-DAG. All the CSEs that can benefit from materialization are captured by the plan-DAG. The second phase optimizes the plan-DAG by pipelining the results of some CSEs in the plan-DAG. Thus, only the results of CSEs that can benefit from materialization are considered for pipelining. This simplification is restrictive since the results of a CSE could be pipelined to improve performance even if materializing and reusing the results of that CSE does not improve performance. Since our work does not materialize the results of any CSEs, their work is not applicable for our context. Furthermore, their work assumes that the pipelined relations/results are not buffered whereas our work focus on efficiently optimizing the buffer allocation for pipelining.

2.3 Multi-Query Optimization in MapReduce Framework

This work presents a more comprehensive study of multi-query/job optimization techniques and algorithms in MapReduce framework. We broadly classify its related work into three categories: job optimization, query optimization and multi-query optimization. In the following, we separately discussed them and position our work.

Job optimization. There are several works [37, 32, 33] on optimizing general MapReduce jobs that are expressed as programs. The work in [37] proposes a system to automatically analyse, optimize and execute MapReduce programs. It works by first analysing the programs to detect optimization opportunities, then applying the detected optimizations such as index selection and data compression to the programs and finally executing the optimized programs. The work in [32, 33] discusses the optimization opportunities presented by the large space of MapReduce configuration parameters such as number of map and reduce tasks, and proposes a cost-based optimizer to choose the best configuration parameters for MapReduce programs. It works by first collecting the profiles through dynamic instrumentation and then estimating the cost through a detailed set of analytical models using the collected profiles. Different from these works where the emphasis is on optimizing single MapReduce program, our work focuses on optimizing multiple jobs specified

in or translated from some high-level query language such that the sharing among the jobs can easily be detected.

Query optimization. The proposal of high-level declarative query languages for MapReduce such as Hive [58, 59], Pig [47, 26] and MRQL [20], opens up new opportunities for query optimization in the framework. As a result, there has been some recent works on query optimization in MapReduce framework similar to query optimization in RDBMS. These works include optimization strategies for Pig [46], multi-way join optimization in MapReduce [5, 72, 30], optimization techniques for Hive [68, 28], algebraic optimization for MRQL [20], theta join processing in MapReduce [45], set similarity join processing in MapReduce [63], and query optimization using materialized results [18]. All these works focus on query optimization techniques for a single query; in contrast, our work focuses on optimizing multiple jobs specified in or translated from some high-level query language.

The work in [18] presents a system ReStore to optimize query evaluation using materialized results. Given a space budget for storing materialized results, ReStore uses heuristics to both decide whether to materialize the complete map and/or reduce output of each job being processed as well as choose which previously materialized results to be evicted if the space budget is exceeded. Our work differs from ReStore in both the problem focus and the developed techniques. The results materialized by our MT technique for a given job could be the partial map output of another job; in contrast, ReStore materializes the complete output of the job being processed. Moreover, whereas the materialized output produced by ReStore might not be reused at all due to the unknown query workload, this is not the case for our context as the query workload is known and our techniques only materialize output that will be reused.

Multi-Query optimization. There are several works on multi-query optimization [44, 40]. The work that is the most closely related to ours is MRShare [44]. Compared with MRShare, our work is more comprehensive with additional optimization techniques (i.e., GGT and MT) which leads to a more complex optimization problem (e.g., the ordering of the map output key of each job becomes important) and a novel cost-based, two-phase approach to find optimal evaluation plans. In MRShare, an input batch of jobs is partitioned based on the following heuristic: the jobs are first sorted in non-descending order of their map output size, and a dynamic-programming based algorithm is used to find an optimal partitioning of the ordered jobs into disjoint consecutive groups. Thus, an optimal job partitioning where the jobs in a group are not consecutively ordered would not be produced by MRShare's heuristic. Note that our partitioning heuristic (with a time-complexity of

$O(n^2)$) does not have this drawback and is more efficient than MRShare's partitioning heuristic ($O(n^3)$ time-complexity).

The work in [40] proposes a transformation-based optimizer for MapReduce workflows (translated from queries). The work considers two key optimization techniques: vertical (horizontal, resp.) packing techniques aim to optimize jobs with (without resp.) producer-consumer relationships; the horizontal packing techniques are based on MRShare's grouping technique. In contrast, our work does not specifically consider MapReduce workflow jobs that have explicit producer-consumer relationships; therefore, their proposed vertical packing techniques are not applicable for our work.

2.4 Optimal Join Enumeration in MapReduce Framework

This work studies the optimal join enumeration (OJE) problem in MapReduce framework. While the OJE problem has attracted much recent attention in the conventional RDBMS context [48, 41, 42, 16, 21, 24, 22, 23, 51, 14, 74], the solutions developed there are not applicable to the MapReduce context due to the differences in the query evaluation framework and algorithms as discussed in Section 1.2.3. In this work, we study both the single-query and multi-query OJE (denoted as SOJE and MOJE respectively) problems as well as their join enumeration algorithms in the MapReduce context. As a result, we broadly classify and discuss its related work in terms of SOJE and MOJE.

SOJE. The SOJE problem is a fundamental query optimization task in RBDMS. A well known join enumeration algorithm for the SOJE problem is dynamic programming which is divided into two categories, i.e., bottom-up enumeration [52, 41] and top-down enumeration [16, 21, 24, 22]. Both approaches have to consider the same enumeration space and neither of them is strictly better than the other. The work in [48] shows that the (optimal) complexity of the SOJE problem depends on the query graph and analyses the (optimal) complexity for chain, cycle, star and clique queries in RDBMS. The work in [41] first shows that the complexity of existing two state-of-the-art dynamic programming algorithms [52, 62] in RDBMS are far from optimal w.r.t. the query graph, and proposes bottom-up dynamic programming algorithms with an optimal complexity. Note that our proposed baseline join enumeration algorithms in MapReduce are adapted from the two state-of-the-art algorithms [52, 62] in RDBMS and thus have a non-optimal time complexity. In addition to the bottom-up dynamic programming algorithms, these works in [16, 21, 24, 22] propose top-down dynamic programming algorithms with an

optimal complexity. However, all these dynamic programming algorithms with an optimal complexity are restricted to binary joins and thus are not applicable in the presence of multi-way joins in the MapReduce context. In addition to the above works, there are also several works [42, 23] on join enumeration algorithms for queries with more complex join predicates such as $R_1.a = R_2.b + R_3.c$ (i.e., their query graphs are hypergraphs). In our work, we do not consider these complex join predicates and leave them as part of our future work.

The MapReduce framework [15] has recently been widely used to process complex analytical queries on large data warehouse systems. As a result, various MapReduce versions of algorithms have been proposed for database operators (e.g., join and aggregation) [10, 5, 45, 72, 30]. In particular, these works in [5, 72, 30] study efficient multi-way join algorithms in MapReduce. Their experimental results show that the performance of multi-way joins and that of a sequence of binary joins can outperform each other in different settings which thus increases the join enumeration space for the SOJE problem in MapReduce. To the best of our knowledge, our work is the first to study the SOJE problem in the MapReduce context. The most related work is a proposal of a greedy heuristic to find a good join order in MapReduce [68].

MOJE. The MOJE problem aims to find global optimal evaluation plans that share CSEs and has been shown to be a very hard problem with a search space that is doubly exponential in the size of the queries [54, 49, 51, 14, 74] in RDBMS. This is due to the pipelining/materialization framework in RDBMS which complicates its MOJE problem as discussed in Section 1.2.3. As MapReduce always materializes intermediate results, the MOJE problem in MapReduce becomes simpler which presents us an opportunity to design an efficient and optimal multi-query join enumeration algorithm. Note that there are also some early works in RDBMS [54, 49] that propose optimal join enumeration algorithms for the MOJE problem using only materialization. However, they simply consider all the plans for each query and stitch them into a global optimal plan which has been demonstrated to be an impractical approach [51, 74]. Our work proposes effective pruning techniques to prune away non-promising plans early and thus reduce the plan combination space for the MOJE problem.

In addition to the above works, there are also several works [18, 44, 40] including our work on multi-query optimization in MapReduce framework on optimizing multiple jobs specified in or translated from some high-level SQL-query language. Our work are orthogonal with these works since our work focuses on optimizing the translation from

queries into jobs (i.e., finding an optimal join plan) while these works focus on optimizing the translated jobs.

CHAPTER 3

EFFICIENT PROCESSING OF ENUMERATIVE SET-BASED QUERIES

3.1 Overview

In this chapter, we study efficient evaluation techniques using RDBMS for enumerative SQs which aim to find a collection of tuples sets that satisfy certain constraints. To the best of our knowledge, there has not been any prior work on the evaluation of enumerative SQs. For convenience, we refer to enumerative SQs as simply SQs in the rest of this chapter.

While SQs can be expressed using SQL, existing relational engines, unfortunately, are not able to efficiently optimize and evaluate such queries due to their complexity involving multiple self joins and/or view expressions. In this chapter, we propose a novel evaluation approach for SQs which works for both in-memory and large, disk-based data. The key idea is to first partition the input relation based on the different combinations of constraints satisfied by the tuples and then compute the answer sets by appropriate combinations of the partitions. In this way, a SQ is evaluated as a collection of cross-product queries (CPQs). However, applying existing MQO techniques for this evaluation problem is not effective for two reasons. First, the scale of the problem could be very large involving

hundreds of CPQ evaluations. Existing MQO heuristics, which are mainly designed for optimizing a handful of queries, are not scalable for our problem. Second, as the queries here are CPQs (and not join queries), existing MQO techniques, which are based on materializing and reusing the results of the CSEs, are not effective as the cost of materialization exceeds the cost of recomputation.

Thus, in this chapter, we propose specialized MQO techniques to optimize the evaluation of a large collection of CPQs. To copy with the high optimization cost, we adapt a well-known two phase approach [73, 57]. The first phase generates local optimal plans for each CPQ by specifying an ordering of the partitions in the CPQ. The second phase uses a trie structure to capture all the CSEs of the CPQs. In this way, our MQO heuristics are able to scale to a large number of CPQs. We further optimize our evaluation approach by exploiting the properties of set predicates in the SQs. We demonstrate the effectiveness of our approach with a comprehensive experimental evaluation on PostgreSQL which shows that our approach outperforms the conventional SQL-based solution by up to three orders of magnitude.

The rest of this chapter is organized as follows. In Section 3.2, we formally introduce set-based queries (SQs) and a fragment of SQs referred to as basic SQs (BSQs). Section 3.3 presents some preliminaries. Section 3.4 presents a baseline SQL-based solution to evaluate SQs. Section 3.5 presents our main-memory based approach to evaluate BSQs, and Section 3.6 extends the approach to evaluate BSQs on disk-based data. In Section 3.7, we extend our approach to evaluate general SQs beyond BSQs. Section 3.8 presents an experimental performance evaluation of the proposed techniques, and we conclude this chapter in Section 3.9.

3.2 Set-based Queries

In the simplest form, a *set-based query (SQ)* Q is defined by an input relation R, which represents a collection of entities of interest, and an input set of predicates P on R. The query's result is a collection of all the subsets of R such that each subset satisfies the predicates in P.

For convenience, we introduce an extended SQL syntax to express SQs more explicitly. The example SQ in Section 1.2.1 can be expressed by the following extended SQL query.

Q_{ext}: SELECT *
FROM SET(R) S
WHERE v_1 in S AND v_2 in S
AND v_3 in S AND v_4 in S
AND v_1.city = S.H. AND v_2.city = S.Z.
AND v_3.type = museum AND v_4.type = park
AND $6 \leq$ SUM(S.duration) ≤ 10

The "SET(R) S" in the from-clause specifies S as a *set variable* whose value is a subset of tuples in relation R. Each of the predicates of the form "v_i in S" specifies v_i as a *member variable* representing a member of the set variable S. Note that the values of member variables are not necessarily distinct. Each of the next four predicates specifies a constraint on an individual member; and the last predicate specifies an aggregation constraint on the set. The output schema of this query consists of all the attributes in relation R and an additional, implicit integer attribute named sid that represents the identifier for an answer set. The values of sid are generated automatically by the database system. The attributes (sid, id) form the key of the output schema where id is the key of input relation R. Thus, each answer set to the query is represented by a collection of output tuples having the same sid value. Table 3.1 shows the output of the example SQ Q_{ext} on the input relation R in Table 1.1 in Section 1.2.1.

Table 3.1: Output of the example SQ

sid	id	type	city	price	duration	rating
1	t_1	museum	S.H.	50	4	7
1	t_2	park	S.Z.	70	3	5
2	t_1	museum	S.H.	50	4	7
2	t_2	park	S.Z.	70	3	5
2	t_3	museum	H.Z.	60	3	8

As the values of member variables are not necessarily distinct, the maximum cardinality of an answer set is bounded either implicitly by the number of member variables in the query (as shown by the example query) or explicitly by a constraint on the set's cardinality (e.g., "COUNT(S) ≤ 3").

There are two types of selection predicates in a SQ. A *member predicate* specifies a constraint on exactly one member variable (e.g., "v_1.city = S.H."). A *set predicate* specifies a constraint on a set variable or more than one member variable; examples include "SUM(S.duration) ≤ 10" and "v_1.price + v_3.price ≤ 100".

Given a set predicate p, it is classified as *anti-monotone* if whenever a set S does not satisfy p, then any superset of S also does not satisfy p; it is classified as *monotone* if whenever a set S satisfies p, then any superset of S also satisfies p. In our example SQ Q_{ext}, the predicate "SUM(S.duration) $\leq$ 10" is an anti-monotone set predicate, while the predicate "SUM(S.duration) $\geq$ 6" is a monotone set predicate. An example of a set predicate that is neither monotone nor anti-monotone is "AVG(S.price) $\leq$ 20". Note that set predicates can also involve other SQL constructs such as *groupby-clause* and *having-clause* which we omit in this chapter.

Since the number of qualifying answer sets could be very large for some SQs, there are two natural ways to limit the size of the query result. The first approach is to retrieve only some fixed number of say k result sets either using a limit clause to retrieve any k sets or via a ranking function to retrieve the top-k sets. The second approach is to retrieve only *minimal sets* that satisfy the query's predicates. A set S is defined to be minimal if no proper non-empty subset of S also satisfies the predicates in P. For example, the answer set $\{t_1, t_2, t_3\}$ for the example SQ Q_{ext} is not minimal since its subset $\{t_1, t_2\}$ also satisfies the query's predicates. Minimal answer sets are interesting as they could save the budgets (e.g., money and time) for users while still guarantee the satisfaction of the query's predicates. They are also of interest on their own as they serve as a concise representation of all the answer sets (i.e., any superset of a minimal answer set is also an answer set) if all the set predicates in the query are monotone. The minimal set constraint can be expressed in our extended SQL syntax by replacing "SET(R) S" by "MINSET(R) S" to indicate that S is a *minimal set variable*.

To simplify the presentation of evaluation algorithms for SQs, we introduce a special fragment of SQs called *basic SQs*. A SQ Q is defined to be a basic SQ (BSQ) if Q retrieves only minimal sets and all the set predicates in Q are anti-monotone. Note that for a BSQ, if a tuple in R does not satisfy any member predicate, then it will not contribute to any answer set and can simply be removed from R.

We should emphasize that the focus of this chapter is not on the design of SQL extensions but on efficient query evaluation. The above example is meant to illustrate how the semantics of SQs can be expressed more explicitly and easily using some SQL extensions instead of using conventional SQL, which we will discuss in Section 3.4.

3.3 Preliminaries

In this chapter, we consider a SQ Q defined over a relation R, where there are n member variables in Q. Thus, the maximum cardinality of the answer sets for Q is n.

Let $V = \{v_1, \cdots, v_n\}$ denote the set of member variables in Q. The predicates P in Q can be partitioned into $n+1$ subsets, P_0, P_1, $\cdots$, P_n, where each P_i, $i \in [1, n]$, denote the set of member predicates in Q that involves the member variable v_i; and P_0 denote the set of set predicates in Q.

In this chapter, we refer to a set S as a *k-set* to mean that the cardinality of S is k. Thus, each answer set for Q is an i-set, where $i \in [1, n]$.

Example 3.1: In our example SQ Q_{ext}, there are four member variables (i.e., v_1, v_2, v_3 and v_4). Therefore, the predicates can be partitioned into five subsets: $P_0 = \{6 \leq SUM(S.duration) \leq 10\}$, $P_1 = \{v_1.city = S.H.\}$, $P_2 = \{v_2.city = S.Z.\}$, $P_3 = \{v_3.type = museum\}$ and $P_4 = \{v_4.type = park\}$. □

3.4 Baseline Solution using SQL

In this section, we first outline a baseline approach to evaluate SQs using conventional SQL in Section 3.4.1. We then illustrate the baseline solution using our example SQ Q_{ext} in Section 3.4.2 by showing the detail SQL queries.

3.4.1 Baseline Solution

In this approach, answer sets are generated iteratively, i.e., answer i-sets are computed before answer $(i+1)$-sets, which is similar to the Apriori-style of using SQL to compute frequent itemsets [34]. Let C_i denote the collection of candidate answer i-sets that satisfy all the anti-monotone set predicates in P_0, and $A_i \subseteq C_i$ denote the collection of answer i-sets. Each C_i/A_i is represented by a relation/view where each tuple in C_i/A_i represents a subset of i tuples from R. Each C_i, $i \geq 2$, is computed using a self-join of C_i and each A_i is derived from C_i. In this approach, the answer sets for a SQ are given by multiple output tables $A_1, \cdots, A_n$, where each tuple in each A_i presents an answer i-set for Q.

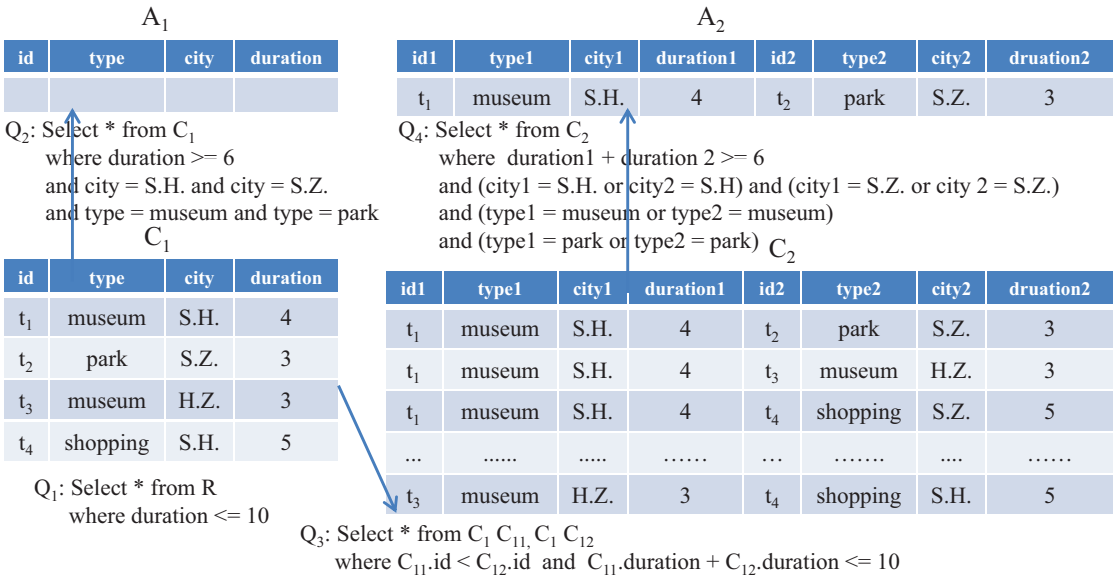


Figure 3.1: Illustration of the first two iterations of the baseline SQL-based solution

In the first iteration, C_1 is the subset of tuples in R that satisfy all the anti-monotone set predicates in P_0. A_1 is the subset of tuples in C_1 that satisfy all the predicates in Q. In the i^{th} iteration, $i > 1$, C_i is computed by a self join of C_{i-1} to ensure two requirements. First, C_i does not contain duplicate candidate answer i-sets[1]. Second, each tuple in C_i satisfies all the anti-monotone set predicates in P_0. A_i is derived from C_i by appropriate selection predicates to ensure that each tuple in A_i must satisfy all the predicates in Q. Thus, this approach is implemented as a sequence of SQL queries where the number of queries is a linear function of n.

Example 3.2: Figure 3.1 illustrates the first two iterations of the baseline approach for evaluating our example SQ Q_{ext} on the input relation R in Table 1.1 (more details are shown in Section 3.4.2). To avoid clutter, the non-relevant attributes (i.e., $price$ and $rating$) are omitted from the figure. In the first iteration, C_1 is computed by Q_1 on R to ensure that each tuple in C_1 (representing a candidate answer 1-set) satisfies all the anti-monotone set predicates. The answer 1-sets are given by A_1 which is computed by Q_2 on C_1; A_1 is empty since there is no answer 1-set for this SQ. In the second iteration, C_2 is computed by Q_3 with a self-join on C_1 and A_2 is computed from C_2 using Q_4. Observe that A_2 contains one answer 2-set $\{t_1, t_2\}$. Since the answer sets for this query has a maximum cardinality of four, this process continues for two additional iterations to find answer 3- and 4-sets (details not shown). □

[1]Following the same principle to avoid duplicates in [34], the self-join of C_{i-1} to compute C_i has $(i-2)$ equi-join predicates requiring that two matching tuples in C_{i-1} (representing two $(i-1)$-sets) have $(i-2)$ identical tuples.

Minimal set constraint. If the query requires only minimal answer sets, then the above approach still works with the following two extensions. First, to generate C_i (representing candidate answer i-sets), the self join is performed on $C_{i-1} \setminus A_{i-1}$ instead of C_{i-1} as all the supersets of answer $(i-1)$-sets in A_{i-1} are not minimal. Second, for each tuple in A_i, in addition to satisfying all the predicates in Q, it must also represent a minimal set. To verify the minimality of a candidate answer i-set $S \in C_i$, all the subsets of S have to be examined to ensure that they do not satisfy all the predicates in Q. However, if P_0 contains only anti-monotone and monotone set predicates, then only subsets with a cardinality of $(i-1)$ need to be examined.

Alternative SQL-based approach for BSQs. For BSQs, there is an alternative SQL-based approach that generates all the answer sets in a single output table with arity equal to the maximum cardinality of the answer sets given by n. This approach consists of two main steps. The first step generates all the candidate answer sets in a relation/view M by computing the cartesian product of n views $M_1, \cdots, M_n$, where each M_i is the set of tuples in R that satisfies P_i. Note that M may contain multiple tuples that represent the same candidate answer set since each tuple in R may appear in multiple M_i's. Therefore, we need to remove the duplicate candidate answer sets from M. The second step computes the answer sets by eliminating those candidate answer sets in M that are duplicates, do not satisfy P_0, or are not minimal. The details of this approach are given in Section 3.4.2.

It is important to note that this alternative approach is not applicable for evaluating SQs since a tuple from R can contribute to an answer set even if it does not appear in any M_i $(1 \leq i \leq n)$. For evaluating BSQs, our experimental results show that the alternative approach is significantly outperformed by the first discussed approach. The main reason is due to the complex SQL queries used to remove duplicate and non-minimal candidate answer sets in the second step. Given its limited applicability and poor performance, we will not consider the alternative approach any further in this chapter.

3.4.2 Detail Illustration of Baseline Solution

In this section, we illustrate the baseline solution for evaluating SQs using our example SQ Q_{ext} and BSQs using the BSQ Q_{der} that is derived from the SQ Q_{ext} by removing its non-anti-monotone set predicate (i.e., $SUM(S.duration) \geq 6$).

Baseline solution to evaluate the SQ Q_{ext}. Figure 3.2 shows the SQL queries to evaluate our example SQ Q_{ext}. To simplify the predicates as well as the minimality checking, we

create view C_1(id,duration,p1,p2,p3,p4) **as select** id, duration,
case city = **S**.H. **then** 1 **else** 0 **as** p1, **case** city = **S**.Z. **then** 1 **else** 0 **as** p2,
case type = museum **then** 1 **else** 0 **as** p3, **case** type = park **then** 1 **else** 0 **as** p4 **from** R **where** duration <= 10

create view A_1 **as select** * from C_1 **where** p1 = 1 **and** p2 = 1 **and** p3 = 1 **and** p4 = 1 **and** duration >= 6

create view C_2(id1,duration1,p11,p12,p13,p14,id2,duration2,p21,p22,p23,p24)
as select * **from** C_1 C_{11}, C_1 C_{12} **where** C_{11}.id < C_{12}.id **and** C_{11}.duration + C12.duration <= 10

create view A_2 **as select** * **from** C_2
where p11 + p21 > 0 **and** p12 + p22 > 0 **and** p13 + p23 > 0 **and** p14 + p24 > 0 **and** duration1 + duration 2 >= 6

create view C_3(id1,duration1,p11,p12,p13,p14,id2,duration2,p21,p22,p23,p24, id3,duration3,p31,p32,p33,p34) **as**
select C_{21}.*, C_{22}.id2* **from** C_2 C_{21}, C_2 C_{22}
where C_{21}.id1 = C_{22}.id1 **and** C_{21}.id2 < C_{22}.id2 **and** C_{21}.duration1 + C_{21}.duration2 + C_{22}.duration2 <= 10

create view A_3 **as select** * **from** C_3 **where** p11 + p21 + p31 > 0 **and** p12 + p22 + p32 > 0 **and**
p13 + p23 + p33 > 0 **and** p14 + p24 + p34 > 0 **and** duration1 + duration 2 + duration3 >= 6

create viewC_4(id1,duration1,p11,p12,p13,p14,id2,duration2,p21,p22,p23,p24,id3,duration3,p31,p32,p33,p34,
id4,duration4,p41,p42,p43,p44) **as select** C_{31}.*, C_{32}.id3* **from** C_3 C_{31}, C_3 C_{32} **where** C_{31}.id1 = C_{32}.id1 **and**
C_{31}.id2 = C_{32}.id2 **and** C_{31}.id3 < C_{32}.id3 **and** C_{31}.duration1 + C_{31}.duration2 + C_{31}.duration3 + C_{32}.duration3 <= 10

create view A_4 **as select** * **from** C_4 **where** p11 + p21 + p31 + p41 > 0 **and** p12 + p22 + p32 + p42 > 0 **and**
p13 + p23 + p33 + p43 > 0 **and** p14 + p24 + p34 + p44 > 0 **and** duration1 + duration 2 + duration3 + duration4 >= 6

Figure 3.2: SQL queries to evaluate our example SQ Q_{ext}

create C_1 to represent the information of POIs that satisfy the anti-monotone set predicate (i.e., $SUM(S.duration) \leq 10$). Each tuple in C_1 represents the information for a POI. Each of the four binary valued attributes p_i $(1 \leq i \leq 4)$ indicates whether a POI satisfies P_i, where a value of 1 indicates that the POI satisfies P_i. Note that in Figure 3.2, to simplify the expression of SQL queries, in the select-clause, $C_i.*$ represents that we retrieve all the attributes in C_i and $C_i.j*$ represents that we retrieve all the attributes from the j^{th} tuple in C_i.

Baseline solution to evaluate the BSQ Q_{der}. Recall that there are two SQL-based approaches to evaluate BSQs. Figure 3.3 shows the SQL queries to evaluate the BSQ Q_{der} that generate answer sets in multiple output tables. In Figure 3.3, we use B_i to denote $C_i \setminus A_i$. Note that for BSQ, C_{i+1} is derived from B_i instead of C_i. In the view A_3, the first four conditions ensure that each answer set in A_3 satisfies all the predicates in Q_{der} and the remaining conditions ensure that each answer set in A_3 is minimal, i.e., for each member in the answer set, there must exist some P_i $(1 \leq i \leq 4)$ that is satisfied by only this member in the answer set.

Figure 3.4 shows the SQL queries to evaluate the BSQ Q_{der} that generate all the answer sets in a single output table whose arity is equal to the maximum cardinality of the answer sets given by n. To avoid clutter, we only keep the key attribute *id*. In this approach, since a tuple may satisfy multiple member predicates, the same tuple may appear multiple

create view C_1(id,duration,p1,p2,p3,p4) **as select** id, duration, **case** city = **S**.H. **then** 1 **else** 0 **as** p1,
case city = **S**.Z. **then** 1 **else** 0 **as** p2, **case** type = museum **then** 1 **else** 0 **as** p3,
case type = park **then** 1 **else** 0 **as** p4 **from** R **where** duration <= 10 and p1 + p2 + p3 + p4 > **0**

create view A_1 **as select** * **from** C_1 **where** p1 = 1 **and** p2 = 1 **and** p3 = 1 **and** p4 = 1

create view B_1 **as select** * **from** C_1 **except select** * **from** A_1

create view C_2(id1,duration1,p11,p12,p13,p14,id2,duration2,p21,p22,p23,p24) **as**
select * **from** B_1 B_{11}, B_1 B_{12} **where** B_{11}.id < B_{12}.id **and** (B_{11}.duration + B_{12}.duration) <= 10

create view A_2 **as select** * **from** C_2 **where** p11 + p21 > 0 **and** p12 + p22 > 0 **and** p13 + p23 > 0 **and** p14 + p24 > 0

create view B_2 **as select** * **from** C_2 **except select** * **from** A_2

create view C_3(id1,duration1,p11,p12,p13,p14,,id2,duration2,p21,p22,p23,p24, id3,duration3,p31,p32,p33,p34) **as**
select B_{21}.*, B_{22}.id2* **from** B_2 B_{21}, B_2 B_{22}
where B_{21}.id1 = B_{22}.id1 and B_{21}.id2 < B_{22}.id2 **and** (B_{21}.duration1 + B_{21}.duration2 + B_{22}.duration2) <= 10

create view A_3 **as select** * **from** C_3 **where** p11 + p21 + p31 > 0 **and** p12 + p22 + p32 > 0 **and** p13 + p23 + p33 > 0
and p14 + p24 + p34 > 0 **and** ((p11 = 1 **and** p21 + p31 = 0) **or** (p12 = 1 **and** p22 + p32 = 0) **or** (p13 = 1 **and**
p23 + p33 = 0) **or** (p14 = 1 **and** p24 + p34 = 0)) **and** ((p21 = 1 **and** p11 + p31 = 0) **or** (p22 = 1 **and** p12 + p32 = 0)
or (p23 = 1 **and** p13 + p33 = 0) **or** (p24 = 1 **and** p14 + p34 = 0)) **and** ((p31 = 1 **and** p11 + p21 = 0) **or**
(p32 = 1 **and** p12 + p22 = 0) **or** (p33 = 1 **and** p13 + p23 = 0) **or** (p34 = 1 **and** p14 + p24 = 0))

create view B_3 **as select** * **from** C_3 **except select** * **from** A_3

create view C_4(id1,duration1,p11,p12,p13,p14,id2,duration2,p21,p22,p23,p24,d3,duration3,p31,p32,p33,p34,
id4,duration4,p41,p42,p43,p44) **as select** B_{31}.*, B_{32}.id3* **from** B_3 B_{31}, B_3 B_{32} **where** B_{31}.id1 = B_{32}.id1 **and**
B_{31}.id2 = B_{32}.id2 **and** B_{31}.id3 < B_{32}.id3 **and** (B_{31}.duration1 + B_{31}.duration2 + B_{31}.duration3 +B_{32}.duration3) <= 10

create view A_4 **as select** * **from** C_4 **where** p11 + p21 + p31 + p41 = 1 **and**
p12 + p22 + p32 + p42 = 1 **and** p13 + p23 + p33 + p43 = 1 and p14 + p24 + p34 + p44 = 1

Figure 3.3: SQL queries to evaluate the BSQ Q_{der} that generate results in multiple output tables

times (under different columns) within a row in the result table representing an answer set. Therefore, this approach uses SQL's case statements to check whether a candidate answer set satisfies a set predicate. All the tuples in the view M satisfy all P_i $(0 \leq i \leq 4)$. The view M' removes the answer sets in M that are not minimal. In the view M', the first four conditions ensure that all the members in the m2 tuple are contained in the m1 tuple, and the remaining four conditions ensure that at least one member from the m1 tuple is different from the m2 tuple which guarantees that the m2 tuple is a proper subset of the m1 tuple. The view M'' removes duplicates in M' and stores the answer sets.

3.5 Basic Approach

To simplify the presentation of evaluation algorithms for SQs, we first present the evaluation of BSQs in this section assuming that all the data and structures can be stored in main

```
create view M as (id1, id2, id3, id4)
select R1.id, R2.id, R3.id, R4.id from R R1, R R2, R R3, R R4
where R1.city = S.H. and R2.city = S.Z. and R3.type = museum and R4.type = park
and (R1.duration + case (R2.id = R1.id) then 0 else R2.duration + case (R3.id = R1.id
     or R3.id = R2.id) then 0 else R3.duration + case (R4.id = R1.id or R4.id = R2.id
     or R4.id = R3.id) then 0 else R4.duration) <= 10

create view M' as select * from M m1 where Not Exists
select * from M m2 where
(m2.id1 = m1.id1 or m2.id1 = m1.id2 or m2.id1 = m1.id3 or m2.id1 = m1.id4) and
(m2.id2 = m1.id1 or m2.id2 = m1.id2 or m2.id2 = m1.id3 or m2.id2 = m1.id4) and
(m2.id3 = m1.id1 or m2.id3 = m1.id2 or m2.id3 = m1.id3 or m2.id3 = m1.id4) and
(m2.id4 = m1.id1 or m2.id4 = m1.id2 or m2.id4 = m1.id3 or m2.id4 = m1.id4) and (
(m1.id1 ≠ m2.id1 and m1.id1 ≠ m2.id2 and m1.id1 ≠ m2.id3 and m1.id1 ≠ m2.id4) or
(m1.id2 ≠ m2.id1 and m1.id2 ≠ m2.id2 and m1.id2 ≠ m2.id3 and m1.id2 ≠ m2.id4) or
(m1.id3 ≠ m2.id1 and m1.id3 ≠ m2.id2 and m1.id3 ≠ m2.id3 and m1.id3 ≠ m2.id4) or
(m1.id4 ≠ m2.id1 and m1.id4 ≠ m2.id2 and m1.id4 ≠ m2.id3 and m1.id4 ≠ m2.id4) )

create view M'' as select * from M' m1 where Not Exist
select * from M' m2 where
(m2.id1 = m1.id1 or m2.id1 = m1.id2 or m2.id1 = m1.id3 or m2.id1 = m1.id4) and
(m2.id2 = m1.id1 or m2.id2 = m1.id2 or m2.id2 = m1.id3 or m2.id2 = m1.id4) and
(m2.id3 = m1.id1 or m2.id3 = m1.id2 or m2.id3 = m1.id3 or m2.id3 = m1.id4) and
(m2.id4 = m1.id1 or m2.id4 = m1.id2 or m2.id4 = m1.id3 or m2.id4 = m1.id4) and
(m2.id1 ≠ m1.id1 or m2.id2 ≠ m1.id2 or m2.id3 ≠ m1.id3 or m2.id4 ≠ m1.id4) and (
(m2.id1 < m1.id1) or (m2.id1 = m1.id1 and m2.id2 < m1.id2) or
(m2.id1 = m1.id1 and m2.id2 = m1.id2 and m2.id3 < m1.id3) or
(m2.id1 = m1.id1 and m2.id2 = m1.id2 and m2.id3 = m1.id3 and m2.id4 < m1.id4))
```

Figure 3.4: SQL queries to evaluate the BSQ Q_{der} that generate results in a single output table

memory, and then describe the extensions to handle large, external data in Section 3.6. We extend our techniques for (general) SQs in Section 3.7.

Recall that a BSQ Q retrieves only minimal sets and all the set predicates in Q are anti-monotone. Our proposed approach evaluates a BSQ Q in two phases. In the first phase, a sequential scan of R is performed to partition R into s disjoint subsets, $R_{V_1}, \cdots, R_{V_s}$, $s \in [1, 2^n]$, where each $V_i \subseteq V$ is a subset of member variables in Q, and $R_{V_i} \subseteq R$ represents the tuples that satisfy all the member predicates (i.e., $\bigcup_{v_j \in V_i} P_j$) associated with the member variables in V_i.

There are two partitions of R, namely, $R_\emptyset$ and R_V, that are not materialized during the partitioning phase [2]. The partition $R_\emptyset$ contains tuples in R that do not satisfy any P_i $(1 \leq i \leq n)$ in Q. For a BSQ Q, none of the tuples in $R_\emptyset$ will contribute to an answer set. Therefore, the partition $R_\emptyset$ is not materialized during the partitioning. At the other

[2]For SQs, both $R_\emptyset$ and R_V have to be materialized as discussed in Section 3.7.1.

extreme, each tuple in R_V satisfies all P_i $(1 \leq i \leq n)$ in Q; therefore, each tuple in R_V forms an answer 1-set if it also satisfies P_0. If a tuple in R_V does not satisfy P_0, it will not contribute to any answer set for a BSQ and can be ignored. Since each tuple in R_V can be either directly output as an answer set or ignored, these tuples will not contribute to additional answer sets; thus, this partition is also not materialized during partitioning. The partitions materialized in the first phase will be used in the second phase to generate further answer sets.

In the second phase, the remaining answer sets are generated by combining tuples from appropriate partitions such that the combined set of tuples qualifies as an answer set; i.e., the set of tuples is a minimal set of tuples that satisfies all the query's predicates. Each such combination of partitions is then evaluated as a cross-product query (CPQ); thus, the remaining answer sets are computed as a union of CPQs. To enumerate these answer sets, we first need to characterize the appropriate combinations of partition sets.

Consider a set of partitions $U = \{R_{V_1}, \cdots, R_{V_k}\}$. We define U to be a *valid partition set (or vpset)* if U satisfies the following two properties: (P1) $\bigcup_{R_{V_i} \in U} V_i = V$; and (P2) no proper subset of U satisfies P1. Property 1 ensures that a candidate answer set S formed by selecting one member from each partition in U will satisfy all the member predicates in Q, while property 2 ensures that S is minimal.

For convenience, we refer to a vpset that is a k-set as a k-vpset. We use $VPSet$ to denote the collection of all vpsets.

Thus, if $U = \{R_{V_1}, \cdots, R_{V_k}\}$ is a k-vpset, then a k-set $S = \{t_1, \cdots, t_k\}$, where $t_i \in R_{V_i}$, $i \in [1, k]$, is an answer set for Q if S satisfies P_0. Therefore, the remaining answer sets for Q is computed by evaluating a collection of CPQs, where each CPQ is associated with a vpset.

Our overall approach evaluates Q based on the following expression:

$$\sigma_{P_0}(R_V \cup \bigcup_{U_i \in VPSet} (\bigtimes_{R_j \in U_i} R_j))$$

$\sigma_{P_0}(R_V)$ is evaluated in the first phase while $\sigma_{P_0}(\bigcup_{U_i \in VPSet}(\bigtimes_{R_j \in U_i} R_j)$ is evaluated in the second phase. The cross-product expression represents a CPQ corresponding to the vpset U_i, the union expression enumerates all the vpsets[3], and the final selection operator

[3]The union operator is used only to combine the results and not to eliminate duplicates as the generated results are all unique.

selects the minimal sets that satisfy all the set predicates P_0.

Example 3.3: Consider the evaluation of the BSQ Q_{der} that is derived from our example SQ Q_{ext} by removing its non-anti-monotone set predicate (i.e., $SUM(S.duration) \geq 6$). In the first phase, R is partitioned into four partitions: $R_{\{v_1,v_3\}} = \{t_1\}$, $R_{\{v_2,v_4\}} = \{t_2\}$, $R_{\{v_3\}} = \{t_3\}$, and $R_{\{v_1\}} = \{t_4\}$. In the second phase, two vpsets, $\{R_{\{v_1,v_3\}}, R_{\{v_2,v_4\}}\}$ and $\{R_{\{v_1\}}, R_{\{v_3\}}, R_{\{v_2,v_4\}}\}$, are enumerated which generate two candidate answer sets $\{t_1, t_2\}$ and $\{t_2, t_3, t_4\}$. Among them, only $\{t_1, t_2\}$ satisfies the anti-monotone set predicate (i.e., $SUM(S.duration) \leq 10$) and forms an answer set. □

In the following, we elaborate on the details of the second phase, namely, how to efficiently enumerate vpsets and evaluate the corresponding CPQs.

Enumeration of vpsets. Given the partitions of R created in the first phase, the collection of all vpsets $VPSet$ is efficiently enumerated based on the following theorem.

Theorem 3.1. *If U is a k-vpset, then it satisfies the following three properties: (1) For each $R_{V_i} \in U$, the cardinality of V_i is at most $n - k + 1$; (2) There must exist a partition $R_{V_i} \in U$ such that the cardinality of V_i is at least $\lceil \frac{n}{k} \rceil$; (3) For any pair of distinct partitions R_{V_i} and R_{V_j} in U, $V_i \not\subseteq V_j$ and $V_j \not\subseteq V_i$.*

Proof. We prove each of the three properties by contradiction.

Suppose the first property is false; i.e., there exists a partition $R_{V_i} \in U$ such that the cardinality of V_i is greater than $n - k + 1$. It follows that U is not a vpset since it does not satisfy the second property of a vpset (i.e., U is not minimal). The reason for this is as follows. To ensure that U is minimal, for any $R_{V_j} \in U$, V_j should contain at least one member variable that other partitions do not contain. Since the cardinality of V_i is greater than $n - k + 1$, the remaining number of member variables is fewer than $k - 1$ which can not ensure that the remaining $k - 1$ partitions in $U \setminus R_{V_i}$ have at least one member variable that other partitions do not contain. Thus, we have a contradiction.

Suppose the second property is false; i.e., for any $R_{V_i} \in U$, the cardinality of V_i is less than $\lceil \frac{n}{k} \rceil$. It follows that U is not a vpset since the number of member variables in $\bigcup_{R_{V_i} \in U} V_i$ is less than n which contradicts the first property.

Suppose the third property is false; i.e., there exists a pair of distinct partitions R_{V_i} and R_{V_j} in U such that $V_i \subseteq V_j$. It follows that U is not a k-vpset since the subset $U \setminus R_{V_i}$ can also satisfy all the member predicates which contradicts the second property. □

Based on the theorem, we enumerate all the vpsets by computing the cartesian product of n sets (with the above three properties enabled to prune the cartesian product space) where each set is $\{R_{V_1}, \cdots, R_{V_s}\}$ representing the set of all generated partitions in the partitioning phase. Thus, the time complexity to enumerate all the vpsets is $O(2^{n^2}n^2)$ where $O(2^{n^2})$ is the time complexity to compute the cartesian product to generate all the candidate vpsets and $O(n^2)$ is the time complexity to determine a candidate vpset is indeed a vpset. As the value n is not expected to be large for BSQs, it is very fast to enumerate all the vpsets by exploiting the above three properties.

Example 3.4: Continue with Example 3.3. Here we have $n = 4$. From the first property, partition $R_{\{v_1,v_3\}}$ will not form a 4-vpset since the cardinality of the partition is 2. From the second property, for a 2-vpset, at least one partition should satisfy two P_i $(1 \leq i \leq 4)$, otherwise the 2-vpset can not satisfy all P_i $(1 \leq i \leq 4)$. From the third property, partitions $R_{\{v_1,v_3\}}$ and $R_{\{v_1\}}$ will not appear in the same vpset since one is a subset of the other. □

Evaluation of CPQs. Each CPQ is evaluated using a *multi-way nested-loop cross-product* (MNLCP) approach, which is a generalization of the well-known binary nested-loop join algorithm. For convenience, we use the notation $(R_{V_1}, \cdots, R_{V_k})$ to refer to a CPQ Q' that is over k partitions $\{R_{V_1}, \cdots, R_{V_k}\}$ as well as the ordering of the partitions in a MNLCP evaluation of Q' where R_{V_1} and R_{V_k} are, respectively, the outermost and innermost relations of the MNLCP evaluation.

With the MNLCP evaluation, for a CPQ $Q' = (R_{V_1}, \cdots, R_{V_k})$, each result tuple $(t_1, t_2, \cdots, t_k)$ of Q' (where each tuple $t_i \in R_{V_i}$) is constructed progressively as a sequence of partial result tuples: (t_1), (t_1, t_2), $\cdots$, and finally $(t_1, t_2, \cdots, t_k)$. To optimize the MNLCP evaluation, for each partial result tuple $t = (t_1, t_2, \cdots, t_j)$ $(1 \leq j < k)$, we check whether t satisfies each anti-monotone set predicate p in P_0. If t does not satisfy p, then this implies that none of the partial result tuples extended from t will satisfy p; therefore, the MNLCP evaluation involving t can be immediately "short-circuited" by dropping t from further processing. Note that similar optimization is also applicable for the monotone set predicates in SQs. Specifically, if each partial results tuple t satisfies p, then we can conclude that each of the partial result tuple extended from t will also satisfy p. Further optimizations for anti-monotone/monotone set predicates evaluation are discussed in Section 3.7.2.

The number of CPQs evaluated for a BSQ can be very large: the maximum number of CPQs when n ranges from 3 to 7 are 7, 48, 461, 6432, and 129424, respectively. There-

fore, there could be considerable efficiency gains by applying MQO techniques to optimize the evaluation of a BSQ. However, MQO is a very hard optimization problem with a search space that is doubly exponential in the size of the queries [49, 51, 54, 74]. As early exhaustive strategies [49, 54] are not practical, many heuristic solutions have been proposed (e.g. [13, 51, 74, 14, 73, 57]). To cope with the high optimization complexity, a well-known strategy for MQO is to adopt a two-phase optimization approach [57, 73]. The first phase generates local optimal query plans for the individual queries, and the second phase generates a global query plan that exploits the common subexpressions (CSEs) in the local query plans.

However, the existing MQO heuristics are not appropriate for our problem context for two main reasons. First, as explained above, the number of CPQs in our problem is very large, which means that it is important to use an efficient heuristic that can scale to thousands of queries. Existing MQO heuristics are, however, not designed for such scale. As an example, the state-of-the-art MQO heuristic [51] took 30 seconds to optimize 22 (which is the maximum number of queries considered) queries without considering cross product joins where each query only references five relations, and was unable to scale when the number of relations in the queries increases or cross product joins are considered. Second, most of the existing MQO works [49, 51, 54, 74, 57] are based on the materialization and reusing the results of CSEs which is not beneficial for our context. This is because for CPQs, the cost of computing, writing and reading a CSE result to/from disk is higher than the cost of recomputing the CSE as shown by our experimental results in Section 3.8.1. Thus, our approach for evaluating CPQs does not employ the materialization technique; instead, we evaluate them by pipelining the results of CSEs to CPQs.

Due to both the scale of the problem as well as the nature of the queries (i.e., CPQs and not join queries), existing MQO heuristics designed for optimizing a moderate number of general join queries are too complex and not sufficiently scalable for our problem. We therefore propose a novel and efficient heuristic, which is also based on the two-phase approach, to optimize the evaluation of a large collection of CPQs. The first phase generates a local optimal evaluation plan for each CPQ and the second phase optimize the collection of local plans by exploiting CSEs.

In the first phase, since each CPQ is evaluated using the MNLCP method, the local evaluation plan for a CPQ is simply a specification of the ordering of the partitions in the CPQ (i.e., from outermost to innermost relation). To optimize the evaluation of CPQs, it is desirable to minimize the cost to check anti-monotone set predicates (to find short-circuited partial result tuples). Therefore, our approach to order the partitions for a CPQ is to order

them in non-decreasing order of their cardinalities. The intuition behind the approach is to minimize the cost to check the short-circuited partial result tuples assuming that any pair of partial result tuples of the same length are equally likely to be short-circuited. As shown by our cost model in Section 3.6.2, our approach to order the partitions for a CPQ in a MNLCP evaluation is indeed optimal.

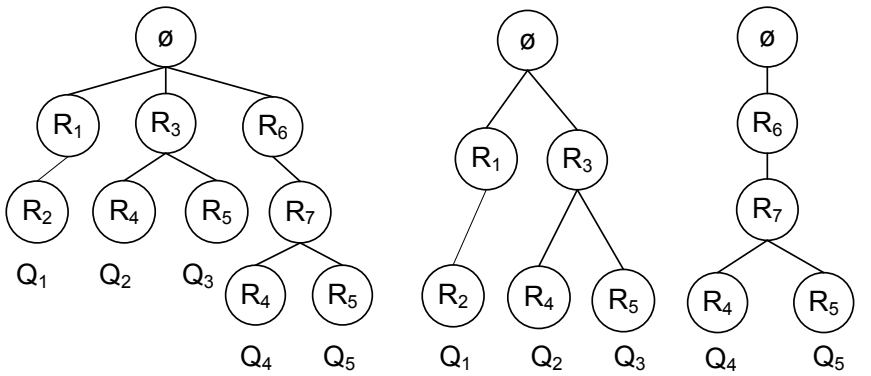
In the second phase, to efficiently identify the CSEs among the local query plans, our heuristic uses a trie to represent all the local query plans. Each node in the trie, except for the root node which is a virtual node, represents a partition, and each path from a child node of the root node to a leaf node corresponds to the sequence of partitions (in non-decreasing order of their cardinalities) in a local query plan. With this simple technique, our heuristic is able to capture the common "prefixes" among the local query plans. The time complexity of constructing the trie is proportional to the total number of partitions in all the CPQs. The simplicity of this structure enables our heuristic to scale to a large number of queries.

Once the trie has been constructed with the local query plans, the global query plan is formed and evaluated by a top-down traversal of the trie structure. Consider a trie node R_i that has multiple child nodes, and let $(R_1, \cdots, R_{i-1})$ be the path of ancestor nodes of R_i in the trie (i.e., R_1 is the child of the root node and each R_j is a child node of R_{j-1}, $j \in [2, i]$). By pipelining the output of $(R_1 \times \cdots \times R_i)$ to each of the child nodes of R_i, the computation of the cross-product expression associated with the common prefix path is shared among the child nodes.

Example 3.5: Consider a BSQ that is evaluated as five CPQs $\{Q_1, \cdots, Q_5\}$ with their local query plans shown by the trie in Figure 3.5(a), where the node labeled $\emptyset$ represents the virtual root node of the trie. Each path from a child node of the root node to a leaf node corresponds to a local query plan for a CPQ. For example, the fourth leftmost path corresponds to the local plan (R_6, R_7, R_4) for Q_4. Observe that the two local plans for Q_2 and Q_3 share the partition R_3. Thus, for every tuple t read from R_3, the global plan evaluation will pipeline t to its child nodes R_4 and R_5. □

3.6 Handling Large Data

In this section, we extend our in-memory approach discussed in the previous section to evaluate BSQs on large, disk-based data.



$\|R_1\|$	$\|R_2\|$	$\|R_3\|$	$\|R_4\|$	$\|R_5\|$	$\|R_6\|$	$\|R_7\|$
1	2	2	3	4	2	2

(c) Size of partition in terms of number of pages

Figure 3.5: An example of CPQ partitions organized as a trie

In the following discussion, we use B to denote the number of main memory buffer pages available for evaluating a BSQ Q on a relation R. For a partition R_{V_i}, we use $|R_{V_i}|$ and $\|R_{V_i}\|$ to respectively denote its size in terms of number of pages and its cardinality in terms of number of tuples. We assume that the answer sets computed for a BSQ are directly output without being buffered.

3.6.1 Phase 1: Partitioning Phase

In the first phase, we need to allocate the available buffer space for reading R as well as creating the partitions of R. This partitioning problem using limited buffer space can be solved with two standard database techniques (i.e., sorting and hashing), which we briefly described in this section.

In the hash-based approach, we allocate one buffer page for reading R and divide the remaining buffer pages uniformly among the maximum number of $2^n - 2$ partitions to be materialized[4]. Each tuple read from R is copied to the appropriate partition buffer, and a partition buffer is flushed to disk when it becomes full. For the case where there is not enough buffer space to even allocate one page for each partition, then R will have to be partitioned in multiple passes instead of a single pass.

In the sort-based approach, each tuple read from R is assigned an appropriate partition identifier (i.e., $1, \cdots, 2^n - 2$) based on the subset of member predicates that it satisfies.

[4]Recall from Section 3.5 that $R_\emptyset$ and R_T are not materialized.

The tuples are then sorted on this identifier using external merge-sort algorithm to create the partitions.

If the buffer space is sufficiently large such that R can be hash partitioned in one scan, then the hash-based approach is generally more efficient as the sort-based approach might require multiple merge passes to sort R. However, if a BSQ contains certain type of set predicates, then the sort-based approach could be optimized to become more efficient; we defer the discussion of the optimization to Section 3.7.2.

3.6.2 Phase 2: Enumeration Phase

The main challenge in the second phase is how to efficiently evaluate a large collection of CPQs given a buffer space constraint of B pages.

Consider a CPQ $Q' = (R_1, \cdots, R_k)$. What is an optimal approach to evaluate Q' such that (1) the buffer space used is minimized and (2) each partition in Q' is read only once? A well-known strategy to achieve this is to load all the partitions of Q', except for the outermost partition (i.e., R_1), into the buffer and to allocate only one buffer page for R_1. As each page R_p of R_1 is loaded into the buffer, the MNLCP method is used to compute $R_p \times R_2 \times \cdots \times R_k$. Thus, the minimum buffer space required for this optimal evaluation is $1 + \sum_{i=2}^{k} |R_i|$ pages. Let $minbuf(Q')$ denote the minimum buffer space requirement (in terms of number of pages) for evaluating a CPQ Q' in this manner.

Given a buffer space of B pages, we classify a CPQ Q' as a *lean query* if $minbuf(Q') \leq B$; otherwise, Q' is classified as a *fat query*. Let Q_{lean} and Q_{fat} denote the set of all the lean and fat CPQs, respectively, from the collection of CPQs to be evaluated. From the optimization viewpoint, Q_{lean} are easier to optimize than Q_{fat}. Therefore, our proposed approach optimizes the evaluation of Q_{lean} and Q_{fat} separately.

Evaluation of Lean Queries. To exploit the CSEs among a collection of lean CPQs, we present an efficient strategy to evaluate them in *batches* such that each batch of queries can be evaluated efficiently similar to the in-memory approach described in Section 3.5 using only B buffer pages. We first formally define a query batch and then present efficient heuristics to optimize both the partitioning of Q_{lean} into query batches as well as the evaluation order of the batches.

Consider a set of lean CPQs $Q_{batch} = \{Q_1, \cdots, Q_m\}$, where $Q_{batch} \subseteq Q_{lean}$ and each $Q_i = (R_{i,1}, \cdots, R_{i,k_i})$. Let $D(Q_{batch}) = \bigcup_{Q_i \in Q_{batch}} \{R_{i,2}, \cdots, R_{i,k_i}\}$ denote the set of

distinct partitions in all the m CPQs from Q_{batch} after excluding the outermost partition from each CPQ (i.e., $R_{i,1}, i \in [1, m]$). We say that Q_{batch} forms a *query batch* if $1 + \sum_{R_i \in D(Q_{batch})} |R_i| \leq B$. Note that a query batch Q_{batch} can be evaluated optimally using only B buffer pages as each partition (involved in Q_{batch}) is read only once from disk.

Example 3.6: Assume that $B = 10$. Consider Q_5 in Figure 3.5. Since $minbuf(Q_5) = 1 + |R_7| + |R_5| = 7 < B$, Q_5 is classified as a lean query. Similarly, all the other queries (Q_1 to Q_4) in Figure 3.5 are also classified as lean queries. Consider a set of lean queries $Q'_{batch} = \{Q_4, Q_5\}$. We have $D(Q'_{batch}) = \{R_7, R_4\} \bigcup \{R_7, R_5\} = \{R_7, R_4, R_5\}$. Since the total size of the partitions in $D(Q'_{batch})$ (i.e., $|R_7| + |R_4| + |R_5| = 9$) is no larger than $B - 1$, Q'_{batch} forms a query batch. On the other hand, for the set of lean queries $Q''_{batch} = \{Q_1, Q_4, Q_5\}$, since the total size of the partitions in $D(Q''_{batch}) = \{R_2, R_7, R_4, R_5\}$ is 11 which is larger than $B - 1$, Q''_{batch} is not a query batch. □

Partitioning of query batches. Since a partition may appear in multiple CPQs which are in different query batches, a partition may still be read into the buffer multiple times. Thus, it is desirable to group CPQs that share some common partition (or more generally, share some CSEs in the form of a subset of partitions) in the same query batch to minimize both the number of times a common partition is read into the buffer as well as the number of redundant computations of the CSEs.

Our heuristic to partition Q_{lean} into query batches applies the same idea from Section 3.5 to organize the CPQs in Q_{lean} using a trie to capture the common "prefixes" among the CPQs. The query batches are then created by a pre-order traversal of the trie as follows. We first initialize the current query batch Q_{batch} to be empty. Whenever the pre-order traversal visits a leaf node in the trie, we have found a CPQ Q' which corresponds to the root-to-leaf path in the trie. If Q_{batch} remains a query batch after Q' is added to it, we add Q' to be part of Q_{batch}; otherwise, we initialize a new query batch with only Q' in it and call this the current query batch. At the end of the traversal, Q_{lean} is partitioned into query batches. By partitioning Q_{lean} in this way, our heuristic is able to capture the CSEs among the CPQs in each batch. Thus, each query batch is a trie which is a subtree of the input trie. The time complexity for the query batch partitioning is linear to the number of nodes in the trie.

Evaluation order of query batches. We now explain how a query batch $Q_{batch} = \{Q_1, \cdots, Q_m\}$ formed using the above approach is evaluated similar to the in-memory approach. Here each Q_i represents a CPQ. For each $Q_i = (R_{i,1}, \cdots, R_{i,k_i}) \in Q_{batch}$, we

load into the buffer all the partitions of Q_i, except the outermost partition $R_{i,1}$. Note that within each query batch, each partition is loaded exactly once in the buffer even if the partition appears in different queries. By the definition of a query batch, the remaining number of pages left in the buffer (denoted by B') after loading all the partitions except for $R_{i,1}$ must be at least one. Therefore, we can incrementally load the outermost partition $R_{i,1}$ for each Q_i into the buffer (B' pages at a time), and pipeline the loaded tuples of $R_{i,1}$ to each child partition to compute the CPQs in the query batch.

The final optimization issue to consider is how to order the query batches formed for evaluation. If two query batches have many partitions in common, then it is desirable to evaluate these two batches consecutively so as to minimize the number of times the same partition is loaded into the buffer (across query batches). This scheduling optimization problem can be formulated as finding the longest Hamiltonian path in a fully-connected, weighted, undirected graph $G = (V', E')$ as follows. Each vertex in V' represents a query batch, and each edge in E' has a weight that is equal to the sum of the sizes of the common partitions (excluding the outermost partition in each CPQ) between the CPQs corresponding to the connected vertices. This optimization problem is in general NP-complete; and we solve this using a simple $3/4$ approximation algorithm [6], which has a time complexity $O(|V'|^3)$ where $|V'|$ is the number of query batches.

Example 3.7: Assume that $B = 10$. Figure 3.5(b) shows two query batches, $Q'_{batch} = \{Q_1, Q_2, Q_3\}$ and $Q''_{batch} = \{Q_4, Q_5\}$, constructed from the trie in Figure 3.5(a) by a pre-order traversal of the trie. Let us assume that Q'_{batch} is evaluated before Q''_{batch}. When evaluating the batch Q'_{batch}, the partitions R_2, R_4 and R_5 are completely loaded into the buffer and the remaining 1 buffer page is used to load in the tuples in R_1 and R_3 sequentially with the tuples being pipelined to the corresponding children partitions. When evaluating the batch Q''_{batch}, as R_4 and R_5 have already been loaded in the buffer, we only need to load in R_7 (i.e., R_2 is evicted from the buffer) and the remaining 1 buffer page is used to load in R_6. □

Evaluation of Fat Queries. Since each fat CPQ can not be evaluated optimally with the available B buffer space, our evaluation approach for lean CPQs is not applicable for fat CPQs. To exploit the CSEs among a collection of fat CPQs, another alternative strategy is to materialize and reuse (instead of pipelining) the results of CSEs. However, since a cross-product result is always larger than the combined size of its input operands, a materialization strategy incurs a high I/O cost to write and read the materialized results. Indeed, as shown by our experimental results, it is overall more efficient to recompute the

results of a CSE (incurring a higher CPU cost) than to materialize and reuse the results of a CSE. Thus, we propose to use the MNLCP method to evaluate each fat CPQ separately without relying on any result materialization. The main challenge here is how to effectively allocate the buffer space among the partitions in the CPQ to optimize both CPU and I/O costs.

In the following, we first analyze the I/O and CPU costs of the MNLCP evaluation method, and then present our heuristic to optimize the buffer allocation based on these cost models.

Cost models. Consider the evaluation of a fat CPQ $Q' = (R_{V_1}, \cdots, R_{V_k})$ using the MNLCP approach. Let $(b_1, \cdots, b_k)$ denote the buffer space allocation for the partitions, where each R_{V_i} is allocated b_i number of buffer pages, such that $\sum_{i=1}^{k} b_i \leq B$. The MNLCP evaluation method will first load the first b_i pages of each R_{V_i} into the buffer and compute the cross-product among the tuples in the buffer, and then load in the next b_1 pages for R_{V_1}, and so on. Whenever all the pages of some R_{V_i} have been read and loaded into the buffer, the method will load in the next b_{i+1} pages for $R_{V_{i+1}}$ and "rewind" each R_{V_j}, $j \in [1, i]$, by loading in the first b_j pages for each R_{V_j}, $j \in [1, i]$. The method terminates when all the pages of R_{V_k} have been read. The I/O cost to evaluate Q' in such a manner is given by

$$C_{i/o} = \sum_{i=1}^{k} c_{i/o} |R_{V_i}| \prod_{j=i+1}^{k} \lceil \frac{|R_{V_j}|}{b_j} \rceil \tag{3.1}$$

where $c_{i/o}$ is the cost ratio to read one page. Each $c_{i/o}|R_{V_i}| \prod_{j=i+1}^{k} \lceil \frac{|R_{V_j}|}{b_j} \rceil$ represents the I/O cost to load in R_{V_i} with $\prod_{j=i+1}^{k} \lceil \frac{|R_{V_j}|}{b_j} \rceil$ representing the times to load in R_{V_i}. The CPU cost to evaluate Q' is given by

$$C_{cpu} = \sum_{i=1}^{k} c_{cpu} S_i \prod_{j=1}^{i} \|R_{V_j}\| \prod_{j=i+1}^{k} \lceil \frac{|R_{V_j}|}{b_j} \rceil \tag{3.2}$$

where c_{cpu} is the cost ratio to process a tuple and S_i is the selective factor of anti-monotone set predicates for $(i-1)$-sets. Each $c_{cpu} S_i \prod_{j=1}^{i} \|R_{V_j}\| \prod_{j=i+1}^{k} \lceil \frac{|R_{V_j}|}{b_j} \rceil$ represents the CPU cost to compute the cross product of $(R_{V_1}, \cdots, R_{V_i})$ with $S_i \prod_{j=1}^{i} \|R_{V_j}\|$ representing the number of cross product results that need to be computed, and $\prod_{j=i+1}^{k} \lceil \frac{|R_{V_j}|}{b_j} \rceil$ representing the times to compute the cross-product results. Note that both $c_{i/o}$ and c_{cpu} are tunable constants commonly used in query optimizers and S_i can be estimated based on conventional RDBMS estimation techniques (e.g. with histograms).

We remark that if each b_i $(1 \leq i \leq k)$ is allocated $|R_{V_i}|$ pages (i.e., in-memory case), then

our approach to evaluate each CPQ with the partitions ordered in non-descending order of cardinality indeed minimizes the CPU cost.

Optimizing buffer allocation. As both $C_{i/o}$ and C_{cpu} are not related to b_1, we will allocate the minimum of one page to b_1. The overall optimization problem is to minimize $C_{total} = C_{i/o} + C_{cpu}$ with the following constraints: (1) $b_1 = 1$, (2) $\sum_{i=2}^{k} b_i \leq B - 1$, and (3) $b_i \leq |R_{V_i}|$ for $2 \leq i \leq k$.

A naive solution to optimize the above is to try all possible assignments for $(b_2, \cdots, b_k)$. However, the time complexity will be $O(B^k)$ which is not feasible when B and k are large. Therefore, we use a simple greedy approach to solve the problem by iteratively selecting the "best" partition to increase its buffer allocation until the buffer space is fully utilized. Initially, each partition is allocated one page (i.e., $b_i = 1$ for $2 \leq i \leq k$). At each iteration, we first compute the *benefit ratio* for each partition R_{V_i}, given by $(C - C'_i)/(b'_i - b_i)$, where b_i is the current buffer allocation for R_{V_i}, C is C_{total} for the current buffer allocation, b'_i is the smallest possible integer such that $b'_i > b_i$ and $\lceil \frac{|R_{V_i}|}{b'_i} \rceil < \lceil \frac{|R_{V_i}|}{b_i} \rceil$, and C'_i is C_{total} after increasing b_i to b'_i. Thus, the benefit ratio measures the reduction in evaluation cost per additional buffer page allocated for a partition. Then we increase b_i for the partition R_{V_i} with the maximum benefit ratio to b'_i. The time complexity of this heuristic is $O(Bk)$ where B is the maximum number of iterations and $O(k)$ is the time complexity of an iteration.

Unlike lean CPQs, where the order of evaluation is optimized, we do not optimize the order of evaluating fat CPQs as the potential benefit is questionable. Since the allocated buffer for a partition is generally less than the partition size, and the allocation could vary among CPQs having that partition, we can only partially share the scan of the partition across CPQs which entails non-trivial bookkeeping to keep track of partially loaded partitions. We therefore do not consider this optimization in this work.

3.6.3 Progressive Approaches

Our proposed two-phase approach is a blocking algorithm in that the enumeration phase can only start after the partitioning phase has completed. For a BSQ that does not require retrieving all the answer sets (e.g., the query has a limit-clause), this approach is not ideal. In this section, we describe how to extend the two approaches (sort-based and hash-based approaches) for the first phase to make them non-blocking (i.e., progressive) so that more

answer sets can be generated earlier during the first phase (beyond those produced by R_T). The challenge is to avoid generating duplicate answer sets that are produced in both the partitioning and enumeration phases.

Sort-based approach. To make the sort-based partitioning phase progressive, we generate answers while creating initial sorted runs as follows. For each set of tuples that form an initial sorted run, we first sort them based on their partition identifiers, and then generate minimal answer sets using these in-memory partitions following the basic approach described in Section 3.5. In this way, we are able to compute some answer sets as initial sorted runs are being created in the partitioning phase. A simple way to avoid generating duplicate answer sets is to simply assign a run number to each tuple in the partitioning phase and detect for duplicate answer sets during the enumeration phase as follows: if all the tuples in a potential answer set have the same run number, then the set is a duplicate and is ignored.

Hash-based approach. To make the hash-based partitioning phase progressive, we simply generate answer sets for each new tuple t read with all the in-memory tuples (i.e., we construct the trie for the partition containing t with all the in-memory partitions). In the event that the buffer space is full, we make room for t by selecting some other in-memory partition and flush it to disk. To detect for duplicate answer sets, we adapted the techniques from [61, 64] as follows. Each tuple t is assigned a timestamp [*begin*, *end*], where *begin* and *end* represent, respectively, the time t is read into memory and the time t is flushed to disk. Thus, for each potential answer set S considered in the enumeration phase, S is a duplicate answer if the intersection of the timestamps of all the tuples is not empty.

Comparing the two approaches, the hash-based approach may produce results earlier than the sort-based approach since the former can produce results immediately for each newly read tuple while the latter can only produce results after it has filled and sorted the buffer with tuples. However, the hash-based approach is likely to run slower than the sort-based approach due to the per-tuple overhead (i.e., trie construction for each tuple).

3.7 Extensions and Optimizations

In this section, we first extend our proposed approaches to evaluate SQs in Section 3.7.1. We then discuss the further optimization of SQ evaluation for sort-based approaches based on the properties of set predicates in Section 3.7.2.

3.7.1 Evaluation of SQs

To evaluate SQs, our proposed approaches for BSQs can be extended as follows.

In the partitioning phase, the input table R is partitioned as before based on the combination of predicates satisfied by the tuples; however, we now need to materialize both partitions $R_\emptyset$ and R_T. This is because for a SQ Q, it is now possible for $S \cup \{t\}$ to be an answer set for Q, where $t \in R_\emptyset \cup R_V$ and S is a set of tuples from the partitions excluding $R_\emptyset$ and R_V. Hence, both $R_\emptyset$ and R_V need to be materialized for generating potential answer sets in the second phase.

Since the answer sets for SQs are not necessarily minimal and the set predicates in SQs are not necessarily anti-monotone, the enumeration phase now requires a weaker definition of vpset (Section 3.5) that satisfies only property P1. This weaker definition has two implications. First, the partitions in a vpset are now not necessarily distinct as it is possible for an answer set to contain multiple tuples from the same partition. However, as the cardinality of answer sets is bounded by n, the maximum number of partitions in a vpset is also bounded by n. Second, it is now possible for one vpset to be a subset of another vpset. For instance, in the example SQ in Section 3.1, if the query is not constrained to retrieve only minimal answer sets, then both $\{R_{\{v_1,v_3\}}, R_{\{v_2,v_4\}}\}$ and $\{R_{\{v_1,v_3\}}, R_{\{v_1,v_3\}}, R_{\{v_2,v_4\}}\}$ are vpsets with one being a subset of the other.

Consequently, after constructing the trie to capture the CSEs for the local query plans, each path from a child node of the root node to any node in the trie now may correspond to a vpset. Note that this is different from the trie constructed for BSQs where only a path from a child node of the root node to a leaf node corresponds to a vpset. Furthermore, since a vpset U could contain multiple instances of the same partition, the CPQ corresponding to U needs to be evaluated such that answer sets with duplicates are not generated by judicious manipulation of tuple pointers using the MNLCP approach[5].

Minimal set constraint. For SQs that are constrained to retrieve only minimal sets, the following additional extensions are required. In the partitioning phase, for R_V, if a tuple t in R_V satisfies P_0, then we simply output t as a singleton answer set; otherwise, we materialize t. Thus, the materialized R_V contains tuples that satisfy all the member predicates

[5]Consider the evaluation of a CPQ $(R_1, R_2, \cdots)$ where R_1 and R_2 are two instances of the same partition R. To avoid generating duplicate answer sets, whenever the tuple pointer for the outer partition R_1 is moved to the i^{th} tuple of R, the tuple pointer for the inner partition R_2 is rewind to the $(i+1)^{th}$ (rather than the first) tuple of R.

but do not satisfy P_0. In the enumeration phase, since the weaker vpset definition does not guarantee that a candidate answer set is minimal, we need to verify its minimality requirement during the enumeration phase as discussed in Section 3.4.

Example 3.8: Consider the example SQ Q_{ext}. In the partitioning phase, R is partitioned into four partitions: $R_{\{v_1,v_3\}} = \{t_1\}$, $R_{\{v_2,v_4\}} = \{t_2\}$, $R_{\{v_3\}} = \{t_3\}$, and $R_{\{v_1\}} = \{t_4\}$. In the enumeration phase, all the vpsets are enumerated using the weaker definition of vpset for SQs. Some example vpsets include $\{R_{\{v_1,v_3\}}, R_{\{v_2,v_4\}}\}$, $\{R_{\{v_1,v_3\}}, R_{\{v_2,v_4\}}, R_{\{v_3\}}\}$ and $\{R_{\{v_1\}}, R_{\{v_3\}}, R_{\{v_2,v_4\}}\}$. Note that the number of vpsets for the SQ is larger than that for the corresponding BSQ (Example 3.3) due to the weaker definition of vpset for SQs. After evaluating the corresponding CPQs and checking the set predicates, two answer sets $\{t_1, t_2\}$ and $\{t_1, t_2, t_3\}$ are formed. □

3.7.2 Optimizations of SQ Evaluation

In this section, we describe how the evaluation of a SQ using MNLCP can be further "short-circuited" for sort-based approach by exploiting the presence of certain set predicates in the SQ. Before we describe the optimizations, we first present some preliminaries.

Given a k-set $S = (t_1, \cdots, t_k)$ and a function F, F is classified as *distributive* if there is a function F' such that $F(S) = F'(F(t_1), \cdots, F(t_k))$. A distributive function F is classified as *monotone* if for any two k-sets $S_1 = (t_{11}, \cdots, t_{1k})$ and $S_2 = (t_{21}, \cdots, t_{2k})$ such that $F(t_{1i}) \leq F(t_{2i})$ for each $i \in [1, k]$, one has $F(S_1) \leq F(S_2)$. The function "SUM(S.duration)" in Section 3.2 is an example of a distributive monotone function.

Anti-monotone set predicates. If a SQ contains an anti-monotone set predicate p of the form $F(S) \leq c$ where F is a distributive monotone function, then we can optimize the sort-based approach of partitioning as follows. Instead of sorting the tuples using only the partition identifier pid, we sort on the composite key $(pid, F(t))$ which generates partitions that are sorted on $F(t)$. When evaluating a CPQ $(R_{V_1}, \cdots, R_{V_k})$ using MNLCP to generate k-sets, if t_j is the first tuple from R_{V_j} $(1 \leq j \leq k)$ that does not satisfy p when combined with a specific combined tuple $(t_1, \cdots, t_{j-1})$ from $(R_{V_1} \times \cdots \times R_{V_{j-1}})$, then we can short-circuit the MNLCP evaluation by dropping $(t_1, \cdots, t_{j-1})$ from further processing. Note that if we do not sort on the composite key $(pid, F(t))$, we can only drop $(t_1, \cdots, t_j)$ from processing.

Monotone set predicates. Consider a SQ that contains a monotone set predicate p of the form $F(S) \geq c$ where F is a distributive monotone function, then we can optimize the sort-based approach of partitioning as follows. Here again, we sort on the composite key $(pid, F(t))$ which generates partitions that are sorted on $F(t)$. When evaluating a CPQ $(R_{V_1}, \cdots, R_{V_k})$ using MNLCP to generate k-sets, if t_j is the first tuple from R_{V_j} $(1 \leq j \leq k)$ that satisfies p when combined with a specific combined tuple $(t_1, \cdots, t_{j-1})$ from $(R_{V_1} \times \cdots \times R_{V_{j-1}})$, then we do not need to check the satisfiability for the partial result tuples extended from $(t_1, \cdots, t_{j-1})$. Note that if we do not sort on the composite key $(pid, F(t))$, we can only avoid the satisfiability checking for the partial result tuples extend from $(t_1, \cdots, t_j)$.

Due to the fixed cardinality of the answer sets for a vpset, the above optimization can also be applied for some functions that are not distributive monotone. One such example is AVG(S.price) $\leq$ (or $\geq$) c.

3.8 Performance Study

In this section, we present an experimental study to compare the performance of our proposed approach against the baseline SQL solution. Our approach was implemented on PostgreSQL 8.4.4, and the experiments were performed on an Intel Dual Core 2.33GHz machine with 3.2GB of RAM and two SATA2 disks running Linux. Both OS and DBMS were installed on a 250GB disk, while the database was stored on a 1TB disk.

Implementation. We implemented our evaluation approach as a new operator inside the PostgreSQL execution engine. An engine-based implementation offers the best performance as it enables the implementation to leverage the existing evaluation code (e.g., external sorting and hashing). Furthermore, it makes the interaction with other database operators much easier. For example, the results of SQs can be pipelined to other database operators like join and set-skyline to perform additional computation.

Algorithms. Table 3.2 shows the notations for the five algorithms (four variants of proposed approach and one baseline SQL solution) compared in the experiments. For each non-progressive algorithm A ($A \in \{ns, nh\}$), we use A-p and A-e to represent, respectively, its partitioning and enumeration phases. For the SQL solution, we actually experimented with two variants: the first variant used virtual views while the second variant used

Table 3.2: Compared algorithms

Notation	Algorithm
ps	progressive, sort-based algorithm
ns	non-progressive, sort-based algorithm
ph	progressive, hash-based algorithm
nh	non-progressive, hash-based algorithm
bs	baseline SQL solution

materialized views. In the experimental results, each running time shown for the SQL solution refers to the timing of the more efficient variant; furthermore, we omit reporting its running time if it exceeds 12 hours.

Datasets. We used both synthetic and real datasets for the experiments. Our real dataset is from the MusicBrainz database [1] which stores music metadata. We created a materialized view by joining several tables from the database as the input relation for our experiments. The schema of the view is *music(mid,mname,duration,language,bname,battribute, btype,bscript,aname,abegindate,aenddate,atype)*, and the detailed information about the attributes can be found in [1]. After removing tuples with non-positive duration attribute value, the size of the materialized view is 1.35GB with 8,507,949 tuples.

Our synthetic dataset was generated based on the schema of the MusicBrainz database [1]. The size of the relation (in the default setting) is 408MB with 1 million tuples. For attributes used for member predicates, their values were generated with a uniform distribution to simplify our control on the selectively factors, while for the attribute (i.e., duration) used for the set predicate, its values were generated with a Gaussian distribution $(\mu = 300, \sigma = 55)$ to ensure that each query returns a reasonable number of answer sets.

Queries. Our experimental queries aim to find different subsets of music files to meet certain constraints. We tested on both BSQs and SQs for the experiments. Each query has between 2 to 6 member variables with exactly one member predicate for each member variable. All the member predicates are on different attributes. Each BSQ also has an anti-monotone set predicate of the form $sum(S.duration) \leq c$, while each SQ has the same anti-monotone set predicate as well as a monotone set predicate of the form $sum(S.duration) \geq c/2$, where c is some constant value. Each query was run three times and we report their average running time.

Parameter settings. Table 3.3 shows the key parameters and their default values used in the experiments; the default parameter values were used unless specified otherwise.

Table 3.3: Key experimental parameters

Parameter	Notation	Default
Cardinality of synthetic input table R	$\|\|R\|\|$	1,000,000
Work memory	B	40MB/200MB
Maximum number of returned answer sets	k	ALL
Number of member predicates	n	4
Selectivity factor of each member predicate	f	0.05
Aggregate value in set predicate	c	Avg

The k parameter represents the maximum number of required answer sets in the query's limit clause and has a default value of "ALL" to retrieve all answer sets. The c parameter is used to control the selectivity of the set predicates and its default value (denoted by "Avg") refers to the average value of the *duration* attribute, which is 230 seconds for the real dataset and 300 seconds for the synthetic datasets.

The work memory parameter B controls the main memory allocated in PostgreSQL for our algorithms as well as for sorting and storing hash tables. Since we are interested in comparing the disk-based variants of our algorithms, we set $B = 40$MB for BSQs and $B = 200$MB for SQs in the default setting. Note that a larger B value was used for SQs since the evaluation of SQs require both $R_\emptyset$ and R_V to be materialized in the partitioning phase which significantly increases the total size of the partitions. However, for the baseline SQL solution, we actually used a larger, fixed value of 256MB of work memory (to improve its performance via speeding up the sort-merge and hash joins in the SQL solution), which is much larger than the typical work memory size recommended for PostgreSQL [4]. Thus, our work memory allocation favors the baseline solution.

Summary of results. For queries where all the query results are returned, our algorithms significantly outperform the SQL solution by up to three orders of magnitude and the non-progressive algorithms are at least as fast as the corresponding progressive algorithms. Furthermore, the sort-based algorithms are significantly faster by up to two orders of magnitude than the corresponding hash-based algorithms due to the optimization technique discussed in Section 3.7.2 for sort-based algorithms. However, the partitioning phase of nh is slightly faster than the partitioning phase of ns as discussed in Section 3.6.

For queries where the maximum number of returned answer sets are limited (i.e., with limit-k clause), our experimental results (with k ranging from 10 to 50) show that both the progressive and non-progressive algorithms outperform the baseline solution by up to one order of magnitude and the progressive algorithms are faster than the corresponding non-progressive algorithms. Furthermore, ph is able to produce results earlier than ps as

ph can start to produce results immediately for each newly read tuple while ps needs to fill and sort the buffer with tuples before producing any results.

3.8.1 Results for BSQs on Synthetic Datasets

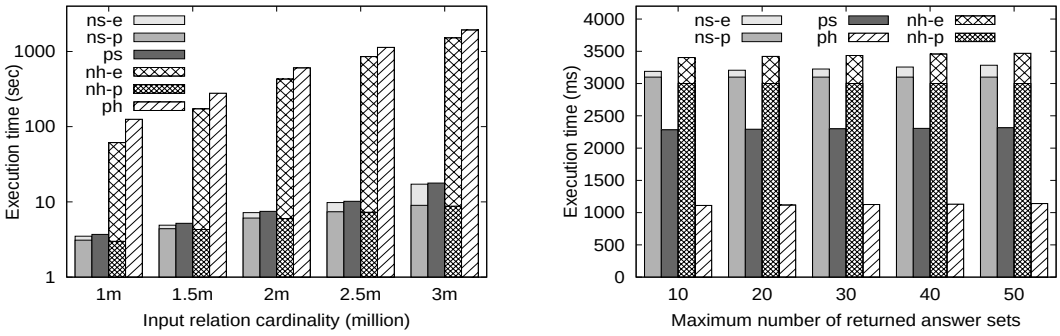
In this section, we first compare our proposed algorithms against the baseline solution, and then study the effectiveness of our optimizations for evaluating lean and fat CPQs, and finally compare the relative performance of our algorithms for different settings.

Comparison with SQL baseline solution. Figure 3.6(a) compares the performance as a function of the input relation cardinality $||R||$. The running times for the baseline solution are not shown on the graph as they are extremely long: for relation cardinality sizes of 1m, 1.5m and 2m, it took 1.2hr, 3.3hr and 6.9hr, respectively; and it exceeded 12hr for cardinality sizes beyond that. Thus, comparing to the cases where the baseline solution run to completion (i.e., under 12hr), our algorithms outperform the baseline solution by up to three orders of magnitude.

As expected, the running times of our algorithms increase with the value of $||R||$. Since a larger input table results in larger partitions, this increases the CPQ processing time for three reasons. First, larger partitions increase the number of results; second, larger partitions cause lean CPQs to be partitioned into more query batches which requires more processing time; and third, larger partitions also increase the number of fat CPQs (which are more costly to evaluate than lean CPQs). For example, when the input cardinality is 1m, 1.5m, 2m, 2.5m, and 3m, the number of answer sets are, respectively, 7942, 15721, 31584, 51247, and 75273; the number of query batches are, respectively, 6, 8, 14, 14, and 15; and the number of fat CPQs are, respectively, 0, 1, 7, 7, and 7.

To enable the baseline solution to complete running within reasonable time, we also compared the algorithms by limiting the maximum number of returned results by varying the k parameter. The comparison is shown in Figure 3.6(b).

For the baseline solution, we manually control its running to obtain k results as follows. Recall that the baseline solution works by generating answer sets iteratively (i.e., 1-sets, 2-sets, etc.) using a sequence of queries. We first try to obtain k answer sets from the query that generates answer 1-sets. If k results are obtained, then we are done; otherwise, we try to obtain the remaining answer sets from the query that generates answer 2-sets, and so on until we get k results.



(a) Effect of $||R||$ (b) Effect of k

Figure 3.6: Comparison with the baseline solution

The performance of the baseline solution (results omitted in Figure 3.6(b)) is almost one order of magnitude slower than our approach: specifically, the running time of bs are 3.6s, 14.9s, 15.0s, 18.1s and 26.9s, respectively, for a k value of 10, 20, 30, 40 and 50. As expected, the execution time of our approach increases as k increases.

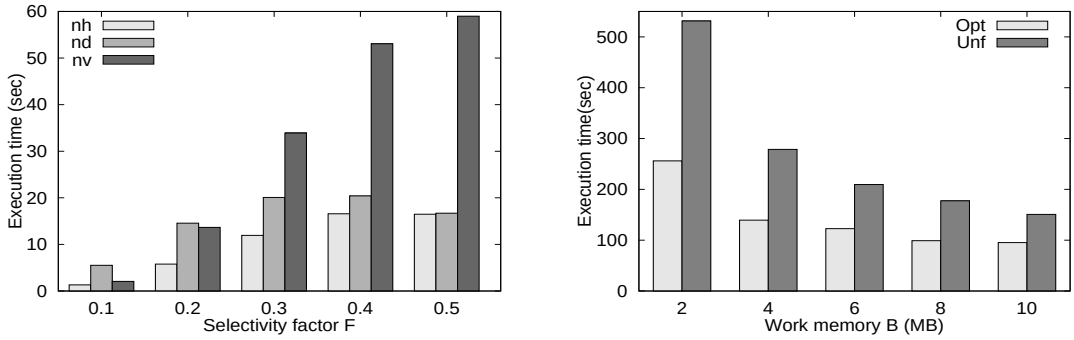
Effectiveness of Optimizations. We now study the effectiveness of our optimizations for evaluating lean and fat CPQs.

To evaluate the effectiveness of our MQO heuristic (denoted by nh[6]) to process lean CPQs, we created two alternative heuristics to compare against nh. The first heuristic (denoted by nd) is equivalent to nh except for nd uses a different strategy to generate the local plans: for each CPQ, its partitions are ordered in non-increasing order of their cardinalities (i.e., opposite to nh's strategy) for the MNLCP evaluation. nd is used to demonstrate the effectiveness of our heuristic to generate local plans. The second heuristic (denoted by nv) uses the same way as nh to generate local plans. However, unlike nh, nv evaluates the CPQs one at a time without sharing the computations of any CSEs; i.e., nv enumerates the vpsets one by one and process the corresponding CPQs one by one. To enable partition scans to be shared, nv employs the following simple buffer replacement strategy: if the buffer is full when a partition P is to be loaded into the buffer, nv randomly evicts some partition(s) that are not needed by the CPQ being evaluated from the buffer to make room for P. nv is used to demonstrate the effectiveness of our heuristic to share computation of CSEs.

Figure 3.7(a) compares the running time of nh, nd and nv as a function of selectivity fac-

[6]We use nh to represent our algorithm since nh is more general than ns (i.e., the optimization technique discussed in 3.7.2 for ns is only applicable for certain set predicates).

tor of member predicates, f [7]. Note that when f increases from 0.1 to 0.5, the cardinalities of the partitions become more balanced. In particular, when $f = 0.5$, the cardinality of all the partitions are almost the same and thus the running times of nh and nd do not show much differences. The experimental results show that nh outperforms nd by 1.1 times on average and up to 3.2 times when $f = 0.1$, which demonstrates the effectiveness of our MQO heuristic to generate local plans, and nh outperforms nv by 1.7 times on average and up to 2.6 times when $f = 0.5$, which demonstrates the effectiveness of our MQO heuristic to share the computation of CSEs.



(a) Lean CPQs (b) Fat CPQs

Figure 3.7: Effectiveness of CPQ optimizations

To evaluate the effectiveness of our heuristic technique (denoted by Opt) for processing fat CPQs, we compare against two other competing techniques (denoted by Mat and Unf). The first, Mat, is the materialization strategy discussed in Section 3.6.2 where a fat CPQ is evaluated as a sequence of binary cross-products with each intermediate result being materialized. The second, Unf, adopts the same MNLCP technique as our Opt but uses a simple buffer allocation strategy that allocates the buffer space uniformly among the query partitions.

To compare the performance of these methods, we created a single fat CPQ with one anti-monotone set predicate that consists of four partitions whose sizes (cardinalities) are, respectively, 3.7MB (7480 tuples), 5.0MB (10084 tuples), 5.1MB (10174 tuples) and 6.3MB (12594 tuples).

Our experimental results show that both Opt and Unf significantly outperform Mat by up to one order of magnitude. As an example, when the work memory is 10MB, the running times for Opt, Unf and Mat are 95s, 151s and 3163s, respectively. Given the poor

[7] To ensure that all the CPQs in the experiment are lean queries when we vary f, we set $||R|| = 10k$ and $n = 6$.

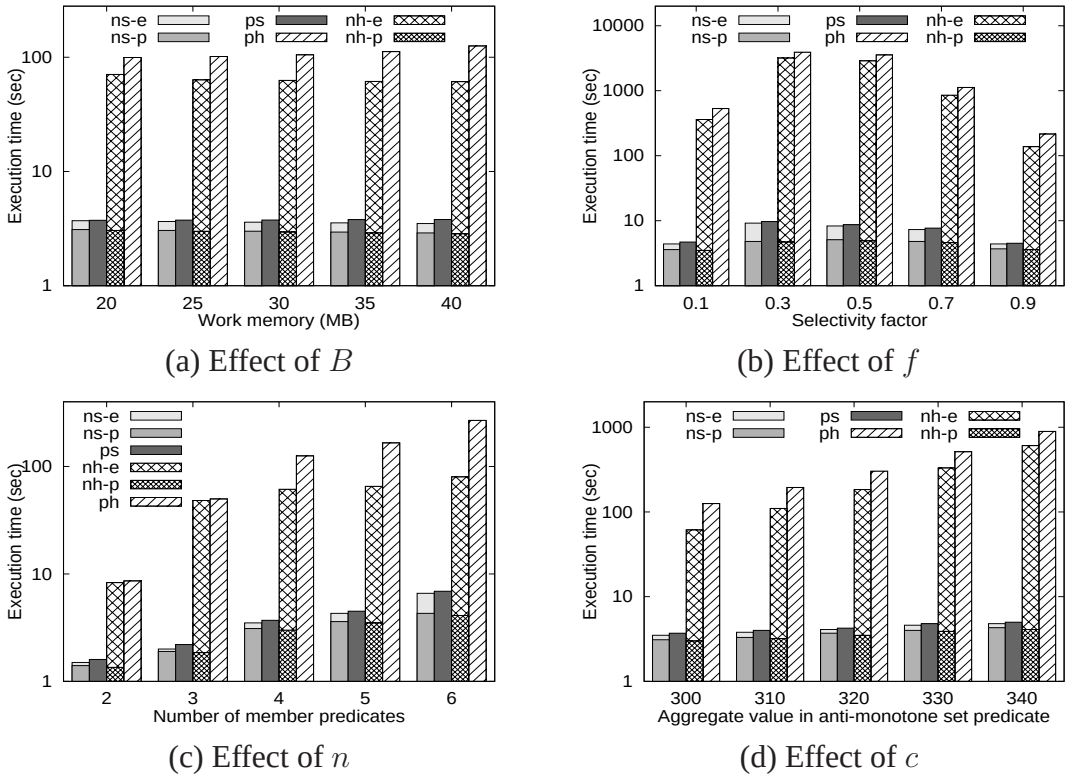
performance of Mat, we next focus on comparing Opt and Unf as a function of the work memory (i.e., B) in Figure 3.7(b). As expected, when B increases, the running times for both Opt and Unf decrease. The experimental results show that Opt outperforms Unf by 83% on average and up to 108% when $B = 10MB$.

Effect of Other Parameters. We compare the effect of other parameters in Figure 3.8; as before, the results for the baseline solution are omitted here as our algorithms outperform the baseline solution by up to three orders of magnitude.

Figure 3.8(a) compares the effect of the work memory size, B. As B increases, the running times for the non-progressive algorithms decrease. This is expected since for non-progressive algorithms, the running times for both the partitioning and enumeration phases decrease when B increases. However, the running times for the progressive algorithms increase with more work memory. The reason is that although a larger B speeds up the enumeration phase of the progressive algorithms, it also increases the running time for the partitioning phase of the progressive algorithms since the larger work memory means that more results are produced during the partitioning phase due to the larger buffer of tuples. For the progressive algorithms, our experimental results show that as B increases, the improvement in the enumeration phase is offset by the slower partitioning phase resulting in an overall slower running time.

Figure 3.8(b) compares the effect of selectivity factor of member predicates, f. We observe an interesting trend where the running time initially increases with increasing f until a certain threshold ($f = 0.3$) after which the running time decreases with increasing f. This is because for BSQs, the value of f affects the type of resultant CPQs and hence the evaluation cost. At one extreme with very small values of f, a tuple is more likely to belong to a partition that satisfies a small number of member predicates. Thus, many tuples will belong to the partition $R_{\emptyset}$ which means that the resultant CPQs can be evaluated efficiently. At the other extreme with very large values of f, a tuple is more likely to belong to a partition that satisfies a large number of member predicates. Thus, the resultant CPQs correspond to vpsets with small cardinality (i.e., CPQs with small number of operand partitions) which can also be evaluated efficiently.

Figure 3.8(c) compares the effect of the number of member predicates, n. Note that the number of partitions increases exponentially with n. Although a larger number of partitions reduces the number of tuples in each partition, it also increases the number of CPQs which increases the running time as shown by our experimental results.



(a) Effect of B

(b) Effect of f

(c) Effect of n

(d) Effect of c

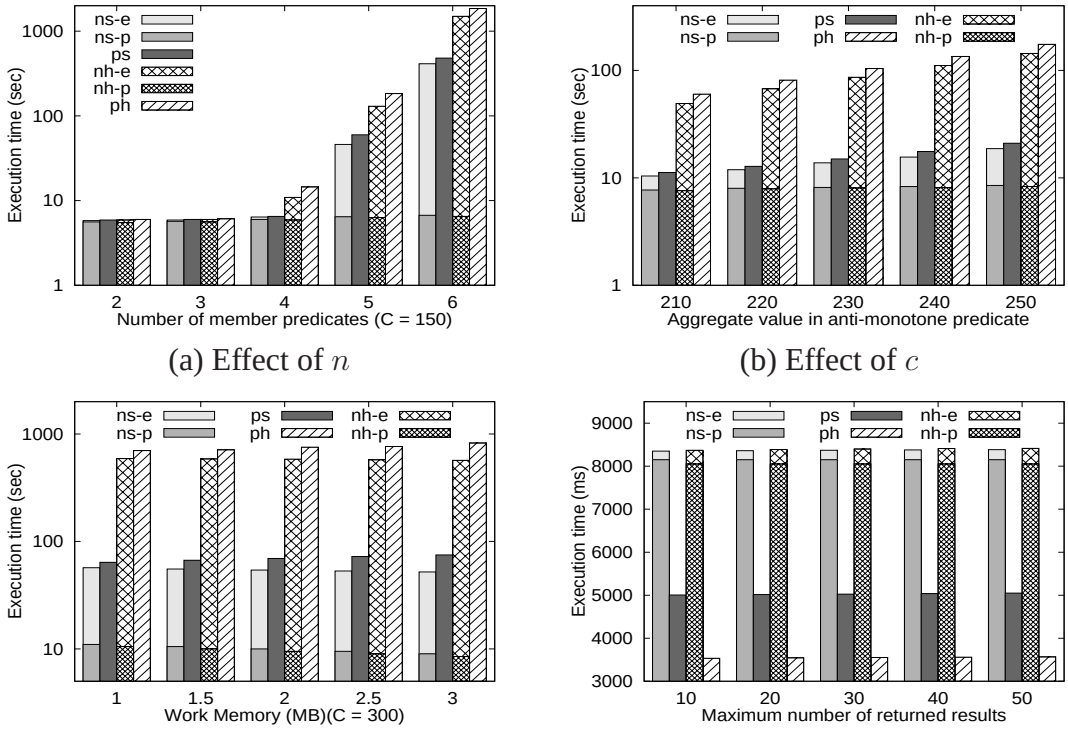
Figure 3.8: Effect of varying parameters on synthetic datasets

Figure 3.8(d) compares the effect of selectivity of the set predicate as we increase the aggregate value c in the set predicate. As the value of c increases, the running times for all the algorithms increase. This is expected since the number of results increases (e.g., the number of answer sets are, respectively, 7942, 14905, 27692, 51243, and 94326 for an aggregate value of 300, 310, 320, 330, and 340) with increasing c value which therefore increases the running time.

3.8.2 Results for BSQs on Real Dataset

In this section, we evaluate the performance of BSQs using the real dataset. Since the cardinality of the real dataset is larger than that of the synthetic datasets, we used smaller selectivity factors for the member predicates for the experiments on the real dataset. In the default setting, each query has four member predicates with the following selectivity factors: 6.1×10^{-4}, 1.1×10^{-3}, 9.4×10^{-4} and 5.8×10^{-4} [8]. Accordingly, we used a

[8] In Figure 3.9(a), the selectivity factors of the additional two member predicates are 3.6×10^{-4} and 2.3×10^{-4}.



(a) Effect of n

(b) Effect of c

(c) Effect of B

(d) Effect of k

Figure 3.9: Effect of varying parameters on real dataset

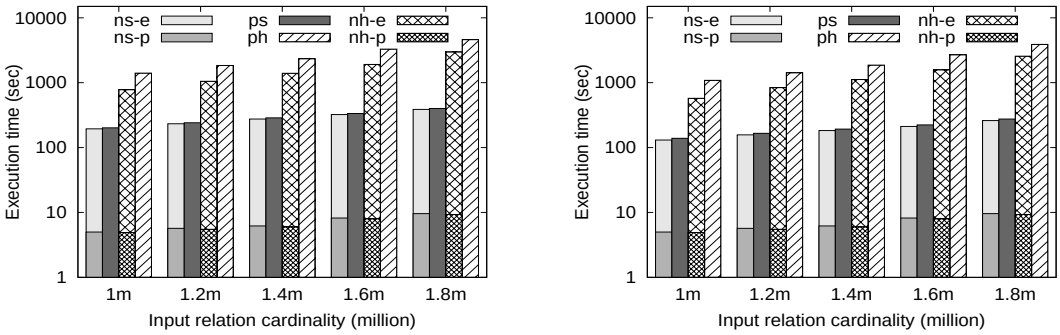
smaller default work memory size of 1MB to ensure that we are comparing the disk-based variants of the algorithms.

In the default setting, the baseline solution did not complete running in 12 hours. In contrast, the running times of ns, ps, nh and ph are $13.8s$, $15.0s$, $86.4s$ and $104s$ respectively. The results shows that our algorithms are at least three orders of magnitude faster than the SQL solution.

Figure 3.9 compares the effect of varying various parameters using the real dataset. Our experimental results for the real dataset exhibit similar trends observed for the synthetic datasets, and we therefore do not repeat the analysis of the results. In Figure 3.9(d), the running times of the baseline solution are not shown as they are one order of magnitude slower than our algorithms. For example, when $k = 10$, the running times of ph, ps, ns, nh and bs are respectively 3.5s, 5.0s, 8.4s, 8.4s and 84s.

3.8.3 Results for SQs on Synthetic Datasets

In this section, we evaluate the performance of SQs on synthetic datasets. Our experimental results for SQs show that both SQs and minimal SQs (i.e., SQs that are constrained to retrieve only minimal answer sets) are more time consuming to evaluate than BSQs. For example, in the default setting, the running times of ph for BSQs, minimal SQs and SQs are, respectively, 61s, 575s and 779s. The reason for this is threefold. First, SQs produce more partitions as both $R_\emptyset$ and R_V have to be materialized in the partitioning phase. Second, SQs require more vpsets to be enumerated (due to the weaker definition of vpsets). For example, when $n = 4$, the number of vpsets for BSQs and SQs are, respectively, 48 and 3229. Third, the number of returned answer sets for SQs are larger. For example, the number of answer sets for BSQs, minimal SQs and SQs are, respectively, 7942, 9214 and 15563 (in the default setting). We also observe that minimal SQs can be evaluated more efficiently than SQs as minimal SQs can prune the cross product space for SQs (i.e., if S is a minimal answer set, then all the supersets of S can be pruned).

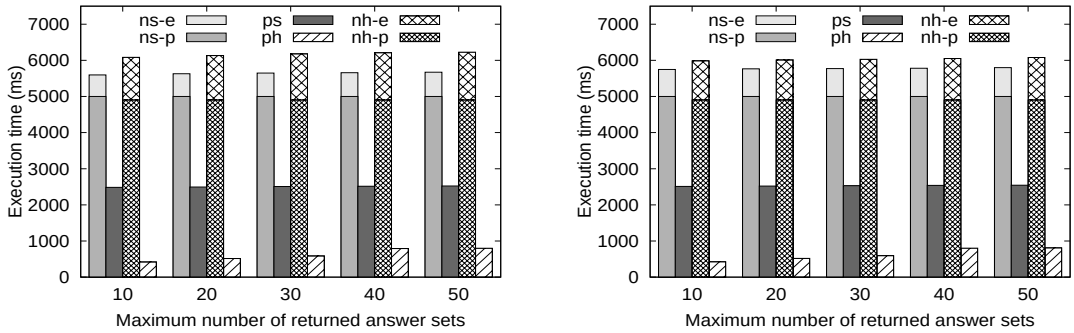


(a) SQs (b) minimal SQs

Figure 3.10: Effect of $||R||$

Figure 3.10 compares the effect of $||R||$ for both SQs and minimal SQs. The baseline SQL solution did not complete execution within 12 hours and we therefore omit its results in the graphs. As expected, the running times of our algorithms increase with $||R||$ as explained in Section 3.8.1.

Figure 3.11 compares the effect of k for both SQs and minimal SQs. As the baseline solution is two orders of magnitude slower than our algorithms, its running times are not shown in the figure. For example, when $k = 50$, the running times of bs are respectively 651.0s and 649.5s for SQs and minimal SQs. As expected, the running times of our algorithms increase slowly with the increasing of k.



(a) SQs (b) minimal SQs

Figure 3.11: Effect of k

We observe that the performance trends for SQs are similar to those for BSQs. Therefore, we do not repeatedly report and discuss them further.

3.8.4 Results for SQs on Real Dataset

In this section, we evaluate the performance of SQs on the real dataset [9]. Here again, the experimental results show that our algorithms significantly outperform the baseline solution. For example, for SQs in the default setting, the running times of ns, ps, nh and ph are respectively, 0.8hr, 1.33hr, 2.25hr and 5.77hr while the baseline solution did not finish running in 12 hours. Furthermore, even for the setting where only k results are returned, both SQs and minimal SQs for the baseline solution did not finish running in 12 hours. This is because the answer sets for queries on the real dataset have large cardinality due to the low selectivity factors of member predicates as discussed in Section 3.8.2, the baseline solution has to spend more time to generate large size candidate answer sets before producing any answer sets. Therefore, the baseline solution runs slowly even for limit-k queries.

We do not repeatedly discuss the results for SQs on the real dataset as the trends are similar to the results for SQs on the synthetic datasets.

[9] To compare the disk-based algorithms and reduce the number of answer sets, we set $c = 100$.

3.9 Summary

In this chapter, we have proposed a novel and efficient approach to evaluate enumerative set-based queries by transforming enumeration set-based queries as a collection of cross product queries. Our extensive experimental results demonstrate that our proposed approach significantly outperforms the conventional RDBMS approach by up to three orders of magnitude.

CHAPTER 4

MULTI-QUERY OPTIMIZATION IN MAPREDUCE FRAMEWORK

4.1 Overview

In this chapter, we study multi-query/job optimization techniques and algorithms for a batch of jobs in the MapReduce framework. The state-of-the-art work in this direction is MRShare [44], which proposed two sharing techniques for a batch of jobs. The *share map input scan* technique aims to share the scan of the input file among jobs, while the *share map output* technique aims to reduce the communication cost for map output tuples by generating only one copy of each shared map output tuple. The key idea behind MRShare is a *grouping technique* to merge multiple jobs that can benefit from the sharing opportunities into a single job. Compared to MRShare, the naive technique of processing each job independently would need to scan the same input file multiple times and generate multiple copies of the same map output tuple. However, MRShare incurs a higher sorting cost compared to the naive technique as sorting a larger map output produced by the merged job is more costly than multiple independent sortings of smaller map outputs produced by the unmerged jobs.

In this chapter, we present a more comprehensive study of multi-job optimization techniques and algorithms. We first propose two new job sharing techniques that expand the opportunities for multi-job optimizations. The first technique is a *generalized grouping technique (GGT)* that relaxes MRShare's requirement for sharing map output. The second technique is a *materialization technique (MT)* that partially materializes the map output of jobs (in the map and/or reduce phase) which provides another alternative means for jobs to share both map input scan and map output. Comparing with the naive technique, GGT incurs a higher sorting cost (similar to MRShare's grouping technique) while MT incurs an additional materialization cost. Thus, neither GGT nor MT is strictly more superior, as demonstrated also by our experimental results.

Given the expanded repertoire of three sharing techniques (i.e., the naive independent evaluation technique, GGT which subsumes MRShare's grouping technique, and MT), finding an optimal evaluation plan for an input batch of jobs becomes an even more challenging problem. Indeed, the optimization problem is already NP-hard when only the naive and grouping techniques are considered in MRShare [44]. We then propose a novel two-phase approach to solve this non-trivial optimization problem.

We conducted a comprehensive performance evaluation of the multi-job optimization techniques using Hadoop. Our experimental results show that our proposed techniques are scalable for a large number of queries and significantly outperform MRShare's techniques by up to 107%.

The rest of the chapter is organized as follows. Section 4.2 introduces the assumptions and notations used in this chapter. Section 4.3 presents several multi-job optimization techniques to share map input scan and map output; their cost models are presented in Section 4.4. Section 4.5 presents a novel two-phase algorithm to optimize the evaluation of a batch of jobs given the expanded repertoire of optimization techniques. Section 4.6 presents a performance evaluation of the presented techniques, and we conclude in Section 4.7.

4.2 Assumptions & Notations

We assume that the input queries are specified in some high-level language (e.g., [58, 59, 47, 26, 20]) which are then translated to MapReduce jobs. By specifying the input jobs via a high-level query language, it facilitates the identification of sharing opportunities among

Table 4.1: Running examples of MapReduce jobs.

Id	Job	<Key,Value>
J_1	***select*** *a, sum(d)* ***from*** *T* ***where*** *a* $\geq$ *10* ***group by*** *a*	<a,d>
J_2	***select*** *a, b, sum(d)* ***from*** *T* ***where*** *b* $\leq$ *20* ***group by*** *a, b*	<(a,b),d >
J_3	***select*** *a, b, c, sum(d)* ***from*** *T* ***where*** *c* $\leq$ *20* ***group by*** *a, b, c*	<(a,b,c),d>
J_4	***select*** *a, sum(d)* ***from*** *T* ***where*** *b* $\leq$ *20* ***group by*** *a*	<a,d >
J_5	***select*** *b, sum(d)* ***from*** *T* ***where*** *a* $\geq$ *20* ***group by*** *b*	<b,d>
J_6	***select*** $*$ ***from*** *T, R* ***where*** *T.a = R.e*	*T:<a, T.$*$>* *R:<e, R.$*$>*
J_7	***select*** $*$ ***from*** *T, R* ***where*** *T.a = R.e and T.b = R.f*	*T:<(a,b), T.$*$>* *R:<(e,f), R.$*$>*

jobs (via their query schemas); and standard statistics-based techniques [38, 55, 68] could be used to estimate the sizes of their shared map outputs. This assumption is also adopted in several related work [44, 18, 40, 68].

In the rest of this chapter, we will use the terms queries and jobs interchangeably. Table 4.1 shows seven jobs (J_1 to J_7) that we will be using as running examples throughout this chapter.

For a job J_i, we use K_i to represent its map output key, A_i to represent the set of attributes in K_i, $|A_i|$ to represent the number of attributes in A_i, f_i to represent its reduce function, M_i to represent its map output [1]., and R_i to represent its reduce output. For example, for J_2 in Table 4.1, $K_2 = (a, b)$, $A_2 = \{a, b\}$ and $|A_2| = 2$.

We use $K_i \preceq K_j$ to denote that K_i is a prefix of K_j, and $K_i \prec K_j$ to denote that K_i is a proper prefix of K_j (i.e., $K_i \neq K_j$). For example, $K_4 \prec K_2$ and $K_5 \not\prec K_2$.

Consider a map output M_i with schema (A_i, V_i) where A_i and V_i refers to the map output key and value attributes, respectively. Given a set of attributes $A \subseteq A_i$, we use M_i^A to denote the map output derived from M_i where its map output key attributes are projected onto A; i.e., $M_i^A = \pi_{A,V_i}(M_i)$. For example, $M_2^{\{a\}} = M_4$.

Consider two jobs J_i and J_j where $A_j \subseteq A_i$. We use $M_{i,j} \subseteq M_i$ to denote the subset of M_i such that $M_{i,j}^{A_j} = M_i^{A_j} \bigcap M_j$ represents the subset of M_j that can be derived from M_i. Furthermore, we use $M_i \bigoplus M_j$ to represent the (key, value-list) representation of the map output $M_i \bigcap M_j$. For example, if $M_i \bigcap M_j = \{(k_1, v_1), (k_1, v_2), (k_2, v_3)\}$, then $M_i \bigoplus M_j = \{(k_1, <v_1, v_2>), (k_2, <v_3>)\}$.

[1] For presentation simplicity, we do not consider combine functions to reduce the size of map output in this chapter; however, our proposed techniques can be easily extended to operate in the presence of combine functions.

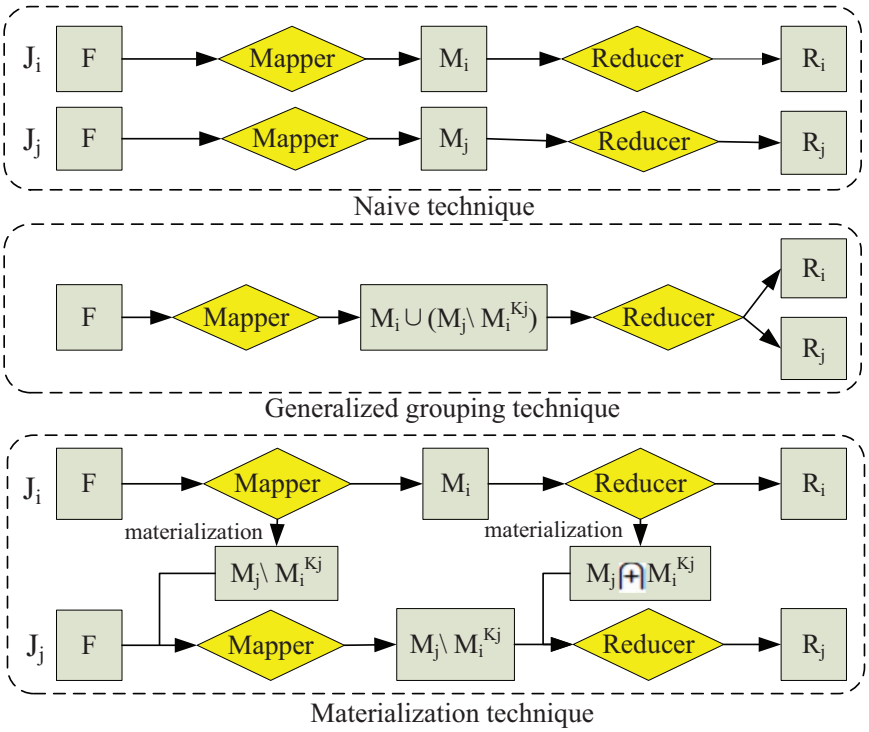


Figure 4.1: Multi-job optimization techniques

4.3 Multi-job Optimization Techniques

In this section, we discuss several multi-job optimization techniques. We first review the grouping technique (GT) in MRShare [44], which is the most relevant work to ours, and then present our proposed generalized grouping technique (GGT) and materialization technique (MT). For simplicity, we first focus our presentation on two single-input jobs J_i and J_j on an input file F and then discuss the generalization for more than two jobs; the handling of multi-input jobs is discussed in Section 4.3.4. Figure 4.1 gives a pictorial comparison of the techniques to process two jobs J_i and J_j, where $K_j \preceq K_i$.

4.3.1 Grouping Technique

In this section, we review MRShare's grouping technique.

Sharing map input scan. For two jobs J_i and J_j to share their map input scan, the input files of J_i and J_j, the input key and value types of J_i and J_j, and the map output key and value types of J_i and J_j must be all the same. We can then combine J_i and J_j into a new

job to share the scan of the map input for the two jobs. We now describe the map and reduce phases of the new job.

In the map phase, the common input file is scanned to generate the map outputs M_i for J_i and M_j for J_j. To distinguish the map outputs of the two jobs in the reduce phase, we use $tag(i)$ to tag the map output M_i and $tag(j)$ to tag the map output M_j. The tags are stored as part of the map output values; thus, each map output tuple is of the form (key,(tag,value)).

In the reduce phase, for each key and for each value associated with the key, if the tag of the value is $tag(i)$, we distribute the value to the reduce function of J_i; otherwise, we distribute the value to the reduce function of J_j. When all the values associated with a key have been examined, we generate the results for that key for the two jobs.

Sharing map output. For J_i and J_j to also share map output besides sharing map input scan, the two jobs must additionally satisfy the requirement that $K_i = K_j$. We can then combine J_i and J_j into a new job to share both their map input scan as well as any common map output (i.e., $M_i \bigcap M_j$). Sharing map output reduces the map output size and hence the sorting and communication cost. We now describe the map and reduce phases of the new job.

In the map phase, the values of the map output are tagged $tag(i)$, $tag(j)$, and $tag(ij)$, respectively, for tuples that belong to $M_i \setminus M_j$, $M_j \setminus M_i$, and $M_i \bigcap M_j$. In this way, tuples that belong to $M_i \bigcap M_j$ are produced only once with the tag $tag(ij)$.

In the reduce phase, for each key and for each value associated with the key, if the tag of the value is $tag(i)$, we distribute the value to the reduce function of J_i; if the tag of the value is $tag(j)$, we distribute the value to the reduce function of J_j; otherwise, we distribute the value to the reduce functions of both J_i and J_j. When all the values associated with a key have been examined, the reducer generates the results for that key for both jobs.

Example 4.1: Consider the two jobs J_1 and J_4. We can combine them into a new job to share both the input file T scan as well as the common map output for $a \geq 10 \wedge b \leq 20$. In the map phase, for each tuple t from T, if $t.a \geq 10 \wedge t.b > 20$, we produce the key-value pair $(t.a, (tag(1), t.d))$ indicating that it is produced by only J_1; if $t.a < 10 \wedge t.b \leq 20$, we produce the key-value pair $(t.a, (tag(4), t.d))$ indicating that it is produced by only J_4; if $t.a \geq 10 \wedge t.b \leq 20$, we produce the key-value pair $(t.a, (tag(14), t.d))$ indicating

that it is produced by both J_1 and J_4; otherwise, we do not produce any map output for the tuple. In the reduce phase, for each key and for each value associated with the key, if the tag of the value is $tag(1)$, we aggregate the value for J_1; if the tag of the value is $tag(4)$, we aggregate the value for J_4; otherwise, we aggregate the value for both J_1 and J_4. When all the values associated with a key have been aggregated, we output the results for that key for J_1 and J_4. □

4.3.2 Generalized Grouping Technique

In this section, we present a generalized grouping technique (GGT) that relaxes the requirement of MRShare's grouping technique (i.e., $K_i = K_j$) to enable the sharing of map output. To motivate our technique, consider the two jobs J_1 and J_2 in Table 4.1. Although $K_1 \neq K_2$, it is clear that the map output of J_2 for $a \geq 10$ could be used to derive the partial map output of J_1. We first present the basic ideas for processing two jobs and then discuss the generalization to handle more than two jobs.

Basic Ideas. To share the map output of two jobs J_i and J_j, GGT requires that $K_j \preceq K_i$ which is a weaker condition than MRShare's grouping technique (i.e., $K_i = K_j$). The jobs J_i and J_j are combined into a new job to enable the map output of J_i to be reused for J_j.

In the map phase of the new job, we generate the map output M_i for J_i and the partial map output $M_j \setminus M_i^{A_j}$ for J_j. The remaining map output of J_j (i.e., $M_{i,j}^{A_j}$) is not generated explicitly since they can be derived from M_i (i.e., $M_{i,j}$). By sharing the map output of J_i and J_j via $M_{i,j}$, we reduce the overall size of the map output. The values of the map output are tagged $tag(i)$, $tag(j)$, and $tag(ij)$, respectively, for tuples that belong to $M_i \setminus M_{i,j}$, $M_j \setminus M_i^{A_j}$, and $M_{i,j}$.

Note that in the MapReduce framework, the map output tuples for a job must all share the same output schema (i.e., same key and value types). While this requirement is satisfied by MRShare's grouping technique (i.e., $K_i = K_j$), the relaxed requirement (i.e., $K_j \preceq K_i$) of GGT may require us to additionally convert the map output of J_i and J_j (produced by our new job) to be of the same type. To achieve this, we use the simple approach of converting both the key and value components of the map output to string values if their types are different. Let us take the conversion of the key component for example. For the key component of a map output tuple, we represent it as a string value that is formed

by concatenating the string representation of each of its key attributes separated by some special delimiter (e.g., ":"). For example, the string representations of the key components of J_2 and J_3 are of the form "a:b" and "a:b:c", respectively. This representation enables each key attribute value to be easily extracted from the string representation of the key component.

Since $K_j \preceq K_i$, the map output of the new job is partitioned on K_j and sorted on K_i. By partitioning on K_j, the map output tuples that have the same K_j values are distributed to and processed by the same reducer thereby enabling the reuse of the map output of J_i for J_j. The sorting on K_i is to facilitate the processing at the reducers (to be explained later); note that this sorting is well defined: for the map output tuples of J_j (whose key values do not contain all the values of K_i), the missing attribute values are treated as being converted to empty string values.

In the reduce phase of the new job, to compute the results of J_i, for each key of J_i, we apply the reduce function on the values associated with that key from tuples tagged $tag(i)$ or $tag(ij)$. To compute the results of J_j, for each key of J_j, besides the values associated with that key (from tuples tagged $tag(j)$), we also need to find the values of J_i that can be reused for J_j; i.e., tuples tagged $tag(ij)$ where the projection of its key on A_j is equal to the key of J_j. The reduce function of J_j is applied on all these values to produce the result for that key. Note that all the relevant tuples needed for the reduce function can be found very efficiently with a partial sequential scan of the map output (which is sorted on K_i).

Unlike the grouping technique where each reduce function is applied on the values associated with one key, GGT may need to apply each reduce function on the values associated with multiple consecutive keys due to the different number of map output key attributes for the jobs. Therefore, in GGT, we have to determine when to apply the reduce functions and output the results for the jobs (the details will be explained later). Figure 4.2 gives a pictorial comparison of applying reduce functions for GGT and GT for two jobs J_i and J_j.

Example 4.2: Consider the two jobs J_1 and J_2. As $K_1 \prec K_2$, GGT is applicable to enable both jobs to share map input scan and map output. In the map phase, for each tuple t from T, if $t.a < 10 \wedge t.b \leq 20$, we produce the key-value pair (t.a:t.b, (tag(2), t.d)) indicating that it is produced and consumed by only J_2; if $t.a \geq 10 \wedge t.b \leq 20$, we produce the key-value pair (t.a:t.b, (tag(12), t.d)) indicating that it is produced by J_2 and consumed by both J_1 and J_2; if $t.a \geq 10 \wedge t.b > 20$, we produce the key-value pair (t.a, (tag(1), t.d))

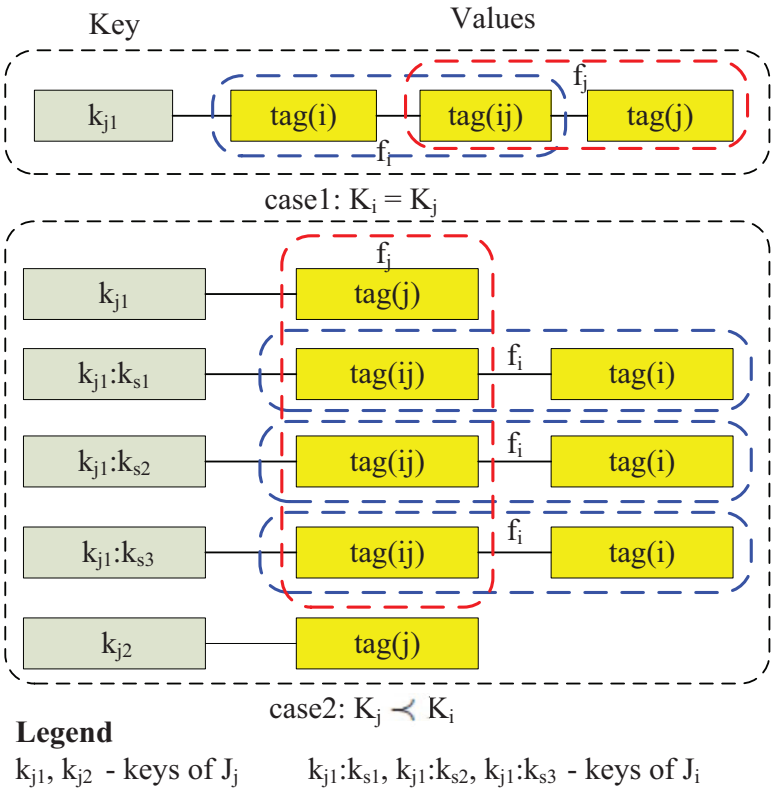


Figure 4.2: A comparison of applying reduce functions for GGT and GT

indicating that it is produced and consumed by only J_1; otherwise, we do not produce any map output for that tuple. We then partition the map output on a and sort the map output on a:b. In the reduce phase, we apply the reduce functions of J_1 and J_2 on the appropriate values to produce the results for J_1 and J_2. □

Generalization. We now discuss how GGT can be generalized to handle more than two jobs.

Consider a batch of jobs $\mathcal{J} = \{J_1, J_2, \cdots, J_n\}$ that are sorted in non-ascending order of $|A_i|$. For each job $J_i \in \mathcal{J}$, let P_{J_i} denote all the jobs preceding J_i in $\mathcal{J}$ whose map output can be reused for J_i; i.e., $P_{J_i} = \{J_j \in \mathcal{J} \mid j < i, K_i \preceq K_j\}$. Furthermore, let $NM_i = M_i \setminus (\bigcup_{J_j \in P_{J_i}} M_j^{A_i})$ denote the map output of J_i that cannot be derived from the map output of any job in P_{J_i}. We refer to NM_i as the *non-derivable map output* of J_i in $\mathcal{J}$. We use $NM = \bigcup_{i=1}^{n} NM_i$ to denote the *non-derivable map output* for all the jobs in $\mathcal{J}$.

GGT combines the batch of jobs $\mathcal{J}$ into a single new job to share map input scan and map output. In the map phase of the new job, for each $J_i \in \mathcal{J}$, we produce and tag the

map output NM_i; each tag here is of the form $tag(S)$, where $S \subseteq \{i, i+1, \cdots, n\}$. In the reduce phase of the new job, we apply the reduce functions on the appropriate values based on their tags to produce the results for the batch of jobs which are discussed below.

Applying reduce functions for GGT. We now discuss how to apply the reduce functions and output the results for the batch of jobs for GGT. For ease of our presentation, we assume sum reduce functions (or other distributive reduce functions as defined in Section 3.7.2).

Algorithm 4.1: Reducer class for GGT

Input: a batch of n jobs $(J_1, J_2, \cdots, J_n)$
Output: reduce output for each job $(R_1, R_2, \cdots, R_n)$

```
1  Method INITIALIZE begin
2      A = new Int[n]; Default value is 0 ;
3      B = new boolean[n]; Default value is false ;
4      C = new String();
5  Method REDUCE (String key, List[(tag, value)]) begin
6      D = decompose key into attributes ;
7      foreach i in [1, Min(|D|,|C|)] do
8          if C[i] ≠ D[i] then
9              foreach j in [1, n] do
10                 if i ≤ |A_j| ≤ |C| and B[j] then
11                     outkey = concatenate the first |A_j| attributes from C ;
12                     emit(outkey, A[j]) ;
13                     A[j] = 0; B[j] = false ;
14             break ;
15     C = D ;
16     foreach (tag, value) in Lists do
17         foreach i in [1, n] do
18             if tag contains i then
19                 A[i] += value ;
20                 if B[i] == false then
21                     B[i] = true;
22 Method Close begin
23     foreach i in [1, n] do
24         if |A_i| ≤ |C| and B[i] then
25             outkey = concatenate the first |A_i| attributes from C ;
26             emit(outkey, A[i]) ;
```

Algorithm 4.1 shows the pseudocode for the reducer class for GGT. A reduce class in Hadoop contains three methods: initialize(), reduce() and close(). Prior to processing any (key, List[(tag,value)]) pair, the initialize method is called. In our reduce class, the initialize method initializes three global variables; one for holding the aggregation values

Figure 4.3: Example illustrating GGT

id	a	b	c	d
t_1	15	30	25	5
t_2	20	30	30	10
t_3	15	15	25	5
t_4	15	20	25	10
t_5	15	15	15	10
t_6	15	15	20	10

(a) An instance of T

Key (a:b:c)	Value-list (tag, d)
15	$t_1 \rightarrow (tag(1), 5)$
15:15	$t_3 \rightarrow (tag(12), 5)$
15:15:15	$t_5 \rightarrow (tag(123), 10)$
15:15:20	$t_6 \rightarrow (tag(123), 10)$
15:20	$t_4 \rightarrow (tag(12), 10)$
20	$t_2 \rightarrow (tag(1), 10)$

(b) (Key,Value-list) layout

for each job (denoted as $A[n]$), one for holding the boolean values for each job which is used to indicate whether some aggregations are performed for each job since its last output (denoted as $B[n]$) and the remaining one for holding the attributes for the previous examined key (denoted as C and the number of attributes in C is denoted as $|C|$). Then the reduce method is applied for each (key, List[(tag, value)]) pair. In our reduce class, for each (key, List[(tag, value)]) pair (the local array D is used to hold the attributes for the examined key and $|D|$ is the number of attributes), the reduce method first checks whether we can output the results for some jobs. This checking is done by finding the first changed attributes (denoted as i where $i \in [1, min(|C|, |D|)]$) between arrays C and D and then for each $j \in [1, n]$, if $i \leq |A_j| \leq |C|$ and $B[j] == true$, we output the results for J_j where its key is formed by extracting the first $|A_j|$ attributes from the array C and its value is simply $A[j]$ (we also have to reset $A[j] = 0$ and $B[j] = false$). The intuition is that if the i^{th} attribute changes, for a job J_j whose $|A_j|$ is at least i, since the map output are sorted on K_1, all the values for the following keys can not be reused for the job for its key. Therefore, the results for J_j for its key can be outputted. Then it updates the previous key to be the current key (i.e., copy D to C). Finally, it applies the aggregations for the current key and accordingly updates A and B. After applying the reduce method for each (key, List[(tag,value)]) pair, the close method is called. In our reduce class, the close method is used to output the remaining results for the jobs (i.e., for each $j \in [1, n]$, $B == true$).

Note that the above algorithm assumes sum reduce functions (or other distributive reduce functions). For non-distributive reduce functions, we need to defer the applying of reduce function until all the required values which may be distributed in multiple consecutive keys are buffered. Therefore, we may need to buffer the values associated with multiple consecutive keys for non-distributive reduce functions.

Example 4.3: Consider using GGT to process three jobs J_1, J_2 and J_3 over the input table in Figure 4.3(a). Since $K_1 \prec K_2 \prec K_3$, the map output of J_3 can be reused for J_1 and J_2 and the map output of J_2 can be reused for J_1. In the map phase, for each

tuple in Figure 4.3(a), we properly tag and produce the map output for the three jobs. For example, for the tuple t_3, since it satisfies the selection conditions for J_1 and J_2 but not the selection conditions for J_3, we produce the map output key-value pair (15:15,(tag(12),5)) indicating that it is produced by J_2 and can be reused for J_1. Figure 4.3(b) shows the (key, value-list) layout in the reduce task [2] for all the map output.

In the reduce phase, when applying the reduce functions for each (key,value-list) pair, for the first three keys (i.e., 15, 15:15, 15:15:15), since the attribute values for each of the three keys does not change by comparing with the previous key, we just apply the reduce functions for them based on the tags. For example, for the second key 15:15, we aggregate the value 5 for both J_1 and J_2 since it is tagged tag(12). For the fourth key 15:15:20, compared to the previous key 15:15:15, the value of the third attribute c changes. Thus, before applying the reduce functions for the key 15:15:20, we need to output the results for a job if its number of map output key attributes is between the number of the changed attribute (i.e., 3) and the number of attributes in the previous key 15:15:15 (i.e., 3). Therefore, we output the results for J_3 for the key 15:15:15 (i.e., extract the first 3 attributes from the previous key 15:15:15). The same procedure is applied for the fifth key 15:20 (i.e., output the results for J_2 for the key 15:15 and J_3 for the key 15:15:20) and the sixth key 20 (i.e., output the results for J_1 for the key 15 and J_2 for the key 15:20). After examining all the (key,value-list) pair, we output the remaining results. In our example, we output the results for J_1 for the key 20. □

4.3.3 Materialization Techniques

In this section, we present an alternative approach, termed *materialization techniques (MT)*, for enabling multiple jobs to share map input scan and map output. Given a batch of jobs, the main idea of MT is to process the jobs in a specific sequence such that the map outputs of some of the preceding jobs can be materialized and used by the succeeding jobs in the sequence. There are two basic materialization techniques, namely, *map output materialization* and *reduce input materialization*, to enable sharing of map input scan and map output, respectively. Here again, we first present the techniques for processing two jobs and then discuss the generalization to handle more than two jobs.

Map Output Materialization (MOM). Our first materialization technique, which enables jobs J_i and J_j to share the scan of the map input file, requires that the input files and

[2] We assume there is only one reduce task for the jobs.

input key and value types of J_i and J_j to be the same. Assume that J_i is to be processed before J_j.

In the map phase of J_i, we read the map input file F to compute both the map output M_i for J_i as well as the map output M_j for J_j. M_j is materialized to the distributed file system (DFS) to be used later for processing J_j. The reduce phase of J_i is processed as usual.

In the map phase of J_j, instead of reading the map input file F a second time, we read the materialized map output M_j from the DFS. The reduce phase of J_j is processed as usual.

This simple materialization technique is beneficial if the total cost of materializing and reading M_j is lower than the cost of reading the input file F.

Reduce Input Materialization (RIM). Our second materialization technique aims to enable jobs J_i and J_j to share map output. This technique requires that $K_j \preceq K_i$, J_i to be processed before J_j, and the map output of J_i and J_j to be partitioned on K_j. The key idea of this technique is to materialize the map output $M_i^{A_j} \upharpoonright\!\!\!\upharpoonright M_j$ in the reduce phase of J_i, to be used later by the reduce phase of J_j. In this way, the sorting and communication cost of the map output $M_i^{A_j} \bigcap M_j$ is eliminated when processing J_j.

The map phase of J_i is processed as usual: we scan the input file F to produce the map output M_i for J_i. To enable the reduce phase of J_i to materialize $M_i^{A_j} \upharpoonright\!\!\!\upharpoonright M_j$ later, the map output M_i is tagged as follows: tuples in $M_{i,j}$ are tagged using $tag(ij)$ while the remaining tuples (i.e., tuples in $M_i \setminus M_{i,j}$) are tagged using $tag(i)$.

In the reduce phase of J_i, for each key, we apply the reduce function of J_i on the values associated with the key to produce the results of J_i. At the same time, for values that are tagged $tag(ij)$, we derive and materialize the sorted map output $M_i^{A_j} \upharpoonright\!\!\!\upharpoonright M_j$ into the DFS so that the materialized output will be later used by the reduce phase of J_j. Note that an optional combine function can be applied to reduce the size of the materialized map output $M_i^{A_j} \upharpoonright\!\!\!\upharpoonright M_j$ and hence the materializing and reading costs.

In the map phase of J_j, we scan the input file F to generate the partial map output $M_j \setminus M_i^{A_j}$ for J_j. The remaining map output of J_j (i.e., $M_{i,j}^{A_j}$) is not generated explicitly since they have already been sorted and materialized by J_i's reduce phase.

In the reduce phase of J_j, we first read the materialized map output $M_j \upharpoonright\!\!\!\upharpoonright M_i^{A_j}$ from DFS and merge them with the map output that are shuffled from the map phase. Then for each

key, we apply the reduce function of J_j on the values associated with that key to produce the results of J_j.

Thus, RIM reduces the sorting and communication costs for J_j by reducing the size of J_j's map output, but incurs an additional cost to materialize and read $M_i^{A_j} \ltimes M_j$.

Combining MOM & RIM. Both MOM and RIM can be applied together as follows. In the map phase of J_i, besides producing the map output M_i for J_i, we also generate the map output $M_j \setminus M_i^{A_j}$ for J_j. $M_j \setminus M_i^{A_j}$ is materialized into the DFS to be reused later for J_j. Then we process J_i as before.

In the map phase of J_j, instead of reading from the input file F, we read the materialized map output $M_j \setminus M_i^{A_j}$ from DFS and simply redirect the read tuples as the map output. Then we process J_j as before. The question of whether MOM and RIM should used together is decided in a cost-based manner depending on whether the total cost of materializing and reading $M_j \setminus M_i^{A_j}$ is lower than the cost of reading the input file F.

Example 4.4: Consider the two jobs J_1 and J_2 again. As $K_1 \prec K_2$, MT is applicable to enable both jobs to share map input scan and map output. As the map output of J_2 can be reused for J_1, we process J_2 before J_1. In the map phase of J_2, for each tuple t from T, if $t.b \leq 20 \wedge t.a < 10$, we produce the key-value pair (t.a:t.b,(tag(2),t.d)); if $t.b \leq 20 \wedge t.a \geq 10$, we produce the key-value pair (t.a:t.b,(tag(12),t.d)); if $t.b > 20 \wedge t.a \geq 10$, we produce the key-value pair $(t.a, t.d)$ and materialize it into DFS to be reused later for J_1 to share map input scan; otherwise, we do not produce any map output for that tuple. In the reduce phase of J_2, for each key, we sum the values associated with the key to produce the results of J_2. At the same time, for each specific key $t.a_i$:$t.b_i$, for all the values $<v_1, \cdots, v_n>$ associated with the key and tagged by $tag(12)$, we materialize $(t.a_i, \sum_{i=1}^{n} v_i)$ into DFS to be reused later for J_1 to share map output. When processing J_1, in the map phase, we read the materialized map output and sort and partition them. In the reduce phase, we first read the materialized map output and merge them with the map output shuffled from the map phase. Then for each key, we sum the values associated with the key to produce the results of J_1. □

Generalization. Given a batch of jobs $\mathcal{J} = \{J_1, J_2, \cdots, J_n\}$ sorted in non-ascending order of $|A_i|$, MT processes the jobs sequentially based on this ordering since the map output of a preceding job can possibly be reused for a succeeding job.

When processing J_1, in the map phase, we first produce NM_1 for J_1 (which is simply M_1), and tag each tuple t accordingly depending on the subset of remaining jobs in $\mathcal{J}$ that t can be used to derive their map outputs. Then for each J_i $(1 < i \leq n)$, if the cost of materializing and reading NM_i is lower than the cost of reading the input file F, we produce, tag, and materialize NM_i for J_i. In the reduce phase, when applying the reduce function for J_1, for each J_i $(1 < i \leq n)$, based on the tags in the values, we materialize the map output $NM_1^{A_i} \pitchfork M_i$ to be reused later for J_i.

When processing J_i $(1 < i \leq n)$, in the map phase, if NM_i has been materialized, we read NM_i and simply redirect the read tuples as the map output; otherwise, we read the input file F to produce and tag the map output NM_i for J_i. In the reduce phase, we first merge NM_i with the map output that are materialized by the previous jobs (i.e., $NM_j^{A_i} \pitchfork M_i$ for each $j \in [1, i-1]$) and then process the reduce function of J_i. When processing the reduce function of J_i, for each J_j $(i < j \leq n)$, based on the tags in the values, we materialize the map output $NM_i^{A_j} \pitchfork M_j$ to be reused later for J_j.

4.3.4 Discussions

In this section, we compare the proposed techniques, discuss the choices for map output keys and show how our proposed techniques apply to multi-input jobs.

Comparison of techniques. Our GGT generalizes and subsumes MRShare's grouping technique. However, there is no clear-cut winner between GGT and MT. Since GGT merges a group of jobs into a single new job, it requires the map output key and value types of the group of jobs to be the same, which may require a type conversion overhead. Moreover, GGT also incurs a higher sorting cost due to the larger map output of the merged job. On the other hand, MT has the limitation that the jobs within a group must be executed sequentially, and MT also incurs the overhead of result materialization and subsequent reading of the materialized results.

Choices for map output keys. For both GGT and MT, the choice of the map output key (i.e., ordering of A_i that specifies the map output key K_i for a job J_i) is important as it affects the sharing opportunities among jobs. For example, consider the jobs J_1, J_2 and J_5 in Table 4.1. Observe that there are two alternative map output keys for J_2: if we choose K_2 to be (a,b), we can share map output for J_1 and J_2; otherwise, with $K_2 = (b, a)$, we can share map output for J_5 and J_2. Thus, to optimize the sharing benefits for a given batch

of jobs, we need to determine the map output key for each job; we defer a discussion of this optimization to Section 4.5.

Handing multi-input jobs. Our proposed techniques can be easily extended to handle multi-input jobs as well. Consider the two jobs J_6 and J_7 in Table 4.1 which have the common input files T and R. For both T and R, the map output key of J_6 is a proper prefix of the map output key of J_7. Therefore, we can apply MT to share both the map input scan as well as map output for the two jobs. Furthermore, by converting the map output keys of the two jobs into the same type, MRShare's grouping technique can share the map input scan for the two jobs while our GGT can share both the map input scan and map output for the two jobs.

4.4 Cost Model

In this section, we present a cost model to estimate the evaluation cost of a batch of jobs $\mathcal{J} = \{J_1, J_2, \cdots, J_n\}$ in the MapReduce framework using the proposed techniques. Similar to MRShare, we model only the disk and network I/O costs as these are the dominant cost components. However, our cost model can be extended to include the CPU cost as well. Table 4.2 shows the system parameters used in our model, where the disk and network I/O costs are in units of seconds to process a page.

We assume the jobs in $\mathcal{J}$ are sorted in non-ascending order of $|A_i|$ and each $J_i \in \mathcal{J}$ is processed as m map tasks and r reduce tasks on the input file F. We use $|R|$ to denote the size of R in terms of number of pages, where R can be an input file or map/reduce output of some job. For a map output M_i, we use $p^m_{M_i} = \lceil \log_D \lceil \frac{|M_i|}{mB_m} \rceil \rceil$ to denote the number of sorting passes of its map tasks where $\frac{|M_i|}{m}$ denotes the average size of a map task, $p^r_{M_i} = \lceil \log_D \lceil \frac{|M_i|}{rB_r} \rceil \rceil - 1$ to denote the number of sorting passes of its reduce tasks where $\frac{|M_i|}{r}$ denotes the average size of a reduce task [3], and p_{M_i} to denote the sum of $p^m_{M_i}$ and $p^r_{M_i}$.

[3] The final merge pass optimization is enabled for sorting in Hadoop's reduce phase.

Table 4.2: System parameters

Parameter	Meaning
C_{lr}	cost of reading a page from local disk
C_{lw}	cost of writing a page to local disk
C_l	sum of C_{lr} and C_{lw}
C_{dr}	cost of reading a page from DFS
C_{dw}	cost of writing a page to DFS
C_d	sum of C_{dr} and C_{dw}
C_t	network I/O cost of a page transfer
D	merge order for external sorting
B_m	buffer size for external sorting at mapper nodes
B_r	buffer size for external sorting at reducer nodes

4.4.1 A Cost Model for MapReduce

Given a job J_i, its total cost (denoted as C_{j_i}) consists of its map and reduce costs (denoted as C_{M_i} and C_{R_i} respectively). The map cost is given by:

$$C_{M_i} = C_{dr}|F| + C_{lw}|M_i| + C_l p^m_{M_i}|M_i| \tag{4.1}$$

where $C_{dr}|F|$ denotes the cost of reading the input file, $C_{lw}|M_i|$ denotes the cost of writing the initial runs of the map output, and $C_l|M_i|p^m_{M_i}$ denotes the cost of sorting the initial runs.

The reduce cost is given by:

$$C_{R_i} = C_t|M_i| + C_l p^r_{M_i}|M_i| + C_{lr}|M_i| \tag{4.2}$$

where $C_t|M_i|$ denotes the transfer cost of the map output, $C_l|M_i|p^r_{M_i}$ denotes the sorting cost of the map output, and $C_{lr}|M_i|$ denotes the reading cost for the final merge pass. We do not include the cost of writing the job results since this cost is common to all the proposed techniques.

Therefore, the total cost can be expressed as follows:

$$C_{R_i} = C_{dr}|F| + (C_t + C_l + C_l p_{M_i})|M_i| \tag{4.3}$$

Our cost model for Hadoop has one major difference from MRShare's cost model. In MRShare's model, the number of initial runs for sorting in the reduce phase is assumed to

be equal to the number of map tasks (i.e., m). Based on this assumption, using the grouping technique does not increase the sorting cost in the reduce phase. However, in practice, Hadoop's reduce phase actually merges the transferred map output in main memory based on B_r to build initial runs which implies that using the grouping technique could increase the sorting cost in the reduce phase. Our cost model does not have this simplifying assumption and it is therefore more accurate than MRShare's model. In our performance evaluation, we apply our more accurate cost model to MRShare's GT technique as well so that all the techniques are compared based on the same cost model.

4.4.2 Costs for the Proposed Techniques

In this section, we use the above cost model to estimate the costs for the naive technique and our proposed GGT (which subsumes MRShare's GT technique) as well as MT techniques.

Naive technique: The naive technique processes each job independently. Thus, the cost of the naive technique is simply the sum of the cost of each job which is given by:

$$C_A = nC_{dr}|F| + (C_t + C_l)\sum_{i=1}^{n}|M_i| + C_l\sum_{i=1}^{n}p_{M_i}|M_i| \tag{4.4}$$

Generalized grouping technique: GGT combines the batch of jobs $\mathcal{J}$ into a single new job whose map output is denoted as $NM = \bigcup_{i=1}^{n} NM_i$. Thus, the cost of GGT is given by:

$$C_G = C_{dr}|F| + (C_t + C_l + C_l p_{NM})|NM| \tag{4.5}$$

Materialization technique: MT processes the jobs in $\mathcal{J}$ sequentially in non-ascending order of $|A_i|$ and materialize and reuse the map output as we have described in Section 3.3. Thus the cost of MT is given by:

$$\begin{aligned} C_M = C_{dr}|F| + \sum_{i=2}^{n}\min\{C_{dr}|F|, C_d|NM_j|\} + (C_t + C_l)|NM| \\ + C_l\sum_{i=1}^{n}p_{NM_i}|NM_i| + C_d\sum_{i=1}^{n-1}\sum_{j=i+1}^{n}|NM_i^{A_j} \bigoplus M_j| \end{aligned} \tag{4.6}$$

Note that $\sum_{i=2}^{n} \min\{C_{dr}|F|, C_d|NM_j|\}$ denote the materialization and reading cost in the map phase, and $C_d \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} |NM_i^{A_j} \leftthreetimes M_j|$ denote the materialization and reading cost in the reduce phase.

4.5 Optimization Algorithms

In this section, we discuss how to find an optimal evaluation plan for a batch of jobs $\mathcal{J} = (J_1, J_2, \cdots, J_n)$.

An evaluation plan for $\mathcal{J}$ specifies the following: (1) the map output key K_i for each job $J_i \in \mathcal{J}$; (2) a partitioning of the jobs in $\mathcal{J}$ into some number of disjoint groups, $G_1, \cdots, G_k$, where $k \geq 1$ and $\mathcal{J} = G_1 \cup \cdots \cup G_k$; and (3) a processing technique T_i for evaluating the jobs in each group G_i. Since MRShare's grouping technique is subsumed by GGT, and the naive evaluation technique is equivalent to partitioning $\mathcal{J}$ into n groups each of which consists of a single job that is processed by GGT, we can simply consider only GGT or MT for each T_i.

Let Cost(G_i, T_i) denote the cost of evaluating the group of jobs $G_i \subseteq \mathcal{J}$ with technique $T_i \in \{GGT, MT\}$. The estimation of Cost(G_i, T_i) has already been discussed in Section 4.4.

The optimization problem is to find an evaluation plan for $\mathcal{J}$ such that the total evaluation cost $\sum_{i=1}^{k}$ Cost(G_i, T_i) is minimized. A simpler version of this optimization problem was studied in MRShare and shown to be NP-hard. The problem is simpler in MRShare for two reasons: first, MRShare considers only the naive and grouping techniques; and second, MRShare does not have to consider the selection of the map output keys as this does not affect the sharing opportunities for the grouping technique. As a result, the heuristic approach in MRShare can not be extended for our more complex optimization problem.

To cope with the complexity of the problem, we present a two-phase approach to optimize the evaluation plan. In the first phase, we choose the map output key for each job to maximize the sharing opportunities among the batch of jobs. In the second phase, we partition the batch of jobs into groups and choose the processing technique for each group to minimize the total evaluation cost.

4.5.1 Map Output Key Ordering Algorithm

In this section, we discuss how to choose the map output key for each job (i.e., determine the ordering of the key attributes) to maximize the sharing opportunities for a batch of jobs. To quantify the sharing opportunities for a batch of jobs $\mathcal{J}$, we use the notion of the *non-derivable map output* for $\mathcal{J}$, denoted by NM, that was defined in Section 4.3.2. Since a smaller size of NM represents a larger amount of sharing among the jobs in $\mathcal{J}$, to maximize the sharing among the jobs in $\mathcal{J}$, the map output key for each job is chosen to minimize the size of NM.

A naive solution to optimize this problem is to enumerate all the combinations of map output keys for the jobs and choose the combination that minimizes the size of NM. However, the time complexity of this brute-force solution is $O(|A_1|!|A_2|! \cdots |A_n|!)$ which is infeasible for large number of jobs[4]. In this work, we propose a greedy heuristic to optimize the map output key for each job.

Our greedy algorithm determines the ordering of the map output key attributes for each job J_i progressively by maintaining a list of sets of attributes, referred to as the ordering list (denoted by OL_i), to represent the ordering relationship for the map output key attributes of J_i. The attributes within a set are unordered, and the attributes in a set S are ordered before the attributes in another set S' if S appears before S' in the list. We use $|OL_i|$ to denote the number of sets in OL_i. For example, in the ordering list $<\{a, b, c\}, \{d\}>$, the attributes in $\{a, b, c\}$ are unordered and they precede the attribute d. Furthermore, given two jobs J_i and J_j, we use $OL_i \preceq OL_j$ to represent that OL_i is a prefix of OL_j, i.e., for each $i \in [1, |OL_i|]$, the i^{th} sets in OL_i and OL_j are the same. For example, $<\{a, b\}, \{c\}> \preceq <\{a, b\}, \{c\}, \{d\}>$.

Besides maintaining OL_i for each job J_i, our approach also maintains a *reuse set*, denoted by RS_i, for each job J_i. The purpose of RS_i is to keep track of all the jobs that can be reused for computing the map output of J_i.

Initially, as we have not chosen any jobs to share map output, the size of NM is simply the sum of each job's map output size. Furthermore, for each $J_i \in \mathcal{J}$, we initialize OL_i to be a list with a single set containing all the attributes in A_i and initialize RS_i to be empty. We then construct a weighted, undirected graph $G = (V, E)$ to represent all the potential sharing opportunities in $\mathcal{J}$ as follows. Each $J_i \in \mathcal{J}$ is represented by a vertex in V. An

[4]For example, we experimented with a batch of 25 randomly generated jobs each with a maximum of four attributes in its map output key, and the brute-force approach did not complete running in 12 hours.

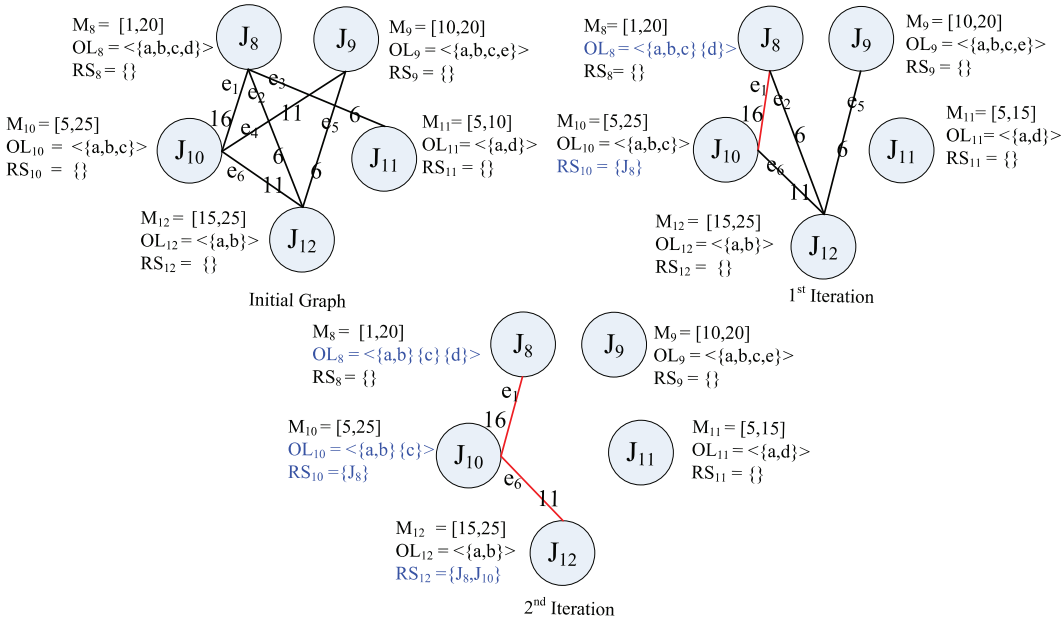


Figure 4.4: An example to illustrate key ordering algorithm.

edge $e = (J_i, J_j)$ is in E if there exists two map output keys K_i and K_j, respectively, for J_i and J_j such that the map output of one job can be reused for the other job (i.e., $K_i \preceq K_j$ or $K_j \preceq K_i$). The weight of (J_i, J_j) is initialized to be the reused map output size for the two jobs (i.e., $|M_i^{A_j} \bigcap M_j|$ if $K_j \preceq K_i$ or $|M_j^{A_i} \bigcap M_i|$ if $K_i \preceq K_j$). All the edges in E are initialized to be unmarked.

Figure 4.4 shows an example of the initial graph constructed for a batch of five jobs $\{J_8, \cdots, J_{12}\}$. For ease of presentation, we use an interval of integers to represent the map output of a job where the size of an integer is 1. For example, the map output size of J_8 is 20 since it contains 20 integers in its map output $[1, 20]$. The initial graph contains the edge $e_1 = (J_8, J_{10})$ since there exists $K_{10} = (a, b, c)$ and $K_8 = (a, b, c, d)$ such that $K_{10} \preceq K_8$; moreover, the weight of e_1 is 16 since there are 16 values (i.e., [5,20]) in the map output of J_8 that can be reused for J_{10}.

For convenience, we use E_{J_i} to denote the set of all the unmarked edges incident on a node $J_i \in V$, and use N_{J_i} to denote the set of all the vertices that have a marked edge with a node $J_i \in V$.

Overall algorithm. Given an initial graph $G = (V, E)$, to reduce the size of NM, our greedy approach iteratively selects and marks one edge from the graph G until all the edges in G have been marked. Algorithm 4.2 shows the pseudocode of our greedy approach. At each iteration, it first chooses an unmarked edge with the maximum weight

Algorithm 4.2: Key Ordering Algorithm

```
Input: An initial graph G = (V, E)
Output: Map output key for each job
while E has unmarked edge do
    choose an unmarked edge e_max ∈ E with the maximum weight to share and mark it ;
    V_1 = nodes whose ordering lists change for e_max ;
    V_2 = nodes whose reusing sets change for e_max ;
    foreach J_i in V_1 do
        foreach e in E_{J_i} do
            if e is not valid then
                remove e from E ;
    foreach J_i in V_2 do
        foreach e in E_{J_i} do
            update the weight for e ;
foreach J_i in V do
    derive the map output key for J_i ;
```

(i.e., the chosen edge represents the largest sharing opportunity and maximizes the reduction of the size of NM) to share and marks the edge. Then based on the chosen edge, it updates the ordering lists and reusing sets for some jobs. We refer to V_1 and V_2 as the set of jobs whose ordering lists and reuse sets, respectively, have been changed in the updating. Finally, for each $J_i \in V_1$, we check the edge validity for all the edges in E_{J_i} and remove the invalid edges (to be explained). For each $J_i \in V_2$, we update the weights for all the edges in E_{J_i} (to be explained). After the iterative process terminates, we derive the map output key for each job based on its ordering list.

In the following, we explain how the graph is updated in each iteration and how the map output key is derived at the end of the iterative process.

Updating ordering lists. Suppose that the edge $e = (J_i, J_j)$ is selected in an iteration. We first update the ordering lists for J_i and J_j. Then for each job $J_k \in \{J_i, J_j\}$, if the ordering list of J_k has changed, we also update the ordering lists for the jobs in N_{J_k} and recursively propagate the updating for the jobs in N_{J_k} whose ordering lists have changed until all the jobs have been examined or there is no more job whose ordering list has changed.

Given an edge $e = (J_i, J_j)$, the main idea to update OL_i and OL_j is to ensure that after the updating, one ordering list is a prefix of the other ordering list (i.e., $OL_i \preceq OL_j$ or $OL_j \preceq OL_i$). For example, the first iteration chooses $e_1 = (J_8, J_{10})$ to share since the weight of e_1 is the highest, and since $OL_8 = <\{a, b, c, d\}>$ and $OL_{10} = <\{a, b, c\}>$, OL_8 is updated to $<\{a, b, c\}, \{d\}>$ to ensure that $OL_{10} \preceq OL_8$. Therefore, to update

Algorithm 4.3: Update Ordering Lists

```
   Input: An edge e = (J_i, J_j)
   Output: Updated ordering lists for J_j and J_j
1  i = 1 ;
2  while i ≤ Min(|OL_i|, |OL_j|) do
3      set1 = OL_i.get(i) ;
4      set2 = OL_j.get(i) ;
5      if set1.equals(set2) then
6          continue ;
7      else if set1.containAll(set2) then
8          set1.removeAll(set2);
9          OL_i.insert(i, set2) ;
10     else if set2.containAll(set1) then
11         set2.removeAll(set1) ;
12         OL_j.insert(i, set1) ;
13     i++;
```

OL_i and OL_j, we iterate through the sets in OL_i and OL_j and accordingly decompose the corresponding sets to maintain the prefix relationship between the two lists. Algorithm 4.3 shows the pseudocode of this updating. The time complexity for this updating is $O(m)$, where m is the maximum number of map output key attributes in a job. Since m is usually very small, we assume this checking can be done in $O(1)$ time.

For example, in Figure 4.4, the first iteration chooses the edge $e_1 = (J_8, J_{10})$ to share. Then OL_{10} and OL_8 are updated as follows: OL_{10} does not change and OL_8 becomes $<\{a, b, c\}, \{d\}>$. The second iteration chooses the edge $e_6 = (J_{10}, J_{12})$, and OL_{12} and OL_{10} are updated as follows: OL_{12} does not change and OL_{10} becomes $<\{a, b\}, \{c\}>$ which triggers the updating for OL_8 since J_8 has a marked edge with J_{10}. Then we update OL_8 to be $<\{a, b\}, \{c\}, \{d\}>$.

Updating reuse sets. The updating of reuse sets is also done recursively similar to the updating of ordering lists. Therefore, we focus on explaining the updating of reuse sets for two jobs.

Given an edge $e = (J_i, J_j)$, the main idea to update RS_i and RS_j is as follows. If $A_i \subset A_j$, we update RS_i by adding the jobs in $RS_j \cup \{J_j\}$ into the set RS_i since all the jobs in $RS_j \cup \{J_j\}$ can be reused for J_i. Similarly, if $A_j \subset A_i$, we update RS_j by adding the jobs in $RS_i \cup \{J_i\}$ into the set RS_j since all the jobs in $RS_i \cup \{J_i\}$ can be reused for J_j. Otherwise, we have $A_i = A_j$, and we update both RS_i and RS_j by assuming that the map output of J_j will be reused for J_i as follows. Let S denote a copy of RS_i. We update

RS_i by adding the jobs in $RS_j \cup \{J_j\}$ into RS_i, and update RS_j by adding the jobs in S into RS_j. The time complexity of the updating is $O(1)$.

After updating the ordering lists and reuse sets as described above, we then use the updated information to update the graph G; this includes identifying invalid edges (to be defined) in G, and updating some edge weights.

Identifying invalid edges. For a job $J_i \in V_1$, since OL_i has changed, for each $e \in E_{J_i}$, we need to check whether e is still a valid edge. An unmarked edge $e = (J_i, J_j)$ in G is defined to be a *valid edge* if we can derive two map output keys K_i and K_j, respectively, for J_i and J_j from OL_i and OL_j such that $K_i \preceq K_j$ or $K_j \preceq K_i$ (i.e., we can share map output for the two jobs); otherwise, e is considered an *invalid edge* and is removed from G.

Algorithm 4.4: Identifying Invalid Edges

```
   Input: An edge e = (J_i, J_j)
   Output: Whether e is a valid edge
 1 i = 1 ;
 2 OL'_i = newList(OL_i); OL'_j = newList(OL_j) ;
 3 while i ≤ Min(|OL'_i|, |OL'_j|) do
 4 |   set1 = OL'_i.get(i) ;
 5 |   set2 = OL'_j.get(i) ;
 6 |   i++ ;
 7 |   if set1.equals(set2) then
 8 |   |   continue ;
 9 |   else if set1.containAll(set2) then
10 |   |   OL'_i.insert(i, set1.removeAll(set2)) ;
11 |   else if set2.containAll(set1) then
12 |   |   OL'_j.insert(i, set2.removeAll(set1)) ;
13 |   else
14 |   |   return false ;
15 return true ;
```

We can check whether an unmarked edge $e = (J_i, J_j)$ is a valid edge or not as follows. If we can derive two ordering lists OL'_i and OL'_j respectively from OL_i and OL_j such that they satisfy the prefix relationship (i.e., $OL'_i \preceq OL'_j$ or $OL'_j \preceq OL'_i$), then the edge is a valid edge; otherwise, the edge is an invalid edge and can be removed from G. This detail process (given in Algorithm 4.4) is similar to the process of updating the ordering lists for two jobs, and the time complexity is also $O(1)$. For example, in Figure 4.4, after choosing e_1 to share in the first iteration, OL_8 becomes $<\{a, b, c\}\{d\}>$ which makes e_3 an invalid edge since OL_{11} is $<\{a, d\}>$.

Updating edge weights. For a job $J_i \in V_2$, since RS_i has changed, for each $e \in E_{J_i}$, we need to update the weight for e. If the updated weight is 0, we can simply remove the edge since sharing the edge will not reduce the size of NM.

Given an edge $e = (J_i, J_j)$, its weight is updated as follows. If $A_i \subset A_j$ (i.e., the map output of the jobs in RS_j can be reused for J_i), then the weight of e is updated to $|S_{i1}| - |S_{i2}|$, where $|S_{i1}|$ and $|S_{i2}|$ denote, respectively, the size of the map output that J_i needs to produce (i.e., the size of the map output of J_i that can not be reused from RS_i) before and after we share e. Note that both $|S_{i1}|$ and $|S_{i2}|$ are computed based on RS_i which has to be updated if we share e. Similarly, if $A_j \subset A_i$ (i.e., the map output of the jobs in RS_i can be reused for J_j), then the weight of the edge is updated to $|S_{j1}| - |S_{j2}|$, where $|S_{j1}|$ and $|S_{j2}|$ denote, respectively, the size of the map output that J_j needs to produce before and after we share e. Otherwise, we have $A_i = A_j$ (i.e., the map output of the jobs in RS_i and RS_j can be respectively reused for J_i and J_j), and the weight of the edge is updated to be $|S_{i1}| - |S_{i2}| + |S_{j1}| - |S_{j2}|$. The time complexity of this updating is $O(1)$.

For example, in Figure 4.4, after choosing e_1 to share in the first iteration, RS_{10} becomes $\{J_8\}$ which triggers the weight updating for the edges in $E_{J_{10}} = \{e_4, e_6\}$. Let us first consider e_4. After choosing e_1 to share, J_{10} only needs to produce the map output [21,25] (i.e., the remaining map output [5,20] can be reused from J_8) and the map output of J_9 can not be reused to reduce the map output [21,25] further. Therefore, the weight of e_4 decreases to 0 and e_4 is removed from the graph. Next, consider e_6. After choosing e_1 to share, both the map output of J_8 and J_{10} can be reused for J_{12}. However, the weight of e_6 remains the same since J_8 does not enable additional reusing for J_{12}.

Deriving map output key. Note that at the end of the iterative process, it is possible for some set in an ordering list OL_i to contain more than one attribute (i.e., the ordering of the key attributes for J_i is not yet a total ordering). To derive the map output key for J_i, we have to determine an ordering for the remaining partially ordered attributes. To correctly derive the ordering of key attributes for such scenarios, we make use of a default ordering for all the attributes. For example, in Figure 4.4, at the end of the iterative process (i.e., after we have chosen the edge e_6 to share), the ordering lists for the five jobs $J_8, \cdots, J_{12}$ all contain at least one set that have more than one attribute. Assuming that the default ordering for all the attributes is (a, b, c, d), then the map output keys for J_{12}, J_{10} and J_8 are, respectively, (a, b), (a, b, c), and (a, b, c, d), which captures all the sharing that our algorithm has chosen. Note that without using a default ordering, we could wrongly

choose the map output key (b, a) for J_{12} and the map output key (a, b, c) for J_{10} which does not allow these two jobs to share their map output.

Time Complexity. The time complexity of the algorithm depends on the number of iterations. The time complexity for the i^{th} iteration is $O(|E_i|)$, where $|E_i|$ is the number of edges in the graph in this iteration. Therefore, the time complexity of the algorithm is $O(In^2)$ where I is the number of iterations and $O(n^2)$ is the maximum number of edges in the graph.

4.5.2 Partitioning Algorithm

In this section, we discuss the second phase of our approach; i.e., how to partition a batch of jobs into multiple groups and choose the processing technique for each group to minimize the overall evaluation cost. We use the notation (G_i, T_i) to denote that a group of jobs G_i is being processed by a technique T_i. Recall that since GGT subsumes MRShare's grouping technique, and the naive evaluation technique is equivalent to partitioning the batch of jobs into single-job groups each of which is processed by GGT, it is sufficient to consider only the GGT and MT processing techniques.

Our partitioning algorithm is based on the concept of *merging benefit* which is defined as follows. Consider two groups of jobs, (G_1, T_1) and (G_2, T_2), where $G_1 \cap G_2 = \emptyset$. We define the *merging benefit* from (G_1, T_1) and (G_2, T_2) to $(G_1 \cup G_2, T_3)$, where $T_3 \in \{GGT, MT\}$, as Cost(G_1, T_1) + Cost(G_2, T_2) - Cost($G_1 \cup G_2, T_3$).

Our partitioning algorithm is a greedy approach that iteratively selects a pair of groups of jobs to be merged based on their merging benefit. Initially, each job is treated as a single-job group processed by GGT (which is equivalent to the naive technique since the group has only one job). At each iteration, it merges the two groups that have the maximum positive merging benefit into a new group. Note that when computing the cost for a merged group, as there are two techniques that we can process the group, i.e, the generalized grouping technique and materialization technique, we will compute the cost for both techniques and choose the better one for the group. The iterative process terminates when the maximum merging benefit is non-positive.

Note that the time complexity of the grouping algorithm is $O(n^2)$, where n is the number of jobs in the batch. In the first iteration, we compute the merging benefit for each pair of groups, and in each subsequent iteration, since there is only one new group produced in

Table 4.3: Compared algorithms

Notation	Algorithm
NA	Naive algorithm that evaluates each job independently
$MRGT$	MRShare's grouping technique combined with its own partitioning algorithm
GT	MRShare's grouping technique combined with our partitioning algorithm
GGT	Our generalized grouping technique combined with our optimization algorithm
MT	Our materialization technique combined with our optimization algorithm
$GGTMT$	Our generalized grouping and materialization techniques combined with our optimization algorithm

the previous iteration, we only need to compute the merging benefit for each group with the new group.

4.6 Experimental Results

In this section, we present an experimental study to evaluate our proposed approach. Section 4.6.1 examines the performance of our approach, Section 4.6.2 evaluates the effectiveness of our map output key ordering algorithm and Section 4.6.3 evaluates the efficiencies of our optimization algorithms.

Algorithms. We compared six algorithms (denoted by NA, $MRGT$, GT, GGT, MT, and $GGTMT$) in our experiments as shown in Table 4.3. The two competing algorithms were NA, which denote the naive approach of evaluating each job independently, and $MRGT$, which denote MRShare's grouping technique combined with its own partitioning algorithm. For $MRGT$, we experimented with two different implementation variants: the original variant [44], which uses only a single global tuning parameter $\gamma \in [0, 1]$ to quantify the sharing among all the jobs in a batch, and an enhanced variant which provides a more fine-grained and accurate approach to estimate job sharing using a tuning parameter $\gamma_{i,j}$ for each pair of jobs J_i and J_j. As our experimental results show that the enhanced variant strictly outperforms the original variant[5], we do not report results for the original variant and use $MRGT$ to denote the enhanced variant.

[5]For example, in the default setting, the running time for the enhanced variant was 3555s while that for the original variant was, 3820s, 3942s, 3931s, 3802s, 3885s, 3860s, 4385s, 4872s, and 4881s, respectively, for a γ value of 1, 0.9, 0.8, 0.7, 0.6, 0.5, 0.4, 0.3 and $\{0.2, 0.1, 0\}$.

Our three main proposed algorithms include: GGT, which denotes the generalized grouping technique (combined with our optimization algorithm); MT, which denotes the materialization technique (combined with our optimization algorithm); and $GGTMT$, which denotes the approach combining both GGT and MT. In addition, to demonstrate the effectiveness of our partitioning heuristic (Section 4.5.2), we also introduce a variant of $MRGT$, denoted by GT, which combines MRShare's grouping technique with our partitioning heuristic.

Datasets and Queries. We used synthetic datasets and queries for our experiments. The schema of the datasets was *Data (key char(8), dim1 char(20), dim2 char(20), dim3 char(20), dim4 char(20), range int, value int)* which consisted of one unique key attribute, four dimensional attributes used as group-by attributes, one range attribute used as the selection attribute, and one value attribute used as the aggregation attribute. Each of the four dimensional attributes had 500 distinct values and all the attribute values were uniformly distributed. The datasets were stored as text format and the size of each tuple was about 100 bytes. The default dataset had 1.7 billion tuples with a size of 160GB.

The synthetic queries were generated from the following template: ***select*** *T, sum(value)* ***from*** *Data* ***where*** $a \leq range \leq b$ ***group by*** *T*, where T was a randomly selected list of dimensional attributes, and a and b were randomly selected values such that $a \leq b$. The default number of queries in a query batch was 20. Each batch of queries was run three times and we report their average running times.

Experimental environment. Our experiments were performed using Hadoop 1.0.1 on a cluster of nodes that were interconnected with a 1Gbps switch. Each node was equipped with an Intel X3430 2.4GHz processor, 8GB memory, 2x500G SATA disks and running CentOS Linux 5.5. The default cluster size was 41 (with 1 master node and 40 slave nodes).

Hadoop configuration. The following Hadoop configuration was used for our experiments: (1) the heap size of JVM running was 1024MB; (2) the default split size of HDFS was 512MB; (3) the data replication factor of HDFS was 3; (4) the I/O buffer size was 128KB; (5) the memory for the map-side sort was 200MB; (6) the space ratio for the intermediate metadata was 0.4; (7) the maximum number of concurrent mappers and the maximum number of concurrent reducers for each node was both 2; (8) the number of reduce tasks was 240; (9) speculative execution was disabled[6]; (10) JVM reuse was enabled; and (11) the default FIFO scheduler was used which supports concurrent execution

[6]Speculative execution is typically disabled in a busy cluster due to its negative impact on perfor-

of jobs; note that for MT, while the jobs within a group were executed sequentially, jobs from different groups were executed concurrently.

Cost model parameters. We ran some I/O benchmarks in the cluster to calibrate our cost model parameters as follows: the cost ratio of local read/write is 1, the cost ratio for DFS read and write are, respectively, 1 and 2 (due to replication factor), and the cost ratio of network I/O is 1.4. Note that the setting of the same cost ratio for both local and DFS reads is reasonable due to the data locality property of the MapReduce framework.

Summary of results. First, our algorithms (GT, GGT, MT, $GGTMT$) significantly outperform NA by up to 167% and $MRGT$ by up to 107%. In particular, GT outperforms $MRGT$ by up to 31% demonstrating the effectiveness of our partitioning algorithm against MRShare's partitioning algorithm. Second, among our algorithms, GT performs the worst, and there is no clear winner between GGT and MT (as explained in Section 4.3): GGT outperforms MT by up to 24% for some cases and MT outperforms GGT by up to 12% for other cases. The overall winning approach is $GGTMT$ which outperforms the best of GGT and MT very slightly. Given this, to avoid cluttering the graphs, we do not explicitly show $GGMT$ in the graphs as its performance is approximated by the best of GGT and MT. Finally, our results show that the optimization overhead incurred by our approach is only a negligible fraction of the total processing time. Thus, the optimization overhead of our approach is negligible even if the queries do not have any sharing opportunities.

4.6.1 Performance Comparison

In this section, we evaluate the effectiveness of our optimization algorithms by varying four parameters, i.e., data size, split size, number of queries and cluster size. Figure 4.5 shows the experimental results with the the improvement factors (in %) of GGT, MT, GT and $MRGT$ over NA indicated.

Effect of number of queries. Figure 4.5(a) compares the performance as the size of a query batch is increased. Observe that our algorithms significantly outperform NA and $MRGT$. For example, GGT outperforms NA by 105% on average and up to 167% when

mance [66]. Indeed, in our preliminary experiments with speculative execution enabled, we observed that the performance of all the algorithms degraded. For example, in the default setting, the running times for both NA and $MRGT$ increased by 10% while that for GGT and MT increased by 6%. Thus, the winning margin of our algorithms increased slightly over NA and $MRGT$ with speculative execution enabled.

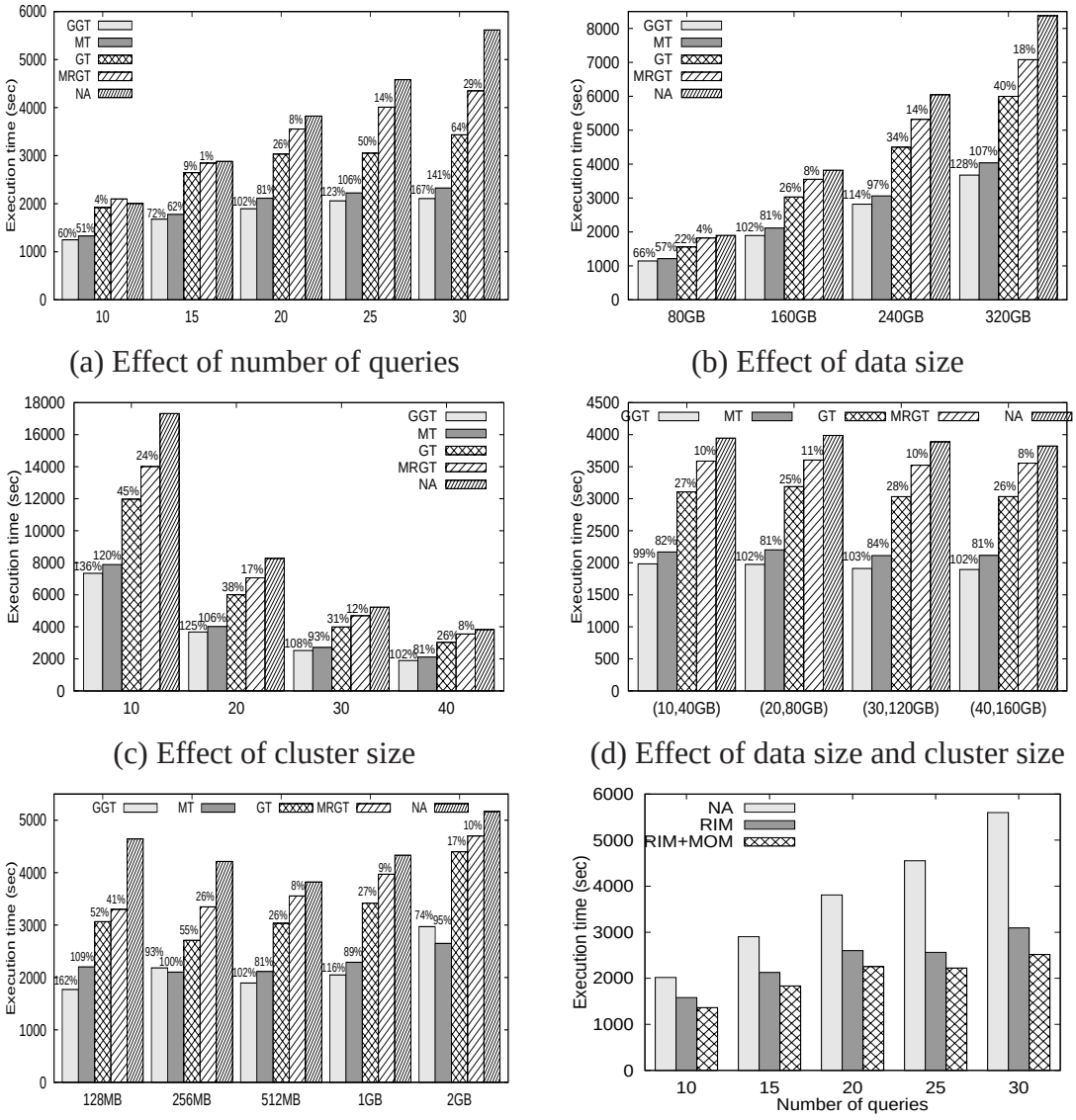
the number of queries is 30, and GGT outperforms $MRGT$ by 85% on average and up to 107% when the number of queries is 30. Furthermore, as the number of queries increases, the winning margin of our algorithms over NA also increases. This is expected as the sharing opportunities among queries also increase with the query batch size.

Effect of data size. Figure 4.5(b) examines the performance as a function of data size. Note that as we increase the data size, we also increase the number of reduce tasks. This is reasonable as the number of reduce tasks is usually proportional to the data size, as noted also in [5, 68]. Therefore, we set the number of reduce tasks to be 120, 240, 360, and 480, respectively, for data size of 80GB, 160GB, 240GB, and 320GB.

Here again, our algorithms significantly outperform NA and $MRGT$. For example, GGT outperforms NA by 103% on average and up to 128% when the data size is 320GB, and GGT outperforms $MRGT$ by 82% on average and up to 93% when the data size is 320GB. Furthermore, as the data size increases, the running time for the algorithms also increases. In particular, the running time for NA increases much faster than for the other algorithms which therefore increases the winning margin of the other algorithms over NA. The reason behind this is that by partitioning the queries into groups, the non-NA algorithms are more scalable. For example, in the default setting (with a batch of 20 queries), NA needs to scan the input table 20 times while GGT, which has partitioned the batch of queries into two groups, only needs to scan the input table twice.

Effect of cluster size. Figure 4.5(c) compares the effect of number of slave nodes in the cluster. Here again, our algorithms significantly outperform NA and $MRGT$. For example, GGT outperforms NA by 118% on average and up to 136% when the number of nodes is 10, and GGT outperforms $MRGT$ by 89% on average and up to 92% when the number of nodes is 10 (the improvement factor of GGT over $MRGT$ does not show significant differences for all the node sizes). Furthermore, as the cluster size increases, the running time for all the algorithms decreases. In particular, the running time for NA decreases much faster than for the other algorithms which therefore reduces the winning margin of the other algorithms over NA as cluster size increases. Thus, the performance improvement from the increased parallelism using a larger cluster benefits the non-optimized NA more than the already optimized non-NA algorithms.

Effect of both data size and cluster size. Besides studying the effect of the data size and cluster size parameters separately, we also conducted an additional experiment to examine the joint effect of both these parameters. In Figure 4.5(d), a cluster size of 10, 20, 30, and 40 slave nodes was used, respectively, for a data size of 40GB, 80GB, 120GB, and



(a) Effect of number of queries

(b) Effect of data size

(c) Effect of cluster size

(d) Effect of data size and cluster size

(e) Effect of split size

(f) Analysis of MT

Figure 4.5: Effectiveness of optimization algorithms

160GB. As the results show, the performance of each algorithm does not vary very much as both the cluster size and data size jointly increase; this demonstrates the scalability of our algorithms wrt these two parameters.

Effect of split size. Figure 4.5(e) compares the effect of the split size. Here again, our algorithms significantly outperform NA and $MRGT$. For example, our best algorithm (i.e., GGT or MT) outperforms NA by 115% on average and up to 162% when the split size is 128MB, and our best algorithm (i.e., GGT or MT) outperforms $MRGT$ by 81% on average and up to 94% when the split size is 1GB. Observe that there is no clear winner between GGT and MT as explained in Section 4.3. For NA, we observe that its running time decreases with increasing split size until a certain threshold (e.g., 512MB for NA) after which its running times increases. This is because when the split size is too small, more map tasks will be launched for processing the job which incurs a higher startup cost; on the other hand, when the split size is too large, each map task will process more data which increases its sorting cost.

Analysis of MT. In this experiment, we analysis the relative effectiveness of the two techniques, MOM and RIM, that form MT. Figure 4.5(e) compares NA against two variants of MT: MT itself (denoted explicitly as RIM+MOM) and MT with only RIM technique (denoted as RIM). As the results show, RIM is more effective than MOM in reducing the running time. However, by further combining with MOM, we can improve the performance of RIM by 17% on average and up to 23% when the number of queries is 30.

4.6.2 Effectiveness of Key Ordering Algorithm

In this section, we evaluate the effectiveness of our key ordering algorithm (denoted by *Pka*) by comparing against two extreme solutions: a brute-force algorithm that generates the optimal key ordering (denoted by *Oka*) and a naive heuristic that uses a random key ordering (denoted by *Rka*).

Recall from Section 4.5.1 that our map output key ordering algorithm is designed to maximize job sharing by minimizing the size of the non-derivable map output (denoted by NM) for the input batch of jobs. To assess its effectiveness, we compare two ratios, $\frac{|NM_{Rka}|-|NM_{Pka}|}{|NM_{Pka}|}$ and $\frac{|NM_{Pka}|-|NM_{Oka}|}{|NM_{Oka}|}$, where $|NM_x|$ denote the size of the non-derivable map output for an input batch of queries using algorithm x, $x \in \{Pka, Oka, Rka\}$. The

Table 4.4: Comparison of key ordering algorithms

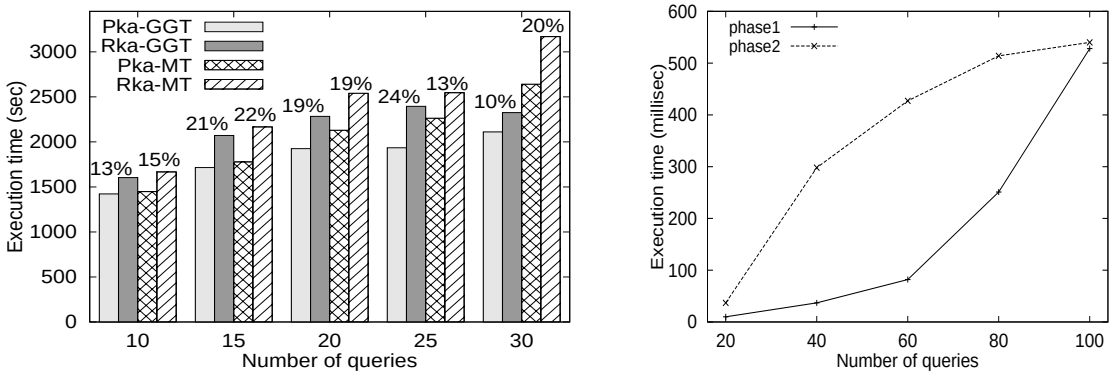
Number of Queries	$\frac{\lvert NM_{Rka}\rvert - \lvert NM_{Pka}\rvert}{\lvert NM_{Pka}\rvert} \times 100\%$			$\frac{\lvert NM_{Pka}\rvert - \lvert NM_{Oka}\rvert}{\lvert NM_{Oka}\rvert} \times 100\%$		
	Min	Max	Avg	Min	Max	Avg
10	10%	26%	16%	0	8%	3%
15	11%	20%	18%	0	7%	2%
20	16%	25%	19%	1%	2%	1%
25	16%	20%	19%	–	–	–
30	14%	22%	19%	–	–	–

first ratio measures the improvement factor of Pka over Rka, while the second ratio measures the improvement factor of Oka over Pka.

Table 4.4 compares these two ratios for various query sizes. For each query size, we randomly generate five batches of queries and report the average, minimum, and maximum values of the ratios. From Table 4.4, the $\frac{|NM_{Rka}|-|NM_{Pka}|}{|NM_{Pka}|}$ values show that our key ordering heuristic is indeed effective in minimizing $|NM|$ compared to the naive random ordering heuristic, while the $\frac{|NM_{Pka}|-|NM_{Oka}|}{|NM_{Oka}|}$ values show that our heuristic is almost as effective as the brute-force approach. Note that for query sizes 25 and 30, we were not able to compute values for $\frac{|NM_{Pka}|-|NM_{Oka}|}{|NM_{Oka}|}$ as Oka did not finish running in 12 hours. Indeed, as expected, Oka is not a scalable solution: for a query size of 20, Oka took about 3 hours to run compared to only 50ms taken by our heuristic Pka.

To evaluate the effectiveness of the key ordering heuristics in terms of their impact on query evaluation time (excluding optimization time), we also compared their running times to evaluate query batches of difference size. In the following, we use the notation X-Y to denote the evaluation algorithm Y when used in combination with the key ordering heuristic X, where $Y \in \{GGT, MT\}$ and $X \in \{Pka, Rka, Oka\}$. Note that the evaluation algorithms NA, $MRGT$, and GT were excluded from the comparison as these algorithms do not require the key ordering step.

Figure 4.6(a) shows the running times for a representative query batch where its value of $\frac{|NM_{Rka}|-|NM_{Pka}|}{|NM_{Pka}|}$ ratio is ranked in the middle among the five batches. As the performance of Oka-Y is very close to that of Pka-Y (e.g., the former outperforms the latter by only 0.7% in the best case), we omit the results for Oka-Y in the graph. For each query size, Figure 4.6(a) also indicates two improvement factors (in %) which represent the performance improvement of Pka-Y over Rka-Y, $Y \in \{GGT, MT\}$. The results show that for both GGT and MT, Pka outperforms Rka by 17% on average.



(a) Comparison of key ordering algorithms

(b) Optimization Time

Figure 4.6: Experimental results

4.6.3 Optimization vs Evaluation time

In this section, we quantify the optimization overhead of our approach and show that the overhead incurs only a very small fraction of the total query processing time. Since the optimization times for our algorithms do not show much differences, we here only report the optimization time for $GGTMT$.

Figure 4.6(b) shows the optimization time for $GGTMT$ as a function of query size. Note that we separately report the optimization times for the two phases of our algorithms. As shown from the figure, the optimization algorithms are very fast. Indeed, by comparing with the evaluation time for the queries, the optimization time can even be ignored. For example, in the default setting, the evaluation time for 20 queries for our best algorithm (i.e., GGT) takes 1895 seconds while the optimization time only takes 50 milliseconds for 20 queries and 1 second for 100 queries. Therefore, our algorithms are very efficient and can scale to a large number of queries.

4.7 Summary

In this chapter, we have presented a comprehensive study of multi-job optimization techniques for the MapReduce framework. We have proposed two new job sharing techniques and a novel two-phase optimization algorithm to optimize the evaluation of a batch of jobs given the expanded repertoire of optimization techniques. Our experimental results show that our proposed techniques outperform the state-of-the-art approach significantly by up to 107%.

CHAPTER 5

OPTIMAL JOIN ENUMERATION IN MAPREDUCE FRAMEWORK

5.1 Overview

In this chapter, we examine the optimal join enumeration (OJE) problem, which is a fundamental query optimization task for SQL-like queries, in the MapReduce paradigm. Specifically, we study both the single-query and multi-query OJE (referred to as SOJE and MOJE respectively) problems and propose efficient join enumeration algorithms for these problems. Our study of the SOJE problem serves as a foundation for our study on the MOJE problem. To reduce the complexity of the OJE problem, we follow a well-accepted heuristic in RDBMS [48, 41, 42, 16, 21, 24, 22, 23] to consider all bushy plans but exclude cross product from the enumeration space. This heuristic is particularly suitable for the MapReduce framework since bushy plans are more suitable for parallel execution via the MapReduce framework than left-deep or right-deep plans. Indeed, the work in [25] shows that bushy plans are usually the optimal plans in distributed environment. Furthermore, since the MapReduce framework always materializes intermediate results for fault tolerance and materializing cross product results is very costly, it is rare that an optimal join

plan in the MapReduce framework will involve cross product. Thus, cross product should be excluded from the enumeration space to reduce the complexity of the OJE problem.

While the OJE problem has attracted much recent attention in the conventional RDBMS context [48, 41, 42, 16, 21, 24, 22, 23], the solutions developed there are not applicable to the MapReduce context due to the differences in the query evaluation framework and algorithms.

There are two major differences between the OJE problem in the MapReduce context and that in the RDBMS context. First, both binary and multi-way joins are implemented in MapReduce while only binary joins are implemented in RDBMS. Specifically, given a join query, RDBMS will evaluate it as a sequence of binary joins while MapReduce will evaluate it as a sequence of binary or multi-way joins. As a result, the SOJE problem in the MapReduce context has a larger join enumeration space than that in the RDBMS context due to presence of multi-way joins. While there has been much recent works in the RDBMS context on the study of the complexity [48] of the SOJE problem and its join enumeration algorithms [41, 42, 16, 21, 24, 22, 23], to the best of our knowledge, there has not been any prior work on the study of these problems in the presence of multi-way joins in the MapReduce context.

Second, intermediate results in MapReduce are always materialized instead of being pipelined/materialized as in RDBMS which simplifies the MOJE problem in the MapReduce context in two ways. First, the MOJE problem in RDBMS may incur deadlock due to the pipelining framework [14] while that in MapReduce does not have the deadlock problem due to the materialization framework. Second, materializing and reusing the results of CSEs in RDBMS may incur additional materialization and reading cost due to the pipelining framework. However, since intermediate results are always materialized in the MapReduce framework, there is no additional overhead incurred with the materialization technique in MapReduce. Although the MOJE problem in RDBMS has been shown to be a very hard problem with a search space that is doubly exponential in the size of the queries [51, 14, 74], due to the simplification in MapReduce, we are able to propose efficient join enumeration algorithms for the MOJE problem in MapReduce.

In this chapter, we first study the SOJE problem in the MapReduce context. Specifically, we first study the complexity of the SOJE problem in the MapReduce context. Since the complexity of the SOJE problem depends on the query graph, we study the complexity for various query graph types (chain, cycle, star and clique) in the presence of multi-way joins. We then propose both bottom-up and top-down join enumeration algorithms for

the SOJE problem with an optimal complexity w.r.t. the query graph based on a proposal of an efficient and easy-to-implement plan enumeration algorithm. Our experimental results demonstrate that our proposed single query join enumeration algorithm significantly outperforms the baseline algorithms by up to 473%.

We then study the MOJE problem in the MapReduce framework. We propose an efficient multi-query join enumeration algorithm for the MOJE problem in the MapReduce framework. The main idea is to first apply the single-query join enumeration algorithm for each query to generate all the interesting plans and then stitch the interesting plans for the queries into a global optimal plan. A query plan is interesting if it is either the optimal plan or produces some output that can be reused for other queries. Our experimental results show that our proposed multi-query join enumeration algorithm is able to scale up to 25 queries where the number of relations in the queries ranges from 1 to 10.

We should emphasize that similar to existing works [48, 41, 42, 16, 42, 21, 24, 22, 23], the focus of this work is on the proposal of efficient join enumeration algorithms for the OJE problem in the MapReduce framework, but not on the effectiveness study of these join enumeration algorithms as it is well known that the runtime of different join orders can vary by orders of magnitude. Note that the proposed join enumeration algorithms could also be served as a foundation for other heuristics to restrict the enumeration space for queries with a large number of relations. To the best of our knowledge, our work presents the first systematic study of the OJE problem in the MapReduce paradigm and proposes efficient join enumeration algorithms for the problem.

The rest of this chapter is organized as follows. Section 5.2 presents some preliminaries. In Section 5.3, we analyse the complexity of the SOJE problem in the MapReduce framework for chain, cycle, star and clique queries. Sections 5.4 and 5.5, respectively, present the join enumeration algorithms for the SOJE and MOJE problems in the MapReduce paradigm. Section 5.6 presents experimental results and we conclude this chapter in Section 5.7.

5.2 Preliminaries

In this section, we introduce the notations and assumptions used in this chapter. Table 5.1 summarizes the notations used through this chapter.

Table 5.1: Notations used in this chapter

Notation	Definition
Q	input query for the study of the SOJE problem
$R = \{R_0, \cdots, R_{n-1}\}$	set of relations in Q
$G = (V, E)$	query graph for Q
$N(R_i)$	set of neighbors for a relation $R_i \in R$ w.r.t. G
$N(S)$	set of neighbors for a set of relations $S \subseteq R$ w.r.t. G
$Min(S)$	relation with the smallest subscript index in a set of relations S
C_i	set of connected subsets of R with a cardinality of i
$C = \bigcup_{i=2}^{n} C_i$	set of connected subsets of R with a cardinality of at least 2
P_S^k	set of k-way partitions of a connected subset S
P_S	set of partitions of a connected subset S
P	set of partitions of all the connected subsets in C
T_S	multiset of connected subsets in all partitions in P_S
T	multiset of connected subsets in all partitions in P
$\mathcal{Q} = \{Q_1, \cdots, Q_n\}$	input batch of queries for the study of the MOJE problem
$U_i = \{U_{i1}, \cdots, U_{i\|U_i\|}\}$	set of all the possible plans for Q_i
$W_i = \{W_{i1}, \cdots, W_{i\|W_i\|}\}$	set of relations in Q_i
I_S	set of interesting plans for a connected subset S
$CSE(U')$	set of CSEs of a plan U' w.r.t. $\mathcal{Q}$
$Cost(U')$	cost of a plan U'
$SubPlan(U')$	set of subplans for a plan U'
$JoinExp(U')$	join expression associated with a plan U'

5.2.1 Notations

Given an input query Q with a set of n relations $R = \{R_0, \cdots, R_{n-1}\}$, its query graph is defined as an undirected graph $G = (V, E)$ such that (1) each R_i $(0 \leq i < n)$ is a vertex in V and (2) an edge $e = (R_i, R_j)$ is in E if R_i and R_j are related by join predicates. In this chapter, we assume the input query graph is connected and use $|S|$ to denote the cardinality of a set S.

Given a query graph $G = (V, E)$, we use $N(R_i) = \{R' | (R', R_i) \in E\}$ to denote the set of neighbors for a vertex $R_i \in V$, and $N(S) = \bigcup_{R_i \in S} N(R_i) \setminus S$ to denote the set of neighbors for a set of vertices $S \subseteq V$. Furthermore, we use $Min(S)$ to denote the relation with the smallest subscript index in a set of vertices $S \subseteq V$.

A subset $S \subseteq R$ is referred to as a connected subset if it induces a connected subgraph of the query graph. We use C_i $(2 \leq i \leq n)$ to denote the set of all connected subsets of R with a cardinality of i, and $C = \bigcup_{i=2}^{n} C_i$ to denote the set of all connected subsets of R with a cardinality of at least 2. All the above definitions follow existing works [41, 42, 16, 42, 21, 24, 22, 23].

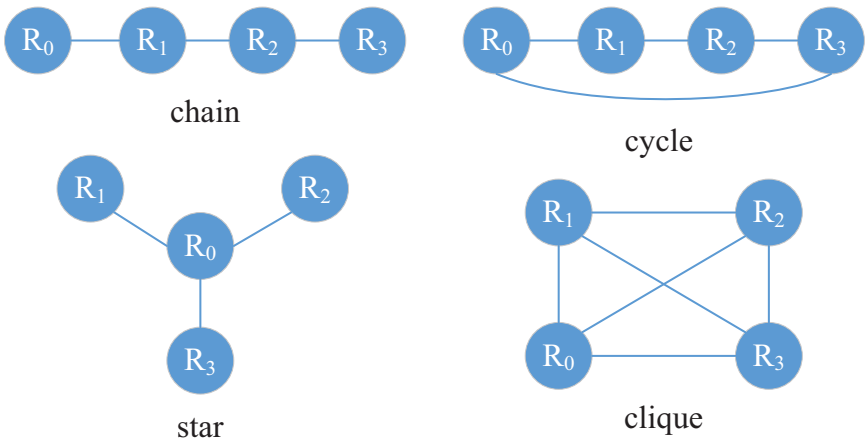


Figure 5.1: Examples of query types

Given a connected subset $S \subseteq R$ ($|S| \geq 2$), a k-way ($2 \leq k \leq |S|$) partition of S partitions S into k disjoint, non-empty sets $\{S_1, \cdots, S_k\}$ such that (1) each $S_i \subseteq S$ is a connected subset and (2) $S = S_1 \bigcup \cdots \bigcup S_k$. Note that each k-way [1] partition $\{S_1, \cdots, S_k\}$ of a connected subset S is associated with a k-way join plan for S constructed by joining the optimal plans for each S_i ($1 \leq i \leq k$).

For a connected subset S, we use P_S^k to denote the set of all k-way partitions of S, $P_S = \bigcup_{i=2}^{|S|} P_S^i$ to denote the set of all partitions of S, $T_S = \biguplus_{P' \in P_s} \biguplus_{S' \in P'} S'$ [2] to denote the multiset of all connected subsets in all partitions in P_S. Furthermore, we use $P = \bigcup_{S \in C} P_S$ to denote the set of all partitions of all connected subsets of R and $T = \biguplus_{P' \in P} \biguplus_{S' \in P'} S'$ to denote the multiset of all connected subsets in all partitions in P. Since each partition of a connected subset S is associated with a join plan for S, $|P_S|$ represent the number of join plans for S and $|P|$ represent the number of join plans for all the connected subsets of R.

Example 5.1: Consider the query graph for a chain query with fours relations $R = \{R_0, R_1, R_2, R_3\}$ in Figure 5.1. First, we have $Min(R) = R_0$, $N(R_1) = \{R_0, R_2\}$ and $N(\{R_1, R_2\}) = \{R_0, R_3\}$. Second, the subset $\{R_0, R_1, R_2\} \subseteq R$ is a connected subset since it induces a connected subgraph while the subset $\{R_0, R_2, R_3\} \subseteq R$ is not a connected subset. Furthermore, we have $C_2 = \{\{R_0, R_1\}, \{R_1, R_2\}, \{R_2, R_3\}\}$ consisting of all the connected subsets with a cardinality of 2. Third, for the connected subset $S = \{R_0, R_1, R_2\}$, it has one 3-way partition (i.e., $\{\{R_0\}, \{R_1\}, \{R_2\}\}$ and two 2-way partitions (i.e., $\{\{R_0, R_1\}, \{R_2\}\}$ and $\{\{R_0\}, \{R_1, R_2\}\}$). Note that $\{\{R_0, R_2\}, \{R_1\}\}$ is

[1] In RDBMS, algorithms for the OJE problem consider 2-way partitions while that in MapReduce consider all the k-way partitions where k ranges from 2 to $|S|$.

[2] $\biguplus$ denote a duplicate preserving union operator.

not a 2-way partition of S since $\{R_0, R_2\}$ is not a connected subset. Thus, we have $P_S = \{\{\{R_0\}, \{R_1\}, \{R_2\}\}, \{\{R_0, R_1\}, \{R_2\}\}, \{\{R_0\}, \{R_1, R_2\}\}\}$, and $T_S = \{\{R_0\}, \{R_1\}, \{R_2\}, \{R_0, R_1\}, \{R_2\}, \{R_0\}, \{R_1, R_2\}\}$. □

5.2.2 Assumptions

Similar to existing works [41, 42, 16, 21, 24, 22, 23], we assume that the number of relations in a query is not large (no more than 64 relations) so that they can be mapped to a machine word size (typically 32 or 64 bits). In this way, any subset of R can be encoded by an integer value where the i^{th} bit in the integer value represents R_i with a value of 1 indicating that R_i is in the subset. Thus, the set operators (i.e., *containment*, *union*, *intersection*, *difference*) can be performed via bitwise operators in constant time. Furthermore, the connectedness checking for a subset of relations $S \subseteq R$ can be done in O($|S|$) time as discussed in [16].

Under this assumption, recent works [41, 42, 16, 21, 24, 22, 23] in the RDBMS context propose both bottom-up and top-down join enumeration algorithms for the SOJE problem with an optimal complexity of $O(|P|)$. In the relational DBMS context, the time complexity of a join enumeration algorithm is optimal if it generates each partition $P' \in P$ in $O(1)$ time. This is realizable in the RDBMS context since each partition consists of two connected subsets which can be generated and output in $O(1)$ time as shown by existing works. However, in the MapReduce context, the number of connected subsets in a partition $P' \in P$ ranges from 2 to $|P'|$ which cannot be generated and output in constant time. Therefore, in the MapReduce context, the time complexity of a join enumeration algorithm is optimal if it generates each partition $P' \in P$ in O($|P'|$) time. Thus, the optimal time complexity of a join enumeration algorithm in MapReduce is $O(|T|)$.

For simplicity, we focus our presentation on bottom-up dynamic programming following the System R approach [52]; the extensions for top-down dynamic programming are straightforward and thus are only discussed if necessary.

5.3 Complexity of SOJE Problem

In this section, we study the complexity of the SOJE problem in terms of both $|P|$ and $|T|$ in the MapReduce context. Since the complexity of the SOJE problem depends on

Table 5.2: Comparison of complexity results for SOJE problem

Type	RDBMS-$\|P\|$	RDBMS-$\|T\|$	MapReduce-$\|P\|$	MapReduce-T
Chain	$\frac{n^3-n}{3}$	$\frac{2n^3-2n}{3}$	$2^{n+1} - \frac{n^2+3n}{2} - 2$	$(n-1)2^n - \frac{n^2+n}{2} + 1$
Cycle	$n^3 - 2n^2 + n$	$2n^3 - 4n^2 + 2n$	$n2^{n-1} + 2^n - n^2 - n - 1$	$(n^2+n)2^{n-2} - n^2$
Star	$(n-1)2^{n-1}$	$(n-1)2^n$	$3^{n-1} - 2^{n-1}$	$(n-1)3^{n-2} + 3^{n-1} - 2^{n-1}$
Clique	$3^n - 2^{n+1} + 1$	$2 \times 3^n - 2^{n+2} + 2$	$B_{n+1} - 2^n$	$B_{n+2} - 2B_{n+1} - 2^n + 1$

the query graph [48], we examine the problem repeatedly for chain, cycle, star and clique queries; an example of these query graphs on four relations $R = \{R_0, R_1, R_2, R_3\}$ are shown in Figure 5.1. Note that in the RBDMS context, since only binary-way joins are considered, we have $|T| = 2|P|$. Table 5.2 compares the complexity of the SOJE problem in the RDBMS context [48] and our cost analysis for the MapReduce framework based on the following theorems where B_n is the n^{th} Bell number [50] and $B_n < (\frac{0.792n}{ln(n+1)})^n$ [8]. In Table 5.2, each column X-Y, $X \in \{$RDBMS, MapReduce$\}$ and $Y \in \{|P|, |T|\}$, denotes the complexity of the SOJE problem in the X context in terms of Y.

Theorem 5.1. *For a chain query with n relations, we have $|P| = 2^{n+1} - \frac{n^2+3n}{2} - 2$ and $|T| = (n-1)2^n - \frac{n^2+n}{2} + 1$.*

Proof. Assume that each (R_i, R_{i+1}) $(0 \leq i \leq n-2)$ is an edge. To generate a connected subset S of R, the relations in S must be consecutive, i.e., $(R_i, R_{i+1}, \cdots, R_j)$ where $1 \leq i < j \leq n-1$. For each C_i $(2 \leq i \leq n)$, the number of all connected subsets in C_i is $(n-i+1)$, i.e., $(R_j, R_{j+1}, \cdots, R_{j+i-1})$ for $0 \leq j \leq n-i$. For each such connected subset $S \in C_i$, the number of all k-way $(2 \leq k \leq i)$ partitions is $|P_S^k| = \binom{i-1}{k-1}$ as we have to delete $(k-1)$ edges from the $(i-1)$ edges in S to partition S into k disjoint, connected subsets, the number of all partitions is $|P_S| = \sum_{k=2}^{i} \binom{i-1}{k-1} = 2^{i-1} - 1$ and the number of all connected subsets in P_S is $|T_S| = \sum_{k=2}^{i} k\binom{i-1}{k-1} = (i+1)2^{i-2} - 1$. Therefore, we have $|P| = \sum_{i=2}^{n}(n-i+1)(2^{i-1}-1) = 2^{n+1} - \frac{n^2+3n}{2} - 2$ and $|T| = \sum_{i=2}^{n}(n-i+1)((i+1)2^{i-2}-1) = (n-1)2^n - \frac{n^2+n}{2} + 1$. □

Theorem 5.2. *For a cycle query with n relations, we have $|P| = n2^{n-1} + 2^n - n^2 - n - 1$ and $|T| = (n^2+n)2^{n-2} - n^2$.*

Proof. Assume that each $(R_{i \bmod n}, R_{(i+1) \bmod n})$ $(0 \leq i < n)$ is an edge. For each C_i $(2 \leq i < n)$, the number of all connected subsets in C_i is n, i.e., $(R_{j \bmod n}, R_{(j+1) \bmod n}, \cdots, R_{(j+i-1) \bmod n})$ for $0 \leq j < n$. For each such connected subset S in C_i, since S is of type chain, the number of all k-way $(2 \leq k \leq i)$ partitions of S is $|P_S^k| = \binom{i-1}{k-1}$, the number of all partitions of S is $|P_S| = \sum_{k=2}^{i} \binom{i-1}{k-1} = 2^{i-1} - 1$ and the number of all connected subsets in P_S is $T_S = \sum_{k=2}^{i} k\binom{i-1}{k-1} = (i+1)2^{i-2} - 1$. For C_n, the number

of all connected subsets is 1 (i.e., R) and R is of type cycle. The number of all k-way $(2 \leq k \leq n)$ partitions of R is $|P_R^k| = \binom{n}{k}$ as we have to delete k edges from the n edges in R to partition R into k disjoint, connected subsets, the number of all partitions of R is $|P_R| = \sum_{k=2}^{n} \binom{n}{k} = 2^n - n - 1$ and the number of all connected subsets in P_R is $T_R = \sum_{k=2}^{n} k\binom{n}{k} = n2^{n-1} - n$. Therefore, we have $|P| = \sum_{i=2}^{n-1} n(2^{i-1} - 1) + 2^n - n - 1 = n2^{n-1} + 2^n - n^2 - n - 1$ and $|T| = \sum_{i=2}^{n-1} n((i+1)2^{i-2} - 1) + n2^{n-1} - n = (n^2 + n)2^{n-2} - n^2$. □

Theorem 5.3. *For a star query with n relations, we have $|P| = 3^{n-1} - 2^{n-1}$ and $|T| = (n-1)3^{n-2} + 3^{n-1} - 2^{n-1}$.*

Proof. Assume that each (R_0, R_i) $(1 \leq i < n)$ is an edge. To generate a connected subset S of R, S must contain R_0. For each C_i $(2 \leq i \leq n)$, the number of all connected subsets in C_i is $\binom{n-1}{i-1}$ as R_0 must be in a connected subset and the remaining $(i-1)$ relations have to be chosen from $\{R_1, \cdots, R_{n-1}\}$. For each such connected subset S in C_i, the number of all k-way $(2 \leq k \leq i)$ partitions is $|P_k^S| = \binom{i-1}{k-1}$ as we have to delete $(k-1)$ edges from the $(i-1)$ edges in S to partition S into k disjoint, connected subsets, the number of all partition is $|P_S| = \sum_{k=2}^{i} \binom{i-1}{k-1} = 2^{i-1} - 1$ and the number of all connected subsets is $|T_S| = \sum_{k=2}^{i} k\binom{i-1}{k-1} = (i+1)2^{i-2} - 1$. Thus, we have $|P| = \sum_{i=2}^{n} \binom{n-1}{i-1}(2^{i-1} - 1) = 3^{n-1} - 2^{n-1}$ and $|T| = \sum_{i=2}^{n} \binom{n-1}{i-1}((i+1)2^{i-2} - 1) = (n-1)3^{n-2} + 3^{n-1} - 2^{n-1}$. □

Theorem 5.4. *For a clique query with n relations, we have $|P| = B_{n+1} - 2^n$ and $|T| = B_{n+2} - 2B_{n+1} - 2^n + 1$, where B_n is the n^{th} Bell number.*

Proof. Assume that each (R_i, R_j) $(0 \leq i < j < n)$ is an edge. For each C_i $(2 \leq i \leq n)$, the number of all connected subsets in C_i is $\binom{n}{i}$ as we have to choose i relations from R. For each such connected subset S in C_i, the number of all k-way $(2 \leq k \leq i)$ partitions is $|P_k^S| = \left\{ {i \atop k} \right\} = \frac{1}{k!} \sum_{j=0}^{k} (-1)^{k-j} \binom{k}{j} j^i$ where $\left\{ {i \atop k} \right\}$ is the Stirling number of the second kind [50], the number of all partition is $|P_S| = \sum_{k=2}^{i} \left\{ {i \atop k} \right\} = B_i - 1$ where B_i is the i^{th} Bell number [50], and the number of all connected subsets in P_S is $T_S = \sum_{k=2}^{i} k\left\{ {i \atop k} \right\} = B_{i+1} - B_i - 1$. Thus, we have $|P| = \sum_{i=2}^{n} \binom{n}{i}(B_i - 1) = B_{n+1} - 2^n$ and $|T| = \sum_{i=2}^{n} \binom{n}{i}(B_{i+1} - B_i - 1) = B_{n+2} - 2B_{n+1} - 2^n + 1$. Note that $B_n < (\frac{0.792n}{ln(n+1)})^n$ as shown in [8]. □

5.4 Single-Query Join Enumeration Algorithm

In this section, we first present two baseline join enumeration algorithms (denoted as DPsize and DPset) for the SOJE problem in the MapReduce framework which are respectively adapted from the two state-of-the-art join enumeration algorithms for RDBMS [52, 62, 41]. Both DPsize and DPset follow a naive generate-and-test approach and thus have a time complexity that is not optimal in the MapReduce context. Then we present an efficient and easy-to-implement plan enumeration algorithm (denoted as PEA) to enumerate all the partitions (i.e., plans) of a connected subset. Finally, we propose both top-down and bottom-up join enumeration algorithms with an optimal time complexity based on Algorithm PEA.

5.4.1 Baseline Join Enumeration Algorithms

Algorithm 5.1: Bottom-up Enumeration:DPsize

Input: A connected query graph with a set of n relations $R = \{R_0, \cdots, R_{n-1}\}$
Output: The optimal join plan for R, BestPlan(R)

```
1  innercounter = 0 ;
2  outercounter = 0 ;
3  BestPlan = new HashTable() ;
4  for i = 0 → (n − 1) do
5  |  create BestPlan({R_i}) ;
6  for i = 2 → n do                                   /* Enumerate i-plans */
7  |  foreach integer partition (i_1, ··· , i_k) of i such that 2 ≤ k ≤ i do
8  |  |  foreach {S_1, ··· , S_k} ∈ C_{i_1} × ··· × C_{i_k} do
9  |  |  |  ++outercounter ;
10 |  |  |  if ∃S_g, S_h, 1 ≤ g < h ≤ k, S_g ⋂ S_h ≠ ∅ then
11 |  |  |  |  continue ;
12 |  |  |  S' = ⋃_{j=1}^{k} S_j ;
13 |  |  |  if S' is not a connected subset then
14 |  |  |  |  continue ;
15 |  |  |  ++innercounter ;
16 |  |  |  newPlan = createPlan(BestPlan(S_1),··· ,BestPlan(S_k)) ;
17 |  |  |  if Cost(BestPlan(S')) > Cost(newPlan) then
18 |  |  |  |  BestPlan(S') = newPlan ;
```

Size-driven Enumeration. For simplicity, we refer to a plan as a i-plan to mean that the number of relations in (i.e., the size of) the plan is i. Our first baseline join enumeration algorithm (referred to as DPsize) enumerates plans iteratively, i.e., i-plans are enumerated before $(i + 1)$-plans, which is adapted from [52, 41] designed for the SOJE problem in

the RDBMS context. Algorithm 5.1 shows the pseudocode for DPsize. In the algorithm, *BestPlan* is a hash table that stores the best plan found so far for each connected subset of R. Two counters *innercounter* and *outercounter* are maintained for complexity analysis for the algorithm (to be explained). The algorithm starts by initializing the best plan for each single relation in R and then enumerates plans in increasing size.

In the i^{th} iteration to enumerate i-plans, DPsize appropriately combines disjoint, smaller-size plans that have already been enumerated. To generate all the possible plan combinations, DPsize first generates all the integer partitions for i [3] using an efficient algorithm from [75]. Each integer partition $(i_1, \cdots, i_k)$ of i $(2 \leq k \leq i)$ essentially represents a way to combine plans of size i_1, $\cdots$, i_k to generate i-plans. Thus, each $(i_1, \cdots, i_k)$ is associated with some candidate partitions denoted as $C_{i_1} \times \cdots \times C_{i_k}$. To make sure that the plan [4] associated with a candidate partition $\{S_1, \cdots, S_k\} \in C_{i_1} \times \cdots \times C_{i_k}$ qualifies as an i-plan (i.e., the candidate partition is indeed a partition), we need to ensure two requirements. First, all the connected subsets in $\{S_1, \cdots, S_k\}$ are disjoint which can be checked in O(k) time. Note that the disjointedness checking is achieved by the bitwise AND operator on the corresponding integer representations for each S_j $(1 \leq j \leq k)$ [5]. Second, $S' = \bigcup_{j=1}^{k} S_j$ is a connected subset which can be checked in $O(|S'|)$ time (i.e., $O|i|$ as $|S'| = i$) as discussed in Section 5.2.

The time complexity of DPsize is not optimal for two reasons. First, it generates more candidate partitions. Specifically, the value of the variable $outercounter$ in Algorithm 5.1 represents the number of generated candidate partitions which is larger than the number of the partitions for all the connected subsets of R (i.e., $|P|$) represented by the value of the variable $innercounter$. Second, the time complexity to verify a candidate k-way partition $\{S_1, \cdots, S_k\}$ is indeed a partition is $O(|\bigcup_{j=1}^{k} S_j|)$ time due to the connectedness checking which is no smaller than the optimal time complexity to generate a k-way partition which is $O(k)$.

Subset-driven Enumeration. Our second baseline join enumeration algorithm (referred to as DPset) enumerates all the subsets of R in increasing order of their integer representations, which is adapted from [62, 41] designed for the SOJE problem in the RDBMS context. For each enumerated subset S of R, if S is not a connected subset, it is immediately dropped; otherwise, we compute the optimal plan for S by generating all the

[3] An integer partition of a number i refers to a way of writing i as a sum of positive integers. Two sums that differ only in the order of their summands are considered to be the same.

[4] The plan associated with a candidate partition $\{S_1, \cdots, S_k\}$ is constructed by joining the optimal plans for each S_i $(1 \leq i \leq k)$.

[5] Two subsets S_i and S_j are disjoint if the result of the bitwise AND operator on their integer representations is 0.

Algorithm 5.2: Bottom-up Enumeration:DPset

Input: A connected query graph with a set of n relations $R = \{R_0, \cdots, R_{n-1}\}$
Output: The optimal join plan for R, BestPlan(R)

```
1  innercounter = 0 ;
2  outercounter = 0 ;
3  BestPlan = new HashTable() ;
4  for i = 1 → (2^n − 1) do                          /* Enumerate subsets */
5      Let S ⊆ R be the subset corresponding to i ;
6      if S is not a connected subset then
7          continue ;
8      if |S| = 1 then
9          create BestPlan(S) ;
10         continue ;
11     foreach {S_1 ⊆ S, S_2 ⊆ (S \ S_1), ··· , S_k = (S \ S_1 \ ··· \ S_{k−1})} such that
       2 ≤ k ≤ |S| and ⋃_{j=1}^{k} S_j = S do
12         ++outercounter ;
13         if ∃i ∈ [1, k] such that S_i is not a connected subset then
14             continue;
15         ++innercounter ;
16         newPlan = createPlan(BestPlan(S_1),··· ,BestPlan(S_j)) ;
17         if Cost(BestPlan(S)) > Cost(newPlan) then
18             BestPlan(S) = newPlan ;
```

partitions of S and enumerating the corresponding join plans for S. Note that to compute the optimal plan for S, all its subsets must be enumerated before itself. This is guaranteed by DPset's enumeration order of the subsets of R. Algorithm 5.2 shows the pseudocode for DPset. Here again, *BestPlan* is a hash table that stores the best plan found so far for each connected subset of R. Two counters *innercounter* and *outercounter* are maintained for complexity analysis for the algorithm (to be explained).

For each enumerated and connected subset S of R, for each candidate partition $\{S_1 \subseteq S, S_2 \subseteq (S \setminus S_1), \cdots, S_k \subseteq (S \setminus S_1 \setminus \cdots \setminus S_{k-1})\}$ such that $2 \leq k \leq |S|$ and $\bigcup_{j=1}^{k} S_j = S$, we need to test for the connectedness for each S_i $(1 \leq i \leq k)$ to ensure that it is indeed a partition of S. As each $S_i \in S$ $(1 \leq i \leq k)$ has already been enumerated before S, the connectedness checking for S_i can be achieved by looking up the hash table *BestPlan* as follows. If S_i is present in *BestPlan*, then it is connected; otherwise it is not connected. Therefore, the connectedness checking for $\{S_1, \cdots, S_k\}$ is done in O(k) time.

To generate all the candidate partitions of a connected subset S, we have to generate all the non-empty subsets $S_1 \subseteq S$, $\cdots$, $S_k \subseteq (S \setminus S_1 \setminus \cdots \setminus S_{k-1})$ such that $2 \leq k \leq |S|$ and $\bigcup_{j=1}^{k} S_j = S$. The generation of each S_i can be done very efficiently in O(1) time

by applying the idea from [62] [6]. Furthermore, to avoid generating duplicate candidate partitions, we restrict each $S_i \subseteq (S \setminus S_1 \setminus \cdots \setminus S_{i-1})$ $(1 \leq i \leq k)$ to contain the relation $Min(S \setminus S_1 \setminus \cdots \setminus S_{i-1})$. The purpose is to give an unique ordering of all the subsets $S_1, \cdots, S_k$ so that duplicate candidate partitions are not generated. For example, consider the chain query in Figure 5.1, for the connected subset $R = \{R_0, \cdots, R_s\}$, if we do not restrict each S_i, we will generate $S_1 = \{R_0\}$ and $S_2 = \{R_1, R_2, R_3\}$, and $S_1 = \{R_1, R_2, R_3\}$ and $S_2 = \{R_0\}$ which essentially represent the same partition $\{\{R_0\}, \{R_1, R_2, R_3\}\}$ of S. However, if we restrict each S_i, we will only generate $S_1 = \{R_0\}$ and $S_2 = \{R_1, R_2, R_3\}$.

Here again, the time complexity of DPset is not optimal. This is because DPset generates more candidate partitions to verify as represented by the value of the variable $outercounter$ in Algorithm 5.1. Indeed, the number of partitions of all the connected subsets of R is equal to the value of the variable $innercounter$ which is smaller than the value of the variable $outercounter$. Note that the time complexity to verify the connectedness for a candidate partition $\{S_1, \cdots, S_k\}$ is equal to the optimal time complexity to generate a k-way partition which are both $O(k)$.

Comparison of DPsize and DPset. Both DPsize and DPset follow a naive generate-and-test approach and thus have a suboptimal time complexity, i.e., DPsize has to test for disjointedness (Line 11 in Algorithm 5.1) while DPset has to test for connectedness (Line 13 in Algorithm 5.2). When only 2-way joins (i.e., in RDBMS) are considered, the experimental results in [41] demonstrate that neither DPsize nor DPset is strictly more superior. This is because in RDBMS, the number of disjointedness checking in DPsize and the number of connectedness checking in DPset can exceed each other for different query types. However, when multi-way joins are considered, our experimental results demonstrate that DPset is significantly faster than DPsize by up to two orders of magnitude. This is due to the large number of integer partitions generated for DPsize which results in a large number of disjointedness checking. For example, when the number of relations in a chain query is 15, DPsize generates 668 integer partitions which results in 17.120.334 disjointedness checking while DPset only needs to check the connectedness 458.073 times.

[6] Given a set S and its integer representation V(S), each integer representation (denoted as V) of the subsets of S is generated using the following recursive formula $V = V(S)\&(V - V(S))$ with the initialization condition $V = 0$ (i.e., empty subset) and the termination condition $V = V(S)$ (i.e., S).

5.4.2 Plan Enumeration Algorithm

Existing works [52, 62, 41] have proposed both bottom-up and top-down dynamic programming algorithms to generate each partition in P in $Q(1)$ time in the RDBMS context. However, their algorithms are limited to 2-way joins and thus are not applicable for multi-way joins in the MapReduce context. Indeed, as discussed in Section 5.2, it is impossible to generate each partition in $O(1)$ time in the context of MapReduce framework. In this section, we present a plan (i.e., partition) enumeration algorithm (denoted as PEA) to efficiently generate all the partitions of a connected subset S (i.e., P_S) with each partition $P' \in P_S$ being generated in $O(|P'|)$ time.

Algorithm 5.3: Connected subset enumeration algorithm (CSEA)

Input: A query graph with a set of relations $R = \{R_0, \cdots, R_{n-1}\}$
Output: All the connected subsets of R containing R_0

```
1 output({R_0}) ;
2 Enumerate({R_0}, {R_0}, N({R_0})) ;
3 Function Enumerate (S, D, H) begin
4     if H is empty then
5         return ;
6     foreach non-empty S' ⊆ H do
7         output(S ⋃ S') ;
8     foreach non-empty S' ⊆ H do
9         Enumerate(S ⋃ S', D ⋃ H, N(S ⋃ S') \ (D ⋃ H))
```

Before we present our algorithm, we first review a connected subset enumeration algorithm [41] (denoted as CSEA shown in Algorithm 5.3) which enumerates all the connected subsets containing a relation R_0 for an input query graph with a set of relations $R = \{R_0, \cdots, R_{n-1}\}$. The main idea of the approach is as follows. Given an already enumerated and connected subset S, the approach extends S by adding the relations S' from its neighbors (i.e., $N(S)$) into it to generate larger connected subsets. To avoid producing duplicates, it maintains a set of relations D that have already been visited. When adding the relations into S, it only adds the relations from $N(S) \setminus D$ (i.e., H). The approach first outputs $\{R_0\}$ as a connected subset and then invokes the function Enumerate with $S = \{R_0\}$, $D = \{R_0\}$ and $H = N(\{R_0\})$ to generate connected subsets which recursively invokes itself with different parameter values until it has generated all the connected subsets.

As a representation of the input query graph for Algorithm CSEA, for each R_i, instead of storing all its neighbors in an adjacency list (a typical way to represent a graph), it is sufficient to maintain an integer to represent $N(R_i)$ to simplify the computation of

neighbors for connected subsets. Furthermore, we have $N(S \bigcup S') = N(S) \bigcup N(S') \setminus (S \bigcup S')$ for incremental neighbourhood computation. To generate all the subsets of U (Line 5 in Algorithm 5.3), we apply the idea from [62] with a complexity of $O(1)$ for each subset as discussed in Section 5.4.1. By combining all these techniques, the time complexity to generate each connected subset is $O(1)$ as shown in [41].

Algorithm 5.4: Plan enumeration algorithm (PEA)

Input: A connected subset $S = \{R_0, \cdots, R_{|S|-1}\}$
Output: All the partitions of S (i.e., P_S)

```
1  for i = 1 → |S| do
2  |   F_i = {R_{i-1}, ··· , R_{|S|-1}} ;
3  |   S_i = getnext(F_i) ;
4  output{S_1, ··· , S_{|S|}} ;
5  i = |S| − 1 ;
6  while i ≥ 1 do    /* Start an iteration to generate a partition */
   |   /* Scan backward to find the first F_i to retrieve S_i      */
7  |   while (S_i = getnext(F_i)) is null do
8  |   |   −− i;
   |   /* Scan forward to rewind each F_i and S_i                  */
9  |   F_{i+1} = F_i \ S_i ;
10 |   while F_{i+1} ≠ ∅ do
11 |   |   S_{i+1} = getnext(F_{i+1}) ;
12 |   |   ++ i ;
13 |   |   F_{i+1} = F_i \ S_i ;
14 |   output(S_1, ··· , S_i) ;
```

We now discuss our plan (i.e., partition) enumeration algorithm PEA. Given a connected subset $S = \{R_0, \cdots, R_{|S|-1}\}$, Algorithm 5.4 shows our approach to generate all the partitions of S (i.e., P_S) where each partition $P' \in P_S$ is generated in $O(|P'|)$ time. In the algorithm, each $F_i = S \setminus \bigcup_{j=1}^{i-1} S_j$ denotes the set of relations in S after excluding $\bigcup_{j=1}^{i-1} S_j$ and can be incrementally computed by the formula $F_{i+1} = F_i \setminus S_i$, and each $S_i \subseteq F_i$ represents a connected subset of F_i containing the relation $Min(F_i)$. Similar to our proposed technique for DPset, we restrict each S_i to contain $Min(F_i)$ to avoid generating duplicate partitions. Furthermore, the function $getnext(F_i)$ is used to get the next connected subset of F_i containing $Min(F_i)$ and will eventually generate all the connected subsets of F_i containing $Min(F_i)$. Thus, for each F_i, Algorithm CSEA is first called to retrieve a sequence of all the connected subsets of F_i containing $Min(F_i)$ and then the function $getnext(F_i)$ is used to retrieve the next connected subset in the sequence.

To generate a partition of S, our algorithm generates a sequence of connected subsets $S_1 \in F_1, \cdots, S_k \in F_k$ until F_{k+1} is empty (i.e., $\bigcup_{j=1}^{k} S_j = S$). Each time when it generates a connected subset S_i of F_i $(1 \leq i \leq k)$, it updates F_{i+1} to be $F_i \setminus S_i$ to ensure

Table 5.3: An example illustrating the plan enumeration algorithm

	S_1	S_2	S_3	S_4	F_1	F_2	F_3	F_4
1	$\{R_0\}$	$\{R_1\}$	$\{R_2\}$	$\{R_3\}$	$\{R_0, R_1, R_2, R_3\}$	$\{R_1, R_2, R_3\}$	$\{R_2, R_3\}$	$\{R_3\}$
2	$\{R_0\}$	$\{R_1\}$	$\{R_2, R_3\}$	$\emptyset$	$\{R_0, R_1, R_2, R_3\}$	$\{R_1, R_2, R_3\}$	$\{R_2, R_3\}$	$\emptyset$
3	$\{R_0\}$	$\{R_1, R_2\}$	$\{R_3\}$	$\emptyset$	$\{R_0, R_1, R_2, R_3\}$	$\{R_1, R_2, R_3\}$	$\{R_3\}$	$\emptyset$
4	$\{R_0\}$	$\{R_1, R_2, R_3\}$	$\emptyset$	$\emptyset$	$\{R_0, R_1, R_2, R_3\}$	$\{R_1, R_2, R_3\}$	$\emptyset$	$\emptyset$
5	$\{R_0, R_1\}$	$\{R_2\}$	$\{R_3\}$	$\emptyset$	$\{R_0, R_1, R_2, R_3\}$	$\{R_2, R_3\}$	$\{R_3\}$	$\emptyset$
6	$\{R_0, R_1\}$	$\{R_2, R_3\}$	$\emptyset$	$\emptyset$	$\{R_0, R_1, R_2, R_3\}$	$\{R_2, R_3\}$	$\emptyset$	$\emptyset$
7	$\{R_0, R_1, R_2\}$	$\{R_3\}$	$\emptyset$	$\emptyset$	$\{R_0, R_1, R_2, R_3\}$	$\{R_3\}$	$\emptyset$	$\emptyset$

that all the generated subsets $S_1, \cdots, S_k$ are disjoint. In this way, $\{S_1, \cdots, S_k\}$ qualifies as a k-way partition of S. Furthermore, since both the retrieval of S_i and the updating of F_i is done in O(1) time, the time complexity to generate a k-way partition is $O(k)$ in our algorithm.

To generate all the partitions of S, our algorithm works as follows. Initially, it has $S_i = \{R_{i-1}\}$ for each $i \in [1, |S|]$ and simply outputs $\{S_1, \cdots, S_{|S|}\}$ as a $|S|$-way partition of S. Then it goes into an iterative process to generate a partition of S. At each iteration, it first scans backwards (from $F_{|S|}$ to F_1) to find the first F_i, where not all the connected subsets have been enumerated, to retrieve the next connected subset of F_i. It then updates each F_j $(j > i)$ and generates the first connected subset S_j of F_j until it has generated a connected subset S_k such that F_{k+1} is empty. Finally, it outputs $\{S_1, \cdots, S_k\}$ as a k-way partition of S. The iterative process terminates when F_2 is empty (i.e., all the partitions of S have been generated). Note that the last generated partition is a 1-way partition of S and should be ignored.

Example 5.2: Consider the chain query in Figure 5.1. Table 5.3 illustrates our plan enumeration algorithm to generate all the partitions of $\{R_0, R_1, R_2, R_3\}$. In the first iteration, we simply output $\{\{R_0\}, \{R_1\}, \{R_2\}, \{R_3\}\}$ as a 4-way partition. In the second iteration, we scan backwards (from F_4 to F_1) and get F_3 to retrieve the next connected subset $S_3 = \{R_2, R_3\}$. Then we have $F_4 = \emptyset$ and output $\{\{R_0\}, \{R_1\}, \{R_2, R_3\}\}$ as a 3-way partition. In the third iteration, we get F_2 to retrieve the next connected subset $S_2 = \{R_1, R_2\}$. Then we have $F_3 = \{R_3\}$, $S_3 = \{R_3\}$, $F_4 = \emptyset$ and output a 3-way partition $\{\{R_0\}, \{R_1, R_2\}, \{R_3\}\}$. We repeat the above process until we generate all the partitions. □

5.4.3 Bottom-up and Top-down Enumerations

In this section, we propose both bottom-up and top-down dynamic programming algorithms with an optimal time complexity of $O(|T|)$ based on Algorithm PEA.

Algorithm 5.5: Bottom-up Enumeration:DPopt

Input: A connected query graph with a set of n relations $R = \{R_0, \cdots, R_{n-1}\}$
Output: The optimal join plan for R, BestPlan(R)

```
1  BestPlan = new HashTable() ;
2  for i = (n − 1) → 0 do
3      foreach S ∈ CSEA({R_i, ··· , R_{n−1}})) (i.e., Algorithm 5.3) do
4          if |S| = 1 then
5              create BestPlan(S) ;
6              continue ;
7          P_S = PEA(S) (i.e., Algorithm 5.4) ;
8          foreach partition {S_1, ··· , S_k} ∈ P_S do
9              newPlan = createPlan(BestPlan(S_1),··· ,BestPlan(S_j)) ;
10             if Cost(BestPlan(S)) > Cost(newPlan) then
11                 BestPlan(S) = newPlan ;
```

Bottom-up enumeration. Algorithm 5.5 shows the pseudocode of our bottom-up enumeration (denoted as DPopt) which makes two changes to DPset to ensure that it has an optimal time complexity. First, DPset follows the approach in [41] to generate all the connected subsets of R with each connected subset being generated in $O(1)$ time [7]. Second, for each enumerated and connected subset S of R, DPopt generates all the partitions of S (i.e., P_S) using Algorithm PEA.

Top-down enumeration. Algorithm 5.6 shows the pseudocode of our top-down enumeration. The algorithm starts by finding the optimal plan for each single relation in R and then invoke the function GenOptimal(R) to construct the optimal plan for R.

Given a connected subset S of R, the function GenOptimal(S) generates the optimal plan for S by enumerating all the partitions of S using Algorithm PEA and recursively construct the optimal plan for each connected subset of S in the enumerated partitions. To avoid redundant construction of optimal plans, a hash table *BestPlan* is used to cache the optimal plan for each connected subset of R.

[7] The approach in [41] generates all the connected subsets of R by applying Algorithm 5.3 on the sequence of inputs $\{R_{n-1}\}$, $\{R_{n-2}, R_{n-1}\}$,..., $\{R_0, \cdots, R_{n-1}\}$. In this way, for a connected subset S of R, all its subsets are enumerated before itself.

Algorithm 5.6: Top-down enumeration

Input: A connected query graph with a set of relations $R = \{R_0, \cdots, R_{n-1}\}$
Output: The optimal join plan for R, BestPlan(R)

```
1  BestPlan = new HashTable() ;
2  for i = 0 → (n − 1) do
3  |   create BestPlan({R_i}) ;
4  return GenOptimal(R) ;
5  Function GenOptimal(S) begin
6  |   if BestPlan(S) is null then
7  |   |   P_S = PEA(S) (i.e., Algorithm 5.4) ;
8  |   |   foreach partition {S_1, ··· , S_k} ∈ P_S do
9  |   |   |   newPlan = createPlan(GenOptimal(S_1),··· ,GenOptimal(S_k)) ;
10 |   |   |   if Cost(BestPlan(S)) > Cost(newPlan) then
11 |   |   |   |   BestPlan(S) = newPlan ;
12 |   return BestPlan(S) ;
```

5.5 Multi-Query Join Enumeration Algorithm

In this section, we present a novel multi-query join enumeration algorithm for the MOJE problem for a batch of queries $\mathcal{Q} = \{Q_1, \cdots, Q_n\}$. The MOJE problem aims to find a global optimal plan for a batch of queries to share computation of their CSEs. As the global optimal plan in general is not simply constructed from the local optimal plan for each query, we have to consider all the possible plans for each query. Due to the large number of possible plans for a query, enumerating all the plan combination space for a batch of queries is usually very costly. In this section, we propose effective techniques to prune away non-promising plans and thus reduce the plan combination space.

Our proposed multi-query join enumeration algorithm consists of two-phases. In the first phase, for each Q_i $(1 \leq i \leq n)$, we apply the single-query join enumeration algorithm (discussed in the previous section) to generate all the interesting plans for Q_i. A plan of Q_i is interesting if it is either the optimal plan or produces some output that could be reused for other queries. In the second phase, we stitch the interesting plans for the queries maintained in the first phase into a global optimal plan.

While there has been some works on the study of the MOJE problem in the RDBMS context [51, 14, 74], they mainly focus on greedy heuristics to find a good global plan for a batch of queries. We present a novel two-phase algorithm to find a global optimal plan for a batch of queries. Specifically, we present novel pruning techniques to prune away non-promising plans as well as a systematic approach to merge the interesting plans for

Table 5.4: Running examples of queries and plans

Query	Plan	CSE set
Q_1: $R_0 \bowtie R_1 \bowtie R_2 \bowtie R_3$	U_{11}: $((R_0 \bowtie R_1) \bowtie (R_2 \bowtie R_3))$	$CSE(U_{11})$: $\{R_0 \bowtie R_1, R_2 \bowtie R_3\}$
	U_{12}: $(R_0 \bowtie R_1 \bowtie (R_2 \bowtie R_3))$	$CSE(U_{12})$: $\{R_2 \bowtie R_3\}$
	U_{13}: $(R_0 \bowtie (R_1 \bowtie (R_2 \bowtie R_3)))$	$CSE(U_{13})$: $\{R_2 \bowtie R_3\}$
	U_{14}: $(((R_0 \bowtie R_1) \bowtie R_2) \bowtie R_3)$	$CSE(U_{14})$: $\{R_0 \bowtie R_1, R_0 \bowtie R_1 \bowtie R_2\}$
	U_{15}: $((R_0 \bowtie R_1) \bowtie R_2 \bowtie R_3)$	$CSE(U_{15})$: $\{R_0 \bowtie R_1\}$
	U_{16}: $((R_0 \bowtie R_1 \bowtie R_2) \bowtie R_3)$	$CSE(U_{16})$: $\{R_0 \bowtie R_1 \bowtie R_2\}$
Q_2: $R_0 \bowtie R_1 \bowtie R_2 \bowtie R_4$	U_{21}: $((R_0 \bowtie R_1) \bowtie (R_2 \bowtie R_4))$	$CSE(U_{21})$: $\{R_0 \bowtie R_1, R_2 \bowtie R_4\}$
	U_{22}: $(((R_0 \bowtie R_1) \bowtie R_2) \bowtie R_4)$	$CSE(U_{22})$: $\{R_0 \bowtie R_1, R_0 \bowtie R_1 \bowtie R_2\}$
Q_3: $R_2 \bowtie R_3 \bowtie R_4$	U_{31}: $((R_2 \bowtie R_3) \bowtie R_4)$	$CSE(U_{31})$: $\{R_2 \bowtie R_3\}$

each query into a global optimal plan. In the following, we elaborate on the details of the two phases.

Notations. For a query Q_i, we use $W_i = \{W_{i1}, \cdots, W_{i|W_i|}\}$ to denote the set of relations in Q_i, $U_i = (U_{i1}, \cdots, U_{i|U_i|})$ to denote the set of all possible plans for Q_i. Table 5.4 shows three example queries and some plans for each query that will be used to illustrate our algorithm.

For a plan U' and its associated partition $\{S_1, \cdots, S_k\}$ [8], we use $JoinExp(U')$ to denote its join expression without any execution order, $SubPlan(U')$ to denote the set of subplans of each S_i, and $Cost(U')$ to denote its evaluation cost based on some cost model. For example, for the plan U_{14} in Table 5.4 and its associated partition $\{\{R_0, R_1, R_2\}, \{R_3\}\}$, we have $JoinExp(U_{14}) = R_0 \bowtie R_1 \bowtie R_2 \bowtie R_3$ and $SubPlan(U_{14}) = \{(R_0 \bowtie R_1) \bowtie R_2, R_3\}$.

5.5.1 First Phase

In the first phase, for each plan U_{ij} of Q_i, we maintain a set of CSEs of U_{ij} whose results could be reused for other queries in $\mathcal{Q}$. We refer to this set as CSE set and use $CSE(U_{ij})$ to denote the CSE set of U_{ij}. For example, consider the plan U_{14} in Table 5.4, we have $CSE(U_{14}) = \{R_0 \bowtie R_1, R_0 \bowtie R_1 \bowtie R_2\}$ since the results of the subexpressions $R_0 \bowtie R_1$ and $R_0 \bowtie R_1 \bowtie R_2$ could be reused for Q_2. Note that there may be several plans corresponding to the same CSE set. For example, the two plans U_{12} and U_{13} in Table 5.4 have the same CSE set (i.e., $\{R_2 \bowtie R_3\}$). We say a plan U_{ij} is interesting if it is either the optimal plan or its CSE set is not empty. Note that even if a plan U_{ij} is not the optimal plan, the global optimal plan may choose U_{ij} for Q_i if U_{ij} produces some output that

[8] Recall that each partition is associated with a plan as discussed in Section 5.2.

could be reused for other queries in $\mathcal{Q}$. Thus, in this phase, we need to maintain all the interesting plans for each Q_i to be further processed in the second phase.

Algorithm 5.7: Interesting plan generation algorithm

Input: An query Q_i in $\mathcal{Q}$
Output: Interesting plans for Q_i (i.e., I_{W_i})

```
1  for j = |W_i| → 1 do
2      foreach S ∈ CSEA({W_ij, ··· , W_i|W_i|}) (i.e., Algorithm 5.3) do
3          if |S| = 1 then
4              I_S = createPlan(S) ;
5              continue ;
6          P_S = PEA(S) (i.e., Algorithm 5.4) ;
7          foreach partition {S_1, ··· , S_k} ∈ P_S do
8              foreach {U'_1, ··· , U'_k} ∈ I_S1 × ··· × I_Sk do
9                  U' = createPlan(U'_1,···,U'_k) ;
10                 CSE(U') = ⋃_{j=1}^{k} CSE(U_j) ;
11                 if JoinExp(U') is a CSE w.r.t Q then
12                     CSE(U') = CSE(U') ⋃ JoinExp(U');
13                 I_S = I_S ⋃ U' ;
14         Apply the two pruning techniques for I_S ;
```

We now discuss how the interesting plans and the corresponding CSE sets for a query Q_i are computed when applying the single query join enumeration algorithm for Q_i. Algorithm 5.7 shows the pseudocode of this process. To support incremental computation, for each connected subset S of W_i, we compute and maintain all the interesting plans for it (denoted as I_S). Note that this is different from the SOJE problem where only one optimal plan is maintained for S. Consider the enumeration of interesting plans and the corresponding CSE sets for S, for each k-way partition $\{S_1, \cdots, S_k\}$ of S and for each $\{U'_1, \cdots, U'_k\} \in I_{S_1} \times \cdots \times I_{S_k}$, the plan (denoted as U') joining U'_1,$\cdots$,U'_k is an interesting plan for S, and the CSE set of U' is simply the union of each $CSE(U'_i)$ $(1 \leq i \leq k)$ plus $JoinExp(U')$ if the results of $JoinExp(U')$ could be reused for other queries. Thus, all the interesting plans and the corresponding CSE sets for S can be computed by enumerating all the partitions of S and examining the expression $I_{S_1} \times \cdots \times I_{S_k}$ associated with each partition $\{S_1, \cdots, S_k\}$ in P_S. After evaluating I_S, we apply two pruning techniques (to be explained) on the plans in I_S to prune away the non-promising plans and thus reduce the optimization cost. We now discuss our pruning techniques.

Pruning techniques. To prune away the non-promising interesting plans for Q_i, we introduces two pruning techniques based on the plan cost and the relationship between CSE sets. The first pruning technique prunes plans with the same CSE set. Specifically, given a set of plans with the same CSE set, it keeps the plan with the minimal cost and prunes

the remaining plans. For example, for plans U_{12} and U_{13} in Table 5.4, we only need to keep the one with the smaller cost and prune the other one.

The second pruning technique prunes plans with different CSE sets. Specifically, consider two plans U_{ij} and U_{ik} with $CSE(U_{ij}) \neq CSE(U_{ik})$ and $Cost(U_{ij}) \leq Cost(U_{ik})$. Although U_{ij} and U_{ik} have different CSE sets, if U_{ij} can compute the results of the CSEs that are present in U_{ik} but not in itself (i.e., $CSE(U_{ik}) \setminus CSE(U_{ij})$) in some cost that is no greater than the value $Cost(U_{ik}) - Cost(U_{ij})$, then we can still prune U_{ik}. Let $Cost(CSE(U_{ik}) \setminus CSE(U_{ij}) \mid CSE(U_{ij}))$ denote the cost to compute the results of the CSEs in $CSE(U_{ik}) \setminus CSE(U_{ij})$ based on the results of the CSEs in $CSE(U_{ij})$. If we have $Cost(U_{ij}) + Cost(CSE(U_{ik}) \setminus CSE(U_{ij}) \mid CSE(U_{ij})) \leq C(U_{ik})$, then we can simply remove U_{ik}. For example, consider the two plans U_{14} and U_{15} in Table 5.4, if $Cost(U_{15}) + Cost(\{R_0 \bowtie R_1 \bowtie R_2\} \mid \{R_0, R_1\}) \leq Cost(U_{14})$, then we can simply prune U_{14}.

To compute $Cost(CSE(U_{ik}) \setminus CSE(U_{ij}) \mid CSE(U_{ij}))$, we have to compute the cost for evaluating the results of each CSE in $CSE(P_{ik}) \setminus CSE(U_{ij})$ based on the results of the CSEs in $CSE(U_{ij})$. Furthermore, if a CSE E' is a subexpression of another CSE E'', we should compute the results of E' first so that they could be reused to compute the results of E''. We assume $CSE(U_{ik}) \setminus CSE(U_{ij}) = \{E_1, \cdots, E_s\}$ and if a CSE E_i is a subexpression of another CSE E_j, then $i < j$. Then we have $Cost(CSE(U_{ik}) \setminus CSE(U_{ij}) \mid CSE(U_{ij})) = \sum_{g=1}^{s} Cost(E_g \mid CSE(U_{ij}) \bigcup \{E_1, \cdots, E_{g-1}\})$. Note that for each CSE E_g, it is associated with multiple interesting plans (maintained when applying the single query join enumeration algorithm) and the costs of these plans are updated based on the CSEs in $CSE(P_{ij}) \bigcup \{E_1, \cdots, E_{g-1}\}$ and the minimal cost is chosen as the cost of E_g. We now discuss how to update the cost of a plan based on a CSE set where the results of each CSE in the set have been materialized.

Algorithm 5.8 shows the pseudocode to update the cost of a plan U' based on a CSE set O' where the results of the CSEs in O' have been materialized. The main idea of the algorithm (denoted as CTUA) is to recursively traverse through the subplans of U' and check whether the results of the CSEs in O' can be reused for some subplans. Initially, it adds U' into a queue. Then it goes into an iterative process to retrieve the subplans of U'. In each iteration, it pulls a plan U'' from the queue. If $JoinExp(U'') \in O'$, then the results of U'' have already been materialized and the cost of U' is updated. Otherwise, if O' has overlap with the CSE set of U'', then the subplans of U'' (i.e., $SubPlan(U'')$) are added into the queue since their results may have already been materialized. The iterative process terminates when the queue is empty.

Algorithm 5.8: Cost updating algorithm (CTUA)

Input: A plan U', a CSE set O'
Output: Updated cost of U' based on the results of the CSEs in O'

```
1  cost = Cost(U') ;
2  queue = newQueue() ;
3  queue.add(U') ;
4  while queue ≠ ∅ do
5  |   U'' = queue.poll() ;
6  |   if JoinExp(U'') ∈ O' then
7  |   |   cost = cost - Cost(U'') ;
8  |   else if CSE(U'') ⋂ O' ≠ ∅ then
9  |   |   queue.addAll(SubPlan(U'')) ;
10 return cost
```

Optimization. As the queries in the batch $\mathcal{Q}$ may have many CSEs, it is unnecessary to redundantly optimize these CSEs in the first phase. For example, consider the CSE $R_0 \bowtie R_1 \bowtie R_2$ for Q_1 and Q_2, after optimizing Q_1, the interesting plans for the CSE are maintained in Q_1's hash table. When optimizing the CSE for Q_2, instead of redundantly optimizing it, we can reuse the results from Q_1's hash table for Q_2. In this way, we share the optimization of CSEs among the batch of queries.

5.5.2 Second Phase

In the second phase, we stitch the interesting plans for the queries maintained in the first phase into a global optimal plan. Our approach constructs the global optimal plan progressively, i.e., the global plans for a set of i queries are constructed before that for a set of $i+1$ queries. Similarly, we maintain a CSE set for each global plan where the results of the CSEs in the set could be reused in future computation. Overall, we construct the global optimal plan by evaluating the expression $(((I_{W_1} \times I_{W_2}) \times I_{W_3}) \times \cdots \times I_{W_n})$ with intermediate global plans being materialized and pruned (via the two pruning techniques). In this way, we are able to prune the combination space of the interesting plans for the batch of queries. Let $M_i \subseteq ((I_{W_1} \times I_{W_2}) \times \cdots \times I_{W_i})$ $(2 \leq i \leq n)$ denote the interesting global plans maintained for the set of queries $\{Q_1, \cdots, Q_i\}$.

Algorithm 5.9 shows the pseudocode to generate the global optimal plan for a batch of queries $\mathcal{Q} = \{Q_1, \cdots, Q_n\}$. In the i^{th} iteration to examine the expression $M_{i-1} \times I_{W_i}$ to construct the global plans for the set of queries $\{Q_1, \cdots, Q_i\}$ $(2 \leq i < n)$, we need to compute both the costs and CSE sets for the global plans. Specifically, consider a combination of plans $(U', U'') \in (M_{i-1} \times I_{W_i})$, its global plan (denoted as GP) is constructed

Algorithm 5.9: Global optimal plan generation algorithm

Input: Interesting plans for each Q_i in $\mathcal{Q} = \{Q_1, \cdots, Q_n\}$ (i.e., $I_{W_i}, \cdots, I_{W_n}$)
Output: The global optimal plan for $\mathcal{Q}$ (i.e., M_n)

```
1 M_1 = I_{W_1} ;
2 for i = 2 → n do
3     foreach (U', U'') ∈ (M_{i-1} × I_{W_i}) do
4         Let GP denote the global plan for (U', U'') ;
5         Cost(GP) = Cost(U') + CTUP(U'', CSE(U')) (i.e., Algorithm 5.8) ;
6         CSE(GP) = CSE(U') + CSUA(U'', CES(U')) (i.e., Algorithm 5.10);
7         M_i = M_i ⋃ GP ;
8     Remove the CSEs in M_i that can not be reused in future computation ;
9     Apply the two pruning techniques for M_i ;
```

by checking the reusable results of the CSEs in $CSE(U')$ for U'' and updating both the cost and CSE set for GP. The cost of GP is simply the cost of U' plus the updated cost of U'' based on the results of the CSEs in $CSE(U')$ which has already been discussed in Algorithm 5.8. Similarly, the CSE set of GP is simply the CSE set of U' union the updated CSE set of U'' based on the results of the CSEs in $CSE(U')$. Algorithm 5.10 shows the pseudocode to update the CSE set of a plan U' based on a CSE set O' where the results of the CSEs in O' are materialized. Similar to Algorithm 5.8, the main idea of the algorithm (denoted as $CSUA$) is to traverse through the subplans of U', check the reusable results of the CSEs in O' for some subplans and update the CSE set of U'. Since the algorithm is very similar to Algorithm 5.8, we do not repeatedly discuss it.

Algorithm 5.10: CSE set updating algorithm (CSUA)

Input: A plan U', a CSE set O'
Output: Updated CSE set of U' based on the results of the CSEs in O'

```
1  cse = CSE(U') ;
2  queue = newQueue() ;
3  queue.add(U') ;
4  while queue ≠ ∅ do
5      U'' = queue.poll() ;
6      if JoinExp(U'') ∈ O' then
7          cse.removeAll(CSE(U'')) ;
8      else if cse ⋂ CSE(U'') ≠ ∅ then
9          queue.addAll(SubPlan(U'')) ;
10 return cse
```

After computing M_i in the i^{th} iteration, we apply the optimization technique (to be discussed) to remove the CSEs in M_i that can not be reused in future computation and apply the two pruning techniques for M_i to prune away the non-promising plans. Note that after the termination of the algorithm, M_n contains only one plan which is the global optimal plan for $\mathcal{Q} = \{Q_1, \cdots, Q_n\}$.

For example, consider the construction of the global plan for the plans (U_{16}, U_{22}) in Table 5.4, the global plan will reuse the results of the CSE $R_0 \bowtie R_1 \bowtie R_2$ from U_{16} for U_{22}. Therefore, the cost and the CSE set of the global plan are simply $Cost(U_{16})$ and $CSE(U_{16})$ respectively.

Optimization. When constructing the global optimal plan for a batch of queries, to further reduce the plan combination space, we remove the CSEs which can not be reused in future computation as early as possible to enhance the effectiveness of the two pruning techniques. For example, after the evaluation of $I_{W_1} \bowtie I_{W_2}$ in Table 5.4, the CSE $R_0 \bowtie R_1 \bowtie R_2$ can not be reused in the evaluation of $M_2 \bowtie I_{W_3}$ and thus can be removed in all the plans in M_2. This early CSE removal optimization will help to prune more plans when applying the proposed two pruning techniques. For example, before applying the optimization, neither the global plan for (U_{15}, U_{21}) nor the global plan for (U_{15}, U_{22}) can be pruned if their costs do not meet certain criterion (as discussed in Section 5.5.1) since their CSE sets (i.e., $CSE(U_{21})$ and $CSE(U_{22})$ respectively) are different. However, since all the CSEs in $\{R_0 \bowtie R_1, R_2 \bowtie R_4, R_0 \bowtie R_1 \bowtie R_2\}$ can not be reused when computing $M_2 \times I_3$, we can remove these CSEs and the two global plans become comparable and only the one with a smaller cost need to be maintained. To achieve this, after the first phase, for each CSE, we maintain an inverted list of queries where each query has at lease one interesting plan with its CSE set containing the CSE. In the second phase, each time when we finish evaluating the expression $M_{i-1} \bowtie I_{W_i}$, we remove Q_i from all the inverted lists it appears. A CSE can be removed from the global plans if its inverted list is empty.

5.6 Experimental Results

In this section, we present an experimental study to evaluate the efficiency of our join enumeration algorithms in terms of query optimization time. Sections 5.6.1 and 5.6.2 respectively study the efficiency of our single-query and multi-query join enumeration algorithms. All the algorithms were implemented in Java and the experiments were performed on an Intel Dual Core 2.33GHz machine with 3.2GB of RAM running Linux.

Generator. We generated different types of queries for our experiments including chain, cycle, star, clique as well as random acyclic and cyclic queries. The random acyclic queries were generated using the approach in [67]. To generate random cyclic queries, we first generated random acyclic queries and then added additional edges into the queries

Table 5.5: Query generation parameters

Relation cardinality ($*10^6$)	Prob.	Domain size	Prob.
10-100	15%	2-10	5%
100-1,000	30%	10-100	50%
1,000-10,000	25%	100-500	35%
10,000-100,000	20%	500-1000	15%

to form cycle. Note that all the edges were uniformly chosen to be added. The default number of edges in a random cyclic query was $(N-1)+\lceil\frac{N(N-1)}{2}\times 0.05\rceil$ where N is the number of relations in the query (i.e., the $(N-1)$ edges were used to generate a random acyclic query and an additional 5 percentage of edges were added in the query to form cycle).

Following the discussion from [56, 43, 24], for each query, we generated random relations and added random attributes with random domain sizes using the parameters in Table 5.5. Furthermore, we generated both foreign-key join predicates as well as non-foreign-key join predicates. For each foreign-key join predicate, its selectivity factor is estimated such that the cardinality of the join result is equal to the cardinality of the relation with the foreign key. For each non-foreign-key join predicate on attributes A_1 and A_2, its selectivity factor is estimated using $\frac{1}{max(dom(A_1),dom(A_2))}$ where $dom(A_i)$ $(i \in \{1,2\})$ is the domain size of A_i. The details of the query generation approach can be found in [56, 43, 24]. Each query was run five times and its average running times was reported. For random acyclic and cyclic queries, since the running times of generated queries vary due to the different query graphs, we report the average running time for 10 random generated queries.

Cost model. To estimate the cost of multi-way joins in the MapReduce framework, we used the cost model in [5] to estimate the communication cost of multi-way joins (i.e., the cost to transfer the map output from map tasks to reduce tasks) and the cost model in Chapter 4.4 to estimate the remaining cost of multi-way joins including map output sorting cost, map input reading cost and so on.

5.6.1 Efficiency of Single-Query Join Enumeration Algorithm

In this section, we study the efficiency of our single query join enumeration algorithm. Figure 5.2 compares our algorithm DPopt against the two baseline algorithms DPset and DPsize for different query types as a function of number of relations in the queries. Note

Table 5.6: Improvement factor of DPopt over DPset

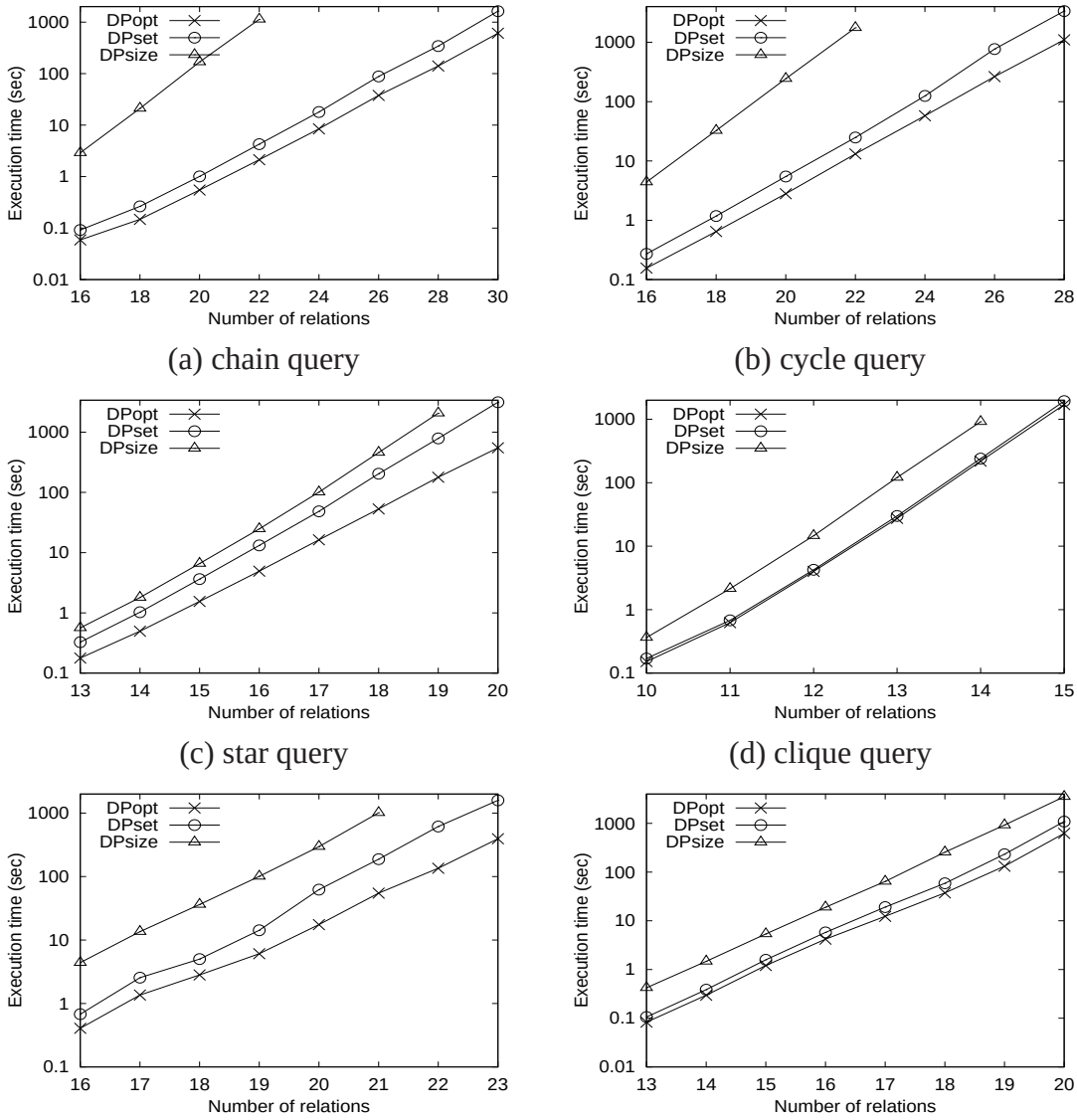
Query type	Minimum	Average	Maximum
Chain	56%	110%	168%
Cycle	74%	122%	206%
Star	84%	223%	473%
Clique	6%	10%	14%
Random acyclic	67%	190%	354%
Random cyclic	28%	49%	76%

that since DPsize ran very slowly, we do not show its running times in Figure 5.2 if it did not finish running within 1 hour. We summarize the results as follows.

First, comparing the two baseline algorithms, DPset significantly outperforms DPsize by up to two orders of magnitude. For example, when the number of relations in a chain query is 22, the running times of DPset and DPsize are respectively 4.3s and 1136.7s. This is due to the large number of integer partitions generated for DPsize which results in a large number of disjointedness checking as explained in Section 5.4.1. For example, when the number of relations in a chain query is 15, DPsize generates 668 integer partitions which results in 17.120.334 disjointedness checking while DPset only needs to check the connectedness 458.073 times. Since DPsize always runs significantly slower than DPset, we focus on our comparison for DPopt and DPset in the following.

Second, as the number of relations in the queries increases, the running times of both DPopt and DPset increase. However, the running time of DPset increases much faster than the running time of DPopt which therefore increases the winning margin of DPopt over DPset. For example, for star queries, the wining percentages of DPopt over DPset are respectively 84%, 107%, 135%, 168%, 197%, 284%, 341% and 473% when the number of relations in the query are 13, 14, 15, 16, 17, 18, 19 and 20. Table 5.6 shows the average, minimum and maximum wining percentages of DPopt over DPset for different query types for the experiments in Figure 5.2. Note that even in the worst case for clique queries, DPopt still outperforms DPset by 10% on average.

Third, the running time of random acyclic queries falls between the running time of chain queries and that of star queries. This is expected since chain queries are the simplest acyclic queries while star queries are the most complex acyclic queries in terms of time complexity. Similarly, the running time of random cyclic queries falls between the running time of cycle queries and that of clique queries. Again here, the reason is that cycle queries are the simplest cyclic queries while clique queries are the most complex cyclic queries in terms of time complexity.



(a) chain query

(b) cycle query

(c) star query

(d) clique query

(e) random acyclic queries

(f) random cyclic queries

Figure 5.2: Efficiency of single query join enumeration algorithms

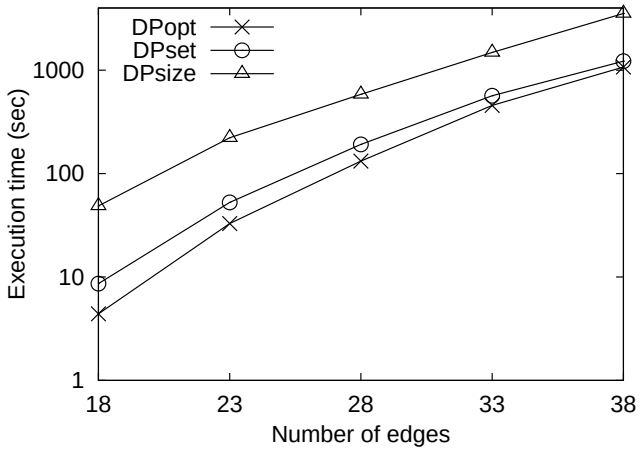
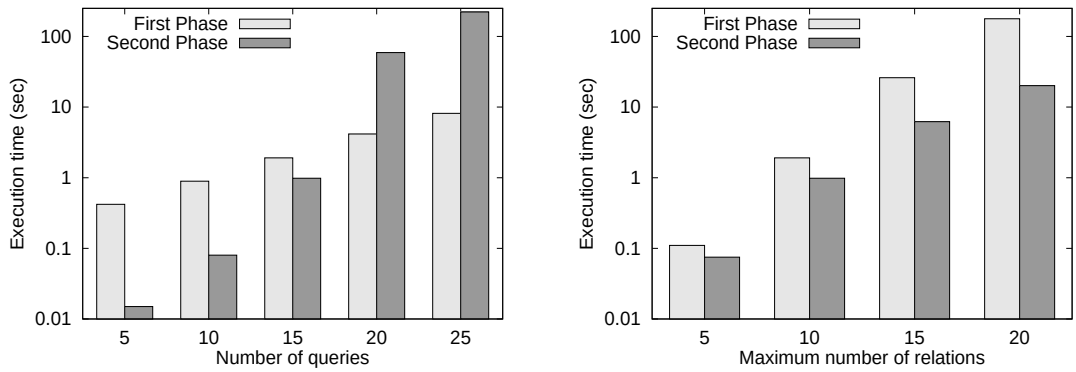


Figure 5.3: Effect of number of edges

In additional to the above experiments, for random cyclic queries, we also conducted an experiment to show the effect of number of edges in the queries. Figure 5.3 shows the running times as a function of number of edges for cyclic queries with 18 relations. As the number of edges in the queries increases, the running times of both DPopt and DPset increase. This is expected since for DPopt, as the number of edges in the queries increases, it generates more partitions which requires more time to enumerate. For DPset, as the number of edges in the queries increases, although the number of candidate partitions remains the same, the connectedness checking for a candidate partition requires more time since more connected subsets and query plans are stored in the hash table. However, the running time of DPopt increases faster than the running time of DPset which therefore decreases the winning margin of DPopt over DPset. For example, the winning percentages of DPopt over DPset are respectively 96%, 60%, 45%, 24% and 14% when the number of edges are 18, 23, 28, 33 and 38.

5.6.2 Efficiency of Multi-Query Join Enumeration Algorithm

In this section, we study the efficiency of our proposed multi-query join enumeration algorithm. To generate a batch of queries, we first generated N (the default value is 10) relations. As discussed previously, we then generated the cardinalities for each relation as well as the selectivity factors for each pair of relations representing a join predicate between them. Finally, each query in a batch was generated as follows: we first randomly chose a subset of the N relations for the query and then generated a random acyclic query for the chosen relations. We chose random acyclic queries since they are more common in real life applications. For example, 20 out of the 22 TPCH queries are acyclic queries.



(a) Effect of number of queries (b) Effect of number of relations N

Figure 5.4: Efficiency of multi-query join enumeration algorithm

The default number of queries in a batch is 15. For each size of query batch, we randomly generated 20 batches and their average running time is reported.

Figures 5.4(a) and 5.4(b) respectively show the efficiency of our multi-query join enumeration algorithm as a function of number of queries and number of relations (i.e., N) in a query batch. For example, when $N = 10$ and the number of queries in a batch is 20, it took only 32 seconds to optimize the queries. Note that we separately report the running times for the two phases of our algorithm. Furthermore, the running times of the first and second phases can dominate each other in different settings depending on amount of sharing among the queries. Specifically, if a batch of queries have a lot of CSEs, then the second phase will run slower than the first phase since the first phase will generate more interesting plans which requires more time to merge in the second phase. For example, in Figure 5.4(a), the first phase took longer time to run than the second phase when the number of queries in a batch are 5, 10 and 15. However, when the number of queries are 20 and 25, the second phase ran longer than the first phase. This is because the number of CSEs becomes larger when the number of queries increases which results in more interesting plans. For example, the number of interesting plans generated in the first phase are respectively 12, 80, 149, 435 and 620 when the number of queries are 5, 10, 15, 20 and 25.

To demonstrate the effectiveness of our proposed two pruning techniques (to prune away non-promising interesting plans), we compare against a naive solution that generates and stitches the interesting plans without any pruning. Our experimental results show that the naive solution consumes a lot of memory space and runs very slowly (due to the large number of interesting plans generated in the first phase and materialized in the second phase). For example, in the default setting, our approach consumed about 30MB Java

heap space and took 3 seconds to run while the naive solution ran out of Java heap space [9] after running 6 minutes. Thus, to enable the naive solution to finish running within the capacity of Java heap size, we set $N = 5$. Note that even for this setting, the naive solution ran out of Java heap size after running about 6 minutes when the number of queries in a batch is no smaller than 15. Thus, we only report the running times when the number of queries in a batch are 5 and 10. The running times of our approach are respectively 8 and 30 milliseconds when the number of queries in a batch are 5 and 10 while that for the naive solution are respectively 140 and 990 milliseconds which demonstrates that our approach is at least one order of magnitude faster than the naive solution due to the proposed pruning techniques. Furthermore, the number of generated interesting plans in the first phase for our approach are respectively 12 and 36 when the number of queries in a batch are 5 and 10 while that for the naive solution are respectively 50 and 111.

5.7 Summary

In this chapter, we have presented a comprehensive study of the OJE problem in the MapReduce framework. We have studied both the SOJE and MOJE problems and proposed efficient join enumeration algorithms for these problems. Our experimental results demonstrate that our proposed single query join enumeration algorithm significantly outperforms the baseline algorithms by up to 473%, and our proposed multi-query join enumeration algorithm is able to scale up to 25 queries where the number of relations in the queries ranges from 1 to 10.

[9] Given the capacity of 3.2GB RAM, we set Java heap size to be 2.5GB which is also the maximum allowed heap size for our system.

CHAPTER 6

CONCLUSION

In this thesis, we studied three problems using novel MQQ techniques, namely, efficient processing of enumerative set-based queries, multi-query optimization in MapReduce framework and optimal join enumeration in MapReduce framework. In this chapter, we summarize our works and highlight some interesting works that are worthy of further exploration.

6.1 Contributions

Our first contribution is the study of efficient evaluation techniques for enumerative set-based queries (SQs). While enumerative SQs can be expressed using SQL, existing relational engines, unfortunately, were not able to efficiently optimize and evaluate such queries due to their complexity as demonstrated by our experimental results. Then we proposed a novel evaluation approach for enumerative SQs. The key idea is to first partition the input table based on the different combinations of constraints satisfied by the tuples and then compute the answer sets by appropriate combinations of the partitions. In this way, an enumerative SQ is evaluated as a collection of cross-product queries (CPQs). We presented efficient and scalable MQO heuristics to optimize the evaluation

of a collection of CPQs. Our experimental results on Postgresql demonstrated that our proposed approach significantly outperform the baseline solutions by up to three orders of magnitude.

Our second contribution is the study of multi-query/job optimization techniques and algorithms for a batch of MapReduce jobs. We first proposed two new techniques for multi-job optimization in the MapReduce framework. The first technique is a generalized grouping technique (which generalizes the recently proposed MRShare technique) that merges multiple jobs into a single job thereby enabling the merged jobs to share both the scan of the input file as well as the communication of the common map output. The second technique is a materialization technique that enables multiple jobs to share both the scan of the input file as well as the communication of the common map output via partial materialization of the map output of some jobs (in the map and/or reduce phase). Then we proposed a new optimization algorithm that given an input batch of jobs, produces an optimal plan by a judicious partitioning of the jobs into groups and an optimal assignment of the processing technique to each group. Our experimental results on Hadoop demonstrated that our new approach significantly outperforms the state-of-the-art technique, MRShare, by up to 107%.

Our third contribution is the study of the optimal join enumeration (OJE) problem and proposed efficient join enumeration algorithms for the problem in the MapReduce paradigm. We first studied the SOJE problem which serves as a foundation for our study on the MOJE problem. Specifically, we first studied the complexity of the SOJE problem in the presence of multi-way joins for different query graph types (chain, cycle, type and clique). We then proposed both bottom-up and top-down join enumeration algorithms for the SOJE problem with an optimal complexity w.r.t. the query graph based on a proposal of an efficient and easy-to-implement plan enumeration algorithm. Based on the proposed single-query join enumeration algorithm, we then presented an efficient multi-query join enumeration algorithm. Our experimental results demonstrated the efficiency of our proposed algorithms.

6.2 Future Work

In this section, we discuss some interesting future directions related to the problems examined in this thesis.

Finding interesting answer sets for enumerative SQs. Since the number of answer sets for some enumerative SQs could be very large, it is essential to help users to browse through all the "interesting" answer sets. Two standard criteria for "interestingness" in the database context are top-k [19] and skyline [11] . Thus, one interesting direction is to examine the evaluation of top-k enumerative SQs. In particular, if the ranking function F is a distributive monotone function as defined in Section 3.7.2, then the sort-based evaluation can be optimized as follows. In the partitioning phase, we generate partitions that are sorted on $F(t)$ by sorting the input relation on the composite key $(pid, F(t))$ where pid is the assigned partition identifier. In the enumeration phase, we apply existing rank join algorithms [35] to incrementally produce the ranked answer sets for each vpset and apply the well-known TA algorithm [19] to retrieve the top-k answer sets for all the vpsets.

Another interesting direction is to investigate the set skyline operator in conjunction with our work to retrieve non-dominated sets which is essentially a generalization of the tuple skyline operator [11] to retrieve non-dominated tuples. To evaluate the set skyline operator in conjunction with enumerative SQs, the most general approach is to first enumerate all the answer sets for enumerative SQs using our proposed approach followed by pruning way the dominated sets. While there has been one preliminary work [71] to integrate these two works to improve the query performance, their work is very limited by assuming either fixed set cardinality or in-memory data which thus can not be applied for our problem. As a result, we plan to investigate techniques to integrate these two works to reduce the evaluation cost for both the set skyline operator as well as the enumerative SQs.

Comprehensive optimization framework in the MapReduce paradigm. Our work on the MOJE problem focuses on CSEs that produce the same results. However, in real life applications, it is common to have some subexpressions whose results have overlap or containment relationships. We denote these subexpressions as sharable subexpressions (SSEs). To explore the sharing for both CSEs and SSEs, a simplistic solution is to apply a two-phase approach. The first phase translates the queries into jobs to share the computation of the CSEs using our multi-query join enumeration algorithm. The second phase applies our multi-job optimization techniques on the translated jobs to share the computation of the SSEs. However, this two-phase solution is suboptimal since we do not consider the SSEs when we choose the global plan in the first phase. Thus, an interesting direction for future work is to investigate a single phase approach to choose the global optimal plan for a batch of queries to share the computation of both CSEs and SSEs.

BIBLIOGRAPHY

[1] http://musicbrainz.org/doc/musicbrainz_database. 43

[2] http://www.aster.com/. 6

[3] http://www.greenplum.com/. 6

[4] http://www.linux.com/learn/tutorials/394523-configuring-postgresql-for-pretty-good-performance. 44

[5] Foto N. Afrati and Jeffrey D. Ullman. Optimizing joins in a map-reduce environment. In *EDBT*, pages 99–110, 2010. 14, 16, 82, 110

[6] A.I.Serdyukov. An algorithm with an estimate for the traveling salesman problem of maximum (in russian). In *Upravlyaemye Sistemy*, pages 80–86, 1984. 36

[7] Senjuti Basu Roy, Sihem Amer-Yahia, Ashish Chawla, Gautam Das, and Cong Yu. Constructing and exploring composite items. In *SIGMOD*, pages 843–854, 2010. 3, 4, 12

[8] Daniel Berend and Tamir Tassa. Improved bounds on bell numbers and on moments of sums of random variables. *Probability and Mathematical Statistics*, 30(2):185–205, 2010. 93, 94

[9] Maxim Binshtok, Ronen I. Brafman, Solomon E. Shimony, Ajay Martin, and Craig Boutilier. Computing optimal subsets. In *AAAI*, pages 1231–1236, 2007. 4

[10] Spyros Blanas, Jignesh M. Patel, Vuk Ercegovac, Jun Rao, Eugene J. Shekita, and Yuanyuan Tian. A comparison of join algorithms for log processing in mapreduce. In *SIGMOD*, pages 975–986, 2010. 16

[11] Stephan Börzsönyi, Donald Kossmann, and Konrad Stocker. The skyline operator. In *ICDE*, pages 421–430, 2001. 118

[12] Upen S. Chakravarthy and Jack Minker. Multiple query processing in deductive databases using query graphs. In *VLDB*, pages 384–391, 1986. 2, 3, 12

[13] Fa-Chung Fred Chen and Margaret H. Dunham. Common subexpression processing in multiple-query processing. *TKDE*, 10(3):493–499, 1998. 12, 31

[14] Nilesh N. Dalvi, Sumit K. Sanghai, Prasan Roy, and S. Sudarshan. Pipelining in multi-query optimization. *J. Comput. Syst. Sci.*, 66(4):728–762, 2003. 1, 2, 3, 6, 7, 12, 13, 15, 16, 31, 88, 103

[15] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004. 2, 10, 16

[16] David DeHaan and Frank Wm. Tompa. Optimal top-down join enumeration. In *SIGMOD*, pages 785–796, 2007. 6, 7, 15, 87, 88, 89, 90, 92

[17] Marie Desjardins and Kiri L. Wagstaff. Dd-pref: A language for expressing preferences over sets. In *AAAI*, pages 620–626, 2005. 4

[18] Iman Elghandour and Ashraf Aboulnaga. Restore: Reusing results of mapreduce jobs. *PVLDB*, 5(6):586–597, 2012. 2, 5, 14, 16, 56

[19] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. In *PODS*, pages 102–113, 2001. 118

[20] Leonidas Fegaras, Chengkai Li, and Upa Gupta. An optimization framework for map-reduce queries. In *EDBT*, pages 26–37, 2012. 2, 14, 55

[21] Pit Fender and Guido Moerkotte. A new, highly efficient, and easy to implement top-down join enumeration algorithm. In *ICDE*, pages 864–875, 2011. 6, 7, 15, 87, 88, 89, 90, 92

[22] Pit Fender and Guido Moerkotte. Reassessing top-down join enumeration. *TKDE*, 24(10):1803–1818, 2012. 6, 7, 15, 87, 88, 89, 90, 92

[23] Pit Fender and Guido Moerkotte. Top down plan generation: From theory to practice. In *ICDE*, pages 1105–1116, 2013. 6, 7, 15, 16, 87, 88, 89, 90, 92

[24] Pit Fender, Guido Moerkotte, Thomas Neumann, and Viktor Leis. Effective and robust pruning for top-down join enumeration algorithms. In *ICDE*, pages 414–425, 2012. 6, 7, 15, 87, 88, 89, 90, 92, 110

[25] Michael J. Franklin, Björn Thór Jónsson, and Donald Kossmann. Performance tradeoffs for client-server query processing. *SIGMOD Rec.*, 25(2):149–160, 1996. 87

[26] Alan F. Gates, Olga Natkovich, Shubham Chopra, Pradeep Kamath, Shravan M. Narayanamurthy, Christopher Olston, Benjamin Reed, Santhosh Srinivasan, and Utkarsh Srivastava. Building a high-level dataflow system on top of map-reduce: The pig experience. *PVLDB*, 2(2):1414–1425, 2009. 2, 5, 14, 55

[27] John Grant and Jack Minker. On optimizing the evaluation of a set of expressions. *International Journal of Parallel Programming*, 11(3):179–191, 1982. 2, 3, 12

[28] Anja Gruenheid, Edward Omiecinski, and Leo Mark. Query optimization using column statistics in hive. In *IDEAS*, pages 97–105, 2011. 14

[29] Sudipto Guha and Divesh Srivastava. Efficient approximation of optimization queries under parametric aggregation constraints. In *VLDB*, pages 778–789, 2003. 3, 4, 12

[30] Himanshu Gupta, Bhupesh Chawda, Sumit Negi, Tanveer A. Faruquie, L. V. Subramaniam, and Mukesh Mohania. Processing multi-way spatial joins on map-reduce. In *EDBT*, pages 113–124, 2013. 14, 16

[31] P. A. V. Hall. Optimization of single expressions in a relational data base system. *IBM Journal of Research and Development*, 20(3):244–257, 1976. 12

[32] Herodotos Herodotou and Shivnath Babu. Profiling, what-if analysis, and cost-based optimization of mapreduce programs. *PVLDB*, 4(11):1111–1122, 2011. 13

[33] Herodotos Herodotou, Fei Dong, and Shivnath Babu. Mapreduce programming and cost-based optimization? crossing this chasm with starfish. *PVLDB*, 4(12):1446–1449, 2011. 13

[34] Maurice A. W. Houtsma and Arun N. Swami. Set-oriented mining for association rules in relational databases. In *ICDE*, pages 25–33, 1995. 22, 23

[35] Ihab F. Ilyas, George Beskales, and Mohamed A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Comput. Surv.*, 40(4):11:1–11:58, 2008. 118

[36] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, pages 261–276, 2009. 11

[37] Eaman Jahani, Michael J. Cafarella, and Christopher Ré. Automatic optimization for mapreduce programs. *PVLDB*, 4(6):385–396, 2011. 13

[38] Jeffrey Jestes, Ke Yi, and Feifei Li. Building wavelet histograms on large data in mapreduce. *PVLDB*, 5(2):109–120, 2011. 56

[39] Theodoros Lappas, Kun Liu, and Evimaria Terzi. Finding a team of experts in social networks. In *KDD*, pages 467–476, 2009. 3, 4, 12

[40] Harold Lim, Herodotos Herodotou, and Shivnath Babu. Stubby: A transformation-based optimizer for mapreduce workflows. *PVLDB*, 5(11):1196–1207, 2012. 2, 3, 5, 14, 15, 16, 56

[41] Guido Moerkotte. Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy join trees without cross products. In *VLDB*, pages 930–941, 2006. 6, 7, 15, 87, 88, 89, 90, 92, 95, 96, 98, 99, 100, 102

[42] Guido Moerkotte and Thomas Neumann. Dynamic programming strikes back. In *SIGMOD*, pages 539–552, 2008. 6, 7, 15, 16, 87, 88, 89, 90, 92

[43] Thomas Neumann. Query simplification: Graceful degradation for join-order optimization. In *SIGMOD*, pages 403–414, 2009. 110

[44] Tomasz Nykiel, Michalis Potamias, Chaitanya Mishra, George Kollios, and Nick Koudas. Mrshare: Sharing across multiple queries in mapreduce. *PVLDB*, 3(1-2):494–505, 2010. 1, 2, 3, 5, 14, 16, 54, 55, 56, 57, 79

[45] Alper Okcan and Mirek Riedewald. Processing theta-joins using mapreduce. In *SIGMOD*, pages 949–960, 2011. 14, 16

[46] Christopher Olston, Benjamin Reed, Adam Silberstein, and Utkarsh Srivastava. Automatic optimization of parallel dataflow programs. In *ATC*, pages 267–273, 2008. 2, 14

[47] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: A not-so-foreign language for data processing. In *SIGMOD*, pages 1099–1110, 2008. 2, 5, 14, 55

[48] Kiyoshi Ono and Guy M. Lohman. Measuring the complexity of join enumeration in query optimization. In *VLDB*, pages 314–325, 1990. 6, 7, 15, 87, 88, 89, 93

[49] Jooseok Park and Arie Segev. Using common subexpressions to optimize multiple queries. In *ICDE*, pages 311–319, 1988. 2, 3, 12, 16, 31

[50] Sriram Pemmaraju and Steven Skiena. *Computational Discrete Mathematics: Combinatorics and Graph Theory with Mathematica*. Cambridge University Press, 2003. 93, 94

[51] Prasan Roy, S. Seshadri, S. Sudarshan, and Siddhesh Bhobe. Efficient and extensible algorithms for multi query optimization. *SIGMOD Rec.*, 29(2):249–260, 2000. 1, 2, 3, 6, 7, 12, 13, 15, 16, 31, 88, 103

[52] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD*, pages 23–34, 1979. 15, 92, 95, 99

[53] Timos K. Sellis. Global query optimization. In *SIGMOD*, pages 191–205, 1986. 12

[54] Timos K. Sellis. Multiple-query optimization. *TODS*, 13(1):23–52, 1988. 1, 2, 3, 12, 16, 31

[55] Yingjie Shi, Xiaofeng Meng, Fusheng Wang, and Yantao Gan. Hedc: a histogram estimator for data in the cloud. In *CloudDb*, pages 51–58, 2012. 56

[56] Michael Steinbrunn, Guido Moerkotte, and Alfons Kemper. Heuristic and randomized optimization for the join ordering problem. *The VLDB Journal*, 6(3):191–208, 1997. 110

[57] Kian-Lee Tan and Hongjun Lu. Workload scheduling for multiple query processing. *Inf. Process. Lett.*, 55(5):251–257, 1995. 12, 19, 31

[58] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive - a warehousing solution over a map-reduce framework. *PVLDB*, 2(2):1626–1629, 2009. 2, 5, 14, 55

[59] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Antony, and Hao Liu. Hive-a petabyte scale data warehouse using hadoop. In *ICDE*, pages 996–1005, 2010. 2, 5, 6, 14, 55

[60] Quoc Trung Tran, Chee-Yong Chan, and Guoping Wang. Evaluation of set-based queries with aggregation constraints. In *CIKM*, pages 1495–1504, 2011. 3, 4, 12

[61] Tolga Urhan and Michael J. Franklin. Xjoin: A reactively-scheduled pipelined join operator. *IEEE Data Eng. Bull*, 23(2):27–33, 2000. 39

[62] Bennet Vance and David Maier. Rapid bushy join-order optimization with cartesian products. In *SIGMOD*, pages 35–46, 1996. 15, 95, 96, 98, 99, 100

[63] Rares Vernica, Michael J. Carey, and Chen Li. Efficient parallel set-similarity joins using mapreduce. In *SIGMOD*, pages 495–506, 2010. 14

[64] Stratis D. Viglas, Jeffrey F. Naughton, and Josef Burger. Maximizing the output rate of multi-way join queries over streaming information sources. In *VLDB*, pages 285–296, 2003. 39

[65] Guoping Wang and Chee Yong Chan. Multi-query optimization in mapreduce framework. *PVLDB*, 7(3):145–156, 2013. 8

[66] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, 2009. 81

[67] David Bruce Wilson. Generating random spanning trees more quickly than the cover time. In *STOC*, pages 296–303, 1996. 109

[68] Sai Wu, Feng Li, Sharad Mehrotra, and Beng Chin Ooi. Query optimization for massively parallel data processing. In *SOCC*, pages 12:1–12:13, 2011. 14, 16, 56, 82

[69] Min Xie, Laks V. S. Lakshmanan, and Peter T. Wood. Composite recommendations: from items to packages. *Frontiers of computer science*, 6(3):264–277, 2012. 3, 4, 12

[70] Min Xie, Laks V.S. Lakshmanan, and Peter T. Wood. Breaking out of the box of recommendations: from items to packages. In *RecSys*, pages 151–158, 2010. 3, 4, 12

[71] Xi Zhang and J. Chomicki. Preference queries over sets. In *ICDE*, pages 1019–1030, 2011. 4, 12, 118

[72] Xiaofei Zhang, Lei Chen, and Min Wang. Efficient multi-way theta-join processing using mapreduce. *PVLDB*, 5(11):1184–1195, 2012. 14, 16

[73] 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. 2, 3, 12, 19, 31

[74] 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. 1, 2, 3, 6, 7, 12, 13, 15, 16, 31, 88, 103

[75] Antoine Zoghbi and Ivan Stojmenovic. Fast algorithms for generating integer partitions. *International Journal of Computer Mathematics*, 70(2):319–332, 1998. 96