2005-08-19

Scalable Integration View Computation and Maintenance with Parallel, Adaptive and Grouping Techniques

Bin Liu
Worcester Polytechnic Institute

Scalable Integration View Computation and Maintenance with Parallel, Adaptive and Grouping Techniques

by

Bin Liu

A Dissertation

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

in

Computer Science

by

August 4, 2005

APPROVED:

Prof. Elke A. Rundensteiner
Advisor

Prof. David Finkel
Committee Member

Prof. Murali Mani
Committee Member

Dr. Paul Larson
Microsoft Research
External Committee Member

Prof. Michael Gennert
Head of Department

Abstract

Materialized integration views constructed by integrating data from multiple distributed data sources help to achieve better access, reliable performance, and high availability for a wide range of applications. In this dissertation, we propose parallel, adaptive, and grouping techniques to address scalability challenges in high-performance integration view computation and maintenance due to increasingly large data sources and high rates of source updates.

State-of-the-art parallel integration view computation makes the common assumption that the maximal pipelined parallelism leads to superior performance. We instead propose *segmented bushy* parallel processing that combines pipelined parallelism with alternate forms of parallelism to achieve an overall more effective strategy. Experimental studies conducted over a cluster of high-performance PCs confirm that the proposed strategy has an on average of 50% improvement in terms of total processing time in comparison to existing solutions.

Run-time adaptation becomes critical for parallel integration view computation due to its long running and memory intensive nature. We investigate two types of state level adaptations, namely, *state spill* and *state relocation*, to address the run-time memory shortage. We propose *lazy-disk* and *active-disk* approaches that integrate both adaptations to maximize run-time query throughput in a memory constrained environment. We also propose *global throughput-oriented* state adaptation strategies for computation plans with multiple state intensive operators. Extensive experiments confirm the effectiveness of our proposed adaptation solutions.

Once results have been computed and materialized, it's typically more efficient to maintain them incrementally instead of full recomputation. However, state-of-the-art incremental view maintenance require O(n^2) maintenance queries with n being the number of data sources that the view is defined upon. Moreover, they do not exploit view definitions and data source processing capabilities to further improve view maintenance performance. We propose novel *grouping* maintenance algorithms that dramatically reduce the number of maintenance queries to (O(n)). A cost-based view maintenance framework has been proposed to generate optimized maintenance plans tuned to particular environmental settings. Extensive experimental studies verify the effectiveness of our maintenance algorithms as well as the maintenance framework.

Acknowledgments

This dissertation and the growth in my knowledge over the last few years owe a great deal to many professors, colleagues, and friends. First among them is my advisor, Prof. Elke A. Rundensteiner. She inspired my interests in database research and gave me direction by suggesting interesting problems. It has been my luck to have her as my advisor. Her technical and editorial advice was essential to the completion of this dissertation. I express my sincere thanks for her support, advice, patience, and encouragement throughout my graduate studies. Her persistence in tackling problems, confidence, and great teaching will always be an inspiration.

My thanks go to the members of my Ph.D. committee, Prof. David Finkel, Prof. Murali Mani and Dr. Per-Ake (Paul) Larson, who provided valuable feedback and suggestions to my comprehensive-exam, my dissertation proposal talk and dissertation drafts. All these helped to improve the presentation and contents of this dissertation. I thank Prof. David Finkel for his time and efforts discussing with me in my research qualifying exam.

I would like to thank Songting Chen for his collaboration on the txn-Wrap system, which composes the basis for my third part of dissertation work. I thank Yali Zhu for her valuable discussions on part of my second dissertation task on adapting operator states of query trees with multiple stateful operators. My thanks also go to Luping Ding, Mariana Jbantova, Rimma V. Nehme, Bradley Momberger, Venkatesh Raghavan and Timothy Sutherland and all other previous and current D-Cape team members for their useful discussions and feedback. The long hours spent in the Fuller Lab would not have been possible but for the company of wonderful office colleagues around me such as Maged F. EL Sayed. The friendship of Song Wang, Xin Zhang, Li Chen and all the other pervious and current DSRG members is much appreciated. They have contributed to many interesting and good-spirited discussions related to this research.

My thanks also go to Josh Brandt, the WPI computing cluster administrator, who helped me setting up my cluster account and gave me quick responses on cluster usage questions. I also thank Worcester Polytechnic Institute and the Computer Science Department for giving me the opportunity to study and also providing TAship during my studies.

Finally, I would like to thank my wife Xiaojie for her understanding and love during the past few years. Her support and encouragement was in the end what made this dissertation possible. My parents receive my deepest gratitude and love for their dedication and the many years of support during my studies. Special thanks also go to my little boy Jeffrey (Linjun), his nice cooperation in the past year made me possible to complete my studies on schedule.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With the information explosion on the World Wide Web, the integration of data from multiple distributed data sources is critical to many modern applications, e.g., data warehousing and data mining systems [46, 53], digital libraries [29], and semantic web [10]. The integration results are usually materialized (referred as *materialized views*) [42] to ensure better access, reliable performance and high availability. The computation of the materialized views can be rather complex and time consuming due to distributed nature of data sources, i.e., evaluating joins across multiple distributed data sources. Thus, it is better to perform the computation process once and materialize the result.

Materialized views need to be maintained given changes on data sources after the integration. This is because stale view extent may not serve well or even mislead user applications. Thus, two essential services need to be provided to realize the benefits of applying materialized views, (1) how to initially compute the view result from multiple data sources (referred as *view computation*), and (2) how to maintain materialized view extents when data sources are changed after the initial computation to provide up-to-date results (referred as *view maintenance*).

These two services face scalability concerns in this modern, networked environment. First, data sources are becoming increasingly large over time. It is not uncommon to see a terabyte warehouse nowdays [37]. Second, rapid changes made to such data sources are common too, e.g., millions of daily transactions [62]. Third, the number of available data sources are increasing due to the information explosion and these data sources tend to be distributed over the network or even over the Web [36]. All these trends demand scalable view computation and view maintenance solutions. Moreover, for time-critical applications such as real-time data integration services [112], the performance of view computation and view maintenance has extremely significant impact on the success of these applications.

This dissertation work is motivated by the above scalability requirement. Corresponding to the two services identified in the materialized view context, we divide the whole work into two parts. The first part relates to efficiently computing views from a large number of distributed data sources, in particular, we focus on investigating research issues related to parallel and adaptive view computation solutions. The second part aims to investigate how to scale materialized view maintenance performance when large batches of source updates need to be maintained. More specifically, the following four research questions are addressed in this dissertation work:

- **Parallel and Adaptive View Computation:**
  - How to design efficient parallel processing strategies for computing materialized views defined over a large number of distributed data sources. That is, given a view definition and a parallel system (i.e., a cluster of high performance PCs), we need to determine a strategy to compute the view results efficiently in terms of the total computation time required.
  - Given a long running computation process with state intensive query operators, it may demand more memory than even a parallel system can provide. Thus, we propose to tackle the challenge of how to efficiently adapt run-time main memory usage to improve the overall performance of parallel view computation process.
- **Scalable View Maintenance for Large Update Batches:**
  - Materialized view maintenance process presents certain regularity in terms of composing and sending maintenance queries to distributed data sources. We investigate whether and how the regularity of the view maintenance process can be exploited to improve the view maintenance performance when maintaining large batches of source updates.
  - Materialized view maintenance over distributed data sources also relates to traditional distributed query processing. We thus also study whether and how the state-of-the-art distributed query

processing techniques could be applied and appropriately extended to improve the view maintenance performance given the distributed nature of the data sources.

1.1 Background and Research Focus

1.1.1 Overall Architecture

The overall architecture of materialized views and their applications is depicted in Figure 1.1. We divide the architecture into three layers, namely, *data sources*, *materialized views*, and *user applications*. The interactions among these different layers can be described as follows. Data from distributed data sources will be first integrated and stored as materialized views by the *view computation engine*. After the integration, source updates will be reported to the *view maintenance engine*. Then the materialized views will be maintained to have up-to-date view extent. Note that both view computation and view maintenance engines are software modules in charge of view computation and maintenance tasks. They do not have to be deployed on the server where materialized views are stored. The user applications can (and also prefer to) directly access materialized views to answer complex analysis queries efficiently even without accessing data sources.

To further understand the research focus, we first describe the components of each layer.

- **Data Sources.** Data sources in a materialized view maintenance context usually play a restricted role [16, 36, 114]. That is, they only pro-

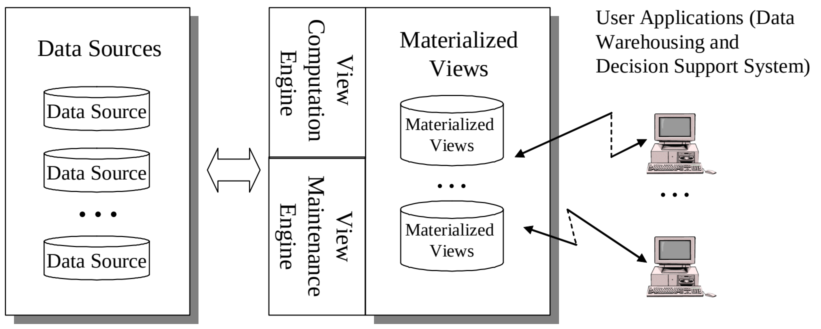


Figure 1.1: Overall Architecture

vide limited query processing capabilities to the outsiders such as materialized views or user applications. This is because (1) data sources may belong to other organizations that are not willing to give full control to the outsiders, (2) the data sources may be too busy doing daily transactions to afford processing additional typically complex queries, i.e., a join query over two data sources, or (3) in some cases, the data sources may indeed only have very limited query processing capabilities or even do not have them at all, for instance, streaming data sources [75].

- **Materialized Views.** View definitions could be defined across multiple data sources in order to achieve the integration of disparate data. However, they usually share a core part, namely, a select-project-join (SPJ) clause that integrates data from multiple data sources. We refer to such an SPJ view as an *integration view*. In this work, we choose integration views as the main focus for the following two reasons. (1) It is a common base for a majority of view definitions, and (2) it is the most expensive part to evaluate and maintain in most cases since it involves joins across multiple distributed data sources. Other parts of a view definition, i.e., aggregations, could be evaluated after the integration view has been processed. Moreover, some principles discussed in this work such as parallel and adaptive computation techniques, can be re-applied in a similar manner. In essence, the integration view definitions can be treated as multi-join queries across multiple distributed data sources.

- **User Applications.** User applications may ask ad-hoc or pre-defined queries against materialized views and data sources. This requires that the materialized views are properly computed and maintained. Research questions related to user applications, i.e., how to answer user queries using materialized views [4, 45], choosing views to be materialized [6, 26, 43], are out of the scope of this dissertation work.

1.1.2 Computing Integration Views

The computation of an integration view can be treated as answering a multi-join query across distributed data sources. However, two major points differentiate a view computation process from that of typical distributed multi-join query processing. First, a typical distributed query processing engine assumes that the data sources are fully cooperative [57]. That is, we often consider to ship data to the data source and to evaluate the query or a subset of the query locally at the data source. However, as we discussed in Section 1.1.1, the roles of data sources involved in the view computation process are restrictive in many cases. We thus cannot make such an assumption in general in computing integration views. For example, data sources may belong to other organizations or the data sources may be too busy handling daily transactions to afford processing additional complex join queries. Second, view computation usually is a fairly long running process since a large volume of data as well as a large number of data sources may be involved. While distributed queries often tend to be ad-hoc queries that need to (and can) be answered fairly quickly.

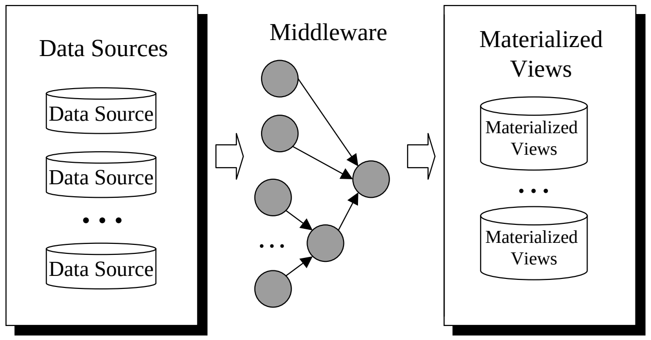


Figure 1.2: View Computation Overview

Figure 1.2 depicts the high level picture of the view computation process. Here the computation process (the middleware part) is represented by a query tree with each node in the tree denoting query operator(s). Given the restricted role that the data sources would play in a materialized view environment, the data source query processing capabilities cannot be counted upon when generating the view computation plan. Thus, we need to have the methods of how to preform the view computation process outside the data sources.

Parallel query processing techniques over a shared-nothing architecture, i.e., a computer cluster, can be naturally applied to this view computation process given its proven scale-up and speed-up properties [31]. As identified in the literature [47], three types of parallelism can be identified. One, operators none of which use data produced by the others may run simultaneously on distinct machines. This is termed *independent parallelism* (inter-operator parallelism). Two, operators may be composed by a producer and consumer relationship. Thus tuples output by a producer can be fed to a consumer as they get produced. Such inter-operator parallelism is termed *pipelined parallelism*. A third form of parallelism, termed *partitioned parallelism*, provides intra-operator parallelism based on the partitioning of the data. That is, several instances of one operation run on different machines, with each instance only processing a partitioned portion of the complete data.

To summarize, the main research focus of the view computation process is to design efficient parallel processing strategies, i.e., to find the best way to incorporate various forms of parallelism for the middleware computation process shown in Figure 1.2.

Uneven workload may happen among machines in a parallel system due to inaccurate cost estimations, or changing cost statistics, or both. This unevenness could impact or even counteract the benefits of the parallel processing. Thus, run time adaptation strategies also need to be investigated especially for such long running computation processes with state intensive queries due to the integration of large volumes of data over distributed data sources. Note that techniques such as load shedding [107] is not a valid option in this integration context since it usually requires complete and accurate results. Moreover, the overall resources (i.e., main memory) of a parallel system remain limited, thus we have to design run-time adaptation strategies for resource restricted environments where the overall resources of the parallel system are not enough for the given computation workload.

1.1.3 Maintaining Integration Views

A large amount of source data updates are common for modern applications, i.e., millions of daily transactions are experienced by modern e-businesses on the internet such as Amazon.com. Thus, efficiently maintaining a materialized view becomes critical in order to provide refreshed results. Incremental materialized view maintenance has been extensively studied in the literature [5, 18, 93, 120, 122, 123] due to the high cost associated with shipping large volumes of data in a distributed environment. That is, instead of completely recomputing the view extent from scratch whenever source updates happen, the delta of the view extent for the given source update is computed and committed to refresh the view extent. The computation of a view delta for join views requires the sending of *maintenance queries* [122] to the remote data sources to determine the changes of the view extent related to the current updates.

Figure 1.3 depicts the high level of the view maintenance process. In general, a *view maintenance engine* is in charge of view maintenance for source updates. The data sources report the source updates to the view maintenance engine. The maintenance engine composes maintenance queries

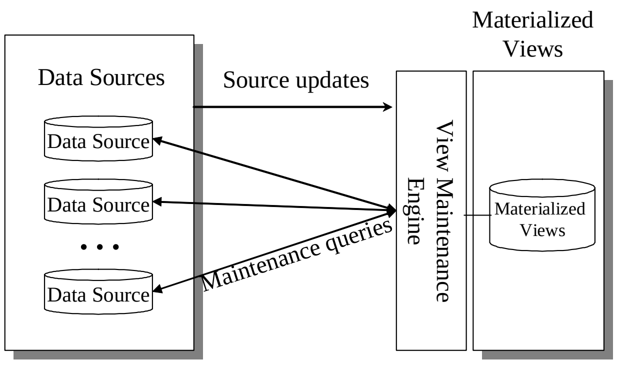


Figure 1.3: View Maintenance Overview

based on the view definition and the updates (or the results of other maintenance queries). The maintenance queries are sent to the data sources. The results are returned back to the view maintenance engine. The maintenance engine computes the changes to the view extent, and finally installs the changes to the materialized views. Note that in an incremental view maintenance context, the data sources are assumed to be able to answer maintenance queries issued by the view manager. Otherwise, it is not possible to perform the incremental maintenance for join views involving distributed data sources [1]. This requirement does not conflict with the restricted role typically assumed for the data sources as we discussed in the overall architecture (Section 1.1.1). This is because the maintenance queries are usually created based on source updates or other maintenance query results. Thus, the maintenance queries are much smaller in size and easier

[1]There are self-maintainable views [87] that can be maintained without issuing maintenance queries. However, the views are rather restricted or it may require copies of data source contents at the view server. In this work, we instead address general non self maintainable join views and assume view server does not have copies of data source contents.

to answer compared to the complex join queries in the view computation process. This is because the later usually relates to join queries involving the whole data sources.

In this work, we would target the view maintenance layer (maintenance queries as shown in Figure 1.3) to address the scalability issue in the view maintenance process. This is because the maintenance queries are the key and the expensive part in a view maintenance process. Moreover, all these queries show a certain regularity (i.e., all of them are join queries involving data sources and the updates) that has the potential to be utilized to improve the overall maintenance performance.

1.2 Contributions of this Dissertation

The main contributions of this dissertation work are described below.

1.2.1 Segmented Bushy Parallel Multi-Join Processing

Evaluating multi-join queries over a shared-nothing architecture has been extensively investigated in the literature [77, 95, 106]. Different parallel processing strategies such as left-deep and right-deep [95], segmented right-deep [21], and zigzag tree [124] have been proposed. These proposed solutions make the common assumption that the maximal pipelined parallelism leads to superior performance. Thus, these approaches tend to maximally apply the pipelined parallelism whenever it is possible.

In this work, we instead illustrate via cost model analysis as well as experimental studies that this commonly accepted assumption does not hold in practical. We investigate how best to combine pipelined parallelism with alternate forms of parallelism to achieve an overall more effective parallel processing strategy. A new parallel multi-join processing strategy, called *segmented bushy* processing, is proposed that brings all three forms of parallelism to bear in the evaluation of multi-join queries. An algorithm is proposed to generate such segmented bushy plans for arbitrary multi-join queries represented by connected join graphs.

To investigate the effectiveness of the proposed parallel processing strategy, we have implemented a parallel multi-join query optimization and processing system, called *PETL*, to conduct extensive experimental studies on a real system (not just a simulation). The experiments are conducted over a computer cluster of 10 high-performance PCs connected by a private network. The experimental results confirm that the proposed parallel processing strategy leads to an on average of 50% improvement in terms of the total processing time in comparison to existing state-of-the-art solutions.

1.2.2 Run-Time Operator State Adaptation

Main memory is a critical resource in an integration view computation process due to the long running nature of multiple join queries composed of state intensive operators. In such environments, the operator state size (so as the main memory consumption) keeps on increasing as more data is being processed. Works in the literature apply *partitioned parallel processing* [41, 94, 99] to alleviate the stringent memory demands. However, uneven workload may appear in distributed and parallel environments due to inaccurate cost estimations, or changing cost statistics, or both. Moreover, main memory of even a parallel system remains limited. Thus, there is a demand for efficient and flexible run-time main memory adaptation solutions for distributed and partitioned parallel queries.

Two types of adaptation solutions are available in partitioned parallel processing environments. First, as discussed in XJoin [109] and Hash-Merge Join [79], main memory resident operator states can be chosen and pushed into local disks when memory overflow happens. As can be seen, this type of approach is designed to delay the processing of certain operator states. We refer this process as *state spill*. Second, in a distributed environment, when only a subset of machines gets overloaded, we can choose states from the overloaded machine and move them over to a less loaded machine. For simplicity, we call this type of adaptation *state relocation*. The potential advantage of this state relocation is that the adapted states remain active in the main memory. However, this type of adaptation may not solve the memory shortage problem by itself since the aggregated main memory of multiple machines remains limited.

We investigate these two adaptations and analyze the tradeoffs regarding the factors and polices to be used when adapting operator states to overcome memory overflow. Two approaches, namely, *lazy-disk* and *active-disk*, are proposed to integrate both the state spill and relocation when the aggregated main memory of a distributed system is not sufficient for the query processing. Both approaches aim to maximize the overall run-time query throughput, defined as the total number of results being output.

We further investigate state spill strategies for complex queries composed of multiple state intensive operators. We observe an interdependency when spilling operator states among different operators in the query. Thus, a consolidated plan level spill strategy must be devised to address this problem. Two global throughput-oriented state spill approaches, namely, *global output* and *global output with penalty*, are proposed aiming for maximal run-time query throughput in memory constrained environments.

The proposed adaptation strategies are implemented in the *D-Cape* system [70, 91, 104]. Extensive experiments have been conducted over the same 10 high performance PC cluster discussed in Section 1.2.1. These experiments confirm the effectiveness of our proposed adaptation solutions.

1.2.3 View Maintenance by Restructuring and Grouping

Incremental view maintenance, instead of completely recomputing the view extent from scratch, has been extensively studied in the literature [5, 18, 93, 120, 122, 123] due to high cost associated with recomputing large volumes of data in a distributed environment. Among these works, the incremental maintaining of batches of updates [27, 63, 66, 93] is of particular interest because it is attractive from both a resource and a performance perspective to most practical systems.

State-of-the-art view maintenance strategies require O(n^2) (batch view maintenance [63, 66, 93]) or more (i.e., sequential maintenance [5, 122]) maintenance queries to remote data sources with *n* being the number of data sources. This mechanism does not scale for a large number of nor for large sized data sources. We propose two novel maintenance strategies, namely, *adjacent grouping* and *conditional grouping*, that are able to dramatically reduce the number of maintenance queries required to maintain the materialized views. This reduction in the number of maintenance queries brings the basic tradeoff between the complexity of each query and the total number of maintenance queries that can be exploited to improve maintenance performance.

The proposed maintenance strategies have been implemented in the TxnWrap system [20]. Extensive experimental studies have been conducted. The results show that our proposed view maintenance strategies are able to achieve about 400% performance improvement in terms of the total processing time compared with existing batch algorithms in a majority of cases.

1.2.4 Optimizing Cyclic Integration View Maintenance

State-of-the-art view maintenance algorithms [5, 63, 64, 66, 93] tend to focus on maintaining simple acyclic join views. Little attention has been paid thus far on more complex view definitions, i.e., cyclic join views that may specify many join conditions between any two arbitrary source relations. Such cyclic join views are being widely used in practical systems [108].

We model view maintenance as the process of answering a set of interrelated distributed multi-join queries. This model enables us to expose several potential optimization opportunities. For example, we can study the techniques of seeking optimal join ordering of a multi-join query or combining queries (sub-queries) to reduce the total number of join queries. We investigate two maintenance strategies that apply the above optimization techniques, namely, *extended batching* and *view graph transformation*, for maintaining general join views where join conditions may exist between any pair of data sources possibly with cycles.

A large amount of of maintenance plans can be built given the complexity of view definitions, we thus propose a cost-driven view maintenance framework which generates optimized maintenance plans taking into consideration the view definition characteristics, the number of source updates and the network costs. The proposed framework has been implemented in the TxnWrap system [20]. Extensive experimental studies illustrate that our proposed optimization techniques significantly improve the view maintenance performance in a distributed environment.

1.3 Dissertation Organizations

This dissertation is organized into three parts. The first part focuses on parallel view computation strategies. It is described in Chapters 2, 3 and 4. The second part, described in Chapters 5, 6, 7 and 8, addresses how to dynamically adapt operator states in partitioned parallel computation environments. While the third part, focusing on incremental batch view maintenance and its optimizations, is described in Chapters 9, 10, 11 and 12. Conclusions of this dissertation and the future work are described in Chapters 13 and 14 respectively.

Part I

Parallel Integration View Computation

Chapter 2

Revisiting Pipelined Parallelism

2.1 Introduction

As discussed in Chapter 1, the integration view computation can be viewed as evaluating multi-join queries assuming the join is evaluated outside the data sources. Without loss of generality, we may interchange the usage of terms multi-join query and integration view in the following of this work.

Two processing strategies at opposite ends of the spectrum, namely, *sequential* processing and *pipelined* processing, have been proposed in the literature [95]. For example, Figure 2.1 illustrates these two approaches when processing a four-way join query $R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4$ on 2 machines. Here, we assume join relations R_1, R_2, ..., R_4 are not in these 2 machines originally. Figure 2.1(a) illustrates an example of sequential processing. That is, we first evaluate $R_1 \bowtie R_2$ over 2 machines and get the intermediate result I_1. We then process $I_1 \bowtie R_3$ on the same 2 machines (indicates by the dashed rectangle) and get the intermediate result I_2. This process repeats until we get the final query results. Figure 2.1(b) shows an example of pipelined processing of this four-way join query. For example, we first distribute (load) R_2, R_3, and R_4 over the 2 machines. Then, tuples read from R_1 probe these relations in a pipelined fashion and generate query results. This pipelined processing of multi-join queries has been shown to be superior to the sequential processing given sufficient resources [95]. As we will discuss shortly, state-of-the-art parallel multi-join query processing solutions tend to maximally apply this pipelined processing as its core execution strategy [21, 95, 124].

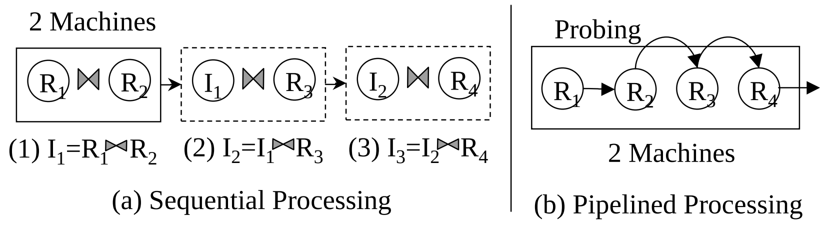


Figure 2.1: A Motivating Example

However, does this commonly accepted solution of maximally applying pipelined parallelism always perform effectively when evaluating multi-join queries? Or, put differently, are there methods that enable us to generate even more efficient parallel execution strategies than this fully pipelined processing? In this part of the dissertation work, we first show via a cost analysis as well as using real system evaluations that such maximally pipelined processing is not always effective. We then propose an *segmented bushy* parallel processing strategy for multi-join queries that outperforms state-of-the-art solutions.

As motivated in Chapter 1, we assume that the multi-join queries are processed outside of any data sources. We focus on complex multi-join queries, i.e., those that involve 10 or more source relations.

We focus on hashing join algorithms [72] since they are among the most popular ones in the literature due to their proven superior performance [72, 94]. Hashing joins provide the possibility of a high degree of pipelined parallelism. Other join algorithms such as sort-merge join do not have this natural property of pipelined parallelism [94]. Furthermore, hashing joins also naturally fit partitioned parallelism.

The key research question that we propose to address in this work is whether maximally pipelined multi-join query processing is indeed a superior solution as commonly assumed in the literature. This pipelined process implies main memory based processing. Hence, we assume that the aggregated memory of all available machines is sufficient to hold the hash tables of the join relations [1]. The rationale behind this is that both the main memory of each machine and the number of machines in the cluster are getting increasingly large at affordable cost.

Due to possibly large volumes of data in each source relation, the main memory of one machine may not be enough to hold the full hash table of one source relation. Thus, partitioned parallelism is applied to each join

[1] In situations when main memory is not enough to hold all hash tables at the same time, we follow the typical approach to divide the query into several pieces with each piece being processed sequentially. We defer this discussion to Section 3.3.

operation whenever it is necessary. That is, a partition (exchange) operator [41] will be inserted into the query plan to partition the input data tuples to multiple machines to conduct a partitioned hashing join processing.

2.2 State-of-the-Art

Various solutions have been investigated for parallel multi-join query processing in the literature [21, 95, 124]. To illustrate, we use the 10-join query depicted in Figure 2.2 to explain the core ideas. The multi-join query is depicted by its join graph. Each node in the graph (R_0, R_1, . . ., R_9) represents one join relation (data source), while an edge denotes a join between two respective data sources.

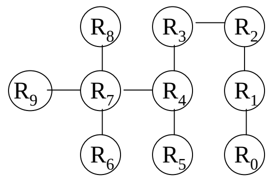
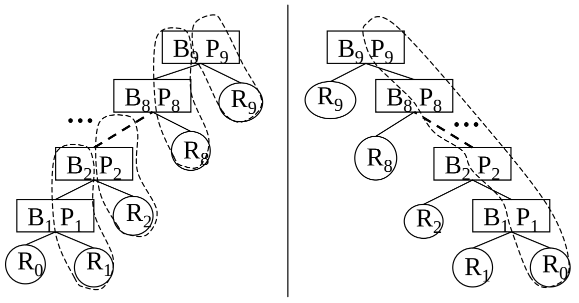


Figure 2.2: An Example Query with 10 Relations

2.2.1 Sequential vs. Pipelined Processing

Two strategies at opposite ends of the spectrum, namely, sequential processing and pipelined processing, have been proposed [95]. Note that partitioned parallelism is applied by default for each join operator. Sequential processing is based on a left-deep query tree. Figure 2.3(a) illustrates one example of sequential processing for the query defined in Figure 2.2. Here

B_i represents the building phase of the i-th join operation, while P_i denotes the corresponding probing phase. This processing can be described by the following steps: (1) scan R_0 and build B_1, (2) scan R_1, probe P_1, and build B_2, (3) scan R_2, probe P_2, and build B_3, and so on. This is repeated until all the join operations have been evaluated. As can be seen, it processes joins sequentially and only partial operations, namely, the probing and the successive building operations, are pipelined.



(a) Sequential (b) Pipelined

Figure 2.3: Sequential vs. Pipelined

Pipelined processing is based on a right-deep query tree [95]. Figure 2.3(b) illustrates an example of pipelined processing for the same query in Figure 2.2. In this case, all the building operations such as scan R_1 and build B_1, scan R_2 and build $B_2, \ldots$, scan R_9 and build B_9 can be run concurrently. After that, the operation of scan R_0 and all the probing operations, probe P_1, probe P_2, $\ldots$, probe P_9 can be done in a pipelined fashion. As demonstrated above, it achieves fully pipelined parallelism.

Note that a pipeline process implies main memory based processing [2].

[2]The term main memory henceforth denotes the sum of memory of all machines in the cluster unless otherwise specified.

That is, it requires there to be enough main memory to hold all the hash tables of the building relations (R_1, R_2, …, R_9 in this case) throughout the duration of processing the query.

As identified in [95], pipelined processing is preferred *whenever main memory is adequate*. This is because (1) intermediate results in pipelined processing exist only as a stream of tuples flowing through the query tree, and (2) even though sequential processing in general may require less memory, this is not always true due to intermediate results have to be stored. A large intermediate result may consume even larger memory than the sum of all building relations.

The simulation results in [95] confirm that the pipelined processing (right-deep) is more efficient than the sequential one (left-deep) in most of the cases they considered. Without loss of generality, we thus associate the *pipelined* processing with a *right-deep* query tree, and the *sequential* processing with a *left-deep* query tree in the following discussions.

2.2.2 Maximally Pipelined Processing

State-of-the-art parallel multi-join query processing solutions maximally pursue the above pipelined parallelism to improve the overall performance [21, 95, 124]. If the main memory is not enough to hold all the hash tables of the building relations, they commonly take the approach of dividing the whole query into “pieces”, with the expectation that the building relations of each piece fit into the main memory. That is, pieces are processed one by one with each piece utilizing the entire memory applying fully pipelined parallelism.

For example, zigzag processing [124] takes a right-deep query tree and slices it into pieces based on the memory availability. As an example, the right-deep tree in Figure 2.3(b) is cut into two pieces, one is R_0, R_1, ..., R_3, and the other is I_1, R_4, ..., R_9 (Figure 2.4(a)). Here, I_1 corresponds to the result of the first piece $R_0 \bowtie R_1 \bowtie \ldots \bowtie R_3$. These two pieces are processed sequentially with fully pipelined parallelism in each piece.

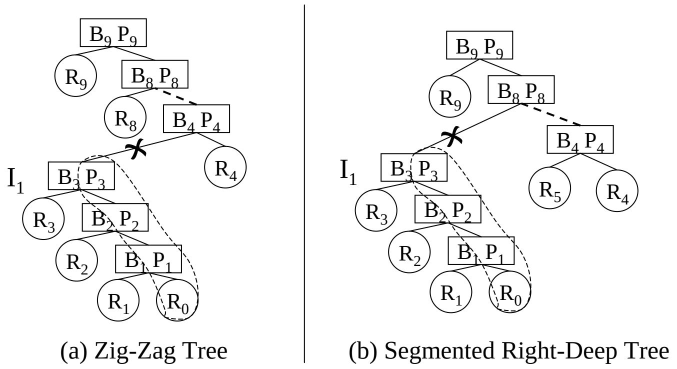


Figure 2.4: ZigZag and Right-Deep Trees

Segmented right-deep processing [21] proposes heuristics, namely, *balanced consideration* and *minimized work*, to generate pieces directly from the query graph based on the memory constraint. The query tree is similar to the zigzag tree. However, each piece can be attached not only at the first join operation of the next piece, but instead also in the middle of it. For example, Figure 2.4(b) illustrates one example of segmented right-deep processing. As can be seen, the output (from P_3) is attached as the building relation of B_8.

To summarize, all the above approaches take the common model of pursuing a maximally pipelined processing of multi-joins via a right-deep query tree, with the number of join relations in the right-deep tree primarily being determined by the main memory available in the cluster.

We now question the performance of such a maximally pipelined processing model. As mentioned earlier, this pipeline process implies a main memory based processing. Clearly, more efficient main memory based processing strategies would lead to an improved overall performance. Without loss of generality, we use the term *pipelined segment* to refer a right-deep query tree that can be fully processed in the main memory.

2.3 A Multi-Phase Optimization Approach

Multi-join query optimization is an expensive process because the number of alternative query plans for a query grows at least exponentially in the number of relations participating in the query [113]. Parallel multi-join query optimization is even harder [35, 51, 101]. Complications arise because the cost to be optimized, either total amount of work to be processed or total processing time, are no longer closely correlated since a query plan with minimal work may have a high sequential dependency that results in high overall processing time. Second, even one sequential query plan can in turn have a huge number of parallel solutions.

We take a *multi-phase* optimization approach [3] to cope with the complexity of parallel multi-join query optimization. That is, we break the optimization task into several phases and optimize each phase individually.

We divide the whole optimization task into the following three phases,

[3]A *single-phase* optimization approach such as [101] could also be applied, our multi-phase approach enables us to focus our attention on the research task we are tackling.

(1) generating an optimized query tree, (2) allocating query operators in the query tree to machines, and (3) choosing pipelined execution methods. We note that even if we divide the optimization task into multiple phases, the complexity of each phase, i.e., phases (1) and (2), still remains exponential in the number of join relations.

The main focus of this work is on investigating the impact of query trees (phase (1)) and different forms of parallelism on the overall performance. To proceed, we first describe the design choices we will assume in the reminder of our work for phases (2) and (3) below. We simplify the operator-machine allocation (for phase(2)) and choose the *concurrent execution* approach [95] as the pipeline execution method (for phase(3)).

Allocating Query Operators. Query operators (joins) need to be allocated to machines in the cluster. However, resource allocation itself is a research problem of high complexity that has been extensively investigated in the literature [39, 59, 71]. Like most work in parallel multi-join query processing literature [21, 95, 124], we focus on main memory in the allocation phase. This is because main memory is the key resource in the above hashing join processing. Other factors such as CPU capabilities of computation nodes are assumed to have less impact on the allocation, i.e., they are often assumed to be sufficient.

The allocation is performed based on pipelined segments to promote the usage of pipelined parallelism [71]. For example, if a right-deep tree is cut into pieces with each piece being processed sequentially due to insufficient memory, then all machines are allocated to each piece. Thus, the whole allocation is performed in a *linear* fashion. As it can be seen, all previous processing strategies described in Section 2.2 fall into this type of *linear allocation*.

Pipelined Execution Method. The building relations of each pipelined segment can entirely fit into the memory of the machines that have been allocated to it. We apply a *concurrent execution* approach [95] to process a pipelined segment [4]. In this execution method, all scan operations are scheduled concurrently. For example, in Figure 2.5, we process a 4 way pipelined segment on 3 machines. Each building relation (R_2, R_3, R_4) is evenly partitioned across all 3 machines. Thus, each machine houses the appropriate partitions from all building relations, denoted as P_i^j. Here, subscript i ($2 \leq i \leq 4$) denotes join relations, while superscript j ($1 \leq j \leq 3$) represents machine ID. The probing relation (R_1) is also partitioned into all 3 machines to probe the appropriate hash tables to generate results.

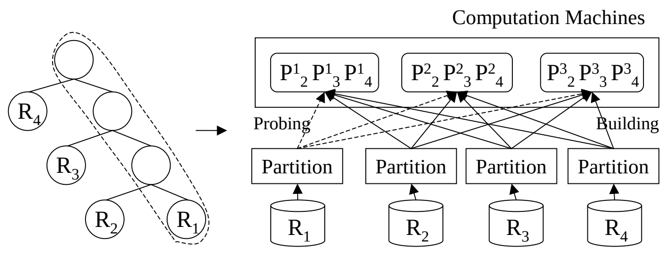


Figure 2.5: Fully Concurrent Execution

[4]Other pipelined execution strategies such as *staged partitioning* [21] have also been proposed. The detailed discussion of these strategies and their impact on parallel processing strategies are left as one future work of this dissertation.

2.4 Cost Analysis of Pipelined Segment

2.4.1 Identifying Tradeoffs

The following two factors need to be considered when analyzing the performance of parallel multi-join query processing via a partitioned hashing: (1) redirection costs between join operations, and (2) optimal degree of parallelism.

Redirection Costs. The basic idea behind the partitioned hash join algorithm is that the join operation can be evaluated by a simple union of joins on individual partitions. For example, an equi-join $A \bowtie B$ can be computed via $(A_1 \bowtie B_1) \cup (A_2 \bowtie B_2) \ldots \cup (A_n \bowtie B_n)$ if A and B are first divided into n partitions (A_1, A_2, . . ., A_n and B_1, B_2, . . ., B_n) using the same hash function. Assume the two partitions in a pair (A_i, B_i) are put in the same machine, while different pairs are spread over the distinct machines. This way, all pairs can be evaluated in parallel.

However, for a right-deep tree segment, it is not possible to always have all the matching partitions reside in the same machine. For example, assume a query tree is defined by "A.A_1 = B.B_1 and B.B_2 = C.C_1". A and B are partitioned based on their common attribute A.A_1 (or B.B_1), while C has to be partitioned based on the common attribute between B and C, namely, B.B_2 (or C.C_1). If we assume A is the probing relation, then the partition function of B.B_2 has to be re-applied to the intermediate result of $A_i \bowtie B_i$ to find the corresponding partitions C_i. However, this corresponding partition C_i might exist in a machine different from where the current B_i resides. Thus redirection of intermediate results is necessary in this situation. For the special case of a right-deep tree when only one attribute per source relation is involved in the join condition, i.e., "A.A_1 = B.B_1 = C.C_1", the same partition function can be applied to all relations. In that case, all the corresponding partitions can be put into the same machine to avoid such redirections. Such redirection affects the probing cost of the query processing.

Optimal Degree of Parallelism. Startup and coordination overhead of different machines may counteract the benefits that could be gained from parallel processing. [78, 116] discuss the basics on how to choose the optimal degree of parallelism for a single partitioned operator, meaning the idea number of machines that need to be assigned to one operator. As one example, if a relation only has 1,000 tuples, it is not a good idea to have it evenly distributed across a large number of machines (i.e., 100) since the startup and coordination costs among these machines might be higher than the actual processing cost. Given the processing of more than one join operators (pipelined segment), we expect this factor has a major impact on the overall performance. That is, it affects the building phase cost of the query processing.

2.4.2 Pipelined Processing Cost Model

For pipelined processing of a right-deep segment, the cost in terms of total work versus the overall processing time may not be that closely correlated. We thus derive two separate cost models. To facilitate the description of cost models, we assume R_0 is the probing relation, while R_1, R_2, ..., R_n are the building relations of the pipelined segment. We also assume k machines are available to process the pipelined segment. These machines are denoted by M_1, M_2, ..., M_k. Without loss of generality, we use I_i to represent the intermediate result after joining with R_i. For example, I_1 denotes the result of $R_0 \bowtie R_1$, while I_2 represents $I_1 \bowtie R_2$. Thus I_n represents the final output of these joins.

Estimating Total Work. The total work of pipelined processing can be described as the sum of the work in the building phase (W_b) and the work in the probing phase (W_p), as listed below.

$$W_b = (t_{read} + t_{partition} + t_{network} + t_{build}) * \sum_{i=1}^{n} |R_i|$$

$$\begin{aligned} W_p &= (t_{read} + t_{partition} + t_{network} + t_{probe}) * |R_0| \\ &+ \frac{k-1}{k} * \sum_{i=1}^{n-1} |I_i| * t_{network} + (\sum_{i=1}^{n-1} |I_i|) * t_{probe} \end{aligned}$$

t_{read}, $t_{partition}$, $t_{network}$, t_{build}, and t_{probe} in the above formulae represent the unit cost of reading a tuple from a source relation, partitioning, transferring the tuple across the network, inserting the tuple into the hash table, and probing the hash tables respectively. They represent the main steps involved in a partitioned hash join processing. In the probing phase work, $\frac{k-1}{k} * \sum_{i=1}^{n-1} |I_i| * t_{network}$ denotes the redirection cost assuming the redirection occurs after each join operation and the output of each join operation is uniformly distributed across all the machines. The cost of outputting the final results is omitted since it is the same for all processing strategies.

Estimating Processing Time. Similarly, estimation of the processing time can be divided into two parts: one, the hash table building time (T_b) and two, the probing time (T_p). The building time of the pipelined processing T_b can be estimated as follows:

$$T_b = \max_{1 \le i \le n} (t_{read} + t_{partition} + t_{network} + t_{build}) * \frac{f(k)}{k} * |R_i|$$

The processing time of the building phase can be estimated as the maximal building time of each individual relation over k machines. Here, f(k) represents the contention factor of the network since the more machines are involved, the more contention of the network caused by transferring tuples of join relations arises. This is used to reflect the optimal degree of parallelism as discussed in Section 2.4.1.

The processing time of the probing phase (T_p) is more difficult to analyze because of the pipelined processing. We use the following formula to estimate the pipeline processing time.

$$T_p = I_{setup} + \frac{W_p}{k} + I_{delete}$$

Here I_{setup} represents the pipeline setup time, while I_{delete} denotes the pipeline depletion time. The steady processing time of the pipeline can be estimated by the average processing time of one tuple ($\frac{W_p}{|R_0|}$) multiplied by the number of tuples ($|R_0|$) that need to be processed over the total of k

machines. Clearly, this is a simplified model representing the ideal steady processing time without including for example variations in the network costs.

From above analysis, we can see that both the number of building relations (n) and the number of machines (k) assigned to the pipeline play an important role in the overall processing time. As we will discuss in Section 3.1, we investigate to break both n and k , and compose smaller pipelined segments to query trees to improve the query processing performance. Note that above cost model is also applied to find the most efficient pipelined processing strategies for each subgraph (Section 3.2).

Chapter 3

Segmented Bushy Parallel Processing

3.1 Breaking Pipelined Parallelism

Query trees of a multi-join query can be classified into two types: sequential trees (i.e., a right-deep tree or a left-deep tree as discussed above), and bushy trees. A right-deep tree has a better performance over a left-deep tree since it has a high potential of pipelined parallelism for a hash-based join algorithm. Thus we now use a right-deep tree as the representative of sequential trees (e.g., Figure 3.1(a)).

A bushy tree has a height of at least $log_2 n$ (given a binary bushy tree that is balanced) with n being the number of join relations involved in the multi-join query. A bushy tree brings new flexibility to the style of processing, such as having multiple probing relations and composing different pipelined segments. Moreover, a bushy tree has the potential of processing independent subtrees (segments) concurrently. However, such flexibility may also bring dependencies to the execution. This dependency may both affect the allocation of query operators and the corresponding parallel processing performance. For example, Figure 3.1(b) illustrates one bushy tree and its possible pipeline segments (each pipeline segment is denoted by one dashed oval). Four segments (P_1, P_2, ..., P_4) can be identified. As can been seen, P_1 and P_3 can be processed in parallel with one another by processing them on different machines. While the execution of P_2 depends on P_1, the execution of P_4 depends both on P_2 and P_3.

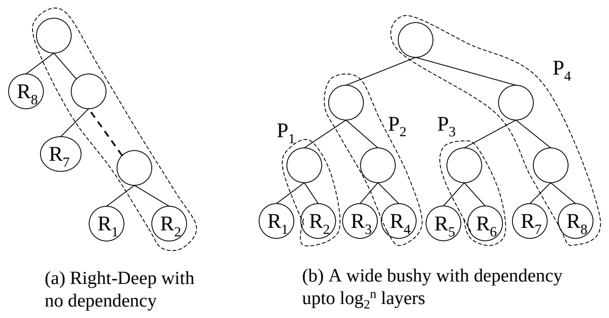

R8
R7
R1
R2
(a) Right-Deep with no dependency
P4
P1
P2
P3
R1
R2
R3
R4
R5
R6
R7
R8
(b) A wide bushy with dependency upto log2n layers

Figure 3.1: Right-Deep vs. Wide Bushy Tree

As can be seen, a right-deep tree has the highest degree of pipelined parallelism without any dependencies because each subtree is a join relation. However, there is no opportunity for independent parallelism except during the initial building phase of the join relations. While a wide bushy tree has many subtrees, it also has up to $log_2 n$ layers of dependencies with n being the number of source relations. These dependencies are likely to impact the overall performance.

3.1.1 Segmented Bushy Tree

Seen from the pipelined cost model discussed in Section 2.4.2, if the results of pipelined segments in a bushy tree are smaller than those of the original join relations, then the bushy tree processing may have less total work ($W_b + W_p$) when compared with the fully right-deep processing. Here we assume all the intermediate results are kept in main memory.

Comparing the overall parallel processing time of fully right-deep and bushy trees is more complicated. As we can see, each pipelined segment in a bushy tree only gets one portion of the total available machines. Thus the network contention ($f(k)$) in the building phase may be less severe than that of the full right-deep case. As a consequence, given the independent processing of these smaller pipelined segments, the processing time of a bushy tree may be better than that of fully pipelined processing. However, as we identified earlier, a bushy tree style processing may be affected by the dependencies among subtrees. Moreover, there may be subtrees (up to $\lceil n/4 \rceil$) that have short pipelined processing stages. For example, P_1 and P_3 only have a pipeline of one probing followed by the building for the next join. These two factors may eventually counteract the benefits gained by introducing independent parallelism and smaller network contention in each segment.

Thus, the key question now is how to balance independent parallelism and pipelined parallelism in parallel multi-join query processing. By reducing each pipelined segment (i.e., identified by dashed oval in Figure 3.1(b)) into one 'mega-node', we can build a dependency tree out of the original query tree. We note that the dependencies are associated with the height of this dependency tree. Thus reducing the height of the dependency tree should effectively reduce the dependencies. We thus propose to utilize a *segmented bushy* query tree. A segmented bushy tree can be controlled to have a dependency tree with height of 2 as long as we increase the number of subtrees of the root node.

Figure 3.2 illustrates the example of a segmented bushy tree of the join query in Figure 3.1. In this example, the whole query is cut into three groups, $R_1 \sim R_3$, $R_4 \sim R_7$, and R_8. Three pipelined segments P_1, P_2, and P_3 can be identified correspondingly. P_1 and P_2 can be processed independently, each with pipelined parallelism. The output from these two segments can be directly fed into P_3. Without loss of generality, the pipelined segment that contains outputs of all other segments is referred to as the *final pipelined segment*. In this case, P_3 is the final pipelined segment. Thus, all pipelined segments except the final one can be executed concurrently without any dependencies given enough main memory. We can see that a segmented bushy tree processing applies independent parallelism with minimal dependencies among subtrees (groups) since it only has one layer of dependencies among pipelines.

Without loss of generality, we always assume that the right-most pipeline of a segmented bushy tree by this convention serves as the probing relation of the final pipelined segment. For example, P_1 is the probing relation of the final segment P_3 in Figure 3.2.

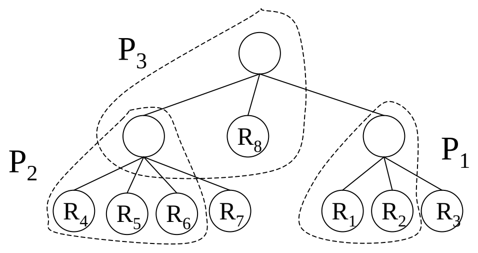


Figure 3.2: A Segmented Bushy Tree

3.1.2 Segmented Bushy Tree Cost Analysis

Similarly, the cost of the segmented bushy tree processing has two different cost measurements as we discussed in Section 2.4.2, one is the total work and the other is the total processing time.

Estimating Total Work. Assume join relations are divided into s groups connected by a segmented bushy tree. Without loss of generality, we assume these groups are denoted by their join relation indices, $(0 \sim m_1)$, $(m_1 + 1 \sim m_2)$, ..., $(m_{s-1} + 1 \sim n)$. The intermediate result of each group is represented by I_{m_1}, ..., I_{m_s}. Correspondingly, we assume each group will be assigned k_{m_i} machines based on its building relation size. The final pipelined segment gets k_f machines. The final query result is represented by I_n. Without loss of generality, we assume that I_{m_1} will be the probing relation of the final pipelined segment. Given these, the total work of the building phase of the segmented bushy processing (W_b') and the total work of the probing phase (W_p') can be described by the following formulae.

$$W_b' = (t_{read} + t_{partition} + t_{network} + t_{build}) * (\sum_{i=1}^{s} \sum_{j=m_i+1}^{m_{i+1}-1} |R_j| + \sum_{i=2}^{s} |I_{m_i}|)$$

$$\begin{aligned} W'_p &= (t_{read} + t_{partition} + t_{network} + t_{probe}) * (|I_{m_1}| + |R_0| + \sum_{i=1}^{s-1} |R_{m_i+1}|) \\ &+ t_{network} * (\sum_{i=1}^{s} \frac{k_{m_i} - 1}{k_{m_i}} \sum_{j=m_i+1}^{m_{i+1}-1} |I_j| + \sum_{i=2}^{s} \frac{k_f - 1}{k_f} |I_{m_i}|) \\ &+ t_{probe} * (\sum_{i=1}^{s} \sum_{j=m_i+1}^{m_{i+1}-1} |I_j| + \sum_{i=2}^{s} |I_{m_i}|) \end{aligned}$$

Estimating Processing Time. The overall processing time of the segmented bushy tree can be treated as the sum of two phases. The first phase, T_{f1}, estimates the time of processing all the pipelined segments (groups) with the results of these pipelines being directly fed into the building phase of the final pipelined segment. The second phase, denoted as T_{f2}, estimates the time of probing the final pipelined segment and outputting the query results.

The processing time of each pipelined segment (m_i) is composed of the following three components. (1) The building phase time of the building relations in m_i, denoted by B_{mi}. (2) The probing phase time of the group m_i, represented by P_{mi}. (3) The building time of the final pipelined segment from the output of group m_i (I_{mi}), denoted as B'_{mi}. The processing time estimations of these components are given below.

$$B_{m_i} = Max_{m_i+1 \leq j \leq m_{i+1}-1}\{\frac{f(k_{m_i})}{k_{m_i}} * |R_j|) * (t_{read} + t_{partition} + t_{network} + t_{build})\}$$

$$P_{mi} = I_{setup} + \frac{W_{p_{m_i}}}{k_{m_i}} + I_{delete}$$

$$\begin{aligned} W_{p_{m_i}} &= (t_{read} + t_{partition} + t_{network} + t_{probe}) * |I_{m_{i-1}}| \\ &+ t_{network} * (\frac{k_{m_i} - 1}{k_{m_i}} \sum_{j=m_i+1}^{m_{i+1}-1} |I_j|) + t_{probe} * (\sum_{j=m_i+1}^{m_{i+1}-1} |I_j|) \\ &+ (t_{partition} + t_{network} + t_{build}) * |I_{m_i}| \end{aligned}$$

$$B'_{mi} = \frac{f(k_f)}{k_f} * |I_{m_i}| * (t_{read} + t_{partition} + t_{network} + t_{build})$$

The cost of the first phase is estimated by $T_{f1} = Max_{1 \leq i \leq s}\{B_{m_i} + P_{m_i} + B'_{m_i}\}$. Note that the P_{m_i} and B'_{m_i} are actually processed in a pipelined fashion, yet here we simplify it by adding the two costs directly.

The processing time of the second phase (T_{f2}) is composed basically of the probing of the first group (I_{m_1}), and the rest of the intermediate results. We estimate the time as $\frac{W'_i}{k_f}$. W'_i can be described below.

$$W'_i = t_{network} * \frac{k_f - 1}{k_f} \sum_{i=2}^{s-1} |I_{m_i}| + t_{probe} * \sum_{i=2}^{s-1} |I_{m_i}|$$

3.2 Composing Segmented Bushy Tree

Now, we address the question how to generate the above segmented bushy tree for a multi-join query. Algorithm 1 sketches our proposed algorithm that incorporates heuristics as well as cost-based optimizations. It consumes a connected join graph **G**. We also input the maximal number of nodes *m* per group (we will discuss how to get this m shortly). We choose the largest join relation as the probing relation of each group since this reduces the time and the memory consumption of the building phase. Once we select the probing relation, we then enumerate all possible groups having a maximum of m join nodes starting from this probing relation. Enumeration is possible since m is usually much smaller than the number of nodes in the join graph. Some of the groups may contain less than m nodes due to the nodes in the group being no longer connected by a join edge. Our goal is to avoid Cartesian products given that each data source may be large, thus resulting in huge intermediate results. After that, we choose the best graph, a partition of the original join graph, from these candidates generated from the enumeration based on the cost model we developed in Section 2.4.2. Alternatively, the selection could also be based on heuristics, i.e., choosing the group in which the join attributes are the same to reduce the possible redirection costs, or selecting the group with the smallest output results.

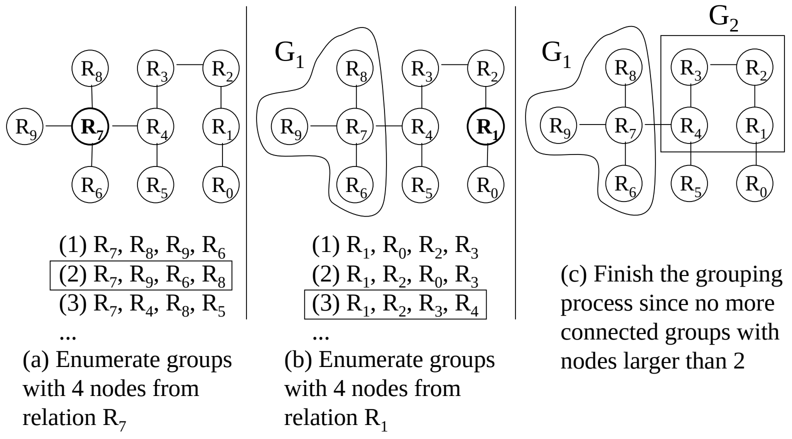
Algorithm 1 ComposeBushyTree(G,m)

Input:*A connected join graph* ***G*** *with n nodes. Parameter m specifies the maximum number of nodes in each group.* ***Output***:*A segmented bushy tree with at least $\lceil n/m \rceil$ groups.*

```
1:  completed = false
2:  while (!completed) do
3:      Choose a node $n$ with largest cardinality not yet been grouped
4:      Mark $n$ as a probing relation
5:      Enumerate all subgraphs starting from $n$ with at most $m$ nodes
6:      Choose the best subgraph $G_i$
7:      Mark nodes in $G_i$ as grouped in graph G
8:      if !(($\exists K$, $K$ is a connected subgraph of G with unselected nodes) $\&\&$
        (K.size() $\geq$ 2)) then
9:          completed = true
10:     end if
11: end while
12: Compose a segmented bushy tree
```

Figure 3.3 illustrates how the example join graph depicted in Figure 2.2

is divided by applying Algorithm 1 when $m = 4$. For example, we start from the relation with largest cardinality, say relation R_7. The enumeration in Step 5 generates all the possible connected groups with 4 nodes starting from R_7, as illustrated in Figure 3.3(a). In this case, we choose R_7, R_9, R_6, and R_8 as the nodes in the first group (pipelined segment). For simplicity, we call this group G_1. After this, if R_1 is the one with the largest cardinality among the nodes that have not yet been grouped, we then choose R_1 as the probing relation for the second group G_2. We repeat the process as illustrated by Figures 3.3(b)-(c). After these steps, only R_0 and R_5 are left. They are not connected. We thus end up with 4 groups. An example segmented bushy tree with these 4 groups can be built as shown in Figure 3.4(a).



R_8 R_3 R_2 R_9 R_7 R_4 R_1 R_6 R_5 R_0
(1) R_7, R_8, R_9, R_6
(2) R_7, R_9, R_6, R_8
(3) R_7, R_4, R_8, R_5
...
(a) Enumerate groups with 4 nodes from relation R_7
G_1
(1) R_1, R_0, R_2, R_3
(2) R_1, R_2, R_0, R_3
(3) R_1, R_2, R_3, R_4
...
(b) Enumerate groups with 4 nodes from relation R_1
G_1 G_2
(c) Finish the grouping process since no more connected groups with nodes larger than 2

Figure 3.3: An Example of the Algorithm

Allocating machines to a segmented bushy is based on the number of building relations in each pipelined segment. For example, for the segmented bushy tree shown in Figure 3.4(a), three pipelined segments can be identified (see dashed cycles in Figure 3.4(b)). The number of machines that are assigned to each pipelined segment, denoted by k_1, k_2, and k_3, can be computed as follows.

$$\begin{aligned} N_b &= \sum_{0 \le i \le 9, i \ne 1,7} |R_i| + |I_1| \\ k_1 &= \lfloor \frac{(|R_6| + |R_8| + |R_9|)}{N_b} \rfloor \\ k_2 &= \lfloor \frac{(|R_2| + |R_3| + |R_4|)}{N_b} \rfloor \\ k_3 &= k - k_1 - k_2 \end{aligned}$$

Here, I_1 and I_2 denote the outputs of groups G_1 and G_2 respectively. N_b represents the total number of tuples that need to be built assuming R_7, R_1, and I_2 are the probing relations of G_1, G_2, and the final pipelined segment respectively. Note that the selection of the probing relation for the final pipeline segment is not straightforward. We will discuss this in more detail in Section 3.4.2.

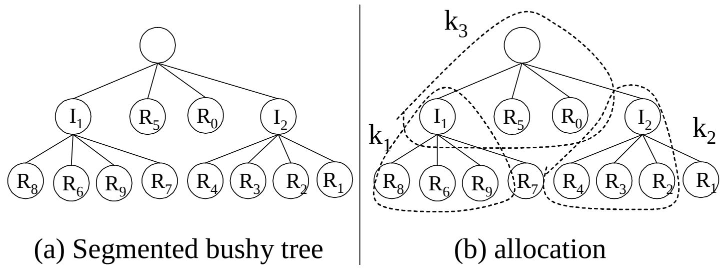


Figure 3.4: Segmented Bushy Tree and Node Allocation

However, the question remains how to decide what may be an appropriate number of groups given a join graph. Let us now use *g* to represent this number. Note that the input of Algorithm 1, the maximum number of nodes in each group *m* is estimated by $m = \lceil n/g \rceil$ with n being the number of join relations in the query. There are two ways to address this issue. The first is a heuristics-based selection approach. For example, we can choose g as the number of nodes that have cardinality larger than 3/2 of the average cardinality. Here, we assume that g has to be bound within $2 \sim n/2$. The rationale behind this selection criterion is that in the best case, we can choose all these large join relations as the probing relations for the generated groups. The second is a cost-based selection approach. Again we note that the range of the number of groups *g* is between 2 to $n/2$ [1]. We thus can repeatedly call the function *ComposeBushyTree* (Algorithm 1) with the number *m* ranging from $n/2$ to 2 (g changes from 2 to $n/2$ correspondingly). We then estimate the cost of the processing strategy from *ComposeBushyTree*. The final output will be the one with the best estimated cost. While this may increase the optimization cost, this has the potential to result in a better processing strategy.

3.3 Handling Insufficient Memory

The problem of handling insufficient memory can be addressed using the "cutting" principle as in [21, 95]. That is, we divide the whole query (joins) into pieces such that each piece can be run in the main memory. Note that in the extreme case, the multi-join query processing would have to be sequentialized due to not enough memory being available to hold more than

[1]In extreme cases, the actual number of groups may be larger than n/2. However, we have less interest in these cases having a large number of groups, while each group may only have one join relation in it.

one join. As we mentioned in Section 2.1, we assume that the aggregated memory can hold at least 2 or more building relations.

Algorithm 2 sketches an incremental approach to address this problem. This incremental approach is based on the static right deep tree [95] or segmented right-deep tree [21] which divides the join query into right-deep segments based on the main memory of the cluster. After that, we further compose each right-deep segment into a segmented bushy tree if it is necessary, i.e., the number of building relations in each piece is larger than a certain threshold. Since each right-deep segment is likely to be more efficiently processed, the performance of the whole query is also expected to be better than the static right-deep or segment-right deep tree processing.

Algorithm 2 SimpleIncSegTree(G,M)

Input: *A connected join graph* ***G*** *with n nodes, total cluster memory* ***M****.* ***Output:*** *A sequence of segmented bushy trees, each processable in M.*

```
1: Compose Static or Segmented Right-Deep Tree
2: for each right-deep segment $r$ do
3:     $m \leftarrow$ Maximal number of relations per group
4:     $t \leftarrow$ ComposeBushyTree(r,m)
5:     Put $t$ into result sequence
6: end for
7: Return result sequence
```

A “top-down cut” approach, dividing the join graph directly such that each group can be processed in the main memory, can also be devised. We then select the groups and process them iteratively. However, as mentioned earlier, the essence of our work is to re-examine the performance of a main memory based maximal pipelined processing. We argue that having a more efficient main memory based processing strategies will also lead to improved overall performance even if we apply a simple incremental optimization algorithm such as Algorithm 2. This claim is confirmed by our experimental studies discussed below.

3.4 Experimental Studies

3.4.1 Prototype System and Setup

We have implemented a distributed query engine to test out the proposed processing strategy. The system is implemented using Java. It is capable of optimizing and executing multi-join queries across a set of shared nothing machines connected by a network. The basic architecture of the system is depicted in Figure 3.5. The architecture consists of two main modules, one is the *controller* module and the other is the *execution* module. The controller module is in charge of managing the computation process. It can be installed on a stand-alone machine or on a machine that also houses other modules. The controller module contains packages that compose multi-join queries, generate parallel execution query plans, and distribute query plans to the participating machines. The parallel query plans (processing strategies) composed of query operators such as scan, partition, hash join, union and load are specified in an xml file format. The query is executed in the *execution* module. This execution engine is installed on each participant machine in the cluster that is involved in the computation process. The execution engine in each node waits for incoming query plans sent by the controller module. Once the execution engine receives the query plan, it parses the query plan, initializes it and starts up the query operators. After that, query operators in different computation machines automatically connect to each other and begin the query processing.

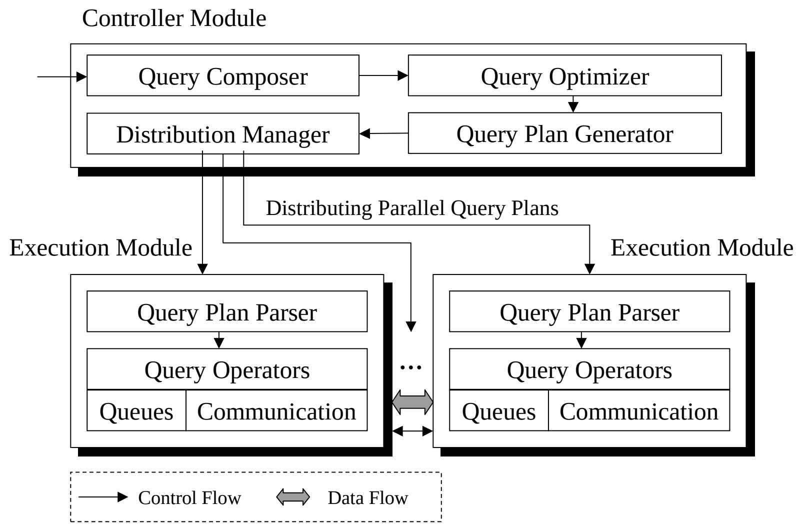


Figure 3.5: Architecture of the System

The system is deployed on a cluster composed of 10 machines, as described in Figure 3.6. Each machine in the cluster has dual 2.4GHz Xeon CPUs with 2GB RAM. They are connected by a private gigabit ethernet switch. In our experimental setting, all source (join) relations are stored in an Oracle database server that reside in a different machine outside the cluster having 2 PIII 1G Hz CPUs and 1G main memory. The query results are sent to an application server with one PIII 800M Hz CPU and 256M Memory. This setup follows a typical data warehouse loading environment (e.g., ETL [92]) where the process has to be performed outside the data sources. This is because the operating data sources may be too busy to process complex join queries or even simply may not be willing to give control to the outsiders.

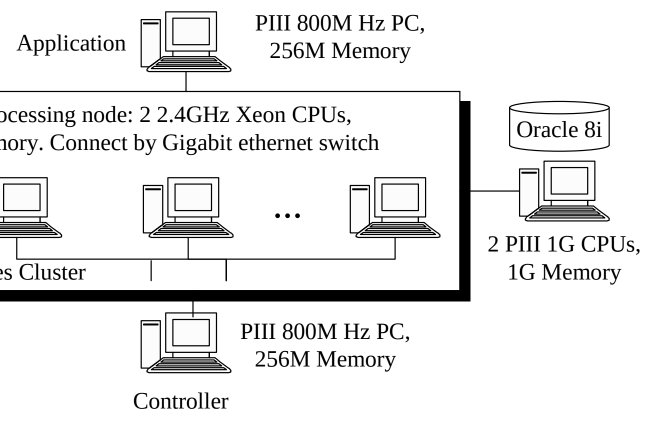

Application
PIII 800M Hz PC, 256M Memory
Each processing node: 2 2.4GHz Xeon CPUs, 2G Memory. Connect by Gigabit ethernet switch
...
10 Nodes Cluster
Oracle 8i
2 PIII 1G CPUs, 1G Memory
PIII 800M Hz PC, 256M Memory
Controller

Figure 3.6: Experimental Environment

As done in [21], we use generated data sets and queries in our experiments. This is because benchmark queries such as TPC-H [108] only have a limited number of queries (around 20), and most of them have less than 5 joins. The multi-join queries used in the experiments are randomly generated with the number of join relations ranging from 8, 12, to 16 [2]. The cardinality of each join relation ranges from 1K $\sim$ 100K tuples, and the average size of each source tuple is about 40 bytes. Each result tuple has about 320 $\sim$ 640 bytes on average, by simply concatenating all tuples from join relations. Thus, the whole data in one test query (including intermediate results) can go up to 600MB. The data size in our experiment is chosen to make sure all the hash tables can fit in the main memory since our main focus of this work is the main memory based processing.

[2]We randomly generate connected acyclic graphs given a specified number of nodes. Each node represents a join relation, while each edge denotes the join condition.

3.4.2 Segmented Bushy Processing Evaluation

Impact of the Number of Data Servers

Initial experiments have been conducted to evaluate the impact of the number of Oracle data servers in the experimental setup on the overall performance. We compare the performance of multi-join queries using a pure right-deep tree (pipelined) processing given different numbers of data servers. The test queries are generated randomly with $8 \sim 16$ join relations. For each query, we vary the number of data servers from 1 to 4. Thus, if we have i data servers with $1 \leq i \leq 4$ and k (either 8, 12, or 16) join relations, then we have each data server hold on average $\lceil k/i \rceil$ join relations. These data servers are deployed on different machines with similar configurations having Oracle 8i installed. The result is shown in Figure 3.7. Each data point in Figure 3.7 reflects an average of 50 randomly generated queries for each query type (queries have the same number of join relations). In Figure 3.7, x-axis denotes the number of join relations in the query, while y-axis represents the total processing time. From Figure 3.7, we can see that the number of data servers in the system only has a minor impact on the overall performance. This is because the total time spend on reading the tuples from data servers only represents a small fraction of the total query processing time in our current experimental settings. Thus, the improvement due to shared read by multiple data servers does not play a major role in the overall performance. This indicates that the data server is not the bottleneck in our experimental environment. Without loss of generality, we report our following experimental results with a setup that stores all join relations in one data server.

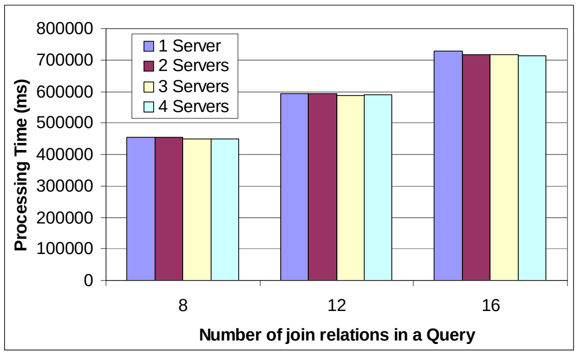


Figure 3.7: Vary the Number of Data Servers

Pipelined vs. Segmented Bushy Processing

Experiments have been conducted to compare the performance (total processing time) of a pure right-deep tree processing having fully pipelined processing to our proposed segmented bushy tree processing that mixes both pipelined and independent parallelism. Figure 3.8 shows the results of 20 randomly generated queries with 8 join relations. Here, the segmented bushy tree has a maximum of 3 join relations per group. In Figure 3.8, we see that a segmented bushy tree processing almost consistently outperforms fully pipelined processing.

Figure 3.9 shows the results of queries with an increasing number of join relations in the query. The number of relations in a query ranges from 8, 12 to 16. The experimental results reflect an average processing time over 50 different randomly generated queries per query type. For example, for queries with 8 join relations, we generate 50 queries randomly. We

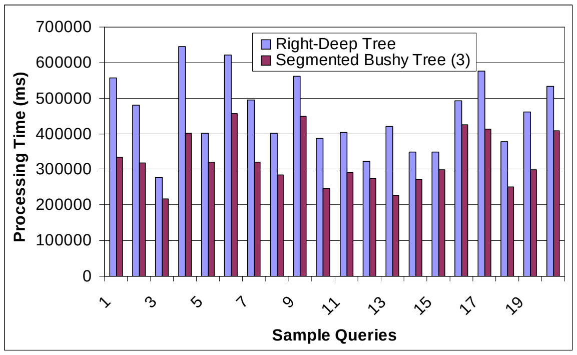


Figure 3.8: Performance of 20 Example Queries

then produce both the fully pipelined processing and the segmented bushy processing strategies for each generated query. In this experimental setup, queries with 8 relations are divided into groups having a maximum of 3 relations, while queries with 12 and 16 relations are divided into groups having a maximum of 4 relations.

In Figure 3.9, we can see that segmented bushy tree processing is consistently better than maximal pipelined parallelism. The performance improvement is around 50% in terms of the total processing time.

Probing Relation Selection for Final Pipelined Segment

Selection of the probing relation of a pipelined segment is usually based on the cardinality of the join relations. This is because choosing a large relation as probing relation can effectively reduce the work and processing time of the building phase. However, for a pipelined segment that involves outputs from other segments (assuming main memory is enough to hold these building relations), the cardinality of the relation alone may no longer be

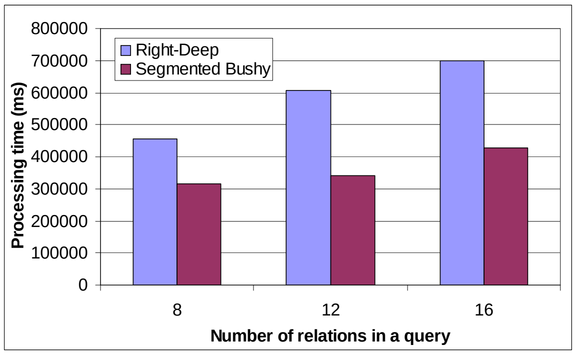


Figure 3.9: Right-Deep vs. Segmented Bushy

the best choice in general. Changing the probing relation of a pipelined segment that only involves source join relations does not change the number of probes in the probing phase. It only changes the number of probing and building tuples. Here we define the number of probe steps as the maximum number of hash tables that a tuple from the probing relation needs to probe to produce the final output. However, for a pipeline segment having outputs from other segments, changing the probing relation will also change the total number of probes.

For example, if we change the probing relation for the pipeline segment P_1 as shown in Figure 3.10(a) from R_7 to R_6, no changes in the number of probe steps occur. Both of them are 3 (Figures 3.10(a)-(b)). However, if we change the probing relation of pipeline P_3 (exchanging P_1 and P_2), then the total number of probe steps changes from 4 to 5 in this case. This is because P_1 itself has 3 probe steps while P_2 only has 2.

Figure 3.11 shows the experimental results of the impact of the probing relation selection for the final pipelined segment. Here, the number on

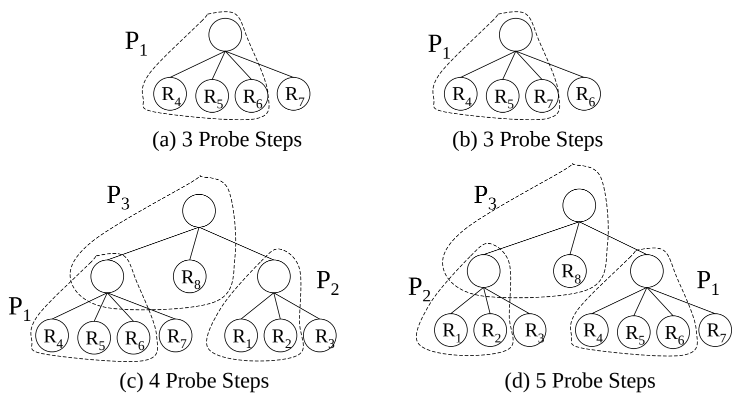


Figure 3.10: Probing Relation Selection

the x-axis denotes the number of relations in the probing relation of the final pipelined segment. The generated queries have 16 join relations. In Figure 3.11, we see that in our current environment, the larger the number of relations in the probing relation of the final pipelined segment, the worse the total processing performance will be. This is because the longer probe steps in the final pipelined segments impair the processing performance. This again confirms our observation that a full pipeline may not be the best performer. Note that the performance degradation for a pipeline that is longer than 8 can be explained by the experiments shown in Figure 3.9. Hence, in Figure 3.11, we conveyed the scope of smaller pipeline sizes.

Number of Join Relations per Group

Figure 3.12 illustrates the impact of the maximal number of join relations per group in our environment. Here, all the tested queries have 16 join relations. We vary the number of join relations per group from 3 to 6. As we can see, if the number of join relations per group increases, the total processing

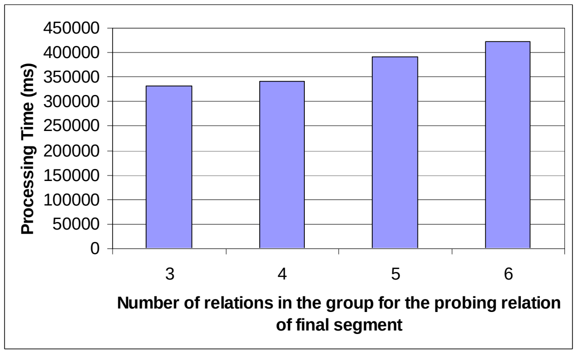


Figure 3.11: Probing Relation Selection

time also increases. This is mainly because given our $ComposeBushyTree$ algorithm, the final pipelined segment tends to choose the largest subgraph (the one with the largest number of join relations) as the probing relation since it usually has the largest intermediate results. As shown in Section 3.4.2, a long pipeline of the final pipelined segment degrades the overall performance. We thus revise our algorithm to choose the subgraph with the smallest number of probing steps as the probing relation of the final pipelined segment. As can be seen, the revised algorithm is less sensitive to the number of join relations in a group.

Insufficient Main Memory

Figure 3.13 shows the experimental results when the aggregated main memory is not sufficient to hold all the hash tables of the building relations. We deploy join queries with 32 join relations. Assume the query will be cut into three pieces with each piece being executed sequentially. Here, the intermediate results of each piece will be first written to the data server,

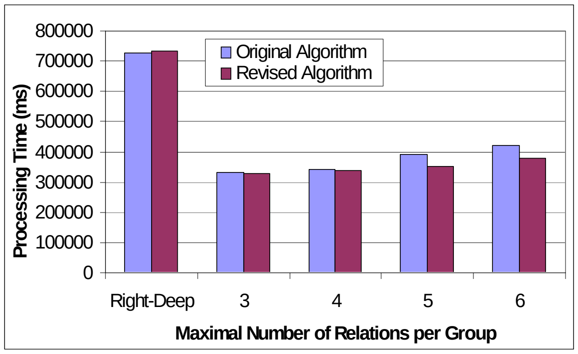


Figure 3.12: Exchanging the Probing Relation

while the next piece will read the intermediate results back into the main memory. We compare the performance of the segmented right-deep tree with our segmented bushy tree generated by Algorithm 2. Note that the segmented right-deep tree has each piece fully pipelined, while the segmented bushy will have the same right-deep segment (piece) further composed into a segmented bushy tree with a maximum of 3 join relations per group. Figure 3.13 reports the comparison between these two approaches for 10 randomly generated queries. As can be seen, the segmented bushy tree processing consistently outperforms the segmented right-deep processing. This is expected because each piece is processed more efficiently given our segmented bushy tree approach. Thus, the overall performance of the query is correspondingly improved.

Concluding Remarks

As can be seen, these experimental results clearly highlight the main message of our work, namely, the long standing assumption that "maximal

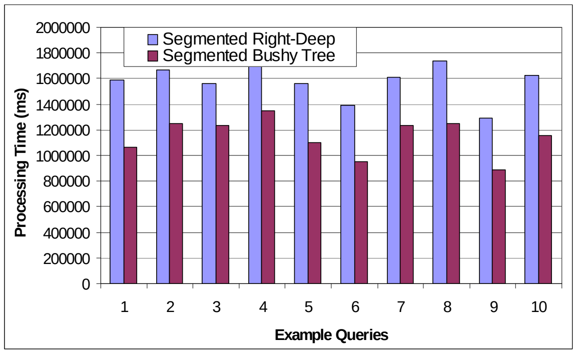


Figure 3.13: Segmented Bushy vs. Segmented Right-Deep

pipelining is preferred" is shown to be wrong. Our proposed segmented bushy processing almost consistently beats full pipelined processing. Given the massive application of pipelined processing, especially in growing areas such as continuous query processing, this observation can also shed some new light on how best to optimize distributed pipelined query plans when the optimization function is related to total processing time.

Chapter 4

Related Work

Parallel query processing has been extensively studied in the literature [17, 22, 25, 31, 41, 48, 51, 59, 77, 78, 95, 99, 116]. Most work in the literature has had rather different research focuses, such as parallel database systems, scheduling algorithms, resource allocation, and load balancing. These works provide the necessary background for this part of my dissertation work.

A lot of early work focused on implementing parallel database systems. For example, the GAMMA Database Machine [32] is implemented on Intel iPSC/2 hypercube with many processors. It provides various partitioning techniques, parallel hashing join algorithms to speedup and scale the query processing. Bubba [12] and Teradata [1] are other similar prototypes. Volcano [41] proposes an operator model by introducing exchange operators in a query dataflow to achieve partitioned parallelism. PRISMA/DB [116] is a main-memory based parallel database prototype that provides a query execution platform to enable experiments related to the performance of parallel processing. Many techniques have been proposed and their performance also has been investigated based on these systems. However, these works usually assume the data sources are fully cooperative in the query processing. Thus, each data source is able to process any part of the queries that are assigned to it. As we have discussed in Section 1.1.2, the view computation is assumed to be processed outside the data sources due to the data sources possibly being too busy in daily transactional processing or the data sources being simply not willing to give control to outsiders. Thus, the data source processing capabilities cannot be taken into consideration in generating or optimizing view computation plans.

As discussed in [47] and other related work, three types of parallelism could be identified, namely, partitioned parallelism, pipelined parallelism, and independent parallelism. Orthogonal to the different types of parallelism, various system architectures can be applied to realize parallel processing, i.e., shared-memory, shared-everything, or shared-nothing. Among these, a shared-nothing architecture is shown to have better speedup and scalable performance [31]. Our work here thus applies this shared-nothing architecture (i.e., a cluster of high-performance PCs) and investigates how all three types of parallelism can be applied in parallel multi-join query processing.

Two heuristics-based algorithms for scheduling a pipelined query tree, localcuts and boundedcuts, are proposed by [48]. They schedule pipelined query operators to a set of shared-nothing processing nodes based on communication and computation costs of each operator which minimizes the total work. Task scheduling in general has been extensively studied in typical parallel processing. [59] provides a recent survey in this area. Numerous algorithms such as list-scheduling [52, 60], clustering [119], task duplication [7, 86], and guided random search techniques [28, 61] have been developed. All these scheduling algorithms usually focus on the inter-operator (task) level. That is, these works usually treat a whole query operator as the smallest unit in the allocation. While our work has the main focus on the partitioned parallelism (intra-operators). Moreover, these works assume a pipelined parallelism whenever it is possible, thus all the operators can be processed simultaneously. In this case, scheduling is not the main focus if enough processing nodes are available and all the available processing nodes are fully utilized.

Resource allocation is related to parallel processing in general. [71] proposes algorithms to achieve optimal processor allocation for pipelined hashing joins in a multiprocessor environment. [39] develops an approach which schedules multi-dimensional resources for pipelined queries. Other allocation algorithms such as memory allocation strategies for complex queries [81] are also related. In this work, we take the similar approach as [71]. Other focuses such as load balancing [13, 33] and degree of parallelism [78, 116] have also been investigated that relate to parallel processing. These ideas have also been applied in our work.

Distributed query processing in general is fundamentally similar to parallel query processing. That is, there is a close relationship between these two areas. [57] provides a recent survey on state-of-the-art distributed query processing and optimization. A lot of optimization techniques, such as two-phase optimization [51] and dynamic programming [58, 96], can be found to be useful in parallel query processing.

All above mentioned related work provides the general background for this parallel view computation work in this dissertation. As we have mentioned in Section 2.2, the closest related work is processing a multiple join query via hashing in parallel over a shared-nothing environment [77, 95, 106]. Different parallel processing strategies such as left-deep and right-deep [95], segmented right-deep [21], and zigzag tree [124] have been proposed, as we have discussed in-depth in Section 2. However, these proposed solutions all share the common approach which is to maximally use pipelined parallelism (i.e., maximally divide a right-deep tree into segments) based on certain objective functions (i.e., memory constraints). Each segment is then processed one by one. In this work, we instead consider more tradeoffs in optimizing such parallel multi-join query processing, i.e., other types of query tree shapes, independent parallelism and its dependencies, properties of the join definitions to reduce redirection costs, etc. Moreover, most of the previous works report their results based on simulations, while we report our results based on a working distributed system.

[117] experimentally compares five types of query shapes such as left-deep, right-deep, wide-bushy and various execution strategies based on the PRISMA/DB system [116]. However, it does not explore how to generate optimized parallel processing query plans. In this work, we propose segmented bushy processing and also provide algorithms to generate such segmented bushy parallel processing solutions.

Part II

Run-time Operator State Adaptation

Chapter 5

Non-Blocking Pipelined Query Processing

5.1 Introduction

5.1.1 Motivation

Current research of non-blocking pipelined query processing often seems to assume that query operators have fairly small-sized operator states, i.e., small-window joins, or stateless operators such as select and project [3, 8, 9, 56]. Query operators with potentially huge operator states, such as multi-way joins, have not been carefully studied in the non-blocking query processing context. However, such query operators are rather common in our target data integration or data warehousing environments. For example, a real-time data integration system as shown in Figure 5.1 helps financial analysts in making timely decisions. Here, stock prices, volumes and external reviews are continuously sent to the integration server during the working hours (i.e., 9AM-4PM). The server is required to process these input streams as fast as it can to output the data in real-time fashion to the decision support system. This way, analysts are able to analyze and make decisions based on the up-to-date information. Two factors are important for such a real-time integration server: (1) The ability to produce as many results as possible given its resources, if not all of them, when data comes through. This ensures that the decision-maker applications could have more information available instantly during working hours. (2) The capability to assure that the whole data processing is conducted in an efficient manner to minimize the overall processing time. This would benefit the analysis applications that rely on the complete data, i.e., quantative analysis. Thus, the overall processing may be composed of two phases: First, a specified *run-time* phase (i.e., from 9AM to 4PM). Second, a *post-run* phase (i.e., to clean up disk resident states if operator states have been pushed to disks during run-time whenever memory overflow occurred).

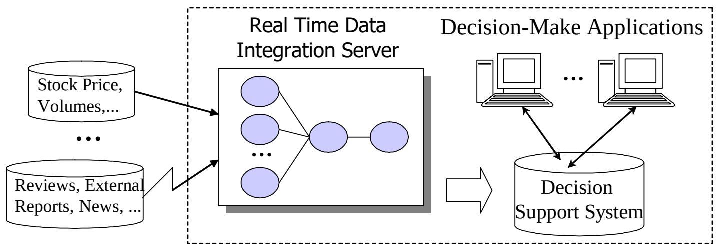

Real Time Data Integration Server
Decision-Make Applications
Stock Price, Volumes,...
Reviews, External Reports, News, ...
Decision Support System

Figure 5.1: A Real-time Data Integration System

The stringent requirement of generating near real time results demands efficient main memory based query processing. This is because any delay in the query processing will cause the input data to accumulate in the system. This in turn would be likely to further accelerate the slow down of the overall processing. Thus, the system may quickly use up all available main memory. This is particularly critical for data integration type queries that are complex and stateful in nature, for example, multi-join queries. Given that the main memory is a limited resource, there is a demand for efficiently utilizing main memory during the query processing.

In this part of dissertation work, we focus on addressing the run-time memory shortage during query processing by adapting operator states, i.e., moving operator states across distributed machines or temporarily pushing operator states into disks. In particular, we focus on queries with state-intensive operators, i.e., queries with multiple-way join [111] operators. These queries are common in data integration related applications as shown in Figure 5.1. Note that stateless operators in the query such as select, project, or operator states are not monotonically increasing (i.e., operators have a small fixed window size) can be addressed without special attentions in the query processing given our focus on run-time main memory usage. This is because these operators are not memory intensive in nature.

As indicated above, we focus on applications that need accurate query results. Thus, all input tuples have to be processed. We thus cannot resort to techniques such as load shedding or approximations [107] to address run-time memory shortage. This also implies that the states of stateful operators could be monotonically increasing during the run-time phase. As motivated in Figure 5.1, we assume the query is long running but finite. However, the techniques we study in this work could also be applied to cases with infinite data streams as long as operators have finite window sizes, a common situation in continuous query processing environments.

5.1.2 State-Level Adaptations

One viable solution to address the problem of main memory shortage, as discussed in XJoin [109] and Hash-Merge Join [79], is to choose memory resident states and push them into disks when memory overflow occurs. This type of approach delays the processing of certain states (the disk resident states) until a later time when more resources would be freed up. The processing of the disk resident states is referred as *state cleanup*. It is required to generate any thus far missed results. We refer this pushing and cleaning process as *state spill* adaptation. Given a monotonic increase of the operator states, it is not likely that we would have the opportunity to perform the state cleanup process during the run-time phase. Thus, these disk resident states would be processed after the run-time phase finishes.

An alternate solution to address memory shortage is to distribute state intensive operators to multiple machines in a shared-nothing architecture, i.e., a cluster of high-performance PCs. Thus, they can be processed in parallel with each machine processing a partition of states (input data). This is referred to as a *partitioned parallel processing* [41, 68, 94, 99]. In such a distributed environment, when only a subset of machines is overloaded at a given time, we then move operator states from the overloaded machine to a less loaded machine. For simplicity, we call this type of adaptation *state relocation*. The potential advantage of this relocation of states is that the adapted states remain active in main memory once the adaptation is completed, avoiding a potentially more long-term delay that would likely be caused by the state spill adaptation. However, this type of adaptation may not in all cases solve the overall memory shortage problem since even the aggregated main memory of multiple machines remains limited.

For this reason, we propose to investigate both type of adaptations in an integrated manner. This should facilitate a more comprehensive solution since state spill may not be efficient due to the access of slow secondary storage, while state relocation alone may not fully resolve the problem of memory shortage. In the rest of this chapter, we first discuss the partitioned non-blocking query processing with multiple input state intensive operators and its performance evaluation. This has not been addressed in previous works such as Flux [99].

In Chapter 6, we then analyze the tradeoffs regarding the factors and policies to be considered when adapting operator states to overcome run-time main memory shortage. Two adaptation approaches that both aim to maximize the overall run-time *query throughput*, namely, *the lazy-disk* and *the active-disk* methods, are proposed. The query throughput here is defined as the total number of tuples have been output thus far. Both approaches integrate state spill and state relocation in memory constrained environments where the aggregated memory of a distributed system is not sufficient for the given query workload.

While in Chapter 7, we investigate the state spill strategies for queries with multiple state intensive operators. We propose *global throughput-oriented* adaptation strategies to handle interdependency when spilling operator states among different operators in the query. These strategies also aim for maximal run-time query throughput.

Extensive experimental evaluations have been conducted based on D-CAPE system [70, 91, 104]. These experiments confirm the effectiveness of our proposed adaptation approaches.

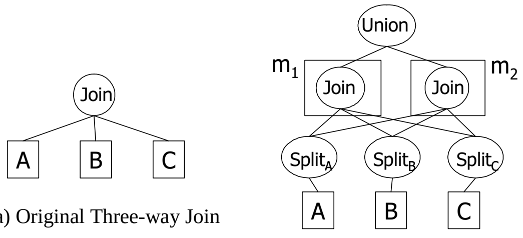
5.2 Partitioned Non-blocking Query Processing

As in [41, 99], we assume that the state intensive operators in the non-blocking query can be so large that they cannot fit in main memory of one single machine. As a result, partitioned parallelism [41, 68, 94, 99] that aims to partition the operator and thus to process it in multiple machines will be applied to these operators. Throughout this work, we use a symmetric multiple-way hash join operator [111] as a representative example of the state intensive operators. Other state intensive operators can also be addressed in a similar manner as long as their functionality can be distributed to multiple machines with each machine only processing non-overlapping partitions of inputs.

5.2.1 Partitioning State Intensive Operators

The first question that needs to be addressed is how to achieve such a partitioned parallel processing for state intensive multi-input operators. As discussed in [41, 99], we insert a *split* operator in front of each input stream of the to be partitioned operator (the state-intensive operator). This split operator partitions the input stream and sends the appropriate partitions to each machine. For simplicity, we will henceforth refer to each instantiation of the operator that is running in a particular machine as an *instance* of the partitioned operator.

For example, assume we process a three-way join query ($A \bowtie B \bowtie C$) as shown in Figure 5.2 (a). The join is defined as A.A_1 = B.B_1 = C.C_1 where A, B, and C denote the three input streams (join relations) and A_1, B_1, and C_1 are the join columns of A, B and C respectively. As shown in Figure 5.2(b), the query plan is partitioned and and run on two machines denoted as m_1 and m_2. The $Split_A$ operator partitions the input stream A based on the column A_1, while the $Split_B$ operator partitions the input stream B based on B_1, and so on [1].



(a) Original Three-way Join

(b) Partitioned Three-way Join

Figure 5.2: Example of Partitioned Processing

Thus, each operator instance only processes non-overlapping partitions of the input stream (a portion of the whole input data). A *union* operator, if needed for result merging, can be inserted into the output streams of all instances of the partitioned operator to combine the results into one output stream for further processing.

[1]Note that for m-way joins (m > 2) that the join conditions are defined on different columns, more data structures are required to support this partitioned m-way join processing. The discussion of this can be found in Appendix A.

Such partitioned processing has advantages over centralized processing. Clearly, more resources are made available given multiple machines are involved in the computation.

5.2.2 Initial Distributions and Connections

An initial distribution can be applied to distribute query operators to multiple machines before the execution. This usually has to be established without statistics information about the operators and input streams. In this work, we utilize operator type in the initial distribution since the operator type convey the potential resource requirements of the operator. For example, a symmetric m-way join operator usually is state intensive, while a select operator is stateless. Basically, we distribute each state intensive operator to all available machines. We then allocate the rest of query operators, such as split and union, using a Round-Robin algorithm to all these machines. The goal of this distribution algorithm is to balance the number of operators as well as the load among different machines. Note that a detailed investigation of such initial distribution algorithm is not the main focus of this work. We design a simple algorithm here to only provide necessary background for the run-time operator states adaptations.

Figure 5.3(b) illustrates the distribution of the three-way join query (as shown in Figure 5.3(a)) over 4 machines (represented by $m_i, 1 \leq i \leq 4$). In our architecture, each machine carries a full copy of the original query plan with all the query operators in each machine being deactivated by default initially. The shading of an operator represents that the operator is activated in that respective machine. In this particular example, each machine has one three-way join operator and one stateless operator assigned as shown in Figure 5.3 (b).

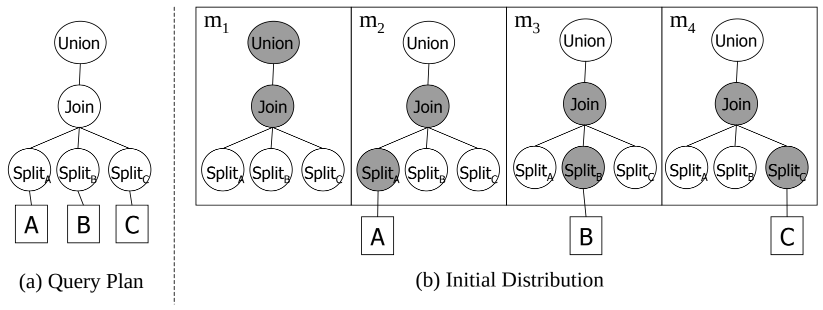


Figure 5.3: Partitioned Query Plan Distribution

The partitioned query plan, operators in the query plan that have been assigned to different machines, needs to be connected together to reflect the pipeline relationship among operators defined in the original query plan. Basically, if two operators with a producer-consumer relationship are activated in the same machine, we then use the memory-based queue to connect them together. If these two adjacent operators are assigned to different machines, we then create a Socket connection to connect them. For example, the connection of the initial distribution as described in Figure 5.3 (b) is shown by Figure 5.4. Solid lines in Figure 5.4 represent the connections between operators (either local or distributed) in the partitioned query plan, while the dashed lines (connections defined in the original query plan) are not used for this particular connection setup. Note that all the network connections and the corresponding data transfers are managed by the *Data Distributor* and the *Data Receiver* (see Figure 5.5) in each machine with dedicated common sending and receiving buffers. Thus, any transient imbalances due to short-term burst arrival rates can be naturally handled as discussed in Flux [99]. The output connections of the split operator are created dynamically based on the number of state-intensive operator instances. In Figure 5.4, each split operator has 4 output connections (one local and 3 distributed connections) given the join is being partitioned into 4 machines. If we change the distribution, these output connections also will be changed correspondingly.

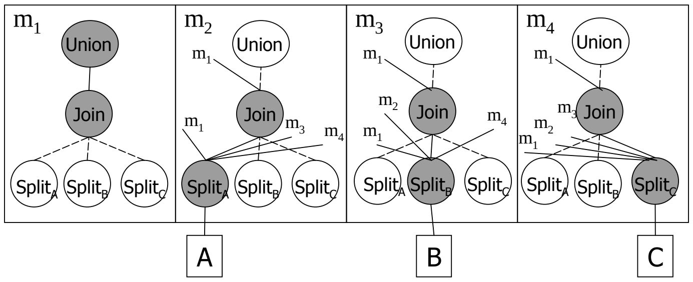


Figure 5.4: Partitioned Query Plan Connection

5.2.3 Experimental Setup

Experimental Environment Description. All our experimental studies in this part of dissertation work are conducted based on the D-Cape system developed at WPI [70]. The overall system architecture is described in Figure 5.5. We have a dedicated *distribution manager* in charge of a set of *query processors*. The distribution manager distributes query plan(s) and connects operators that are distributed into different query processors (as discussed in Section 5.2.2). It collects and analyzes running statistics of each query processor. It makes *global adaptation decisions* such as relocating states from one query processor to the other. Each query processor employs a Cape continuous query engine [91] as its core. The Cape engine takes care of executing the continuous query plans (operators) assigned to it. Each query processor also has modules to collect running statistics and report statistics to the distribution manager, to receive tuples from up stream operators and distribute tuples to down stream operators if they are activated in other query processors. A *local adaptation controller* is responsible for choosing operator states to be spilled or relocated. Note that in this architecture, the state relocation decision is made by the distribution manager based on the collected statistics, while the decision of choosing which partitions to spill/relocate is handled by the local adaptation controller in each query processor. Unlike Flux [99], which puts all the adaptation and partitioning functionalities into the Flux operator, our architecture applies 'light' split operators with a tiered decision architecture. It enables more global adaptation decision making especially given multiple state-intensive operators in the query plan.

The D-Cape system is deployed on a 10-machine cluster. Each machine in the cluster has dual 2.4Hz Xeon CPUs with 2G main memory. These machines are connected via a private *gigabit* ethernet. We dedicate three of them to run the distribution manager, stream generator, and application server respectively. The stream generator continuously generates stream input tuples for queries to process, while the application server processes the output results of the query plan. All the other machines are deployed as query processors as necessary for the given experiment.

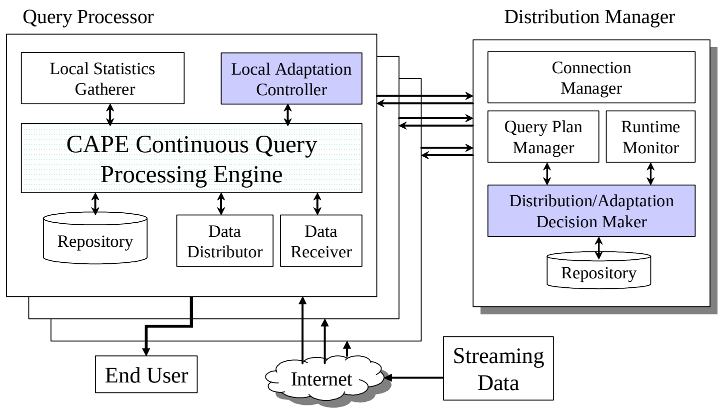


Figure 5.5: D-Cape System Architecture

Experimental Data Sets and Queries. We use a three-way symmetric hash join query plan as described in Figure 5.2(a) to report our experimental results in the following sections. The join is defined on the first column of each input stream. We partition each input stream into 300 partitions based on its join column values. As we will discuss shortly, the run-time adaptation policies we developed, as the main focus of this work, are based on main memory usage and the statistics of each individual partition, they are insensitive to the query plan itself. Other query plans will shown similar results as we will describe below.

We vary the following variables in generating input streams in our experimental studies. We use term *tuple range* to indicate the possible number of tuples with distinct join values in a given set of tuples. We define *join multiplicative factor* as the average number of tuples with the same join value in a given set of tuples. Here, the given set can be tuples have been processed thus far from one input stream, or it can be the tuples in one particular partition of the input stream. Clearly, this join multiplicative factor is not a static number in our environment. This is because operator states are accumulated as more inputs get processed (given our no state purging assumption). Thus, the join multiplicative factor keeps on increasing as more inputs get processed.

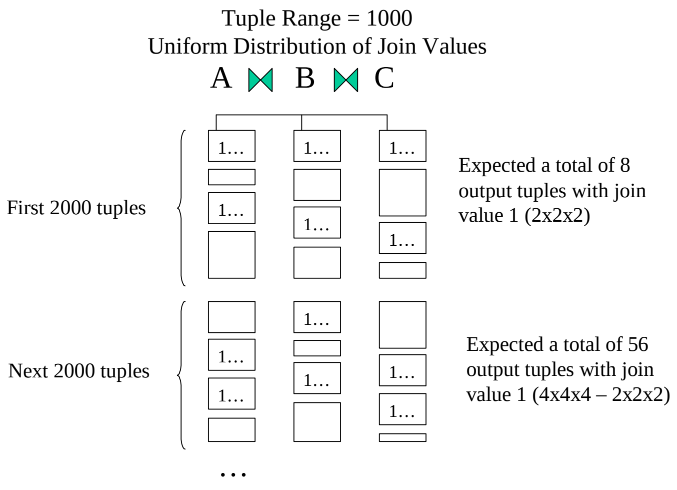

Tuple Range = 1000
Uniform Distribution of Join Values
A ⋈ B ⋈ C
First 2000 tuples
1…
Expected a total of 8 output tuples with join value 1 (2x2x2)
Next 2000 tuples
Expected a total of 56 output tuples with join value 1 (4x4x4 – 2x2x2)
…

Figure 5.6: Join Factor and the Number of Join Results

Above join multiplicative factor is closely related to the the number of join results generated. For example, as shown in Figure 5.6, assume we have a three way join. Each input stream has a tuple range of 1000. We assume join values are uniformally distributed. Thus, after processing 2000 tuples per input stream, we expected two tuples with a join value 1 from each input stream. It generates a total 8 ($2 \times 2 \times 2$) output tuples with the join value 1. While after the next 2000 tuples has been processed from each input stream, we then would expect a total of 4 tuples with the join value 1 from each input stream. It thus generates a total of 64 ($4 \times 4 \times 4$)

output tuples with the join value 1. Thus, a high join multiplicative factor indicates that a larger number of tuples will be generated during the query processing.

As we will discuss shortly, we will partition each input stream into a large number of non-overlapping partitions to help the run-time adaptation. Thus, different partitions may have different join multiplicative factors over time depending on the partitioning and the distribution of join values. We now define term *join multiplicative factor increase rate* (r), or simply *join rate*, to describe that the join multiplicative factor increases r after processing every k tuples. For example, as illustrated in Figure 5.7, we assume partition P_1 has join rate 1, while partition P_2 has join rate 3 for every 1000 tuples being processed. As can be seen, a higher join rate implies a higher join multiplicative factor value over time, which in turn results in generating a larger number of output tuples.

Given uniform distribution of join values, the relationship among k, join rate r, and the tuple range can be illustrated by Figure 5.8. We assume that the tuple range of the input stream is 1000. We assume that the join rate of partitions P_1 and P_2 are 2 and 3 respectively. Thus, k can be computed as (2+3)*1000=5000. That is, after processing every 5000 tuples from this input, join multiplicative factor of P_1 increases by 2, while join multiplicative factor of P_2 increases by 3.

5.2.4 Partitioned Parallel Evaluation

The performance analysis of partitioned parallel processing is divided into two parts. One, we assess the performance of partitioning a non-memory

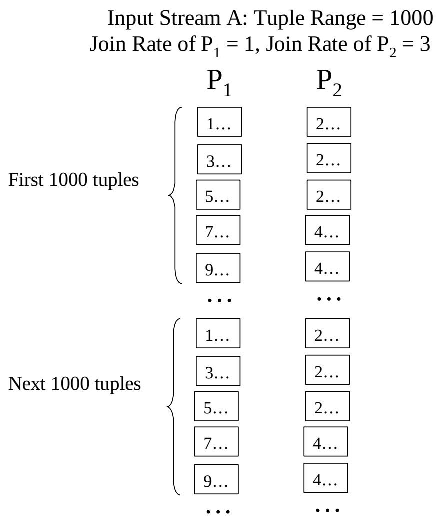

Input Stream A: Tuple Range = 1000
Join Rate of P1 = 1, Join Rate of P2 = 3
P1
P2
First 1000 tuples
Next 1000 tuples
1…
3…
5…
7…
9…
2…
2…
2…
4…
4…

Figure 5.7: Join Rate Example: Non-Uniform Distribution

intensive query that can be fully processed by one single machine. Two, we then study the case that the query workload is beyond the processing capability of one single machine.

Figure 5.9 shows the performance of partitioning a *non-memory intensive* query. In this experiment, the stream generator sends one tuple per 20 ms on average for each input stream. The tuple range is set to 50k, while the join rate is set to 1. The query runs over 40 minutes. The maximal available memory of each processor is set large enough to run the query completely in its own main memory. We then distribute the query to run it on 2 up to 5 machines. Note that in this work, we rely on Java Virtual Machine (JVM) to manage the run-time main memory usage. In most of our experiments, the overall memory usage of one single machine in the query processing is less than 300MB. Given 2GB physical main memory of each machine, run-time

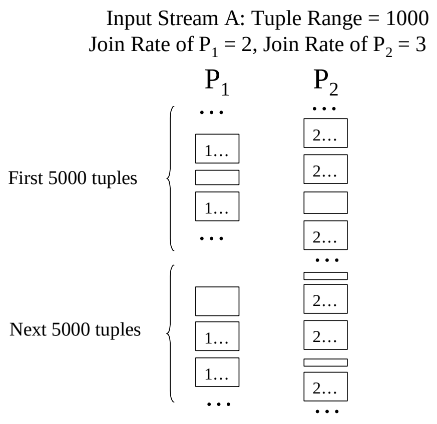

Input Stream A: Tuple Range = 1000
Join Rate of P1 = 2, Join Rate of P2 = 3
P1
P2
First 5000 tuples
Next 5000 tuples
1...
2...

Figure 5.8: Join Rate Example: Uniform Distribution

swapping of main memory pages by the JVM (or the operating system) is not likely to occur that may affect the query processing performance.

In Figure 5.9, the X-axis represents the time (in terms of minutes) of execution up to that point, while the Y-axis denotes the *overall throughput* [2] of the query up to that point of time. Throughput is defined as the total number of tuples output thus far. Here, the line denoted by M_i represents the query throughput when the query is partitioned into i number of machines. As can be seen, there is a negligible effect on the overall throughput when we assign more machines to this query plan. This is because adding extra resources (memory) beyond the maximally needed does not help the overall performance.

However, since each split operator has to partition the input data to all instances of the partitioned operator, this may incur noticeable overhead in

[2]In this work, we focus on query throughput as the measurement of run-time query processing performance. The discussions of using other matrics are beyond the scope of this dissertation. They are left as one possible future work.

the query processing if an unnecessary large number of machines were assigned to the query. For example, it would not be worthwhile to distribute the query into 100 machines while it can be fully processed by one single machine. This is conferred by Figure 5.9 where the overall throughput of 'M_5' already has a slight drop compared with the throughput of 'M_1'.

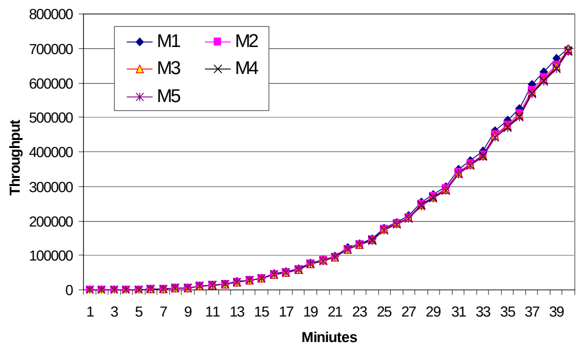


Figure 5.9: Partitioning Non Memory-Intensive Query into 1 $\sim$ 5 machines

Figure 5.10 shows the case for *memory intensive* queries. In this setup, we set the maximal available main memory of each processor to 200MB. We change the data stream generator so that each input stream sends out one tuple per 10ms on average, 2 times more input tuples are generated than the settings in Figure 5.9. The tuple range is set to 50K, while join rate is set to 2 to produce a lot more output tuples. In Figure 5.10, we see that the query throughput drops after about 20 minutes of running compared with full main memory based processing if it is run on one single machine. This is because the memory consumption exceeding the 200MB limitation. Note that it actually stops running after about 25 minutes of running. This is because the JVM tries to take some actions to handle the main memory overflow, i.e., to call garbage collections more frequently. However, this affects the run-time query throughput and eventually fails due to main memory states keep on increasing. If we distribute this query plan on two machines, it runs about 30 minutes before main memory of each machine exceeds 200MB. However, as we discussed above, once we have enough resources to run the query, adding more machines will not improve the overall throughput. In this experiment, we can thus again observe that 'M_4' and 'M_5' have almost the same throughput over the 40 minute run.

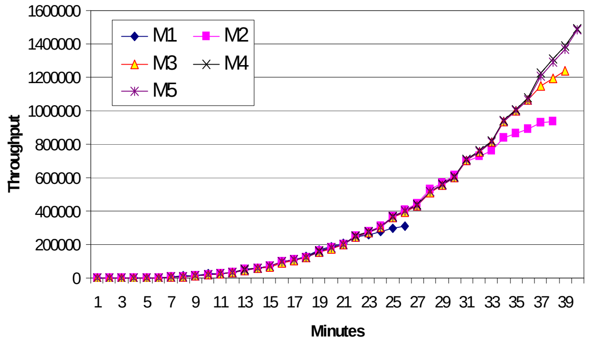


Figure 5.10: Partitioning Memory-Intensive Query into 1 $\sim$ 5 machines

Chapter 6

Run-time Adaptation Strategies

6.1 State Spill

6.1.1 State Partitions and Partition Groups

State spill refers to the processing of selecting memory resident states and pushing them into disks when memory overflow happens. Given a monotonic increase of memory usage during the run-time phase, these spilled states will be kept in disk (inactive) until memory overflow has been addressed, i.e., at the end of the query processing.

To facilitate this run-time adaptation, we divide the input streams into a much larger number of partitions than the number of machines. For example, we might work with 500 partitions over 10 machines. This enables us to effectively redistribute partitions at minimal cost without affecting any of the partitions that are not adapted. This is because we can simply choose appropriate partitions to adapt at run time, while avoiding repartitioning during the adaptation process. This method has first been applied in early data skew handling literature such as [33] as well as the recent partitioned continuous query processing work Flux [99].

We organize operator states based on the input partitions. Since each input partition is identified by a unique partition ID, thus the operator states can also be identified by the IDs of the input partitions. For simplicity, we also use the term partition to refer to the corresponding operator state partition if the context is clear.

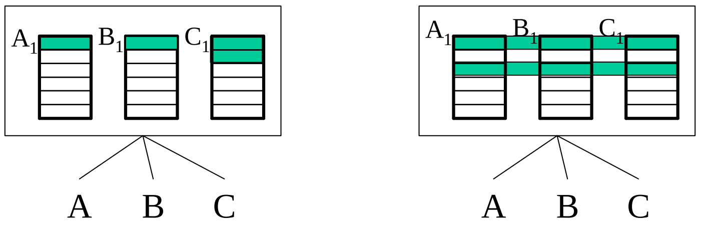
For a single input query operator, as tackled in Flux [99], it is natural to choose partitions from one input stream since there is only one input stream for the single input query operator. However, for a multiple-input operator, there are partitions from different inputs in the operator states with the same partition ID. Thus, multiple ways of organizing partitions are possible, as we will discuss below. Note that we discuss the possible smallest unit in the adaptation here. The strategy of selecting partition units to adapt will be described later.

As discussed in XJoin [79, 109], we could choose partitions from one input at a time as shown in Figure 6.1(a). However, this strategy has two potential drawbacks in processing of partitioned multi-way join queries. (1) It increases the complexity in the cleanup process. This is because if the partitions have been pushed to disk, this requires us to keep track of the timestamps of when each of these partitions was pushed, and the timestamps of each tuple in order to avoid duplicates in the cleanup process. For example, partition A_1 has been pushed into the disk at time t during the execution. Here, we use A_1^1 to denote this part of partition A_1. Then all the tuples from B_1 and C_1 with a timestamp greater than t have to join with the A_1^1 in the cleanup. Given A_1, B_1, and C_1 could be pushed into the disk more than one time, the cleanup process needs to be carefully synchronized with the timestamps of the input tuples and the timestamps of the partitions being pushed. (2) If we were to move partitions from individual inputs to other machines, this then would force us to process tuples for that partition with across machine joins. For example, if we have partition A_1 in machine M_1, while having partitions B_1 and C_1 in machine M_2, then a new coming tuple that belongs to A_1 has to access both M_1 and M_2 to produce the join result. Clearly, if instead we were to put all three partitions A_1, B_1, and C_1 (partitions with the same ID) in the same machine, we could access one machine only to produce the result.

In this dissertation work, we thus propose to group the partitions from all involved input streams that have the same partition ID as the smallest unit to be adapted. For simplicity, we call this smallest adaptation unit as one *partition group*. For example, as illustrated in Figure 6.1(b), A_1, B_1, and C_1 together is referred as one partition group. This avoids the expensive processing of queries across multiple machines. It also greatly simplifies the cleanup process as we will discuss shortly. Without loss of generality, we may use the term partition to refer to partition group if the context is clear.

6.1.2 Clean Up Disk Resident Partitions

When memory becomes available, disk resident states have to be brought back to main memory to produce missing results. This *state cleanup process*



(a) Select partitions from one individual input steam

(b) Select partitions from all input streams with the same ID

Figure 6.1: Variations in Choosing Partitions to Spill

can be performed at any time when memory becomes available during the execution. It does not have to be at the end of the run-time phase. In the cleanup, we should produce all missing results due to these disk resident states while preventing duplicates. Note that multiple partition groups may exist in disk given one partition ID. This is because once a partition group has been pushed into disk, new tuples with the same partition ID may accumulate a new partition group in main memory. Later, as needed, this partition group could be pushed into the disk again.

The tasks that need to be performed in the cleanup can be described as follows: (1) Organize the disk resident partition groups based on their partition ID. (2) Merge partition groups with the same partition ID and generate missing results. (3) If a main memory resident partition group with the same ID exists, then merge this memory resident part with the disk resident ones.

Figure 6.2 illustrates an example of the partition groups before and after the cleanup process. Here, the example query is defined as $A \bowtie B \bowtie C$. We use a subscript to indicate the partition ID, while we use a superscript

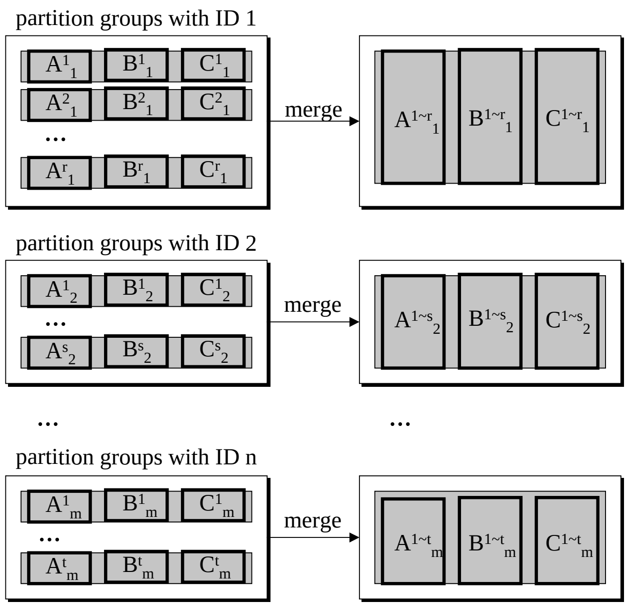

partition groups with ID 1
A^1_1 B^1_1 C^1_1
A^2_1 B^2_1 C^2_1
...
A^r_1 B^r_1 C^r_1
merge
A^{1~r}_1 B^{1~r}_1 C^{1~r}_1
partition groups with ID 2
A^1_2 B^1_2 C^1_2
...
A^s_2 B^s_2 C^s_2
merge
A^{1~s}_2 B^{1~s}_2 C^{1~s}_2
...
...
partition groups with ID n
A^1_m B^1_m C^1_m
...
A^t_m B^t_m C^t_m
merge
A^{1~t}_m B^{1~t}_m C^{1~t}_m

Figure 6.2: Example of Cleanup Process

to distinguish between the partition groups with the same partition ID that have been pushed at different times. The collection of superscripts such as $1 \sim r$ represents the merge of partition groups that respectively had been pushed at time $1, 2, \ldots, r$.

The merge of partition groups with the same ID can be described as follows. For example, assume that a partition group with partition ID i has been pushed k times to disk, represented as (A_i^1, B_i^1, C_i^1), (A_i^2, B_i^2, C_i^2), ..., (A_i^k, B_i^k, C_i^k) respectively. Here (A_i^j, B_i^j, C_i^j), $1 \leq j \leq k$ denotes the j-th time that the partition group with ID i has been pushed into the disk. For ease of description, we denote these partition groups by $P_i^1, P_i^2, \ldots, P_i^k$ respectively.

The results generated within members of each partition group have already been produced during the run-time phase execution. In other words, all the results such as $A_i^1 \bowtie B_i^1 \bowtie C_i^1$, $A_i^2 \bowtie B_i^2 \bowtie C_i^2$, ..., $A_i^k \bowtie B_i^k \bowtie C_i^k$ have been generated during the run-time phase. For simplicity, we denote these results as V_i^1, V_i^2, ..., V_i^k. These partition groups can thus be considered to be *self-contained* partition groups given the fact that all the results have been generated from the operator states that are included in the partition group.

Merging two partition groups with the same partition ID results in a combined partition group having the operator states from both partition groups. For example, the merge of P_i^1 and P_i^2 results in a new partition group $P_i^{1,2}$ now containing the operator states $A_i^1 \cup A_i^2, B_i^1 \cup B_i^2, C_i^1 \cup C_i^2$. Note that the output $V_i^{1,2}$ from partition group $P_i^{1,2}$ should be $(A_i^1 \cup A_i^2) \bowtie (B_i^1 \cup B_i^2) \bowtie (C_i^1 \cup C_i^2)$. Clearly, a subset of these output tuples have already been generated, namely, V_i^1 and V_i^2. Thus now we must generate the missing part $\Delta V_i^{1,2} = V_i^{1,2} - V_i^1 - V_i^2$ in the merging process for these two partition groups in order to make the resulting partition group $P_i^{1,2}$ self-contained. Here, we observe that the incremental batch view maintenance algorithm [63, 66] can be applied to merge partition groups and produce missing results.

Lemma 6.1 *A combined partition group $P_i^{r,s}$ generated by merging partition groups P_i^r and P_i^s using incremental batch view maintenance algorithm is self-contained if P_i^r and P_i^s were self-contained before the merge.*

Proof. Without loss of generality, we treat partition group P_i^r as the base state, while P_i^s as the incremental change to P_i^r. Incremental batch view maintenance equation as described in 9.4 produces the following two results [1]: (1) the partition group $P_i^{r,s}$ having both states of P_i^r and P_i^s and (2) the incremental changes to the base result V_i^r by Δ = $V_i^{r,s}$ - V_i^r. Since two partition groups P_i^r and P_i^s already have results V_i^r and V_i^s generated, the missing result of combining P_i^r and P_i^s can be generated by Δ - V_i^s. As can be seen, $P_i^{r,s}$ is self-contained since it has generated exactly the output results $V_i^{r,s}$ = (Δ - V_i^s) + (V_i^r + V_i^s). ■

For example, let us assume A_i^1, B_i^1 and C_i^1 are the base states, while treating A_i^2, B_i^2 and C_i^2 as the incremental changes. Then, by evaluating the view maintenance equation seen in 6.1, we get the combined partition group $P_i^{1,2}$ and the delta change $\Delta = V_i^{1,2} - V_i^1$. By further removing V_2^2 from Δ, we generate exactly the missing results by combining P_i^1 and P_i^2.

$$\begin{array}{rcl} V_i^{1,2} - V_i^1 & = & A_i^2 \bowtie B_i^1 \bowtie C_i^1 \\ & \cup & (A_i^1 \cup A_i^2) \bowtie B_i^2 \bowtie C_i^1 \\ & \cup & (A_i^1 \cup A_i^2) \bowtie (B_i^1 \cup B_i^2) \bowtie C_i^2 \end{array} \tag{6.1}$$

Lemma 6.2 *Given a collection of self-contained partition groups* $\{P_i^1, P_i^2, \ldots, P_i^m\}$*, a self-contained partition group* $P_i^{1\sim m}$ *can be constructed using incremental view maintenance algorithm in* $m-1$ *steps.*

Proof. A straightforward iterative process can be applied to combine such a collection of partition groups. The first combination consumes two partition groups, while the remaining m-2 partition groups are combined one at a time. Thus the combination ends after m-1 steps. Given each combi-

[1]The detailed discussion and its correctness proof of incremental batch view maintenance algorithm can be found in Part III of the dissertation.

nation results in a self-contained partition group based on Lemma 6.1, the final partition group is self-contained. ■

Based on Lemmas 6.1 and 6.2, we can see that the cleanup process (merging partition groups with same partition ID) ends successfully with all missing results. Note that any memory resident partition groups can be combined with the disk resident parts in the same manner as we discussed above. As can be seen, the cleanup process does not rely on any timestamps. We thus do not have to keep track of any timestamps during the state spill process.

6.1.3 Throughput-Oriented State Spill

Now, we need to address what amount of states and which partition groups to be pushed into disks when main memory overflows. Pushing different partition groups may have different impact on the overall performance. As motivated in Section 5.1.1, we propose a throughput-oriented state spill strategy that aims for high overall run-time query throughput. That is, we aim to generate as many output results as possible given part of the memory resident states are pushed into disks and thus are temporarily inactive. Ideally, given a high overall throughput in the run-time phase, this should also reduce the efforts in the cleanup process as more work would have already been completed [2].

The intuition behind our throughput-oriented state spill strategy is to identify productivities of partitions and push partitions that are less pro-

[2]This may not be true given multiple stateful operators that depend on each other. We will discuss this in more detail in Chapter 7.

ductive in each spill process. Thus, memory space is utilized by productive partitions that could potentially generate more output results.

The usage of partition groups as adaptation unit helps us to realize this throughput-oriented strategy. This is because it simplifies the statistics collection to the granularity of each partition group. Otherwise, one may need to collect a more general histogram for all possible values at the individual tuple level. Instead, for each partition group, we record the current size of each partition group, represented by P_{size}. We also record how many tuples have been generated from this partition group, denoted by P_{output}. We define the *productivity* of each partition group as P_{output}/P_{size}. Given a similar size P_{size}, a small P_{output}/P_{size} value indicates that only few output results have been generated so far. We thus prefer to spill the partitions with smaller productivity values into the disk. The intuition is that the partitions left in the main memory are likely to produce more results than the ones that have been pushed into disks.

Other factors could also be considered when choosing candidate partitions to push such as the *activity* value of each partition group. For example, we can record the time when an input tuple is being inserted into the hash table of the partition, denoted by T_{access}. We then compute *inactive time*, defined as the current time T_{curr} - T_{access}, to indicate the activity of each partition group. If the inactive time is larger than a certain threshold τ, we then choose this partition group as the candicate to be pushed. The intuition of identifying activity of partition is to push the partitions that are least often being used. This is similar in concept to the commonly used LRU cache replacement strategy.

Note that both the productivity and activity values of each partition group only reflect the input data that has been processed so far. They are updated when new data gets processed. Here we assume that the value we observed so far would be indicative of the trends of behavior of the partition group. Given a more dynamic environment where the properties of input data keeps on changing, we may have to apply other techniques to predict the productivity/activity value of each partition group. Clearly, various alternate ways of computing the above productivity and activity values exist. For example, we can maintain a list of history values and assign different weights to each value using some amortized weight function, i.e., the latest one has the highest weight, to compute partition productivity and activity values. However, any new cost model of identifying productivities/activities can be easily plugged into our system if it turns out to be necessary. This is because we work on the mechanisms and policies of run-time state adaptation, which are independent of such low level cost models.

The state spill decision is made by the *local adaptation controller* (shown in Figure 5.5) of each processor. The controller continuously checks the current memory usage and compares it against a threshold θ_s, i.e., 200 MB. If the current memory usage is larger than θ_s, then the adaptation controller initiates the throughput-oriented state spill adaptation process.

The high level description of how to choose the candidate partitions to be pushed to disk is sketched in Algorithm 3. The algorithm returns the partition IDs that will be spilled based on the collected cost statistics, i.e., the productivity values of the partition groups. Here p denotes the percentage of operator states to be pushed in this state spill process. We will experimentally evaluate how to choose p in Section 6.1.4.

Algorithm 3 ComputePartitionsToPush(p)

Input: *p, the percentage of states to be pushed.*
Output: *retID, a list of Partition IDs to be pushed.*

```
1: retID = null; /*list to hold the Partition IDs to be pushed*/
2: pStats = getPartitionStats(); /*get statistics of the partitions*/
3: pStats.computeProductivity(); /*compute & order productivity values*/
4: pct = 0; /*the percentage of states being selected*/
5: while (pct < p) do
6:   if (pStats.hasNext()) then
7:     retID.add(pStats.getNext());
8:     pct = computePercent();
9:   else
10:    break;
11:  end if
12: end while
13: return retID;
```

6.1.4 State Spill Evaluation

We first need to address how much state is to be pushed to disk when main memory gets overloaded. We run the query on one single machine in the cluster for over 50 minutes. The input tuples are set to arrive at a rate of every 30 ms on average from each input stream. The tuple range is set to 30K. The join rate of each partition group is set to 3. In this experiment, the state spill is triggered when the memory usage of the machine is over 200MB. In the experimental studies, we have each partition group frequentlly accessed, thus measuring the activity value in our setup would not turn out to be of much interest. We thus focus on investigating the impact of the productivity of partitions on the state spill performance.

Figure 6.3 depicts the overall throughput with different percentages of states being pushed when overload happens. A k%-push means that k% of the main memory states are chosen to be pushed to disk in each adaptation. We vary k from 10 to 100 in this experiment. We randomly choose partition groups from the operator state for this experimental setup since we investigate the impact of which amount of state to be pushed in each adaptation. As a comparison, we also provide the throughput of the query when it is fully processed in main memory (labeled as 'All-Mem'). Seen from Figure 6.3, the more states are being pushed into the disk each time, the smaller the overall throughput. This is as expected since the states being pushed are no longer active.

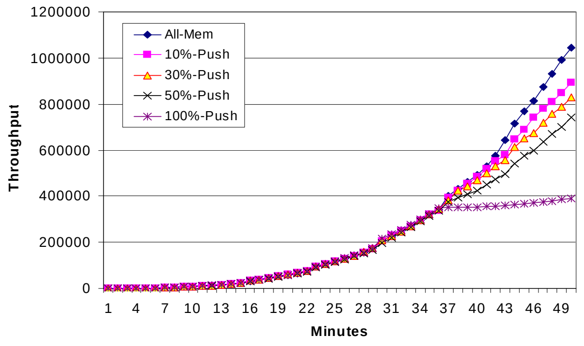


Figure 6.3: Percentage of States Pushed in Each Adaptation

Figure 6.4 shows the corresponding memory usage for above k%-push strategies. The memory is projected based on the memory usage of all active operator states as well as the input and output tuples in queues in the query processor. Seen from Figure 6.4, the main memory utilization can be effectively controlled by the adaptation to avoid system crashs arising due to memory overflow. We also see that the more states (a higher percentage) we push in each adaptation, the fewer times we need to trigger the state-spill process. In Figure 6.4, each zag in the line represents one adaptation process.

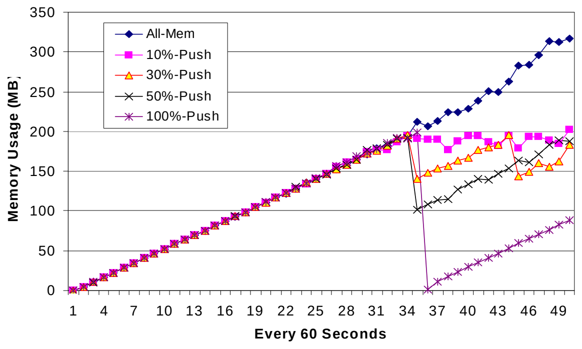


Figure 6.4: Memory Usage vs. Percentage Pushed

Without loss of generality, we will use the 30%-push strategy for further analyzing state spill adaptations in the following experiments unless specified otherwise. This is because the 30%-push strategy has a small number of adaptations, while at the same time it has a reasonable small impact on the overall query throughput.

Given a specified percentage of states to be pushed, i.e., 30%, the question remains which partition groups to be pushed. Figures 6.5 and 6.6 show the impact of choosing different partition groups on the overall run-time phase throughput. In Figure 6.5, the input stream has 1/3 of the partitions with an average join rate of 4, 1/3 of partitions with an average join rate of 2, while the rest have a join rate of 1. We compare the strategies of pushing more productive partition groups with the pushing of less productive ones. The 'push-30% | less-productive' line corresponds to the case of pushing partition groups with the smallest P_{output}/P_{size} value first. As a comparison, the 'push-30% | more-productive' line denotes the pushing of partition groups with the largest P_{output}/P_{size} value first. Seen from Figure 6.5, the 'push-30% | less-productive' strategy has a much higher run-time throughput. This is because that leaving the partition groups with high productivity values in main memory is more likely to generate more output results as input tuples come through.

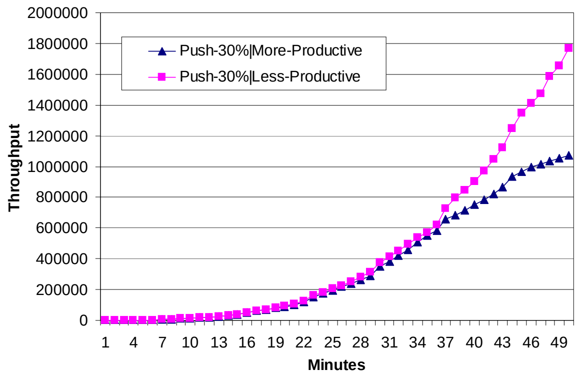


Figure 6.5: Throughput-Oriented Spill: Various Join Rates

A high overall run-time throughput also helps to reduce the cleanup efforts in this case. This is because more work has been accomplished before the cleanup starts. In the above experiment, the 'push-30% | less-productive' strategy uses $26,879$ ms to generate $194,308$ tuples during the cleanup, while the 'push-30% | more-productive' one generates $992,893$ tuples in around $359,396$ ms.

Even if the join factors of different partitions are similar, the throughput-oriented strategy of pushing less productive partition groups does not hurt the overall performance. Figure 6.6 depicts the run-time phase throughput when we set the join factor of all partitions to be around 3. As can be seen, the 'push-30% | less-productive' strategy still slightly outperforms the 'push-30% | more-productive' since it is able to capture even minor differences of partition productivities.

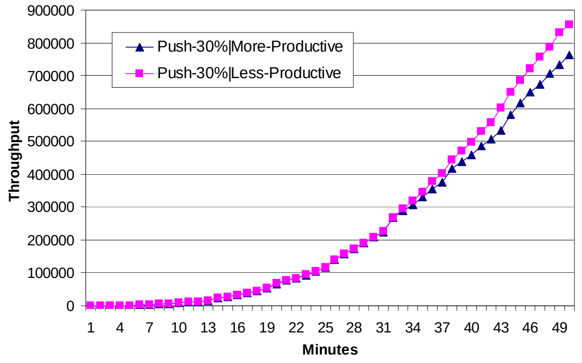


Figure 6.6: Throughput-Oriented Spill: Similar Join Rates

6.2 State Relocation

Uneven workload may arise among machines in a distributed environment. Thus, while one machine runs out of memory, another machine may still have memory at its disposal for holding additional states. Unlike the state spill that results in temporarily inactive adapted states, moving operator states across machines will have the adapted operator states stay in main memory. Thus, the adapted states are still actively involved in the query processing. Given that, state relocation, which moves operator states across machines, may be a preferred choice over state spill to maximally utilize all memory resources.

State relocation requires knowledge from several machines to make an adaptation decision. In our system, the distribution manager has the global knowledge including main memory usage of each machine by monitoring run-time statistics of the distributed system continuously. The state relocation process is triggered once the distribution manager observes an accelerated uneven memory usage among the machines.

State relocation is performed in a pair-wised scheme in our system. That is, once the distribution manager finds the difference between the maximal amount of memory used (referred as M_{max}) and the least amount of memory used (referred as M_{least}) reaches a certain threshold (θ_r) in the cluster, i.e., $M_{least}/M_{max} < \theta_r$, it then initiates the state relocation process. In each relocation process, the distribution manager asks the most used machine to move $(M_{max} - M_{least})/2$ amount of states to the least used one. Note that the actual partition groups to be moved are decided by the local adaptation controller of the machine with the most used memory. Given such tiered decision architecture, the distribution manager only requires to collect very little running statistics such as main memory usage to make globally adaptation decisions. It thus helps to increase the scalability of the distribution manager. Ideally, both machines will have about $(M_{max} + M_{least})/2$ amount of states after the adaptation.

Theorem 6.1 *The pair-wised adaptation scheme converges to a balanced load for any load distribution.*

Proof. Assume n machines with their load (i.e., memory usage) denoted by l_1, l_2, …, l_n in an ascending order. E represents the mean value of the load. σ^2 denotes the current load variance. After one pair-wised adaptation, both l_1 and l_n become (l_1 + l_n)/2. We assume the load variance after the adaptation to be σ_1^2. Here the changes on the load variance Δ = σ^2 - σ_1^2 can be derived as follows:

$$\begin{aligned}
\Delta &= [\sum_{i=1}^{n}(l_i - E)^2] - \{2[(l_1 + l_n)/2 - E]^2 + \sum_{i=2}^{n-1}(l_i - E)^2\} \\
&= [(l_1 - E)^2 + (l_n - E)^2] - 2[(l_1 + l_n)/2 - E]^2 \\
&= l_1^2/2 + l_n^2/2 - l_1 * l_n = (l_1 - l_n)^2/2 > 0
\end{aligned}$$

As can be seen, Δ is always positive. This means that the load variance reduces after applying each pair-wised adaptation. Thus, the minimal load and the maximal load ratio would converge to 1 after enough number of adaptations. This indicates a balanced load distribution. ■

Note that in a dynamic environment with the load of the machines continuously changing, it is not worthwhile (or even possible) to achieve a perfectly balanced load in each adaptation. We thus perform only one pair-wised adaptation in each adaptation phase. For simplicity, we call the machine with the maximally used memory the *sender*, and the machine with the least memory usage the *receiver*.

6.2.1 Moving States Across Machines

For the state relocation process, we need to make sure no data (operator states) are missing, duplicated or corrupted. To achieve that, we design the following 8 steps protocol, denoted as steps i with $1 \leq i \leq 8$. These steps are illustrated in Figures 6.7, 6.8, 6.9, and 6.10. Note that the sender, the receiver and the amount of state to be moved are known before the state relocation process starts.

In this process, the distribution manager and the local adaptation controller of each machine are responsible of executing the moving protocols. Here, the distribution manager is responsible of the overall control of the moving process. It accomplishes that by sending protocols (messages) to the local machines and waiting for appropriate responses. While the local adaptation controller in each machine is in a wait mode until it is woken up by the distribution manager via the moving protocols. The local adaptation controller then performs the requested actions, and returns messages back to the distribution manager. Different local machines have different roles in the relocation process such as sender, receiver, or machines with active split operator(s). The distribution manager has the responsibility to send appropriate messages to the right machines.

The goal of the first two steps of the protocol is to figure out which partitions in the sender's machine are to be moved. These two steps are depicted in Figure 6.7.

In **step 1**, the distribution manager sends a *ComputePartitionsToMove* message to the sender. This message tells the sender the amount of operator

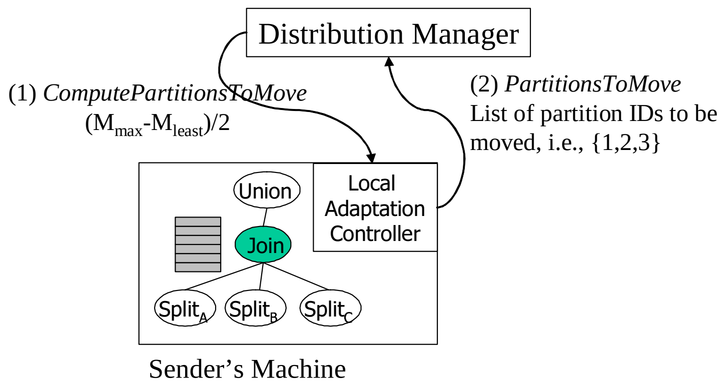


Figure 6.7: Compute Partitions to Move

states ((M_{max} - M_{least})/2) that is to be moved out. In **step 2**, the sender calls its *local adaptation controller* to identify the actual partition groups that it suggests to move. Various local heuristics could be employed here to make this decision, for example, the throughput-oriented strategy discussed in Section 6.1.3. After that, the IDs of these partition groups to be moved are returned to the distribution manager via a *PartitionsToMove* message, as illustrated in Figure 6.7.

After the partitions to be moved have been computed, two issues need to be addressed before moving the selected partitions: (1) We need to create temporary space for holding new incoming tuples belonging to partitions to be moved and (2) we need to make sure all on-the-fly tuples have been processed before moving partitions. Protocol steps that address these two issues are illustrated in Figure 6.8.

In **step 3**, after receiving the *PartitionsToMove* message from the sender, the distribution manager sends a *DeactivatePartitions* message with the par-

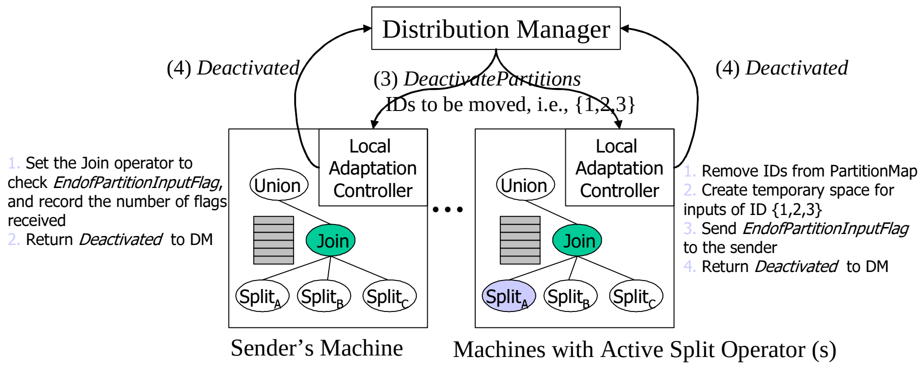

Distribution Manager
(4) Deactivated
(3) DeactivatePartitions
IDs to be moved, i.e., {1,2,3}
(4) Deactivated
Local Adaptation Controller
Union
Join
SplitA
SplitB
SplitC
1. Set the Join operator to check EndofPartitionInputFlag, and record the number of flags received
2. Return Deactivated to DM
1. Remove IDs from PartitionMap
2. Create temporary space for inputs of ID {1,2,3}
3. Send EndofPartitionInputFlag to the sender
4. Return Deactivated to DM
Sender's Machine
Machines with Active Split Operator (s)

Figure 6.8: Deactivate Partitions to be Moved

tition IDs to be moved to the sender and to all machines where split operators are activated.

After receiving the message *DeactivatedPartitions*, the sender is triggered to start checking whether the flag *End-Of-Moved-Partitions* from the split operators has been received. While the machine on which split operator(s) is running notifies the split operator to perform the following two tasks. (1) The split operator creates a temporary storage space. (2) The split operator stops sending the tuples with IDs that will be moved and puts them in the temporary storage space. Then it sends out a special control tuple called *End-of-Moved-Partitions* to the sender to indicate that no more tuples will arrive belonging to the partitions that are to be moved. Note that once the sender receives the *End-of-Moved-Partitions* tuples from all the split operators, this indicates that all on-the-fly tuples have been processed.

The distribution manager will first send the *DeactivatedPartitions* message to the sender, and then to the machines with split operators being activated [3]. This is to make sure the *End-Of-Moved-Partitions* flag will not arrive the sender's machine before the sender is set to receive these flags.

After that, in **step 4**, the split operator returns the *Deactivated* message back to the distribution manager. The distribution manager knows that the partitions to be moved have been successfully deactivated once the distribution manager receives *Deactivated* messages from all split operators. That is, no more input tuples belonging to the partitions to be moved will be sent to the sender.

Steps 5, 6, and 7 together perform the actual movement of states from the sender to the receiver, as illustrated by Figure 6.9.

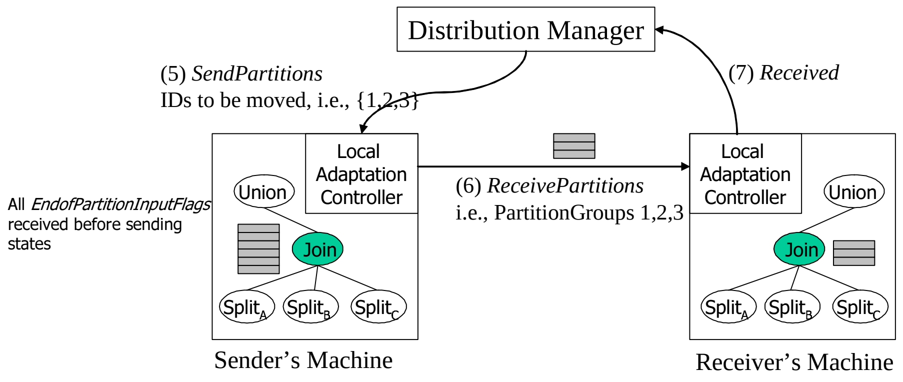


Figure 6.9: Deactivate Partitions to be Moved

In **step 5**, the distribution manager sends a *SendPartitions* message to the sender after it has received all *Deactivated* messages from machines with active split operators. Then, in **step 6**, the sender prepares to move parti-

[3]Note that in a slow (and nonstable) network environment, we may need another round of protocol to make sure that the *DeactivatedPartitions* message is received at the sender before sending it to machines with active split operator(s). However, given a local highspeed cluster environment, a simple wait at the distribution manager is sufficient for this purpose.

tions. As discussed in step 4, the sender has to wait until all *End-of-Moved-Partitions* messages from split operators have been received. This indicates all on-the-fly tuples have been processed. After that, it gets the partition groups to be moved and puts them into a *ReceivePartitions* message. Then it sends the message to the receiver. In **step 7**, the receiver install the partition groups extracted from the *ReceivePartitions* message. The receiver then returns a *Received* message back to the distribution manager to indicate that the partitions have been received and successfully installed.

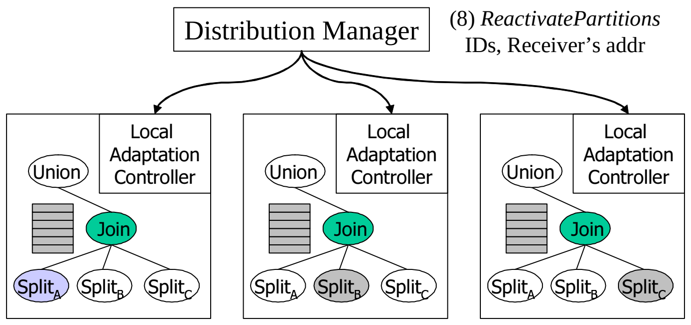


Figure 6.10: Reactivate Partitions that Moved

Finally, in **step 8**, after the distribution manager receives the *Received* message from the receiver, it sends the *ReactivatePartitions* message to machines where the split operators are running. This notifies the split operator to change the partition mapping of the moved IDs. That is, the split operator will now send the tuples with partition IDs that just moved to the receiver machine. Note that the split operator will first process the data in its temporary space, if it is not empty before working with tuples in its input queue.

To summarize, the overall interactions between the distribution manager and the local adaptation controller of each individual machine (for those machines that involved in the relocation process) can be described by the sequence diagram illustrated in Figure 6.11. Here, the local adaptation controller in each processor is responsible for parsing moving protocols and performing corresponding actions.

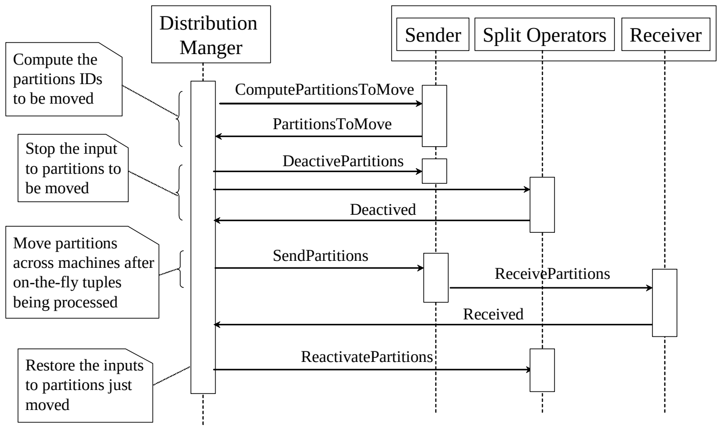

Distribution Manger
Sender
Split Operators
Receiver
Compute the partitions IDs to be moved
ComputePartitionsToMove
PartitionsToMove
Stop the input to partitions to be moved
DeactivePartitions
Deactived
Move partitions across machines after on-the-fly tuples being processed
SendPartitions
ReceivePartitions
Received
Restore the inputs to partitions just moved
ReactivatePartitions

Figure 6.11: Sequence Diagram of State Relocation Protocols

Algorithms 4 and 5 sketch the high level interactions between the distribution manager and the local adaptation controller during the state relocation process.

Algorithm 4 describes the basic operations of the distribution manager. The algorithm is triggered to move operator states from the sender to the receiver when the distribution manager observes $M_{least}/M_{max} < \theta_r$. The algorithm basically follows the actions in the sequence diagram (Figure 6.11) by sending protocol messages and waiting for the corresponding responses. Here, the **send** and **wait** are primitive operators designed to send or wait for messages across machines.

Algorithm 4 State-Relocation:Manager(sender, receiver, amt)

*/*It controls state relocation process by sending moving protocols to local machines and waiting for corresponding responses.*/*

```
1: send ComputePartitionsToMove(amt) msg to sender;
2: wait until get PartitionsToMove msg;
3: send DeactivatePartitions to sender & machines with split operator(s);
4: wait until get all Deactivated msgs;
5: send SendPartitions msg to sender;
6: wait until get Received msg;
7: send ReactivatePartitions msg to machines with split operator(s);
```

Similarly, Algorithm 5 describes the main steps performed in the local adaptation controller during the state relocation process. Here, the algorithm keeps on listening to moving protocols. It performs corresponding actions based on the protocols it has received.

In **step 3** (*DeactivatePartitions*), each split operator creates a temporarily storage space for holding the new incoming tuples that belong to the partition groups to be moved. It then sends a special control tuple (*End-Of-Moved-Partitions*) to the sender. While in **steps 5 and 6**, the sender moves partition groups after it has received all *End-Of-Moved-Partitions* control tuples. Given an ordered (FIFO) message transfer, this guarantees all on-the-fly tuples belonging to the moved partition groups have been processed and included in.

While in *step 8*, each split operator redirects the tuples belonging to the partition groups that just moved to the receiver's machine. This makes sure all new incoming tuples to those moved partitions continue to be processed correctly. Each split operator processes tuples stored in its temporarily stor-

Algorithm 5 State-Relocation:Local()

/ to receive messages, perform corresponding actions, and return message(s) to the distribution manager.*/*

```
1: while (keepGoing) do
2:   wait for moving protocols;
3:   switch(protocol)
4:   ComputePartitionsToMove: /*compute partitions to be moved*/
5:     compute partitions to move;
6:     send PartitionsToMove msg to Distribution Manager;
7:   DeactivatePartitions: /*stop the inputs to partitions to be moved*/
8:     deactivate partition inputs;
9:     send Deactivated msg to Distribution Manager;
10:  SendPartitions: /*send out partitions*/
11:    wait on-the-fly tuples being processed;
12:    send partitions via ReceivePartitions msg to receiver;
13:  ReceivePartitions: /*receive and install partitions*/
14:    install partitions received;
15:    send Received msg to Distribution Manager;
16:  ReactivatePartitions: /*resume & redirect inputs for moved partitions */
17:    reactivate moved partitions;
18:    redirect moved partitions' input;
19: end while
```

age first (if any) to make sure tuples get processed in an ordered manner.

Note that the state relocation protocols are performed at the granularity of the partition group. Thus, the processing of partition groups other than those to be moved would not be affected by this relocation process.

6.2.2 State Relocation Evaluation

We study the following two parameters in evaluating the state relocation performance: (1) threshold θ_r , and (2) minimal time-span between two consecutive relocations τ_m. The distribution manager triggers state relocation if and only if when $M_{least}/M_{max} < \theta_r$ and the time elapsed since the last relocation is greater than τ_m.

Figures 6.12 and 6.13 aim to investigate the impact of these two parameters. Here, the query is partitioned to run in two machines. Each machine processes about half of all partitions. Maximal memory of each machine is set large enough to have the query completely run in main memory. We use a worst case situation in terms of input stream fluctuations having each machine alternatively change its demand of main memory. For example, partitions assigned to machine 1 get 10 times more tuples than those of machine 2 for the first five minutes. After that, the machine 2 gets 10 times more tuples than machine 1 for the next 10 minutes, and so on. Thus the main memory usage of these two machines alternates dramatically every 10 minutes. Given this setup, the state relocation may keep on moving states among two machines, i.e., thrashing by wasting time on moving states.

Figure 6.12 shows the impact of choosing the threshold θ_r. We vary θ_r from 50% to 90%. A high percentage indicates that a larger number of adaptations is triggered with each adaptation only moving a small amount of states. τ_m is set to 45 seconds in this experiment. Note that the impact of τ_m will be further discussed in Figure 6.13.

Seen from Figure 6.12, the throughput when choosing different θ_r is almost the same. All of them experience throughput similar to that of the pure main memory processing without any adaptations ('All-mem'). This implies that the cost of state relocation is low. We thus potentially could perform such state relocations frequently without impacting the overall performance. In Figure 6.12, a total of 24 relocations has been conducted when θ_r is set to 90%, while only 2 adaptations when θ_r equals 50%.

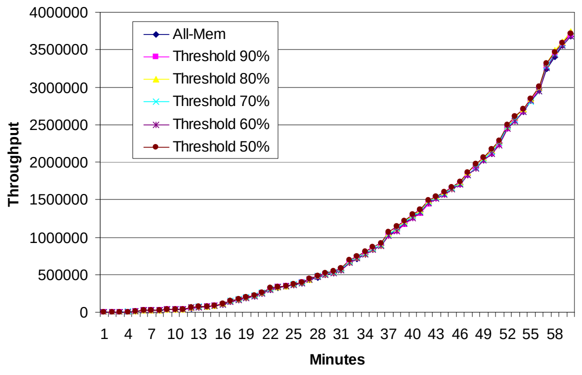


Figure 6.12: Varying Threshold (θ_r) that Triggers Relocation

Figure 6.13 shows the impact of τ_m on the overall performance. We set θ_r equal 90%, the one with a large number of adaptations. We then change τ_m from 15 seconds to 45 seconds. In Figure 6.13, we again see that the overall throughput also does not change too much. In this run, there are 31 relocations when τ_m = 15 seconds, 27 relocations when τ_m = 30 seconds, and 24 relocations when τ_m = 45 seconds.

Seen from Figures 6.12 and 6.13, our pair-wised relocation does not incur significant overhead on the query processing.

We compare the memory usage with/without state relocations, as shown in Figure 6.14. We set θ_t = 90% and τ_m = 15 seconds for the state relocation. The 'no-relocation-M1' and 'no-relocation-M2' show the memory usage respectively of machines M_1 and M_2 without state relocations. As can be seen, the memory consumption alternatively changes due to our input data pattern. 'with-relocation-M1' and 'with-relocation-M2' indicate the memory usage after the state relocations. We can see that the main memory

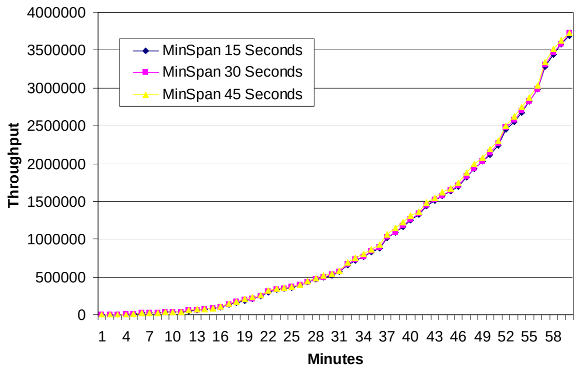


Figure 6.13: Impact of Minimal Time-span (τ_m) on Throughput

usage remains largely balanced due to the relocation.

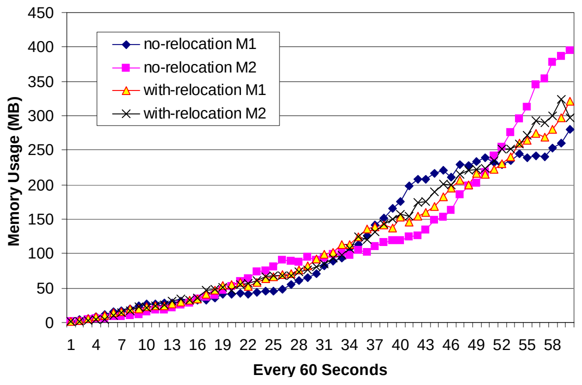


Figure 6.14: Memory Usage with/without Relocation

Applying state relocation maximizes the opportunity for a full main memory based processing if the aggregated main memory is sufficient for a given query workload. It thus has the potential to result in a much higher overall throughput since the cost of state relocation is not expensive as shown by Figures 6.12 and 6.13.

Figure 6.15 illustrates the benefits of the state relocation. The query is run over three machines. We change the initial distribution of partitions to make one machine process 60% of all partitions, while the other two have 20% of partitions respectively. We set θ_r = 80% and τ_m = 45 seconds. In this setup, state spill is triggered when the main memory usage of the machine is over 200MB.

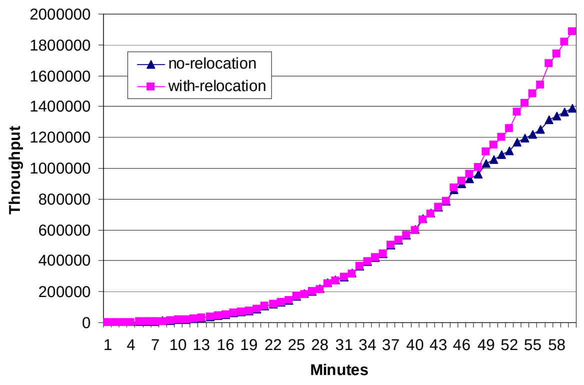


Figure 6.15: Throughput: with/without State Relocation

Seen from Figure 6.15, the throughput of the 'no-relocation' drops after running for 40 minutes. This is because main memory of the machine having 60% of the partitions overflows and starts pushing states into disks. On the other hand, the 'with-relocation' adapts these states to other machines having all states kept in main memory. Thus, it generates output continuously at maximal rate during the run-time phase instead of waiting until the cleanup stage.

6.3 Integrating State Spills and Relocations

6.3.1 Lazy Disk and Active Disk Approaches

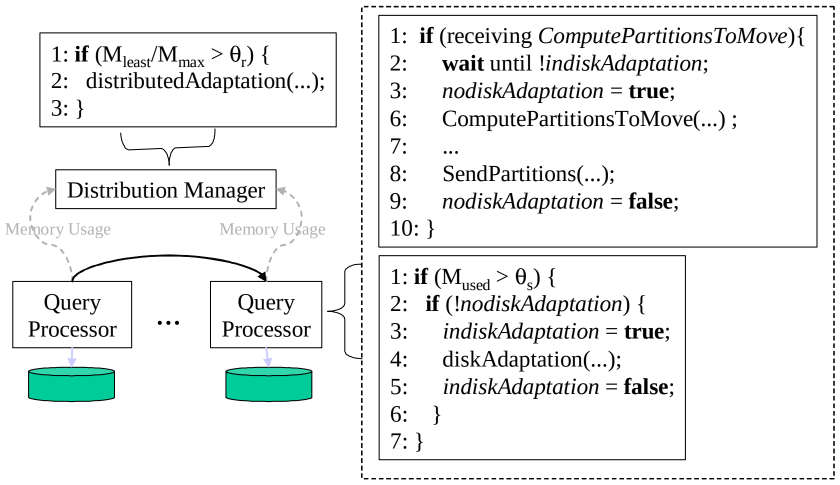
In case of the aggregated memory of all machines not being sufficient for the given query workload, then the state spills cannot be avoided even by relocating states across machines. This is because some machines (or even all machines) in the cluster may suffer due to memory overflow. We now propose two strategies to combine both state spill and state relocation aiming for achieving maximal overall throughput in run-time phase in such memory-constrained environments.

The first solution, called *lazy-disk* approach, is to postpone the state spill until there is no main memory in the cluster that can hold the states from the overloaded machine. That is, when the distribution manager observes that the difference between the maximal used memory and the least used memory reaches a certain threshold, then the state relocation is started to balance the memory usage among machines. This relocation aim to have as much as states kept in main memory. While the local adaptation controller observes that the memory usage of the machine is beyond a certain threshold, it then triggers the local state spill adaptation on that particular machine. Note that these two adaptations are not concurrent. This means only one adaptation will be processed at a time. In the state relocation process, we prefer to choose partitions that have not been pushed into disks to adapt. This helps to avoid unnecessary overhead and complexity in the cleanup stage.

The interactions of two adaptations in this lazy-disk approach are described by the pseudo code sketched in Figure 6.16. That is, the distribution manager starts the state relocation process when the $M_{least}/M_{max} < \theta_r$. While the state spill is triggered for each individual machine if $M_{used} > \theta_s$. Here, M_{used} denotes the main memory that has been used so far for that machine. The boolean *nodiskAdaptation* is set by the sender's machine in the state relocation process (when the sender receives the *ComputePartitionsToMove* message) to make sure no state spill adaptation will be processed in that machine when its local adapter is computing the partitions to be moved across machines. The boolean *indiskAdaptation* is set when the state spill adaptation starts. It will force the sender's machine in a state relocation process to wait until the state spill finishes. Note that the state spill adaptation usually finishes quickly. Thus the sender will not need to wait a long time before resuming the state relocation process. A time-out mechanism is also implemented in the system to make sure the state relocation process will not halt for a long time due to the state spill adaptations.

As can be seen, this lazy-disk approach focuses on the main memory usage only since both types of adaptations are driven purely by main memory usage. In the lazy-disk approach, we push the less productive partitions (with small P_{output}/P_{size} values) to disk in the state spill process, while we choose the productive partitions (with large P_{output}/P_{size} values) to move in the state relocation adaptation. Given that, productive partitions are likely to be kept in main memory that would result in a high throughput in the run-time phase.

The state spill in the above approach is a local decision. This means the decision is made by the query processor as the memory overflow happens



```
1: if ($M_{least}/M_{max} > \theta_r$) {
2:   distributedAdaptation(...);
3: }
```

```
1:  if (receiving ComputePartitionsToMove){
2:      wait until !indiskAdaptation;
3:      nodiskAdaptation = true;
6:      ComputePartitionsToMove(...) ;
7:      ...
8:      SendPartitions(...);
9:      nodiskAdaptation = false;
10: }
```

```
1: if ($M_{used} > \theta_s$) {
2:   if (!nodiskAdaptation) {
3:     indiskAdaptation = true;
4:     diskAdaptation(...);
5:     indiskAdaptation = false;
6:   }
7: }
```

Figure 6.16: Lazy-Disk Adaptation

at a local machine. However, the productivity of partitions among machines might not be the same. For example, the least productive partition in one machine, as the candidate to be pushed into disks, may still be much more productive than many other partitions in another machine. Thus, if we raise the state spill adaptation decision to a global level (to the distribution manager), we could instead globally choosing the least productive partitions among all machines to be pushed into disks. This should free more aggregated main memory space across the cluster for the productive partitions.

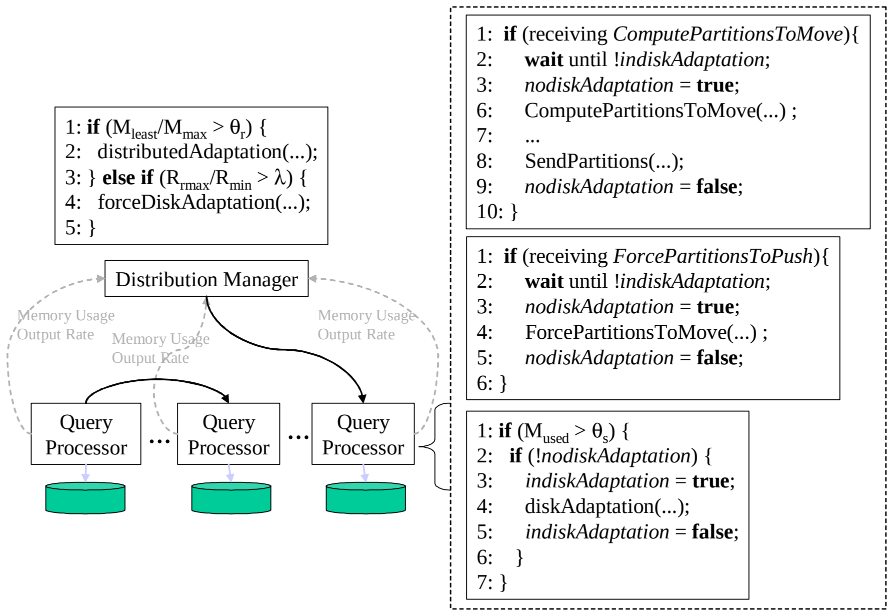
Corresponding to this idea, we now propose an *active-disk* approach which actively performs state spill adaptations. As illustrated in Figure 6.17, the distribution manager monitors both the main memory usage and the average productivity rate of machines in the cluster. Here, the *average productivity rate* of one machine is defined as the total number of tuples that have been generated from this machine during the sampling time divided by the number of partition groups in the machine.

As in the lazy-disk approach, if $M_{least}/M_{max} < \theta_r$, then the state relocation is triggered. If the memory usage across machines in the cluster is balanced, i.e., the $M_{least}/M_{max} \geq \theta_r$, then we compare the average productivity rate (R) of each machine. If one machine has a much lower average productivity rate, for example, $R_{max}/R_{min} > \lambda$, we then force the partitions of the lower average productivity rate machine to be pushed into disks. Given this, we would leave main memory space for the high productive partitions in other machines to be relocated into these machines. This would help the overall performance since higher productive partitions remain in main memory. Note that the state spill is triggered independently when $M_{used} > \theta_s$ as described in the lazy-disk approach.

However, pushing more states into disks than necessary could decrease the overall performance as well. In the active-disk strategy, we set the maximal amount of states being pushed by the distribution manager to be less than $M_{query} - M_{cluster}$, where M_{query} denotes the estimation of the overall main memory consumption for the query, while $M_{cluster}$ is the overall available main memory of the cluster.

6.3.2 Lazy-Disk and Active-Disk Evaluation

A lazy-disk adaptation approach has the potential to fully utilize all available main memory in the cluster. As we have shown in Section 6.2.2, the state relocation only causes little overhead on the query processing. Thus, this has a high chance of resulting in a high run-time throughput. Figure



```
if ($M_{least}/M_{max} > \theta_r$) {
  distributedAdaptation(...);
} else if ($R_{rmax}/R_{min} > \lambda$) {
  forceDiskAdaptation(...);
}
```

```
if (receiving ComputePartitionsToMove){
    wait until !indiskAdaptation;
    nodiskAdaptation = true;
    ComputePartitionsToMove(...) ;
    ...
    SendPartitions(...);
    nodiskAdaptation = false;
}
```

```
if (receiving ForcePartitionsToPush){
    wait until !indiskAdaptation;
    nodiskAdaptation = true;
    ForcePartitionsToMove(...) ;
    nodiskAdaptation = false;
}
```

```
if ($M_{used} > \theta_s$) {
  if (!nodiskAdaptation) {
    indiskAdaptation = true;
    diskAdaptation(...);
    indiskAdaptation = false;
  }
}
```

Figure 6.17: Active-Disk Adaptation Approach

6.18 shows the performance of the lazy-disk approach in a memory constraint environment when the memory of all machines is not enough for the query processing. The query is deployed on three machines. We set a skewed initial distribution with one machine being assigned 2/3 of all partitions, while another two machines share evenly the rest 1/3 of partitions. In this setup, if we do not apply state relocation, then only one machine gets overloaded. While the other two can process its partitions fully in main memory. We call this 'no-relocation' approach. Using the lazy-disk approach, all three machines will eventually get overloaded and trigger state spill processes.

Seen from Figure 6.18, the lazy-disk approach has a higher overall throughput than the 'no-relocation'. This is because the lazy-disk approach fully makes use of available main memory in the cluster during the query processing.

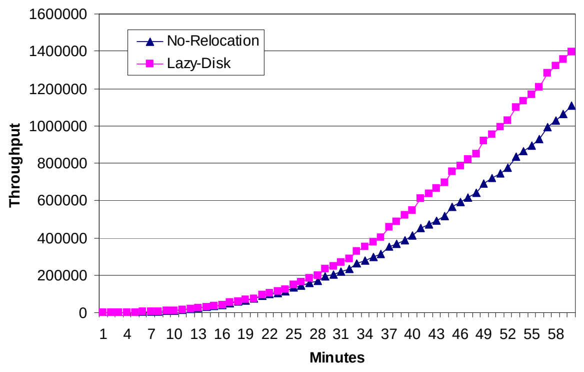


Figure 6.18: Lazy-Disk vs. No State Relocation

Even for an extremely heavy query workload where each machine in the cluster does not have sufficient memory to process the partitions assigned to them, a lazy-disk approach still has benefits. To illustrate, we again deploy the given query into three machines and have one machine get more partitions than the others. We run the query for 6 hours so that each machine has a large amount of states beyond its capacity, i.e., its available main memory. We again compare the performance of lazy-disk and no-relocation. In the experiment, the overall results generated in this 6 hour run by these two approaches are similar since they have similar amount of states being pushed into the disk.

However, the clean up process of these two approaches are dramatically different. The no-relocation approach takes more than 1600 seconds to produce 2,023,781 tuples in the clean up stage. This is because most of work is done by one machine. While the lazy-disk approach only takes less than 400 seconds to clean up. This is because the work is already evenly distributed among all three machines before cleanup starts.

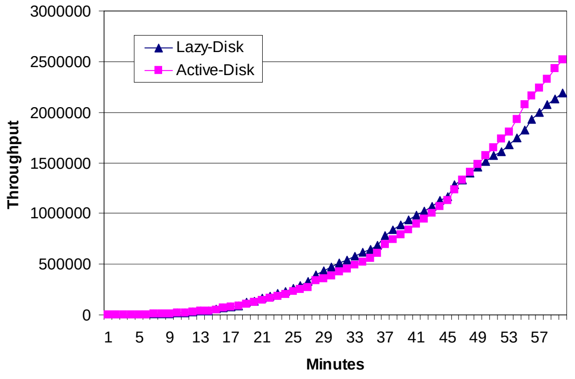


Figure 6.19: Lazy-Disk vs. Active-Disk: Comparison One

We now illustrate that the active-disk approach could further improve run-time query throughput if the distribution manager observes major differences of productivity among machines while the memory usage is balanced. Figure 6.19 shows one comparison of these two approaches. In this experiment, we set the tuple range of the input stream to 30K. We set the partitions assigned to machine m_1 to have a high average join rate of 4, while partitions in the other two machines have a low average join rate of 1. The lazy-disk approach does nothing at the distribution manager level if the memory usage among the three machines is balanced. While the active-disk approach forces lower productive partitions to be pushed into disks since the average productivity of partitions in machine m_1 is much larger than that of the other two. Note that in both approaches, each machine triggers the state spill process as its memory usage reaches its threshold

($\theta_s \geq 60MB$). Here, the state relocation threshold θ_r is set to 0.8, while the minimal time span of two relocations τ_m is set to 45 seconds. The productivity threshold λ that triggers a 'force state spill adaptation' is set to 2. Seen from Figure 6.19, the active disk strategy experiences a slight drop in the throughput after it starts pushing partitions into disks. However, the active-disk strategy outperforms the lazy-disk gradually since more high productive partitions remain in main memory.

We need to control the total amount of states that are pushed by the distribution manager. This is because too many pushes than necessary could slow down the overall performance. In the above experiment, we set the total amount of states being pushed by the distribution manager to less than 100 MB.

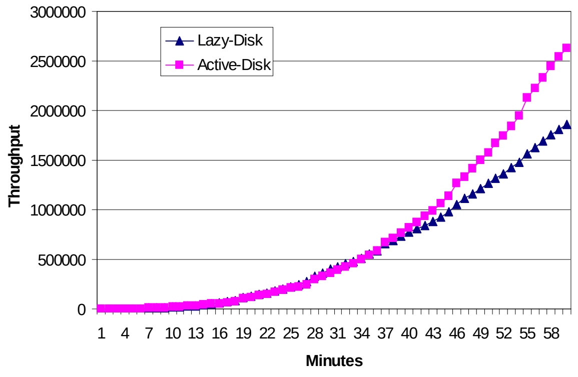


Figure 6.20: Lazy-Disk vs. Active-Disk: Comparison Two

As the difference of average productivity of different machines increases, then the active-disk approach can further improve the run-time query throughput compared to the lazy-disk approach. We set partitions assigned to machine m_1 (with a join rate 4) to have a small tuple range (15K), while set the partitions assigned to the other two machines (with a join rate 1) to have a large tuple range (45K). This setup further differentiates the average productivity values of machines. Having a smaller tuple range indicates a larger join factor value given the same number of input tuples. It thus further increases the number of output tuples. As expected, the active-disk approach has a major throughput improvement compared with that of the lazy-disk approach (see Figure 6.20).

Chapter 7

Spilling States of Pipelined Query Trees

In Chapter 6, we have focused on adapting states of queries with one single state intensive multiple input operator to address the run-time main memory shortage problem. However, in a data integration context, queries with multiple state intensive operators are common. In this chapter, we thus investigate how to spill operator states (partition groups) from pipelined state intensive operators in the query tree.

Given multiple state intensive operators, interdependency among different operators cannot be avoided. For example, as shown in Figure 7.1, two state intensive operators OP_i and OP_j with the output of OP_i directly being pipelined as input stream into OP_j. Now, if we apply state spill strategies for one single operator, i.e, with the adaptation decision aimed to maximize output streams generated by OP_i when spilling states from

OP_i. Now this would in turn increase the main memory consumption of OP_j. This is because the operator states of OP_j are directly dependent on the output of OP_i. On the other hand, these states in OP_j may not necessarily contribute to the final output of the query. For example, OP_j may have a rather low selectivity, thus only very few results tuples will be generated from OP_j.

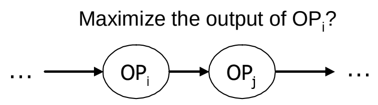


Figure 7.1: A Chain of Stateful Operators

As we have discussed in Chapter 6, we also aim for maximal run-time phase throughput when the main memory of the system is not enough for the query processing. Again, we use symmetric m-way hash join [111] as the example of state intensive operators.

7.1 Global Throughput-Oriented State Spill

We first define the operator state size and the state size of the query tree since different operators in the query tree can have different contributions to the overall state size. The size of the operator state can be estimated based on the average size of each tuple and the total number of tuples in the operator. The total state size of the query tree is defined as the sum of all the operator sizes. For example, the state size of $Join_1$ (see Figure 7.2) can be estimated by $S_1 = u_a * s_a + u_b * s_b + u_c * s_c$. Here, s_a, s_b, and s_c denote the number of tuples have been stored in $Join_1$ from input streams

A, B and C respectively. While u_a, u_b, and u_c represent the average sizes of each input tuple from the corresponding input streams.

In Figure 7.2, I_1 and I_2 denote the intermediate results from $Join_1$ and $Join_2$ respectively. Note that the average size of each tuple in I_1 can be represented by u_a + u_b + u_c, while the average size of one tuple in I_2 can be denoted by u_a + u_b + u_c + u_d. Here, we assume no projection is applied in the query plan. However, this simple model can be naturally extended to situations when projection does exist.

The size of operator states to be pushed during the spill process can be computed in a similar manner. For example, assume d_a tuples from A, d_b tuples from B, and d_c tuples from C are to be pushed. Then, the pushed state size can be represented by $D_1 = u_a * d_a + u_b * d_b + u_c * d_c$.

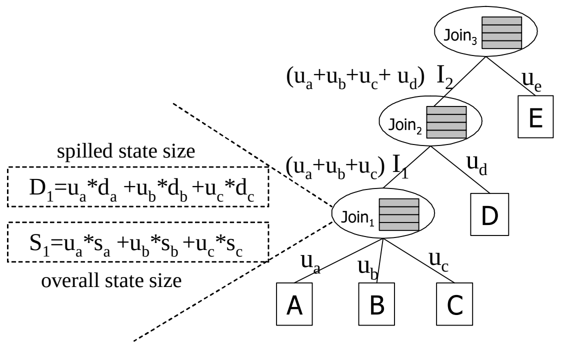


Figure 7.2: Operator/Query Tree State Size

Thus, the percentage of states been pushed for the query tree can be computed by the sum of state size being pushed divided by the current total main memory resident state size. For the query tree depicted in Figure 7.2, it is denoted by (D_1 + D_2 + D_3)/(S_1 + S_2 + S_3). Here S_i represents the current total state size of operator $Join_i$, while D_i denotes the operator states being pushed from $Join_i$ ($1 \leq i \leq 3$).

7.1.1 Choosing Candidate Partitions to Spill

Given multiple stateful operators in the query tree, partition groups from different operators can be considered as the candidates to be pushed when main memory overflows. Again, the question is how to spill the right partition groups in order to have the least effect on the overall run-time query throughput.

In Section 6.1, we have investigated different strategies on how to push partition groups from one single operator into disks in order to have the least affect on the overall run-time phase throughput. While given multiple stateful operators in the query tree, we have to further figure out which operator(s) and how many partition groups from each operator need to be pushed. As we will discuss shortly, a direct extension of our strategies described in Section 6.1.3 does not perform well in the multiple stateful operator situation. We instead propose various strategies for how to choose partition groups from multiple stateful operators. As discussed in Chapter 6, we will continue to push k% of operator states in each adaptation.

Let us first investigate the impact of pushing operator states in a chain of operators. Figure 7.3 illustrates an example chain of operators: OP_1, OP_2, $\ldots$, and OP_n. Here OP_i ($1 \leq i \leq n$) in the chain can be viewed as an abstract stateful operator in the query tree, it does not have to be a single input operator. While s_i represents the selectivities of operator OP_i respectively.

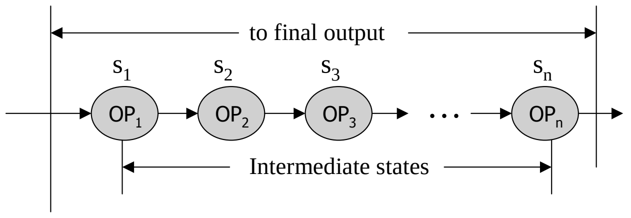


Figure 7.3: An Operator Chain

For such an operator chain, Equation 7.1 estimates the possible number of output tuples (u) from OP_n given a set of t input tuples to OP_1.

$$u = \prod_{i=1}^{n} s_i * t \tag{7.1}$$

The total number of tuples that will be stored in the chain due to these t tuples, that is, the indicator of the increase on the operator state size, can be computed as follows [1].

$$I = \sum_{i=1}^{n} [(\prod_{j=1}^{i-1} s_j * t)] \tag{7.2}$$

More precisely, OP_1 stores t tuples, OP_2 stores $t * s_1$ tuples, OP_3 stores $t * s_1 * s_2$ tuples, and so on. Thus, if we were to drop all t tuples at OP_1, then all the corresponding intermediate results due to these t tuples that have to be stored in OP_2, OP_3, …, OP_n on the other hand now would not appear any more. Note that dropping any of these intermediate results would also have the same overall effect on the final output, i.e., dropping the $t * s_1$

[1]Note that we assume that all the input tuples to the stateful operators have to be stored in the operator as operator states, i.e., join operators. In principle, other stateful operators such as those impose a window constraint for state purging can be addressed in a similar manner.

tuples at OP_2 has the same overall effect on the final output as estimated by the Equation 7.1.

Bottom-up Pushing. Inspired by the above analysis, we propose one naive solution, referred to as *bottom-up pushing*, to spill operator states of a query tree with multiple stateful operators. That is, we always choose operator states from the bottom operator(s) in the query tree until the selected states reach k% compared to the overall operator state size. Here, the bottom operator is defined as the stateful operators having the highest height in the query tree assuming the root operator has a height of zero, i.e., OP_1 in Figure 7.3. Partition groups from the bottom operator are chosen randomly, or alternatively, we can employ a more sophisticated strategy based on certain statistics such as the throughput-oriented strategy discussed in Section 6.1.3. Note that for a query tree having multiple bottom operators, we randomly choose one of them to start. We only spill states of their parent operators until states from all bottom operators do not fill up the k%.

As can be seen, if we spill partition groups of the bottom operator, we would have less intermediate results stored in the query tree compared with pushing states in the other operators. Thus, the bottom-up pushing strategy has the potential of requiring a smaller number of state spill processes while still achieving the same reduction in memory space usage. This is because less states (intermediate results) are expected to be accumulated during the query processing.

However, having a smaller number of state spill processes does not naturally result in a high overall throughput. This is because (1) the partitions being pushed in the bottom operator can be the parent partitions in its downstream operators. While these downstream partitions may experience a high output rate. (2) the cost of each state spill process may not be high, thus instead opting for a large number of state spill processes may not incur any significant overhead on the query processing. That is, which partitions to be spilled may be more important than how many time of state spill processes in terms of the effect on run-time query throughput.

Given a partition based query tree, the output of a particular partition of any operator (in particular a bottom operator) is likely to be sent into multiple different partitions of its downstream operator(s). For example, as illustrated in Figure 7.4, assume *t* input tuples into OP_1 are partitioned to become member of the partition group P_1^1. Here the superscript represents the operator ID, while the subscript denotes the partition ID. After the processing in OP_1, $(t_1^1 + t_2^1)$ result tuples are output to OP_2. Of those, t_1^1 tuples are partitioned to P_1^2 of OP_2, while t_2^1 tuples are partitioned to P_2^2 of OP_2. Now P_1^2 and P_2^2 of OP_2 may have rather different selectivities. For example, the number of output t_2^2 from P_2^2 may be much larger than the output t_1^2 from P_1^2, while the size of these two partitions may be similar. Thus, it may be worth while to keep P_1^1 in OP_1 even though certain states (in P_1^2 of OP_2) will be accumulated as a side-effect of keeping P_1^1.

As can be seen, the relationship between partitions among adjacent operators is a many to many relationship. Thus, pushing partition groups at the bottom operator may affect multiple partition groups at any of its down stream operators. From above, we can see that this naive bottom-up pushing strategy does not have a as clear connection to the overall throughput as one may assume at first.

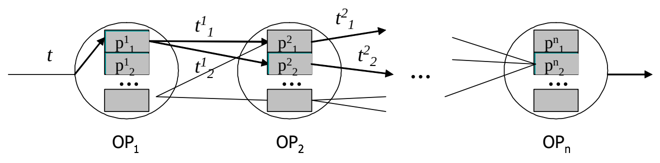


Figure 7.4: A Chain of Partitioned Operators

To design a better state spilling strategy, we need to globally select partition groups in the query tree as candidates to be pushed. Figure 7.5 illustrates the basic idea of this type of approach. That is, instead of pushing partition groups from particular operator(s) only, we conceptually view partition groups from different operators at the same level. While we choose partition groups among all operators based on the cost statistics collected about each partition.

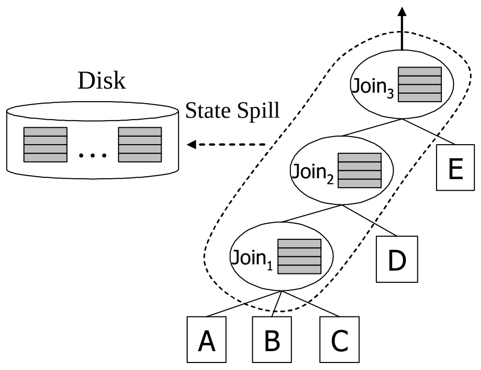


Figure 7.5: Globally Choose Partition Groups

The basic statistics we collect for each partition group are P_{output} and P_{size}. P_{output} indicates the total number of tuples that have been output from the partition group, while P_{size} refers to the operator state size of the partition group. These two values together can be utilized to identify the productivity of the partition group. We now describe three different strategies on how to collect P_{output} and P_{size} values of each partition group, and how partition groups can be chosen based on these values with less impact on the run time throughput.

Local Output. The first strategy, referred to as *local output*, treats each operator individually when updating its statistics such as P_{output} and P_{size} values of each partition group. This strategy actually is inspired by existing strategies as described in Section 6.1.3. P_{size} of each partition group is updated whenever the input tuples are inserted into the partition group. While P_{output} value is updated whenever output tuples are generated from the operator.

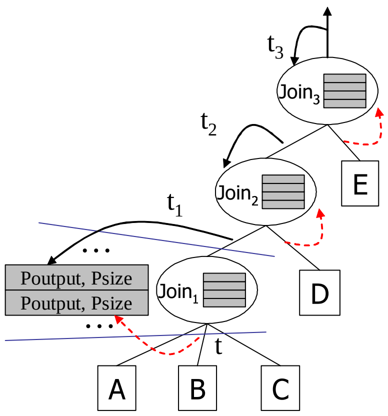


Figure 7.6: A Localized Statistics Approach

Figure 7.6 illustrates this localized approach. For example, *t* tuples input into $Join_1$, we then update P_{size} of the corresponding partition groups in $Join_1$. When t_1 tuples are generated from $Join_1$, then P_{output} value of the corresponding partition groups in $Join_1$ and the P_{size} value of related paritition groups in $Join_2$ are updated. Similarly, if we get t_2 from $Join_2$, then P_{output} of the corresponding partition groups in $Join_2$ and P_{size} in $Join_3$ are updated.

The selection of partition groups to be pushed to disk is based on the *productivity value* (P_{output}/P_{size}) of each partition group. In this local output strategy, we select the partition group with the smallest productivity value among all partition groups in the query as candidates to be pushed.

However, this approach does not provide a global productivity view of the partition groups. For example, if we leave partition groups of $Join_1$ in main memory that exhibit high productivity values, then this in turn would contribute to generating more output tuples to be sent as inputs to $Join_2$. All these outputs will be stored in $Join_2$. It thus increases the main memory consumption of $Join_2$. This may cause the main memory to be filled up quickly. Note that these intermediate results may not necessarily help the overall throughput since these results may be dropped by any one of its down-stream operators if it happens to have a low selectivity.

Global Output. In order to maximize the run-time throughput after pushing states into disks, we need to have a global view of partition groups that reflects how each partition group contributes to the final output of the query. That is, the productivity value of each partition group needs to be defined in terms of the whole query tree (not just its local effect).

This requires us to have the P_{output} value of each partition group correspond to the number of final output tuples generated from the query. In this case, the productivity value, P_{output}/P_{size}, denotes how 'good' the partition group is in terms of contributing to the final output of the query. Thus, if we have the partition groups with high global productivity value in main memory, the overall throughput of the query tree is likely to be high compared with other pushing strategies.

To achieve this, we design a *tracing algorithm* to update the P_{output} value of each partition group. The basic idea is whenever output tuples are generated from the query tree, we then figure out the lineage of each output tuple. That is, we trace back to the respective partition groups of different operators that contributed to the output. For join operators, we note that the tracing back to the contributing partition groups that contributed to the output can be computed by reapplying the corresponding split operators. Here we assume the output tuple contains at least all join columns along the query tree. Thus, applying corresponding split functions on each output tuple (on the corresponding join column value) exactly re-produces the lineage of partition groups that contributed to the output. Note that the update of the P_{size} value remains the same as we have discussed in the local output approach.

Note that for other operators that do not have such partition information automatically embedded in each output tuple, we may have to encode such lineage information into the output tuple. In this case, techniques such as discussed in [30] can be applied.

For example, as shown in Figure 7.7, if *10* tuples are generated from partition group 2 (P_2^3) of $Join_3$, we directly update the P_{output} values of P_2^3 by $P_{output} \leftarrow P_{output} + 10$. To find out the partition groups in the $Join_2$ that contribute to the outputs, we then apply the partition function of $Split_2$ on each output tuple. Note that multiple partition groups in the $Join_2$ may contribute to even to the same partition group in $Join_3$. In this example, partition groups with ID 1 (P_1^2) has output 6 tuples, partition group P_4^2 has sent 2 tuples, while P_6^2 have sent 2 tuples to P_2^3 repectively. We thus update the P_{output} values for partition groups with IDs 1, 4 and 6 in $Join_2$ based on the number of output tuples they have sent to $Join_3$. Similarly, we apply the partition function of $Split_1$ to find the corresponding partition groups in operator $Join_1$ and update their P_{output} values.

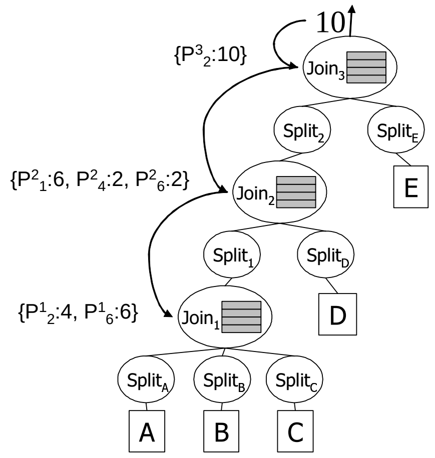


Figure 7.7: Tracing and Updating the P_{output}

Such tracing and updating may incur a certain overhead on the query processing. We thus do not have to trace and update P_{output} values for each output tuple, we can only update the value with a certain probability, say, 10% of the output tuples using some random sampling method.

The high level picture of this tracing algorithm is sketched in Algorithm 6. Here, we assume in each stateful operator in the query tree has references to its immediate upstream stateful operator and its immediate split operator. Note that for a query tree, then multiple pairs of immediate stateful operator references and its immediate split operator references may exist. We thus can similarly use a breadth-first/depth-first query tree search algorithms to update the P_{output} values of corresponding partition groups.

Algorithm 6 updateStatistics(tpSet)

```
/*Tracing and updating the $P_{output}$ values for a given set of output tuples $tpSet$.*/
1: prv_join_ref ← this.getUpStreamJoinReference();
2: prv_split_ref ← this.getUpStreamSplitReference();
3: while ((prv_join_ref $\neq$ null) && (prv_split_ref $\neq$ null)) do
4:   for each $tp \in tpSet$ do
5:     $cPID$ ← Compute partitionID of $tp$ in prv_join_ref;
6:     Update $P_{output}$ of partition group with ID $cPID$;
7:   end for
8:   prv_split_ref ← prv_split_ref.getUpStreamSplitReference();
9:   prv_join_ref ← prv_join_ref.getUpStreamJoinReference();
10: end while
```

Algorithm 6 is activated when output tuples of the query tree have been generated, i.e., from $Join_3$ as shown in Figure 7.7.

Given the above tracing, the P_{output} value of each partition group indicates the total number of outputs that have been generated that had some part of the output tuple come from this partition group. Thus, P_{output}/P_{size} indicates the *global productivity* of the partition group. By pushing partition groups with a lower global productivity, we expect that the overall run-time phase throughput would be better optimized than when using the localized approach or the bottom-up approach.

Global Output with Penalty. In the above approaches, the size of the partition group P_{size} only reflects the main memory usage of the current partition group. However, the operators in a query tree are not independent. That is, output tuples of an upstream operator have to be stored in its downstream stateful operators. This indirectly affects the P_{size} of the corresponding downstream operator partition groups.

For example, as shown in Figure 7.8, both partition groups P_1^1 and P_2^1 of OP_1 have the same P_{size} and P_{output} values. Thus, these two partitions would have been assigned the same productivity value given the global output approach. However, P_1^1 produces 2 tuples on average (have to be stored in OP_2) given one input tuple. While one tuple input to P_2^1 generates 20 tuples on average and stores in OP_2. Given that all such intermediate results have to be stored in the downstream stateful operators, pushing P_2^1 instead of P_1^1 can help to reduce the storage requirement demanded by intermediate results. This in turn reduces the number of state spill processes required overall.

To capture this, we now define an intermediate result factor in each partition group P_{inter}. This factor indicates the possible intermediate results that will need to be stored in its downstream operators in the query tree. This intermediate result storage factor can be computed similarly as the tracing of the final output. That is, if a final output is produced from the query tree, we update the P_{output} value of the corresponding partition

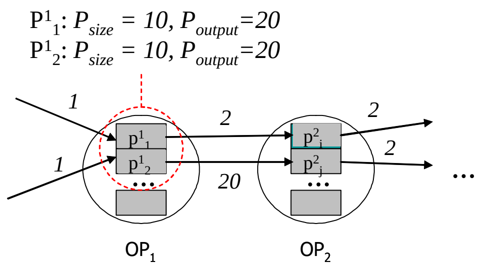


Figure 7.8: The Impact of the Intermediate Results

groups in the operators. While for all the intermediate results generated, we update the P_{inter} values of the upstream operators. We then define the productivity value of each partition group as $P_{output}/(P_{size} + P_{inter})$ [2].

Figure 7.9 illustrates how tracing algorithm can be utilized to record the intermeidate result strorage factor P_{inter}. For example, one input tuple to OP_1 eventually generates 2 output tuples from OP_4. For simplicity, we assume all these tuples are partitioned to the partition group 1 of each operator. Here, the number indicated in the square box represent the number of intermediate results tuples generated along the processing of this input tuple. Thus, once the 2 output tuples are produced by partition group P_1^1, the tracing algorithm updates the $P_{inter} \leftarrow P_{inter} + 2$ for P_1^1. After the 3 tuples are produced from P_1^2 (in OP_2), the tracing algorithm updates the P_{inter} ($P_{inter} \leftarrow P_{inter} + 3$) for both P_1^1 and P_1^2. Similarly, once the 4 tuples are produced from P_1^3, P_{inter} values of P_1^1, P_1^2 and P_1^3 are updated. Thus, P_{inter} of P_1^1 increases by 9, P_{inter} of P_1^2 increases by 7 tuples, while P_{inter} of P_1^3 increases by 4. As can be seen, P_{inter} value of an operator indicates the

[2]Variations exist on how to define the productivity, i.e., to emphasize the P_{inter} value. Again, this can be investigated in the future work.

number of intermediate result tuples need to be stored in its downstream operators.

While for the final 2 output tuples, the tracing algorithm updates the P_{output} values of partition group 1 in all operators.

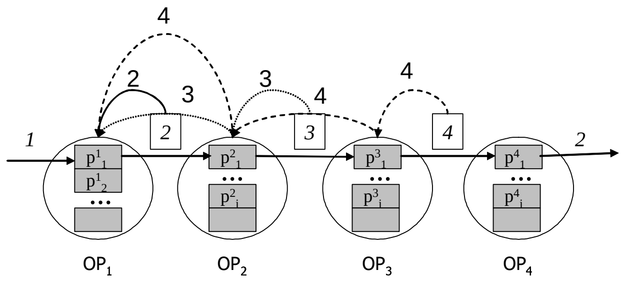


Figure 7.9: Tracing and Updating P_{inter} Values

Only little change is required in Algorithm 6 to to support this tracing of both intermediate results and final output tuples. Now we update P_{output} value if the $tpSet$ is the set of final output tuples of the query, while update P_{inter} otherwise.

7.1.2 Clean Up Multiple Stateful Operators

Given a query tree with multiple stateful operators, when operator states from any of the stateful operators have been pushed into the disk at the run-time phase, then the cleanup stage cannot be performed in a random order. This is because the operator has to incorporate the missing results generated from the cleanup process of its up stream operator. That is, the cleanup process of join operators has to conform to the partial order as defined in the query tree.

Figure 7.10 illustrates a 5-join query tree ((A $\bowtie$ B $\bowtie$ C) $\bowtie$ D) $\bowtie$ E with three joins being denoted as $Join_1$, $Join_2$, and $Join_3$ respectively. Assume we have operator states pushed into the disk from all three operators. The corresponding join results from these disk resident states are denoted by ΔI_1, ΔI_2, and ΔI_3. From Figure 7.10, we can see that the cleanup results of the $Join_1$ (ΔI_1) have to be joined with the complete operator states of D to produce the cleanup result for $Join_2$. Here, the complete states D includes states from the disk resident part ΔI_2 and the corresponding main memory operator states. While the cleanup result of $Join_2$, (ΔI_2 + $\Delta I_1 \bowtie D$), has to join with the complete operator states from E to produce the missing results.

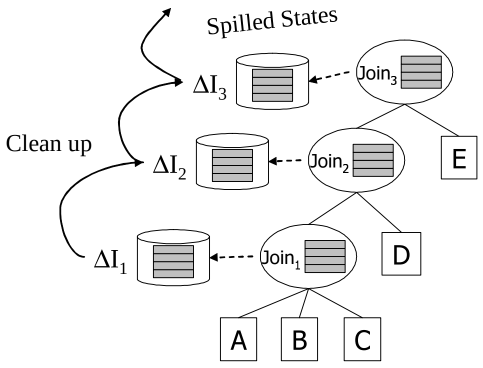


Figure 7.10: Clean Up the Operator Tree

Given this constraint, we design a *synchronized cleanup process* to combine disk resident states and to produce missing results. That is, we order the cleanup process based on the height in the query tree. We first cleanup the operator(s) with the largest height.

The clean up process for a particular operator is the same as we discussed in Section 6.1.2. Note that the cleanup process for operators with the same height can be processed concurrently. Once the up stream operator completes its cleanup process, it notifies its down stream operator using a control message to indicate no more input tuples will come. Then the cleanup process of the down stream operator can be started. Note that all the other operators (the stateful operators that have not been cleaned and other stateless operators such as split) keep on running as usual. Here, the results of the current cleanup process will continue feeding the down stream operators as during the normal run-time query processing. Once the cleanup process of the operator has been completed, the operator then will no longer be scheduled.

Given the example illustrated in Figure 7.10, we first start the cleanup process of $Join_1$. Note that in the cleanup process, no more input tuples will come from the input streams such as A, B and C. The missing results generated from the cleanup process of $Join_1$ will be immediately sent to the down stream operators. Once the cleanup process of $Join_1$ completes, i.e., ΔI_1 has been generated. Then, the $Join_1$ generates a special control tuple '*End-of-Cleanup*' to indicate the end of the cleanup process. After the down stream stateful operator, $Join_2$ in this example, receives the tuple, it starts its cleanup process. Note that all the other non-stateful operators between these two stateful operators, such as split operators, will simply pass the '*End-of-Cleanup*' tuple to the down stream operator. This process continues until all cleanup processes have been processed.

Note that it is possible to start the cleanup process of all stateful operators at the same time. However, this may require a large amount of main memory space since each cleanup process will bring disk resident states into the memory. On the other hand, the operator states of the down stream operators cannot be released until its up stream operators finish their cleanup and compute the missing results. While for the synchronized method, we bring these disk resident states sequentially and discard them once the cleanup process of this operator completes.

7.2 Partitioning Query Trees

The approach of partitioning input streams, that is, operator states, helps to achieve a partitioned parallel query processing [21, 68, 95]. This is because we can simply spread the partitions into different machines with each machine thus only processing a portion of all inputs. This is useful given our focus on queries with multiple state intensive operators that are resource demanding in nature. In this section, we now extend our global state spill strategies to also work for partitioned parallel query processing environments.

The following two issues first need to be solved for supporting a partitioned parallel query processing: (1) allocation of stateful operators to available machines, and (2) composition of partitioned query plan that run on multiple machines.

Allocating Multiple Stateful Operators. The allocation of stateful operators refers to the distribution of stateful operators to available machines. In this work, we choose to allocate all stateful operators in the query tree to all the machines in the cluster, as shown in Figure 7.11(b). Thus, each machine will have exactly the same number of stateful operators defined in the query tree activated. Each machine processes a partition of all input streams of the stateful operators.

Note that we only focus on the complex stateful operators in the allocation since these operators have the potential of requiring partitioned processing. While the allocation of non-stateful operators in the query tree is simple, i.e., a round robin approach that aims to distributed these operators evenly to the available machines.

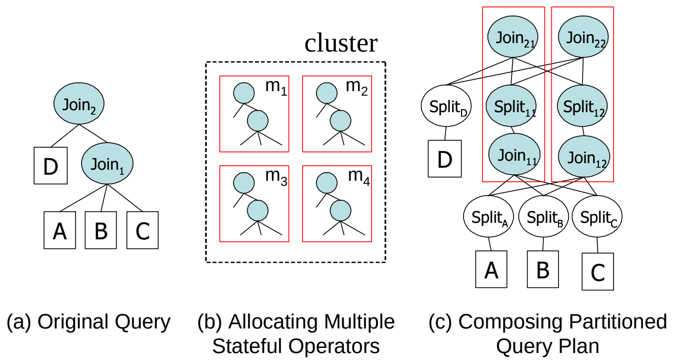


Figure 7.11: Partitioned Parallel Processing of Query Trees

Composing Partitioned Query Tree. The composition of the partitioned query plan focuses on how to connect partitioned stateful operators. It needs to be addressed after the distribution of stateful operators as discussed above has been completed.

Here we use the split per instance approach illustrated in Figure 7.11(c). We directly insert one split operator after each instance of the stateful operator. Both the split operator and the operator instance are activated in the same machine. Thus, the output of the operator instance is directly partitioned and then shipped to the appropriate down stream operators.

Note that other approaches exist for both allocating stateful operators and composing partitioned query plan. However, the main focus of the work is to adapt operator states in order to address run time main memory shortage problem. Thus, the exploration of other partitioned parallel processing approaches as well as their performance are beyond the scope of this dissertation work.

State Spilling in Partitioned Parallel Environments. The global throughput oriented state spilling strategies discussed in Section 7.1.1 naturally apply to the partitioned parallel processing environments. This is because the cost statistics we collected are purely based on main memory usage and operator states only. It is not sensitive to the distribution of the query plans.

However, given partitioned parallel processing, the update of the P_{output} value can be across different machines. For example, as shown in Figure 7.12, the query plan is deployed in two machines. If *k* tuples are generated from $Join_3$, we directly update the P_{output} values of partition groups in $Join_3$ that produces these outputs. To find out the partition groups in $Join_2$ that contribute to the outputs, we then apply the partition function of $Split_2$ on each output tuple. Note that given partitioned parallel processing, partition groups from different machines may contribute to the same partition group of the down stream operator. Thus, the tracing and updating of P_{output} values may involve multiple machines. In this work, we design a *UpdatePartitionStatistics* message to notify other machines the update of P_{inter} and P_{output} values. Since each split operator knows exactly the mapping between the partition groups and the machines having these partitions, it is feasiable to only send the message to the machine having the partition groups to be updated.

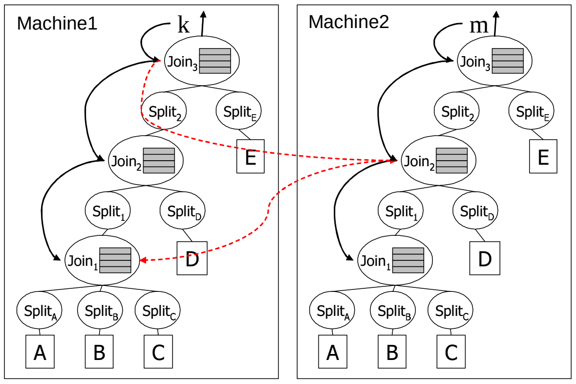


Figure 7.12: Tracing the Number of Output

The revised updateStatistics algorithm is sketched in Algorithm 7. We classify partition group IDs by applying the current split function into *localIDs* and *remoteIDs* depending on whether the ID is mapped to the current machine. Then for the partition groups with *localIDs*, we update either P_{inter} or P_{output} based on whether the current $tpSet$ is a set of intermediate results. While for the *remoteIDs*, we compose *UpdatePartitionStatistics* messages with appropriate information and then send the messages to the machine with the partition groups having IDs in the *remoteIDs*.

Algorithm 7 updateStatisticsRev(tpSet,intermediate)

*/*Tracing and updating the P_{output}/P_{inter} values for a given set of output tuples $tpSet$. intermediate is a boolean values indicate whether the tpSet is the intermediate results of the query tree*/*

```
1: prv_join_ref ← this.getUpStreamJoinReference();
2: prv_split_ref ← this.getUpStreamSplitReference();
3: while ((prv_join_ref $\neq$ null) && (prv_split_ref $\neq$ null)) do
4:    for each $tp \in tpSet$ do
5:       $cPID$ ← Compute partitionID of $tp$ in prv_join_ref;
6:       Classify $cPID$ into $localIDs/remoteIDs$;
7:    end for
8:    if (intermediate) then
9:       Update $P_{inter}$ of $localIDs$;
10:   else
11:      Update $P_{output}$ of $localIDs$;
12:   end if
13:   Compose & send UpdatePartitionStatistics msg(s) for $remoteIDs$;
14:   prv_split_ref ← prv_split_ref.getUpStreamSplitReference();
15:   prv_join_ref ← prv_join_ref.getUpStreamJoinReference();
16: end while
```

7.3 Global State Spill Evaluation

The above state spill strategies for query trees with multiple stateful operators have been implemented in the D-Cape system. The performance studies are conducted on the 10-machine cluster we already describe in Section 11.6. We use a five-join query tree illustrated in Figure 7.12 as an example to report our experimental results. The query is defined on 5 input streams denoted as A, B, C, D, and E with each input stream having two columns. Here $Join_1$ is defined on the first column of each input stream A, B, and C. $Join_2$ is defined on the first join column of input D and the second join column of input C, while $Join_3$ is defined on the first column of input E and the second column of input D. Note that other types of query plans have also been used in the experimental studies. They all result in similar results as will be reported here.

We deploy the query on two machines with each machine processing about half of all input partitions. All input streams are partitioned into 300 partitions. We set the memory threshold to 60 MB, which means the system starts spilling operator states into the disk when the memory usage of the system is over 60 MB. In each state spill process, we push 30% of all states into disks. We vary the join rate of the join operators in the query tree. The average tuple inter arrival time is set to 50 ms for each input stream.

Figure 7.13 compares the run-time phase throughput of different state spilling strategies. Here we set the average join rate of $Join_1$ 3, while the average join rates of $Join_2$ and $Join_3$ are both 1. In Figure 7.13, the X-axis represents the minutes have been run up to the point, while the Y-axis denotes the overall run time throughput.

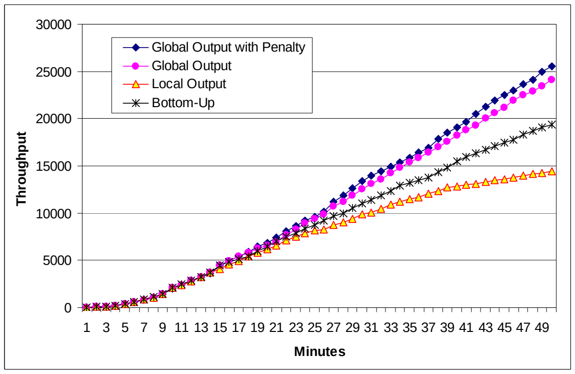


Figure 7.13: Comparing the Run-time Throughput

From Figure 7.13, we can see that both the *local output* approach and the *bottom-up* approach perform much worse than the *global output* and the *global output with penalty* approaches. This is as expected because the *local output* and the *bottom-up* approaches do not consider the productivity of partition groups at a global level. From Figure 7.13, we also see that the *global output with penalty* approach performs even better than the *global output* approach. This is because the global output with penalty approach is able to efficiently use the main memory resource by considering both the partition group size as well as the possible intermediate results that have to be stored in the query tree.

Figures 7.14 and 7.15 show the the corresponding memory usage when applying different spilling strategies. Figure 7.14 shows the memory usage of the *global output* approach and *global output with penalty* approach. Note that each 'zig' in the lines denotes one state spill process. From Figure 7.14, we can see that the *global output* approach has a total of 13 state spill processes in the 50 minutes running. While the *global output with penalty* approach only has a total of 10 times of spills. Again, this is as expected since the *global output with penalty* approach considers both the size of the partition group and the overall memory impact on the query tree in each adaptation.

As discussed in Section 7.1.1, having a smaller number of state spill processes does not imply a high overall run time phase throughput. From Figure 7.15, we can see that the *bottom-up* approach only has 7 adaptations. However, the run time phase throughput of the *bottom-up* approach is much less than the *global output with penalty* approach as seen from Figure 7.13. This is because having high productive partition groups in main memory helps to the overall throughput.

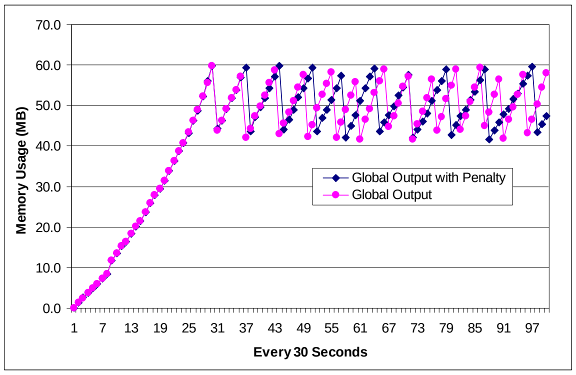


Figure 7.14: Global Output vs. Global Output with Pentalty

In Figures 7.16 and 7.17, we show the run time phase throughput if we vary join rates of the operators. As can be seen, they exhibit the results similar to what we have shown in Figure 7.13. In Figure 7.16, we set the join rate of $Join_1$ 1, while the join rates of $Join_2$ and $Join_3$ 3. In Figure 7.17, we set the join rate of $Join_1$ to 3, Join rate of $Join_2$ to 2, and the join rate of $Join_3$ to 3. From both figures, we can see that the *global output with penalty* approach alway outperforms other state spill strategies, while both the *bottom-up* and the *local output* approaches are much worse than the two global approaches.

Note that the run-time throughput of *bottom-up* approach and the *local output* approach are not always consistent. It is not that one approach is always better than the other. Figure 7.16 shows that the *bottom-up* approach has a higher run-time throughput than that of the *local output* approach, while Figure 7.17 instead shows that the *local output* approach is better than the *bottom up* approach. This is because both approaches do not consider the productivity of partition groups at a global level.

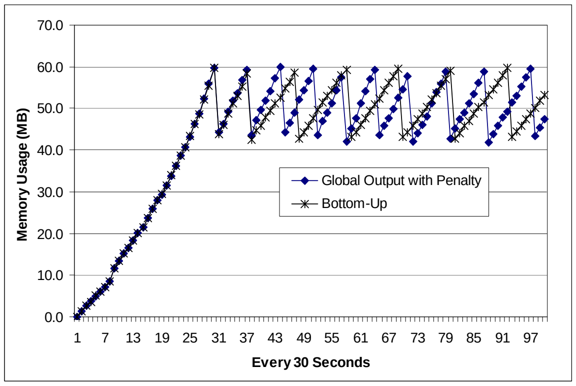


Figure 7.15: Global Output with Penalty vs. Bottom-up

The main memory usage of these two experiments also show a similar pattern as we illustrated in Figures 7.14 and 7.15. That is, the *global output with penalty* approach requires less number of adaptations compared with the *global output* approach. While the *bottom-up* approach requires even less number of adaptations than the *global output with penalty* approach. As can be seen, less number of adaptations does not imply a high run-time throughput.

The cleanup process time depends on where the operator states are pushed in the query tree. As we discussed in Section 7.1.2, the lower level partition groups are pushed (from operators with a higher height), the higher the clean up cost. This is because the clean up process needs to be sequentialized according to the partial order defined in the query tree.

The factor of cleanup process time has not been incorporated in the current global state spilling strategies. The cleanup processing times of these approaches vary depending on the queries and the settings. In experiment

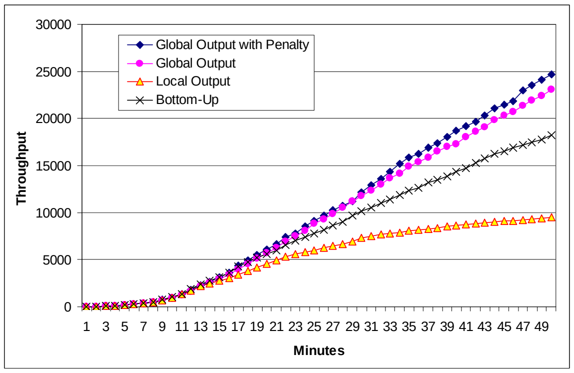


Figure 7.16: Run Time Throughput For Join Rates 1-3-3

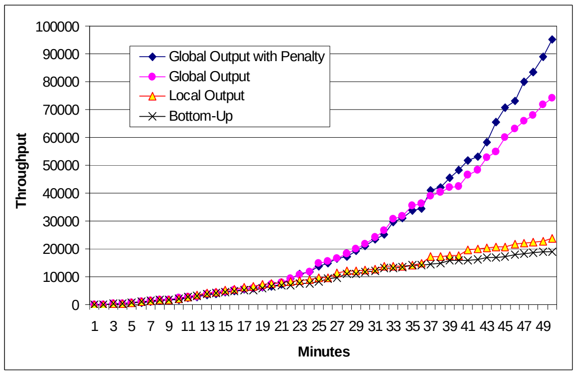


Figure 7.17: Run Time Throughput For Join Rates 3-2-3

shown in Figure 7.13, the total cleanup time of the *global output with penalty* approach takes *495,741* ms, while the cleanup time of the *global output* approach takes *305,997* ms. While in experiment shown in Figure 7.16, the cleanup time of the *global output with penalty* approach takes *278,234* ms, while the *global output* approach takes *362,752* ms. However, in all above experiments, the *bottom-up* approach takes much longer time to clean up disk resident states since this strategy tends to push partitions at the bottom operators.

The cleanup processing time can also be incorporated into the state spilling strategies if it is necessary. For example, we can assume a slightly higher weight for the partition groups in the upper level operators when calculating the productivity value (as part of the partition group size). Thus, it will promote the pushing of partition groups that require less cleanup process time. However, this in turn may impact the run time throughput since it indirectly influences the selection of the partition groups to be pushed. The evaluation of different weight or different productivity functions is not the main focus of this work. They are left as the feature work of this dissertation work.

Chapter 8

Related Work

Continuous query processing [3, 8, 15, 19, 76, 115] is closely related to our work in that it applies a push based non-blocking processing model. Continuous query processing also faces scalability concerns due to high rates of inputs and possibly infinite data streams. A lot of techniques with different research focuses have been investigated to address this problem. For example, load shedding techniques [3, 107] aim to drop input tuples to handle the run time resource shortage while having the query results within certain predefined QoS requirements. In this work, we instead require an accurate query results, thus load shedding is not an option in our context. Operator-state purging [34] relies on certain semantics of the input streams, e.g., puncations, to purge useless states. This is orthogonal to our current focus of the work since we only temporarily move states and do not focus on the semantics of the states. Adaptive scheduling and processing [9, 76] techniques have also been proposed. But they focus on adapting the order of operators or tuples being processed. While in this work, we instead focus on adapting the memory usage for complex stateful query operators with possible huge volumes of states. Note that this issue has not yet been carefully addressed in the continuous query processing literature.

Distributed continuous query processing over a shared nothing architecture, i.e., a computing cluster, has been investigated in the literature to address the resource shortage and the scalability concerns [2, 25, 105, 99]. In existing systems such as Aurora* [25] and Borealis [2], operators are assumed to be small enough to fit completely within one single machine. Thus, their main focus is how to distribute the query plan over multiple machines while treating each operator as one atomic unit. The adaptation in such systems [118] mainly focuses on balancing the load by moving query operators across machines. Thus, the basic unit to be adapted in the system is always at the granularity of one complete operator. D-Cape [105] also distributes and adapts continuous queries at an operator-level. While in this work, we instead investigate methods of adapting operator states to optimize the main memory usage.

Flux [99] is the first work in the literature to discuss the partitioned parallel processing and the distributed adaptation in a continuous query processing context. It makes use of the exchange architecture that was proposed by Volcano [41] by inserting split operators into the query plan to achieve partitioned processing for large stateful query operators. However, Flux mainly focuses on single input query operators. Given complex stateful query operators such as multiple-way join, more issues such as how to organize states from different input steams need to be addressed even for a pure distributed adaptation. Flux also discusses how to spill operator states into disks. However, it does not consider the state spill process at a global level. As we have discussed in Section 6.3.2, our proposed active disk strategy which makes state spill decisions across multiple machines, helps to further improve the overall run time throughput. Flux does not address how to adapt operator states for a full query tree with multiple stateful operators. Moreover, Flux tends to put all the adaptation logic and decisions to the split operator. This will make the coordination among different split operators complex when adapting multiple stateful operators. While in our architecture, we employ light split operators and leave the adaptation logic and decisions to separate modules in each query processor and in the distribution manager. This helps to achieve a better adaptation decisions especially in our target environment, i.e., local computer clusters.

State spill adaptation for non-blocking query operators has also been investigated in the literature. As discussed above, both XJoin [109] and Hash-Merge Join [79] adapt memory resident states from individual input streams to disks when memory overflow happens. As we have discussed in Chapter 6, this strategy does not work well for multiple input query operators, especially in a partitioned parallel processing environment. Moreover, these strategies are designed to work in a central environment. In an environment where both state spill and state relocation are necessary, again, issues such as how to integrate them need to be considered.

Parallel and distributed query processing has been the focus of both academia and industry for a long time [31, 35, 57]. Partitioned parallel processing, especially for complex operators such as joins, has also been studied in [21, 67, 95]. Correspondingly, data skew handling techniques [33] have been proposed. All these works provide the necessary background for our distributed non-blocking query processing and its forms of adaptations. However, they are typically studied under a traditional database processing model assuming static queries. Unique properties such as push-based processing (requires a non-blocking pipelined processing), little statistics about input data streams at query definition time (requires adaptation at run time) and long running or even infinite data streams (high demand on the system resources) differentiate this work from traditional distributed and parallel query processing.

Main memory allocation and management for distributed systems has also been extensively studied [14, 39, 85]. However, they usually focus on static resource allocations. While in this work, we instead focus on the run time adaptation. This is because little statistics about input streams are available initially. Moreover, the adaptation techniques proposed in this work consider both the state spill and state relocation.

Part III

Integration View Maintenance and Optimization

Chapter 9

Introduction and Background

As motivated in Chapter 1, materialized views need to be maintained upon source changes since a stale view extent may not help or even mislead user applications. Incremental view maintenance, which aims at only computing the deltas of the view result instead of recomputing the view from scratch upon data source changes, has been extensively studied in the past [5, 11, 18, 24, 27, 93, 120, 123, 122]. Among these works, the incremental maintenance of batches of updates [27, 63, 66, 93] is of particular interest because it is attractive from both a resource and a performance perspective to most practical systems/applications. The benefits are two fold. One, better overall maintenance performance can be achieved. Two, fewer conflicts of the maintenance tasks with users' read sessions upon the view extent may arise.

In an incremental view maintenance context, especially when the materialized view is defined upon distributed data sources, *maintenance queries* [122] need to be composed and processed to compute the view delta. Figure 9.1 illustrates the basic architecture of an incremental view maintenance framework. First, data sources report source updates to the materialized view manager. Then, the view manager composes maintenance queries and sends them to distributed data sources (or their corresponding wrappers if necessary) to compute the view delta change. Note that all maintenance queries are created by the view manager and the query results will also be returned to the view manager.

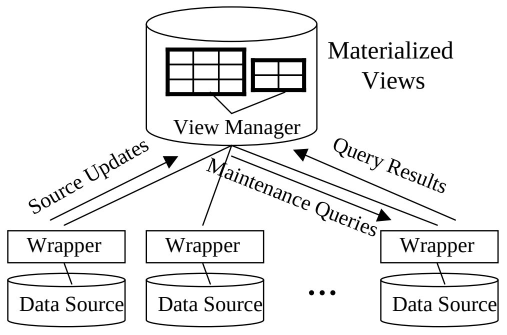


Figure 9.1: Incremental Maintenance over Distributed Data Sources

State-of-the-art view maintenance strategies require O(n^2) (batch view maintenance) maintenance queries to remote data sources with *n* being the number of data sources in the view definition. They usually only batch the updates specified against the same data source [63, 66, 93]. This mechanism does not scale for large sized nor for a large number of data sources.

On the other hand, state-of-the-art view maintenance algorithms [5, 63, 64, 66, 93] also tend to focus on maintaining simple acyclic join views. Little attention has been paid thus far to more complex view definitions, i.e., cyclic join views that may specify many join conditions between any two arbitrary source relations. Such cyclic join views are also being widely used in practical systems [108].

In this part of dissertation work, we first investigate scalable view maintenance algorithms for maintaining large batches of source updates (Chapter 10). The basic approach we take is to reduce the number of maintenance queries to remote data sources by effectively restructuring and grouping the batch view maintenance plans. Though such reduction in the number of maintenance queries will increase the complexity of each query, we find that it outperforms existing batch view maintenance strategies in a rather significant manner (around 400% improvement) in a majority of the cases.

We then focus on maintaining and optimizing cyclic join views over distributed data sources (Chapter 11). Many maintenance plans are available given the complexity of view definitions. We propose a cost-based view maintenance optimization framework that is able to generate optimized maintenance plans that incorporate variations of view definitions, data source processing capabilities and network cost. Such cyclic join view maintenance as well as cost-based view maintenance optimization have not been carefully addressed in state-of-the-art solutions.

9.1 Sequential vs. Batch Maintenance

We use the following concrete example to illustrate two of the most prevailing classes of existing incremental view maintenance strategies, namely, sequential maintenance and batch maintenance. The basic tradeoff that will be exploited in this work is revealed by analyzing these two strategies. Table 9.1 describes three data sources with one relation each that will be used in the example. A view *Tour-Customer* is defined as depicted in Query 9.1.

$$\begin{array}{ll} \text{CREATE VIEW} & Tour-Customer\ \text{AS} \\ \text{SELECT} & C.Name,\ C.Age,\ T.TourID, \\ & F.FlightNo,\ F.Dest \\ \text{FROM} & Cust\ C,\ FlightRes\ F,\ Tour\ T \\ \text{WHERE} & C.Name\ =\ F.Name\ AND \\ & F.Name\ =\ T.CustName \end{array} \tag{9.1}$$

R_1: Cust (Name, Age, Address, Phone)
R_2: FlightRes (Name, FlightNo, Source, Dest)
R_3: Tour (TourID, CustName, Type, Days)

Table 9.1: Data Sources Descriptions

Sequential Maintenance. Sequential maintenance refers to maintaining one single source update at a time. As one typical example of such strategy, we illustrate the SWEEP algorithm introduced in [5]. For example, one data update "*U_1 = Insert into Cust Values ('Ben', 28, 'WPI', 6136)*" happened at R_1. In order to determine the delta effect on the view extent, this requires us to send two maintenance queries, one to R_2 and another to R_3. In this case, one maintenance query (Query 9.2) is generated based on U_1 and send to source R_2. After we get the result, say ('Ben', 28, 'AA69', 'Mia'), another maintenance query (Query 9.3) will be generated and sent to R_3 to get the delta change on the view extent.

$$\begin{array}{ll} \text{SELECT} & 'Ben'\ as\ Name,\ 28\ as\ Age \\ & F.FlightNo,\ F.Dest \\ \text{FROM} & FlightRes\ F \\ \text{WHERE} & F.Name\ =\ 'Ben' \end{array} \tag{9.2}$$

$$\begin{array}{ll} \textsf{SELECT} & 'Ben' \; as \; Name, \; 28 \; as \; Age, \; T.TourID \\ & 'AA69' \; as \; FlightNo, \; 'Mia' \; as \; Dest \\ \textsf{FROM} & Tour \; T \\ \textsf{WHERE} & T.CustName \; = \; 'Ben' \end{array} \tag{9.3}$$

Thus, to maintain one source update using SWEEP, we may have to send maintenance queries to all data sources besides the one where the source update originated from to compute the delta effect on the view extent. If multiple source updates need to be maintained, as illustrated in Table 9.2, we would repeat this process for each and every update until all updates have been processed [1].

U_1: Insert ('Ben', 28, 'WPI', 6136) into Cust
U_2: Insert ('Tom', DL169, 'Lax', 'Bos') into FlightRes
U_3: Insert (63, 'Tom', 'Lux', 10) into Tour
U_4: Insert ('Joe', AA189, 'Bos', 'Paris') into FlightRes
U_5: Delete ('Ken', 27, 'WPI', 5857) from Cust

Table 9.2: Data Updates Descriptions

Batch Maintenance. Batch maintenance refers to maintaining the view extent using source-specific *deltas* [63, 66] where one *source delta* describes a set of changes made to a data source in a certain time period. For example, instead of maintaining five updates listed in Table 9.2 individually as described above, we construct a delta specific for each source. Thus,

[1]Concurrent source updates could happen during the maintenance process. Thus additional concurrency control is necessary to keep the view extent consistent [24, 123]. We discuss this with more detail in Section 10.3.1.

ΔR_1 = { +('Ben', 28, 'WPI', 6136), -('Ken', 27, 'WPI', 5857) } [2], ΔR_2 = { +('Tom', DL169, 'Lax', 'Bos'), +('Joe', AA189, 'Bos', 'Paris') }, and ΔR_3 = { +(63, 'Tom', 'Lux', 10) }. Thereafter, the incremental view extent (*view delta*) for all five updates can be logically computed in three steps (one step per source delta). Within each step, maintenance queries are built based on the source-specific delta and submitted to the other data sources to compute the maintenance result.

Batch view maintenance reduces the time taken for maintaining a large set of source updates [27, 63, 66, 93]. Sequential maintenance involves many maintenance queries (depending on both the number of source updates and the number of data sources) to be sent with each maintenance query reflecting a single source update. Batch maintenance typically has a smaller number of maintenance queries (depending only on the number of data sources) with each maintenance query being more complex that is reflecting a set of source updates. This now opens the opportunity to group multiple source updates and to construct a combined maintenance query that may outperform handling each individual update one by one. Exploitation of this tradeoff between the number of maintenance queries and their complexity (size) leads to novel view maintenance algorithms that improve maintenance performance.

[2]For simplicity, we use '+' to represent an insert operation and '-' to denote a delete operation. Here, each source delta represents the updates at a logical level, we separate the processing of insert and delete operations in the real implementation.

9.2 Abstraction of View Maintenance Process

For ease of describing our proposed maintenance strategies, we first present an abstraction capturing the essence of the state-of-the-art batch view maintenance algorithms below. Assume a materialized view V is defined as an n-way join upon n distributed data sources. It is denoted by $R_1 \bowtie R_2 \ldots \bowtie R_n$ [3]. There are n source deltas (ΔR_i, $1 \leq i \leq n$) that need to be maintained. As was mentioned earlier, each ΔR_i denotes the changes (the collection of insert and delete tuples) on R_i at a logical level. An actual maintenance query will be issued separately, that is, one for insert tuples and one for delete tuples.

Given the above notations, the batch view maintenance process can be depicted as in Equation 9.4. Here R_i refers to the original data source state without any changes from ΔR_i, while R'_i represents the state that reflects $R_i + \Delta R_i$ ('+' denotes the union operation). The discussion of the correctness of this batch view maintenance itself can be found in [63, 66]. Note that concurrency control strategies either compensation-based [5, 20, 122] or multiversion-based [24] need to be employed if additional source updates happen concurrently. Without loss of generality, we now only focus on the maintenance queries and ignore any concurrent source updates. The discussion of handling concurrent updates will be deferred to Section 10.3.1.

[3]Discussions of the handling of more general SPJ views will be deferred to Section 11.2.

$$\begin{aligned}\Delta V &= \Delta R_1 \bowtie R_2 \bowtie R_3 \ldots \bowtie R_n \\ &+ R_1' \bowtie \Delta R_2 \bowtie R_3 \ldots \bowtie R_n \\ &+ \ldots \\ &+ R_1' \bowtie R_2' \bowtie R_3' \ldots \bowtie \Delta R_n\end{aligned} \tag{9.4}$$

We call Equation 9.4 a **batch maintenance plan**. It specifies how to maintain the view at an abstract level. Each "line" in Equation 9.4 is referred to a **maintenance step**, i.e., $\Delta R_1 \bowtie R_2 \bowtie R_3 \ldots \bowtie R_n$. A maintenance query needs to be composed for each join ($\bowtie$) either from the source delta (ΔR_i) or the intermediate results from previous queries, i.e., $\Delta R_1 \bowtie R_2$. For ease of description, we may interchange the term 'maintenance query' and 'delta' (either ΔR_i or the result of a maintenance query) in the following sections. Two ways of composing a maintenance query from a delta will be discussed in Section 10.5.2. Note that the evaluation of each maintenance step is expected to start from the source delta (ΔR_i) and goes over all the other data sources. This is because each source delta is usually much smaller in terms of the number of tuples compared to the size of a data source. Seen from the above discussion, *n*(*n*-1) (O(n^2)) maintenance queries are required for the batch maintenance to compute the delta change (ΔV) of the view extent.

However, two questions remain. First, is it possible to further reduce the number of maintenance queries, say to less than $O(n^2)$? Second, does a lower number of maintenance queries imply a reduction in total maintenance time? Or, put differently, what are the key factors that affect the maintenance performance? Here, we use the batch maintenance plan (Equation 9.4) as the baseline algorithm and exploit it to form our proposed strategies.

Traditional distributed query optimization techniques [57] could be applied to improve view maintenance performance, e.g., to select an optimized join execution order for each maintenance step. Clearly, this is orthogonal to what we will explore in this Chapter since our focus is to find new maintenance algorithms by restructuring the batch maintenance process. The cost-based optimization techniques will be discussed in Chapter 11. We note that the size of each source delta usually is much smaller than that of the data source. Hence in the view maintenance context, finding the common expressions such as $R_3 \bowtie R_4$, which is investigated in traditional multiple query optimization [98], may not be beneficial since the common parts are too large to be evaluated.

Chapter 10

Grouping and Restructuring Maintenance Queries

10.1 Adjacent Grouping Algorithm

One way to reduce the number of maintenance queries is to exploit the regularity in a maintenance plan to promote sharing of common accesses to data sources. Studying the batch maintenance plan (Equation 9.4), we observe that a large number of common data source accesses exist in different maintenance steps. For example, the first two maintenance steps both have $R_3 \bowtie R_4 \bowtie \ldots \bowtie R_n$ in common, while the second and the third steps both have R_1' and $R_4 \bowtie \ldots \bowtie R_n$. If we share the accesses to these common data sources, the number of maintenance queries (join operations) would be reduced.

The matrix-like depiction of the batch maintenance plan as in Figure

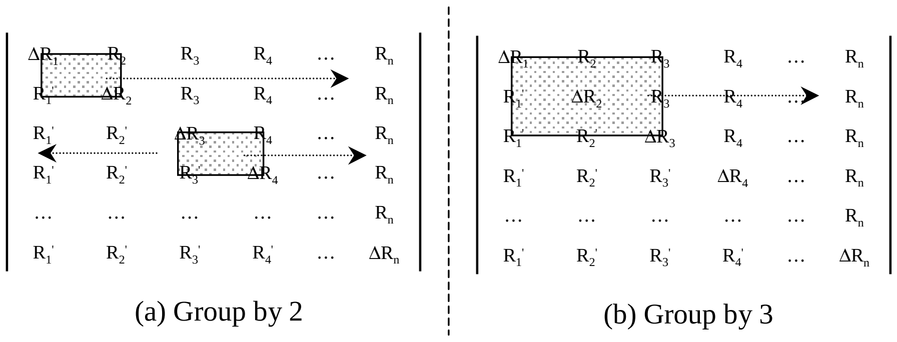


Figure 10.1: Group Adjacent Maintenance Steps

10.1 highlights the regularity and also the common items between adjacent maintenance steps. The basic idea underlying the adjacent grouping strategy is illustrated in Figure 10.1. Namely, we divide maintenance steps and group the deltas from different maintenance steps along the main diagonal. Then we share the accesses to common data sources.

For example, Figure 10.1(a) illustrates the grouping by two. The first two maintenance steps can be rewritten into one expression, namely, ($\Delta R_1 \bowtie R_2$ + $R_1' \bowtie \Delta R_2) \bowtie R_3 \bowtie \ldots \bowtie R_n$. Thus, the total number of maintenance queries for evaluating these two maintenance steps is reduced from 2(*n*-1) to *n*. While for the third and the fourth steps, we rewrite them to $R_1' \bowtie R_2' \bowtie$ ($\Delta R_3 \bowtie R_4$ + $R_3' \bowtie \Delta R_4) \bowtie \ldots \bowtie R_n$, and so on. Grouping maintenance steps by three can be done in a similar manner (see Figure 10.1(b)).

If we divide steps equally, i.e., we group every *m* ($m < n$) adjacent steps along the main diagonal, the total number of maintenance queries (N_m) can be described by Equation 10.1. Here, $\Re = (n - \lfloor \frac{n}{m} \rfloor m)(n-1)$ includes the leftover factors of *n* that can't be divided by *m*. By solving

$\frac{\partial N_m}{\partial m} = 0$, we know that the total number of queries reaches its minimum when *m* is around $\sqrt{n}$. Note that other grouping heuristics are also possible. For example, we could group maintenance steps unevenly based on the estimated respective delta sizes.

$$\lfloor \frac{n}{m} \rfloor (m(m-1) + (n-m)) + \Re \tag{10.1}$$

By adjacent grouping, we are able to reduce the total number of maintenance queries to $O(n^{3/2})$ when $m = \sqrt{n}$. We note that this approach only combines temporary results having the same schema. For example, the combination of the result from $\Delta R_1 \bowtie R_2$ and $R_1' \bowtie \Delta R_2$. This of course limits the type of query shrinking that can be considered. To further reduce the number of accesses to data sources, we must take a different approach. A new type of solution is outlined below.

10.2 Grouping Heterogenous Deltas

10.2.1 Basic Notations

To keep our description simple, we first introduce the following two notations. We use δ to represent the operation that takes a list of deltas as input and combines them together except those bracketed. For example, $\delta([\Delta R_1], \Delta R_2, \Delta R_3)$ equals a combined delta containing both ΔR_2 and ΔR_3. Note that we focus on the logical expressions only for now. The engineering problem of how to actual combine different deltas will be discussed in more detail in Section 10.2.4. Given this notation, a join operator that involves a δ

can be treated as the computation of each delta in δ individually by simple combining and decomposing rules. For example, $\delta([\Delta R_1], \Delta R_2, \Delta R_3) \bowtie R_i$ equals the collection of result deltas $\Delta R_2 \bowtie R_i$ and $\Delta R_3 \bowtie R_i$, represented by $\{\Delta R_2 \bowtie R_i$, $\Delta R_3 \bowtie R_i\}$. To further simplify the notations, we may omit the $\bowtie$ sign in the result set if the context is clear, i.e., $\{\Delta R_2 \bowtie R_i$, $\Delta R_3 \bowtie R_i\}$ will be represented by $\{\Delta R_2 R_i$, $\Delta R_3 R_i\}$.

We assume that each ΔR_i has been installed to R_i before it is reported to the view manager for maintenance. Thus, each maintenance query result will be evaluated based on R'_i instead of R_i. Compensations are needed to get the maintenance query results based on the original state R_i. We introduce θ_i to represent the compensation process using ΔR_i. For example, assuming $\mathcal{D}$ is a delta (either ΔR_i or previous maintenance query result), then $\theta_i(\mathcal{D} \bowtie R'_i)$ = $\mathcal{D} \bowtie R'_i$ - $\mathcal{D} \bowtie \Delta R_i$ = $\mathcal{D} \bowtie R_i$. The rationale behind this compensation process can simply be illustrated as follows. $\mathcal{D} \bowtie R'_i$ = $\mathcal{D} \bowtie (R_i + \Delta R_i)$ = $\mathcal{D} \bowtie R_i$ + $\mathcal{D} \bowtie \Delta R_i$. Note that both $\mathcal{D}$ and ΔR_i are available at the view manager. Thus such compensation can be computed locally at the view manager when we get the result of $\mathcal{D} \bowtie R'_i$.

10.2.2 A Greedy Grouping Algorithm

To maintain n source deltas $\Delta R_1, \Delta R_2, \Delta R_3, \ldots, \Delta R_n$ on an n-way join view, one extreme solution is to group all the intermediate results (deltas) computed in maintenance steps (ΔR_i or any previous maintenance query result) to construct a combined query. We are thus able to access each data source (R_i, $1 \leq i \leq n$) once to evaluate the maintenance process as represented by Equation 9.4. In this way, we only require n combined maintenance queries (the theoretically minimal number).

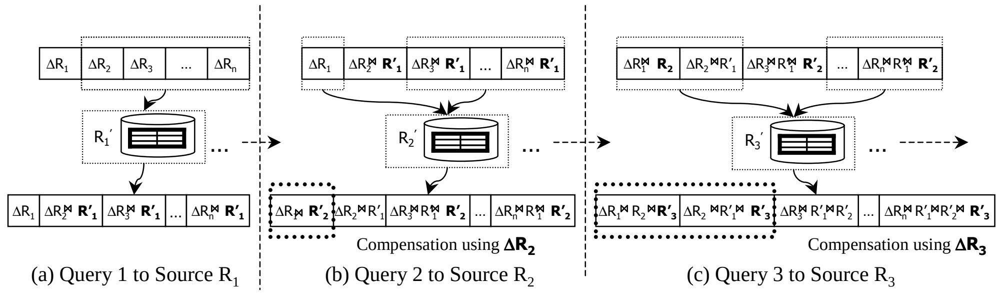


Figure 10.2: The First Three Greedy Grouping Queries

These n combined maintenance queries will be evaluated in a sequential manner by sending them to the data sources R_1, R_2, ..., R_n respectively. For simplicity, these queries are represented by Q_1, Q_2, ..., Q_n, as we describe them below.

- Q_1**:** We send all source deltas except ΔR_1 to the data source R_1 and evaluate the query result. This process can be expressed by $\delta([\Delta R_1]$, ΔR_2, ΔR_3, ..., $\Delta R_n) \bowtie R_1'$ = $\{\Delta R_1$, $R_1'\Delta R_2$, $R_1'\Delta R_3$, ..., $R_1'\Delta R_n\}$ (see Figure 10.2a).
- Q_2**:** We combine all result deltas from Q_1 except the one containing ΔR_2 and submit it to R_2 (referred as the evaluation step). After we get the query result, we compensate it using ΔR_2 for those result deltas containing ΔR_1 (referred as the compensation step). The following two capture this process (see Figure 10.2b).
  - Evaluation step:

$\delta(\Delta R_1, [R'_1 \Delta R_2], R'_1 \Delta R_3, \ldots, R'_1 \Delta R_n) \bowtie R'_2$
$= \{\Delta R_1 R'_2, R'_1 \Delta R_2, R'_1 R'_2 \Delta R_3, \ldots, R'_1 R'_2 \Delta R_n\}$.

– Compensation step:
$\{\theta_2(\Delta R_1 R'_2), R'_1 \Delta R_2, R'_1 R'_2 \Delta R_3, \ldots, R'_1 R'_2 \Delta R_n\}$
$= \{\Delta R_1 R_2, R'_1 \Delta R_2, R'_1 R'_2 \Delta R_3, \ldots, R'_1 R'_2 \Delta R_n\}$

- Q_3: Similarly, we combine all result deltas except the one containing ΔR_3 from query results of Q_2, we then ship them to the data source R_3 and evaluate the query. We compensate the results using ΔR_3 for deltas containing ΔR_1 or ΔR_2 after we get result deltas (see Figure 10.2c for the illustration).

- Q_i $(1 < i \leq n)$: To generalize, for any query Q_i, we combine the result from query Q_{i-1} except the one containing ΔR_i and then ship them to data source R_i for evaluation. The result deltas that contain ΔR_j $(j < i)$, which correspond to the data sources that have been visited, will be compensated using ΔR_i. Similarly, this evaluation and compensation process can be described as follows.

  – Evaluation:
  $\delta(R'_1 R'_2 \ldots \Delta R_k R_{k+1} \ldots R_{i-1}$ $(1 \leq k < i)$, $[R'_1 R'_2 \ldots R'_{i-1} \Delta R_i]$, $R'_1 R'_2 \ldots R'_{i-1} \Delta R_k$ $(i < k \leq n)) \bowtie R'_i$
  $= \{R'_1 R'_2 \ldots \Delta R_k R_{k+1} \ldots R_{i-1} R'_i$ $(1 \leq k < i)$, $R'_1 R'_2 \ldots R'_{i-1} \Delta R_i$, $R'_1 R'_2 \ldots R'_{i-1} R'_1 \Delta R_k$ $(i < k \leq n)\}$.

  – Compensation:
  apply θ_i to $R'_1 R'_2 \ldots \Delta R_k R_{k+1} \ldots R_{i-1} R'_i$ $(1 \leq k < i)$, we get

the final result of Q_i as $\{R'_1R'_2 \ldots \Delta R_k R_{k+1} \ldots R_i$ $(1 \leq k < i)$, $R'_1R'_2 \ldots R'_{i-1}\Delta R_i$, $R'_1R'_2 \ldots R'_i\Delta R_k$ $(i < k \leq n)\}$.

Thus, after the n-th query Q_n, we get $\{R'_1R'_2 \ldots \Delta R_k R_{k+1} \ldots R_i$ $(1 \leq k < n)$, $R'_1R'_2 \ldots R'_{i-1}\Delta R_n\}$. This exactly equals $\{\Delta R_1 \bowtie R_2 \bowtie R_3 \ldots \bowtie R_n$, $R'_1 \bowtie \Delta R_2 \bowtie R_3 \ldots \bowtie R_n$, $R'_1 \bowtie R'_2 \bowtie \Delta R_3 \ldots \bowtie R_n$, $\ldots$, $R'_1 \bowtie R'_2 \bowtie R'_3 \ldots \bowtie \Delta R_n\}$. Clearly, this is the same with the equation we have shown for the batch maintenance plan (Equation 9.4). The correctness of the approach can also be shown similarly with the above step-by-step re-transformation of Equation 9.4. Thus, by issuing only n combined queries to the underlying data sources, we indeed compute the incremental view extent ΔV.

However, one weakness of this approach is the possibly **large intermediate result set** caused by the lack of a join condition between some of the intermediate results and the data source. For example, we send $\delta([\Delta R_1]$, ΔR_2, ΔR_3, $\ldots$, $\Delta R_n)$ to data source R_1 in Q_1. Only R_2 has the join condition with R_1 given the view is defined by $R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n$. Thus, to evaluate the result $\Delta R_k \bowtie R'_1$ $(3 \leq k \leq n)$, we may have to compute the Cartesian product instead. Given that the size of each data source may be huge, this approach is thus likely not feasible for practical settings.

10.2.3 Conditional Grouping Algorithm

To address the large intermediate result set problem arising in the above greedy approach, we now take a different approach and propose the *conditional grouping* strategy. The basic idea is to make use of join conditions in the view definition. This is because a maintenance query composed from join conditions is much cheaper to process than a Cartesian product in a view maintenance context.

The whole maintenance process in the conditional grouping is divided into two phases, called *scroll up* phase and *scroll down* phase. In each phase, we only group the deltas having common join conditions with the data source.

Scroll Up Phase. $n-1$ queries, represented by Q^u_1, Q^u_2, ..., Q^u_{n-1}, will be evaluated sequentially in this phase. We describe each query below.

- Q^u_1**:** We send ΔR_1 to R_2, evaluate $\Delta R_1 \bowtie R'_2$ and then compensate the result using ΔR_2. These two steps can be expressed by $\delta(\Delta R_1) \bowtie R'_2$ = $\Delta R_1 R'_2$ and $\theta_2(\Delta R_1 R'_2)$ = $\Delta R_1 R_2$ (see Figure 10.3(a)).

- Q^u_2**:** We combine the result of the first query ($\Delta R_1 R_2$) with ΔR_2 and send them to R_3. We then compensate this query result using ΔR_3. The following steps capture this: (1) $\delta(\Delta R_1 R_2, \Delta R_2) \bowtie R'_3$ = $\{\Delta R_1 R_2 R'_3$, $\Delta R_2 R'_3\}$, and (2) $\{\theta_3(\Delta R_1 R_2 R'_3)$, $\theta_3(\Delta R_2 R'_3)\}$ = $\{\Delta R_1 R_2 R_3$, $\Delta R_2 R_3\}$ (see Figure 10.3(b)).

- Q^u_3**:** Similarly, we have the third query expressed as (1) $\delta(\Delta R_1 R_2 R_3$, $\Delta R_2 R_3$, $\Delta R_3) \bowtie R'_4$, and (2) apply θ_4 to compensate the query results. After the compensation, we get $\{\Delta R_1 R_2 R_3 R_4$, $\Delta R_2 R_3 R_4$, $\Delta R_3 R_4\}$ as the result of the third query (see Figure 10.3(c)).

- Q^u_i $(1 < i \leq n-1)$: To generalize, we do the following three operations for any query Q^u_i in the scroll up phase.

- Build the maintenance query by combing Q^u_{i-1} query result with ΔR_i. We get $\delta(\Delta R_1 R_2 R_3 \ldots R_i, \Delta R_2 R_3 \ldots R_i, \ldots, \Delta R_{i-1} R_i, \Delta R_i)$.
- Send the combined query to R_{i+1} and evaluate it against R_{i+1}. We get the query result $\{\Delta R_1 R_2 R_3 \ldots R_i R'_{i+1}, \Delta R_2 R_3 \ldots R_i R'_{i+1}, \ldots, \Delta R_{i-1} R_i R'_{i+1}, \Delta R_i R'_{i+1}\}$.
- Compensate the result using ΔR_{i+1} (θ_{i+1}). We then get $\{\Delta R_1 R_2 R_3 \ldots R_i R_{i+1}, \Delta R_2 R_3 \ldots R_i R_{i+1}, \ldots, \Delta R_{i-1} R_i R_{i+1}, \Delta R_i R_{i+1}\}$.

After processing query Q^u_{n-1}, we get $\{\Delta R_k \bowtie R_{k+1} \ldots \bowtie R_n$ $(1 \leq k \leq n)$ $\}$ as the result of the scroll up phase (Figure 10.3(d)).

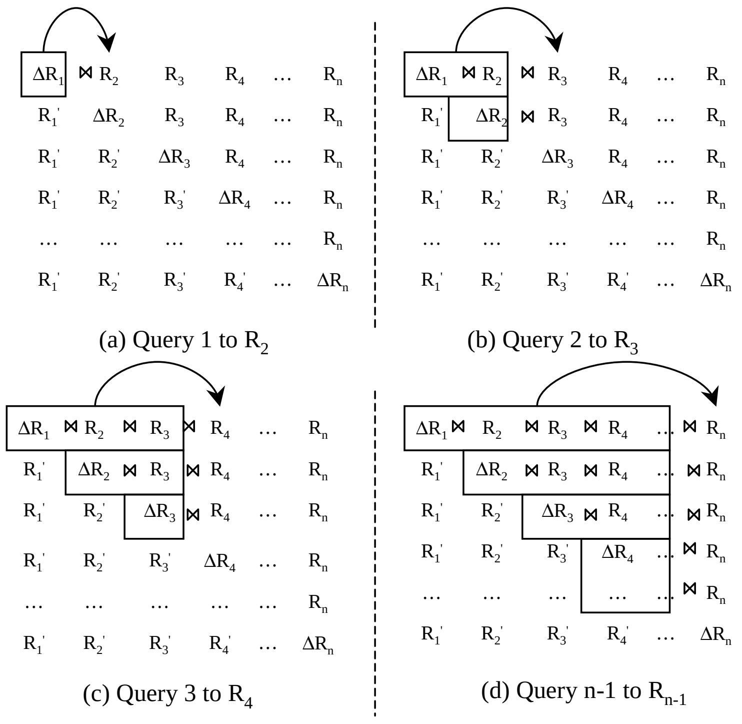


Figure 10.3: Scroll Up Phase

Scroll Down Phase. There are also $n-1$ queries in the scroll down phase represented by Q_1^d, Q_2^d, ..., Q_{n-1}^d. These queries take the result from the scroll up phase as input. Below, we again describe this phase by its queries.

- Q_1^d**:** We first evaluate $\delta(\Delta R_n) \bowtie R'_{n-1}$ and get $R'_{n-1}\Delta R_n$. Note that no compensation needs to be applied in this phase (Figure 10.4(a)).
- Q_2^d**:** We combine the result of the first query ($R'_{n-1}\Delta R_n$) with the result from the scroll up phase containing ΔR_{n-1} ($\Delta R_{n-1}R_n$ in this case). This results in $\delta(R'_{n-1}\Delta R_n, \Delta R_{n-1}R_n)$. We send it to R_{n-2} to evaluate $\delta(R'_{n-1}\Delta R_n, \Delta R_{n-1}R_n) \bowtie R'_{n-2}$. We get $\{R'_{n-2}R'_{n-1}\Delta R_n$, $R'_{n-2}\Delta R_{n-1}R_n\}$ (Figure 10.4(b)).
- Q_i^d ($1 < i \leq n-1$): To generalize, we take the following two steps for any query Q_i^d in the scroll down phase.
  - Combine previous query (Q_{i-1}^d) result (denoted by $\{R'_{n-i+1}R'_{n-i+2} \ldots \Delta R_{n-k+1}R_{n-k+2} \ldots R_n$, $1 \leq k \leq i-1\}$) with the result from the scroll up phase that contains ΔR_{n-i+1} ($\Delta R_{n-i+1}R_{n-i+2} \ldots R_n$).
  - Submit the combined query to R_{n-i} and evaluate it against R_{n-i}. We get result $\{R'_{n-i}R'_{n-i+1}R'_{n-i+2} \ldots \Delta R_{n-k+1}R_{n-k+2} \ldots R_n$ $(1 \leq k \leq i)\}$.

Thus, after processing query Q_{n-1}^d, we get $\{R'_1R'_2R'_3 \ldots \Delta R_{n-k+1}R_{n-k+2} \ldots R_n$ $(1 \leq k \leq n-1)\}$. As we can see, this equals $\{\Delta R_1 \bowtie R_2 \bowtie R_3 \ldots \bowtie R_n$, $R'_1 \bowtie \Delta R_2 \bowtie R_3 \ldots \bowtie R_n$, $R'_1 \bowtie R'_2 \bowtie \Delta R_3 \ldots \bowtie R_n$, ..., $R'_1 \bowtie R'_2 \bowtie R'_3 \ldots \bowtie \Delta R_n\}$ (See Figure 10.4(d)). It is clear that this is also the same as Equation 9.4.

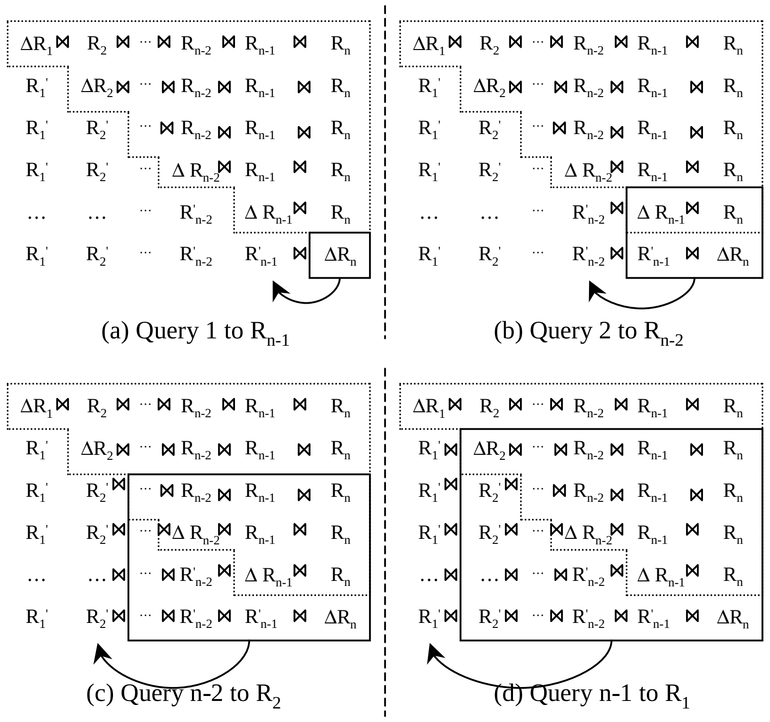


Figure 10.4: Scroll Down Phase

To summarize, the *scroll up* phase calculates the upper part along the main diagonal of the batch maintenance plan (Equation 9.4) using n-1 queries, while the *scroll down* phase computes the remaining part in another n-1 queries.

10.2.4 Unifying Deltas Together

Next, we address the engineering problem of combining the heterogeneous deltas. For example, consider building a combined delta for $\delta(\Delta R_1 \bowtie R_2, \Delta R_2)$. If the query engine at the data source were advanced, it could exploit the similarity among the deltas to scan the source relation once when processing this δ operator even if we send them separately. However, data sources may not be that advanced. Thus, we instead propose a non-intrusive method to address this issue of unifying various deltas from different data sources.

The basic idea is to construct one large table that contains the schema of different deltas and fill the respective unrelated fields with default values. This table is shipped to the data source as one large delta and evaluated together. The view manager splits the large query result back into different deltas per source. We may append certain identification related information to the delta so we can split the query result back into deltas more easily.

As shown in Figure 10.5, instead of sending delta tables $\Delta R_1 \bowtie R_2$ and ΔR_2 to the data source R_3 separately, we build a union table which contains the information of both deltas and send them together to R_3 to evaluate the maintenance result in one pass. For the issues of building a maintenance query from a delta table, either a composite SQL query or temporary table approach can be applied based on whether the data source is cooperative or not. We will discuss this in more detail in Section 10.5.2.

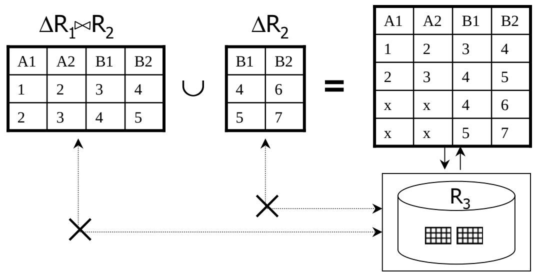


Figure 10.5: Example of Unifying Different Deltas

10.3 Generalizing the Maintenance Strategies

10.3.1 Concurrent Updates

In the grouping strategies proposed above, we have assumed that there is no concurrency interfering with the current view maintenance plan. This can be easily achieved by a multi-version system [24] because we can always retrieve the right data source states from the versioned source data. However, if a compensation-based approach were to be used such as [20], concurrent updates would have to be considered. To address this, we propose to apply the following method to maintain the view even in concurrent environments.

We use two vectors to hold source updates: the **current vector** (**CV**) holds the *deltas* per source that currently is being maintained, while the **concurrent vector** (**CRV**) holds all updates that occur concurrently to the current maintenance plan. Initially, CRV is empty because all source updates will be put into CV. After we begin to maintain the deltas in CV, newly incoming updates will be put into CRV. As usual, we use R_i ($1 \leq i \leq n$) to represent its original data source state, and R'_i ($R'_i = R_i + \Delta R_i$) to represent the state that incorporates the effect of source updates in CV. We use R^c_i to represent the state that reflects $R'_i + \Delta R^c_i$, where ΔR^c_i denotes the corresponding deltas accumulated in CRV that are concurrent with the current maintenance plan.

As done in most of the literature [5, 122], we assume that all message transfers between sources and the view manager use a FIFO scheme. That is, all updates that happen on a data source after the evaluation of the maintenance query upon this source will also arrive at the view manager (vector CRV) after the arrival of the result of this maintenance query. That is, we can use *deltas* in both vectors ($\Delta R_i, \Delta R_i^c$) to restore the appropriate data source states (either R_i' or R_i), when the view manager gets the result of a maintenance query.

Now, we are ready to extend the original compensation operator θ_i to θ_i^{i+c} and θ_i^c. Here θ_i^{i+c} compensates the query result using $\Delta R_i + \Delta R_i^c$. That is $\theta_i^{i+c}(\mathcal{D} \bowtie R_i^c) = \mathcal{D} \bowtie R_i$. The θ_i^c compensates the result using ΔR_i^c. That is, $\theta_i^c(\mathcal{D} \bowtie R_i^c) = \mathcal{D} \bowtie R_i'$. Given that, above *conditional grouping* algorithm can be simply adapted as follows for a concurrent environment: (1) For any query Q_i^u in the *scroll up* phase, we use $\theta_{i+1}^{(i+1)+c}$ to compensate the result. (2) For any query Q_i^d in the *scroll down* phase, we then use θ_{n-i}^c to compensate the result.

Thus, we compute view delta (ΔV) which exactly only reflects the source updates in CV. Once we refresh the view extent, we simply move the deltas in CRV to CV and set $R_k = R_k'$ $(1 \leq k \leq n)$. Thereafter, we can repeat the maintenance process for the next set of collected updates.

10.3.2 General View Definitions

The grouping strategies we have described so far assume a linear join view definition, i.e., $R_1 \bowtie R_2 \ldots \bowtie R_n$, as also implicitly assumed by many previous works [5, 64, 66, 93]. However, practical view definitions may have other shapes, such as a star-shaped view definition. For these, we use a *join graph* to represent the view definition. A node in a join graph represents the data source, while an edge denotes the join conditions that appear in the view definition. We then propose to apply the following graph transformation technique, as briefly described below. (1) Find a linear path and apply the grouping strategies for parts of the view definition related to the linear path [1]. (2) Transform the graph using the partial results from (1) and recursively apply this Find-and-Transform technique.

For example, Figure 10.6(a) represents a star-shaped view (V) that involves 5 data sources. To maintain this view using grouping strategies, we first find a linear path, i.e., $R_1 \bowtie R_2 \bowtie R_3$. For simplicity, we use G_1 to represent this part of the view definition. We then maintain G_1 by the grouping strategy (Figure 10.6(b)). After that, we transform the original graph by replacing the linear path using G_1. Here, edges that connects to any of nodes in the linear path are changed to G_1, and multiple edges between two nodes are combined into one. The delta change of G_1 (ΔG_1) can be got from the maintenance result of G_1 (Figure 10.6(c)). We repeat the above processes until we get the final view maintenance result ΔV. Note that we do not have G_1 materialized, thus, a maintenance query involving G_1 (or G_1' = G_1 + ΔG) has to go through each of the underlying data sources, i.e., $R_1 \bowtie R_2 \bowtie R_3$ in this case.

10.4 Cost Model and Analysis

We now introduce cost models we have developed to analyze proposed maintenance strategies. Here, we focus on the following two cost variables since they are the main factors that affect the overall performance: the cost

[1] In Chapter 11, we will provide a cost-based optimization algorithm to find a good linear path from the view graph.

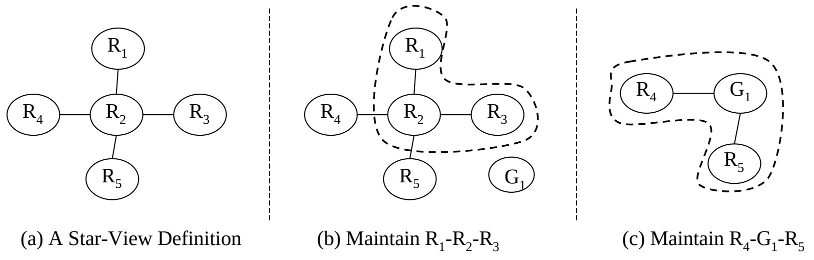


Figure 10.6: Handling General View Definitions

of transferring data between the view manager and the data sources, and the cost of evaluating maintenance queries (join operations). We note that no compensation cost would exist if we were to apply a multiversion based concurrency control strategy [24]. This happens indeed to be the environment we have at our disposal for our experimental study (Section 10.5). Hence, in the cost model, we do not consider the compensation cost.

We use the following assumptions to further simplify the models we develop: **(1)** Assume all data sources are identical in terms of the cost of answering similar maintenance queries. Thus, we use R to represent each data source R_i ($1 \leq i \leq n$). **(2)** Assume all ΔR_i ($1 \leq i \leq n$) are identical in terms of the cost when evaluating against a data source R, i.e., all ΔR_i have same number of insert and delete tuples involved. Thus, we use $\mathcal{D}$ to represent each delta ΔR_i.

To represent the result delta of a maintenance query composed from a source delta $\mathcal{D}$, we define $\mathcal{D}_{i+1} = \mathcal{D}_i \bowtie R$ ($1 \leq i \leq n-1$) with $\mathcal{D}_1 = \mathcal{D}$. For simplicity, we use S_i to represent the size of a delta $\mathcal{D}_i$.

The cost of the batch maintenance is given by T_b with $T_b = n \sum_{i=1}^{n-1} [N(S_i)$ + $J(S_i)$ + $N(S_{i+1})]$, which is a summation of individual maintenance query costs. Here $N()$ and $J()$ represent the magic unit cost functions of data transfer and maintenance query answering respectively [2]. $N(S_i)$ represents the network cost of sending $\mathcal{D}_i$ from view manager to the data source. $N(S_{i+1})$ denotes the network cost of transferring the corresponding query result from the data source to view manager. $J(S_i)$ denotes the join cost of evaluating the corresponding maintenance query.

The cost of adjacent grouping can be described by T_a assuming that we divide the maintenance steps evenly into groups of size *m* where $m < n$. $m\sum_{i=1}^{m-1}[N(S_i) + J(S_i) + N(S_{i+1})]$ represents the cost of grouping and processing m source deltas (a $m \times m$ matrix along the main diagonal in Equation 9.4), while $\sum_{i=m}^{n-1}[N(mS_i) + J(mS_i) + N(mS_{i+1})]$ denotes the cost of processing the result of above $m \times m$ matrix on the remaining $n-m$ data sources.

$$\begin{aligned} T_a &= \frac{n}{m}\{m\sum_{i=1}^{m-1}[N(S_i) + J(S_i) + N(S_{i+1})] \\ &+ \sum_{i=m}^{n-1}[N(mS_i) + J(mS_i) + N(mS_{i+1})]\} \end{aligned}$$

The cost of conditional grouping is given in T_c. Here, $\sum_{i=1}^{n-1}$ $[N(\sum_{j=1}^{i} S_j)$ $+J(\sum_{j=1}^{i} S_j)+N(\sum_{j=2}^{i+1} S_j)]$ represents the scroll up phase cost, while $\sum_{i=1}^{n-1}$

[2]We omit the discussion of detailed cost functions in our model in order to illustrate the main tradeoff on the number of maintenance queries and the complexity of each query clearly.

$[N(iS_i)\ +J(iS_i) + N(iS_{i+1})]$ denotes the scroll down phase cost.

$$\begin{aligned} T_c &= \sum_{i=1}^{n-1}[N(\sum_{j=1}^{i} S_j) + J(\sum_{j=1}^{i} S_j) + N(\sum_{j=2}^{i+1} S_j)] \\ &+ \sum_{i=1}^{n-1}[N(iS_i) + J(iS_i) + N(iS_{i+1})] \end{aligned}$$

The above formulae show the basic relationship between the number of maintenance queries and the complexity (size) of each query as expected. To accentuate this difference, we use S to represent each S_i (assume the size of each delta $\mathcal{D}_i$ is the same, $1 \leq i \leq n-1$). The relationship among these approaches is described in Figure 10.7. Here the x-axis represents the number of maintenance queries required, while y-axis denotes the average delta size. $\sum N$ represents the total data transfer cost. If the cost (network transfer and the query answering) for a large delta is less than that of the sum of the costs of handling multiple smaller deltas, performance improvements are expected by reducing the number of maintenance queries.

10.5 Experimental Evaluations

10.5.1 Experimental Testbed

We have implemented the proposed strategies based on the TxnWrap system [24]. TxnWrap is a multiversion-based view maintenance system which removes concurrency control concerns from its maintenance logic. Thus, it is not necessary to apply compensation for handling concurrent source updates in our setting. The basic TxnWrap system maintains one single source

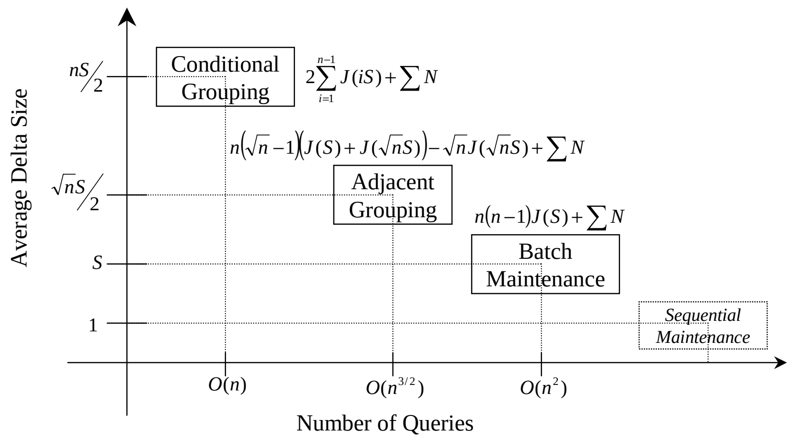


Figure 10.7: Relationship in Maintenance Strategies

update at a time using the known SWEEP algorithm [5]. The batch TxnWrap [66] combines the updates from the same data source and maintains the view extent using the source specific deltas.

We have conducted our experiments on four Pentium III 500MHz PCs connected via a local network. Each PC has 512M memory with Windows 2000 and Oracle 8*i* installed. We employ six data sources with one relation each over three PCs (two data sources per PC). Each relation has 1,000,000 (1M) tuples with 64 bytes on average of each tuple size. A materialized join view is defined through equi-joins upon these six source relations residing on a separate (the fourth) machine. The view has 1M tuples with each tuple having 384 bytes on average (having all source relations' attributes included). All the source deltas are composed of approximately the same number of insert and delete tuples. Note that two actual queries are needed when a single delta contains both insert and delete tuples.

10.5.2 Composing Maintenance Queries

Two ways of composing a maintenance query from a delta can be distinguished based on source dependent properties, namely, either **cooperative** or **non-cooperative** data sources. A non-cooperative source only answers maintenance queries (SQL queries), but offers no other services or control to the view manager. A cooperative data source would cooperate with the view manager by allowing to synchronize processes or to lock its data. To compose an appropriate maintenance query from a delta submitted to a non-cooperative data source (i.e., evaluating $\Delta R_i \bowtie R_j$), we have to use a composite SQL query which unions maintenance queries for a single source update to evaluate the result. A cooperative source would allow the view manager to build a temporary table directly at the data source, ship the delta data, evaluate it locally and send the result back.

The performance of these two methods of evaluating maintenance queries are different as we experimentally explore below by comparing batch maintenance costs using these two methods against sequential maintenance. In Figures 10.8, 10.9 and 10.10, we vary the number of data updates from 10 to 100 (and then from 500 to 3000) with all updates from the same data source (on x-axis). The y-axis represents the total maintenance query processing time.

From Figure 10.8, the processing time using a composite query increases slowly. For the temporary table approach, the increase of the total cost is even slower than that of using a composite query. This is due to the fact that the setup cost (create temporary table and populate its extent) dominates

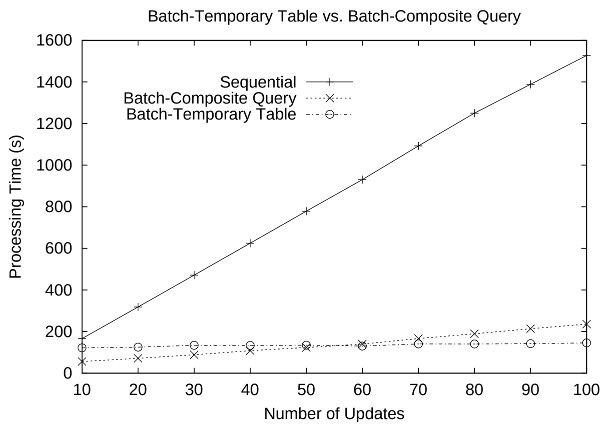


Figure 10.8: Batching a Small Number of Updates

the actual maintenance query expenses for small cases. This also explains that with a small number of updates, a temporary table approach is more expensive than that of the composite query. The sequential maintenance processing time increases linearly as expected.

Figure 10.9 displays the ratio of the sequential processing time divided by batch processing using the data gotten from Figure 10.8. The higher the ratio, the larger a performance improvement is achieved. We observe that the improvement of the composite query approach slows down when the number of updates is larger than 50 in our current setting. While for batch maintenance using temporary tables, the ratio increases steadily.

In Figure 10.10, we see that the cost of batch maintenance using the composite query approach becomes increasingly high when the number of updates increase. This is because a composite query composed of the union of a large number of queries will result in a huge cost increase. We thus instead suggest to divide such a large number of updates into smaller

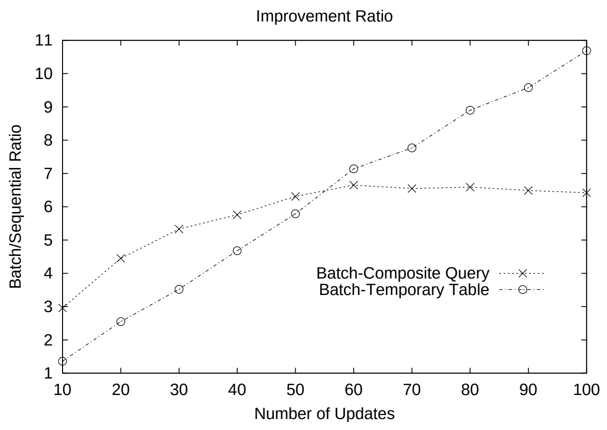


Figure 10.9: Performance Ratio (Seq. divided by Batch)

subbatch queries of size *k* based on the ratio measured in Figure 10.9. The cost of the sum of these subqueries will be smaller than that of the one large composite query. As seen in Figure 10.10, when we choose *k* equal 50, the total maintenance cost using a composite query approach will reach its optimum in our setting. However, if we use the temporary table approach, the total cost is even much lower than that of the optimized composite query approach. This is because the ratio of the increase of each such batch maintenance query to the increase in the number of source updates is very low. Without loss of generality, from now on we utilize this more efficient temporary table approach to compose maintenance queries from deltas when comparing our proposed strategies.

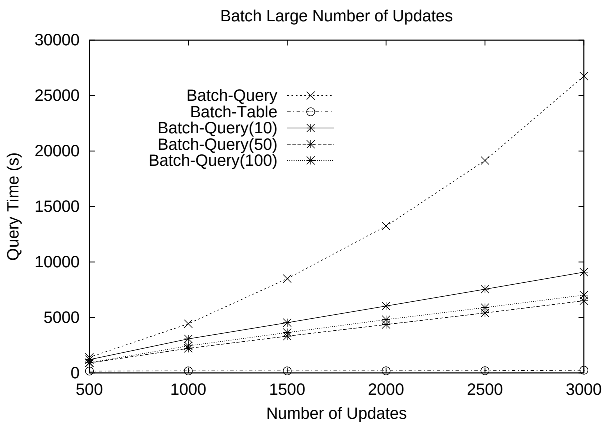


Figure 10.10: Batch a Large Number of Updates

10.5.3 Grouping Maintenance Performance

Change the Number of Source Updates

Figure 10.11 shows the average maintenance time (on the y-axis) of different maintenance approaches by varying the number of source updates from 100 to 1000 (on the x-axis). These updates are evenly distributed among six data sources. That is, for the k updates in the setting, each source delta experience approximately $k/6$ updates. From Figure 10.11, the maintenance cost of all these strategies increases very slowly because we compose and issue maintenance queries using the temporary table approach. Seen from Figure 10.11, the batch processing is almost 4 times slower than the conditional grouping. We also see the following maintenance cost relationship: *conditional grouping* $<$ *adjacent grouping* $<$ *batch processing*. Thus, with less number of maintenance queries, we do have less processing time even when the complexity (size) of each maintenance query increases. Given that the adjacent grouping is a medium performer between the batch and conditional grouping, we will focus on comparing batch with conditional grouping in more depth below.

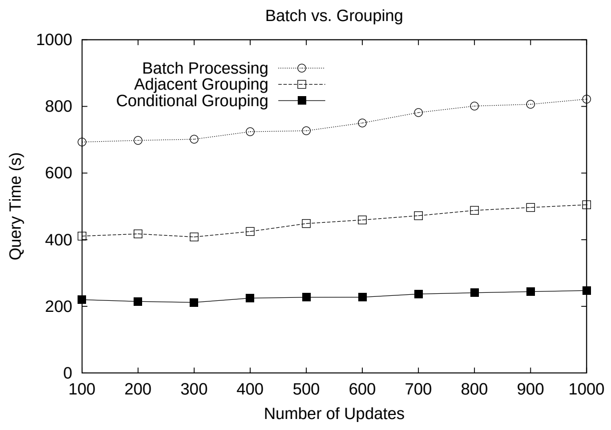


Figure 10.11: Group a Small Number of Source Updates

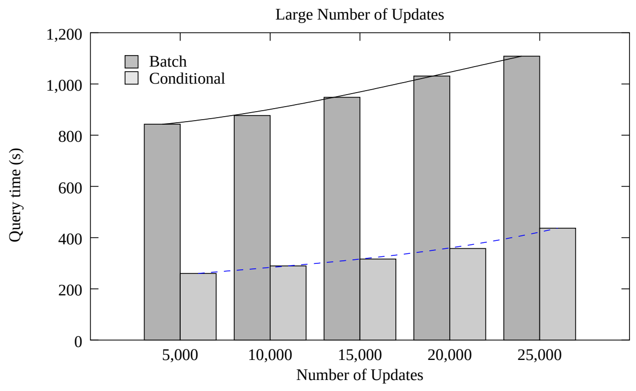


Figure 10.12: Group a Large Number of Source Updates

Figure 10.12 shows the performance changes of batch and conditional grouping given an increasing number of source updates. The maintenance cost of both approaches increases steadily as the size of each delta increases.

The conditional grouping still outperforms batch maintenance due to the size of the delta not being a major factor on the Oracle query cost if we use the temporary table approach and the conditional grouping has a smaller number of maintenance queries.

Impact of the Join Ratio

We set up 200 updates on six sources (each source delta change experience about 30 updates) and vary the join ratio from 0.5 to 3.0 (on x-axis). Join ratio here represents the average number of tuples affected by a source change. For example, a join ratio equals to 2 means that a single update which changes a tuple in the source may cause 2^5 tuples to be updated in the view extent given the view is defined over six sources. From Figure 10.13, we see that the higher the join ratio, the higher both maintenance costs. A high join ratio increases the size of each temporary maintenance result, which in turn increases the time to answer the maintenance query. Also, the higher the join ratio, the closer these two maintenance costs become. This is because any change in the temporary result size will be amplified by the join ratio and the conditional grouping has extra data (null values) need to be processed in the scroll up phase. Thus, the benefit of having a smaller number of maintenance queries will be slowly overtaken by the increase of each query cost.

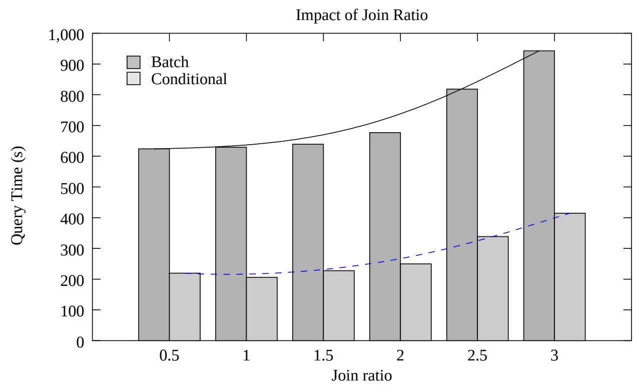


Figure 10.13: Change the Join Ratio in the View

Change the Distribution of Source Updates

We examine the impact of the distribution of 1,000 updates among the data sources on the maintenance performance (Figure 10.14). On the x-axis, a distribution of 1 denotes that we only have one source delta with 1000 updates, while k $(2 < k \leq 6)$ indicates that k source deltas with each delta change has around $1000/k$ updates. Figure 10.14 presents the cost ratio (batch maintenance cost divided by conditional grouping cost). Clearly, the more data sources are involved, the higher the performance improvement. This is because the total number of maintenance queries in batch maintenance changes from 5 to 30 queries if we increase the distribution from 1 to 6 sources, while the conditional grouping only changes from 5 to 10 correspondingly. Thus more improvement is achieved by further reducing the number of maintenance queries.

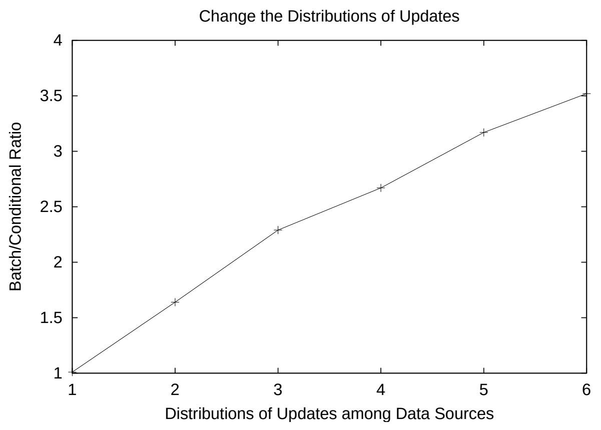


Figure 10.14: Change the Distributions of Updates

Impact of the Network Delay

To evaluate the impact of different data transfer rates of the network, we insert delay factors to model the data shipping costs. The delay is generated based on the average time to transfer one tuple. For example, if we assume that the average time of transferring a tuple with 64 bytes is ℓ, then it takes 100*2*ℓ to transfer one delta with 100 tuples with 128 bytes each. We set up six source delta changes with about 180 updates each (a total of 1000 data updates) and vary ℓ from 0 ms to 200 ms. On Figure 10.15, both maintenance costs grow steadily as the network cost of each maintenance query is increasing. In a typical network environment where the transfer time of one tuple with 64 bytes is less than 100 ms, conditional grouping is more efficient than the batch method because we have a smaller number of maintenance queries. However, in a slow network, i.e, when the average transfer time for one tuple is larger than 200 ms, then the gain gotten by

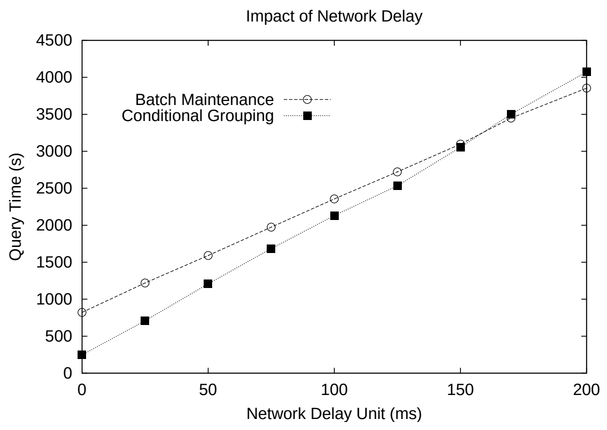


Figure 10.15: The Impact of Network Delay

reducing the number of maintenance queries is overtaken by the increase in the network cost of each query. This is because we may have some extra data (null values) to be transferred in the conditional grouping. This extra data becomes a burden in a slow network.

Chapter 11

Maintaining and Optimizing Cyclic Join Views

As we mentioned in the introduction section, state-of-the-art view maintenance algorithms usually focus on maintaining simple acyclic join views [63, 64, 66, 93]. They also have not investigated on potential optimization opportunities by exploring environmental settings such as view definitions and data source processing capabilities. In this chapter, we first describe possible view maintenance optimization opportunities in Section 11.1. While in Section 11.2, we discuss the cyclic view maintenance strategies. In Section 11.3, we then provide a cost-based view maintenance optimization framework to generate optimized view maintenance plans tuned to particular environmental settings

11.1 View Maintenance Optimizations

The maintenance process as identified in Equation 9.4 can be viewed as the process of answering n inter-related distributed queries. That is, each maintenance step corresponds to a distributed query that involves joins on n data sources. Given that, two general optimization opportunities, namely, choosing optimal join orders for maintenance queries and and sharing common accesses to data sources to reduce the number of maintenance queries, can be naturally applied to a view maintenance process. We briefly describe them below.

11.1.1 Choosing Optimized Join Orders

In an incremental view maintenance context, the size of source deltas is usually much smaller compared with the size of data source relations. Hence, without loss of generality, the evaluation of each maintenance step can be expected to start from the source delta. That is, maintenance queries in one maintenance step are processed in a sequential manner. The view manager first composes a maintenance query based on the source delta. While the maintenance query to the next data source will be composed and processed after the results of the previous maintenance query have been returned to the view manager. This is to avoid maintenance queries that directly join over data sources. Yet, multiple ways of executing each maintenance step exist. For example, for the second maintenance step that contains ΔR_2, we could either evaluate $\Delta R_2 \bowtie R_3$ or $\Delta R_2 \bowtie R'_1$ first. Different join orderings bring variations such as different intermediate results that affect the overall performance. Thus, the selection of optimal join orders for a multi-join query, which has also been investigated in traditional distributed query optimization such as in [57, 44], could be applied here to improve the view maintenance performance.

Considering view definitions beyond simple acyclic join views, i.e., those having multiple join conditions between arbitrary data sources possibly with cycles, the selection of such join orderings is likely to have a major impact on the view maintenance performance.

However, we note that such optimization only manipulates the ordering of maintenance queries, it does not change the maintenance logic itself. For instance, it does not combine multiple maintenance queries into one customized query to reduce the number of accesses to remote data sources.

11.1.2 Reducing the Number of Maintenance Queries

Reducing the number of accesses (maintenance queries) to remote data sources has the potential to improve the overall maintenance performance. For example, a batch maintenance that maintains multiple updates from the same data source together (*n*(n-1)* maintenance queries) is shown to have a superior performance compared with maintaining one single source update at a time (*k*(k-1)* queries) [66, 93]. Here n is the number of data sources, while k is the total number of source updates that need to be maintained. The number k usually is much larger than n.

As discussed in Chapters 10, we have proposed a *grouping maintenance* algorithm that maintains a materialized view defined as $R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n$ only using 2*(n-1) maintenance queries. In grouping maintenance, n

maintenance steps (Equation 9.4) is visualized as a computational matrix. The basic idea is to group deltas (source deltas or intermediate maintenance results) and construct combined maintenance queries whenever it is possible. Compared with state-of-the-art batch maintenance which as illustrated by Equation 9.4 having O(n^2) remote maintenance queries, this further reduction of maintenance queries has been shown to lead to major performance improvements in a majority of cases (see Section 10.5). However, the basic grouping maintenance algorithm does not address the views beyond simple acyclic join views.

11.2 Cyclic Join View Maintenance

11.2.1 View Definition Graph

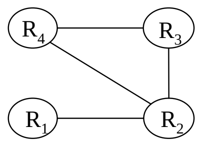
We use a *view graph* to represent a general join view definition (including cyclic views). Each node in the graph represents a data source that appears in the view definition. An edge indicates a join condition in the view definition between two respective data sources. For example, the graph depicted in Figure 11.1(c) represents the view *Tour-Customer* defined by the SQL query in Figure 11.1(a) based on the data source descriptions in Figure 11.1(b). Other operations in the view definition such as projection and selection are assumed to be applied locally at each data source. Thus they are not explicitly depicted in the view graph. As we will discuss in Sections 11.3.1 and 11.4.2, these operations are implicitly captured by our cost regression model.

```
CREATE VIEW  Tour-Customer AS
SELECT       C.Name, F.Dest, F.FlightNo, T.TourID, P.StartDate
FROM         Customer C, FlightRes F, Tour T, Participant P
WHERE        C.Name=F.Name and F.Name=T.Name
             and T.Name=P.Name and P.Loc=F.Dest
             and F.Age<=`65'
```

(a) SQL Query View Definition

R1: Customer(Name, Address, Phone)
R2: FlightRes(Name, Age, FlightNo, Dest)
R3: Participant(Name, TourID, StartDate, Loc)
R4: Tour(TourID, Name, Type, Dest)

(b) Description of Data Sources



(c) View Definition Graph

Figure 11.1: Model View Definition

11.2.2 Extended Batching and Graph Transformation

One key issue in maintaining a general join view is how to handle extra join conditions that compose cycles, i.e., the join between R_2 and R_4 in Figure 11.1(c). We now propose the following two strategies to address this: (1) *extended batching* that incorporates extra join conditions in each maintenance step whenever it is applicable, and (2) *view graph transforming* that transforms view graph into simpler forms and then applies existing algorithms for simple join views recursively.

Extended Batching. In this extended batching approach, we aim to incorporate such extra join conditions between data sources in each maintenance query whenever it is possible. For example, after we get the maintenance query results of $\Delta R_2 \bowtie R_3$, we can combine the join conditions indicated by edges R_2-R_4 and R_3-R_4 together and send a combined maintenance query to R_4. Thus, both join conditions can be evaluated at the same time.

We prefer such combinations in a distributed environment because this reduces the number of accesses to remote data sources. Such reductions have the potential to improve the view maintenance performance.

Note that extra join conditions also bring more options regarding the join orders that can be considered for each maintenance step. For example as shown in Figure 11.1(c), R_2 has join conditions both with R_3 and R_4. Thus a ΔR_2 can first join with either R_3 or with R_4.

For simplicity, we use $\bowtie_{ij}$ to represent the edge in the view graph which denotes the join condition between data sources R_i and R_j. We define $\mathcal{R}_c$ as the set of all data sources that have been evaluated thus far. For example, $\mathcal{R}_c = \emptyset$ initially. After we have evaluated the join condition between R_2-R_3 ($\bowtie_{23}$), then $\mathcal{R}_c = \{R_2, R_3\}$. We define $\bowtie_{>j}$ as the collection of all join conditions (edges) that can be evaluated at the data source R_j together. More formally, it has the following two properties: (1) each $\bowtie_{ij}$ in $\bowtie_{>j}$ is an edge in the view graph, and (2) each R_i of $\bowtie_{ij}$ has $R_i \in \mathcal{R}_c$ and $R_j \notin \mathcal{R}_c$. For example, if we have $\mathcal{R}_c = \{R_2, R_3\}$, then $\bowtie_{>4} = \{\bowtie_{24}, \bowtie_{34}\}$. After $\bowtie_{>4}$ has been evaluated, $\mathcal{R}_c = \{R_2, R_3, R_4\}$.

Thus each $\bowtie_{>j}$ contains all the join conditions that can be combined into one maintenance query to be submitted to the source R_j. The join conditions in each $\bowtie_{>j}$ depend on the actual execution order of the maintenance queries (the data sources that have been visited so far) in each maintenance step. For example, both Formulae 11.1 and 11.2 are possible ways of execution to maintain a delta change ΔR_2 for the view modeled by Figure 11.1. Here $\bowtie_{>4}$ of Formula 11.1 includes $\{\bowtie_{24}, \bowtie_{34}\}$, while $\bowtie_{>3}$ in Formula 11.2 denotes $\{\bowtie_{43}, \bowtie_{23}\}$.

$$\Delta R_2 \bowtie_{23} R_3 \bowtie_{>4} R_4 \bowtie_{12} R_1 \tag{11.1}$$

$$\Delta R_2 \bowtie_{12} R_1 \bowtie_{24} R_4 \bowtie_{>3} R_3 \tag{11.2}$$

Based on the above notations, the batch maintenance process for general join views containing n data sources can be represented by Equation 11.3. Compared with state-of-the-art batch process denoted by Equation **??**, join conditions in the maintenance query modeled by $\bowtie_{>j}$ in Equation 11.3 are more flexible.

$$\begin{aligned}\Delta V \quad &= \Delta R_1 \bowtie_{>2} R_2 \bowtie_{>3} R_3 \ldots \bowtie_{>n} R_n \\ &+R_1' \bowtie_{>1} \Delta R_2 \bowtie_{>3} R_3 \ldots \bowtie_{>n} R_n \\ &+\ldots \\ &+R_1' \bowtie_{>1} R_2' \bowtie_{>2} R_3' \ldots \bowtie_{>n-1} \Delta R_n\end{aligned} \tag{11.3}$$

We refer to one possible execution such as Formulae 11.1 or 11.2 as an *instance* of a maintenance step. The term *extended batch maintenance plan*, or *batch plan* for short, refers to a collection of instances with one instance per maintenance step. It specifies at the logical level the specific evaluation of a maintenance process. As can be seen, many maintenance plans exist for one given view definition since each maintenance step may have multiple instances. These maintenance plans may exhibit different performances due to differences among join conditions, network costs, and other factors.

The problem of choosing the best instance for each maintenance step is similar with finding the *optimal join ordering* that has been investigated in traditional distributed query optimization [44, 57, 103]. However, in the view maintenance context, each instance will start from the source delta change since it is much smaller than that of data sources. We also promote the combination of join edges (conditions) to reduce the number of access to data sources. These two heuristics together reduce the overall search space we need to go through in the optimization. As described in Section 11.3.1, we will first enhance our view definition model with appropriate cost information and then apply search techniques such as dynamic programming [97] to generate optimized extended batching maintenance plans.

Note that this extended batching approach still requires O(n^2) join queries over distributed data sources to compute the view delta.

View Graph Transformation. Another approach for maintaining a general join view is to transform the view graph into simpler structures, and then to recursively apply the existing algorithms. For example, the view graph defined in Figure 11.2(a) can be divided into two parts as shown in Figures 11.2(b) and 11.2(c). Existing join view maintenance algorithms could be applied to Figure 11.2(b) first. Once we get the maintenance result for Figure 11.2(b), let us call it ΔV_1, then the remaining join conditions could be directly applied to ΔV_1 to get the final maintenance result.

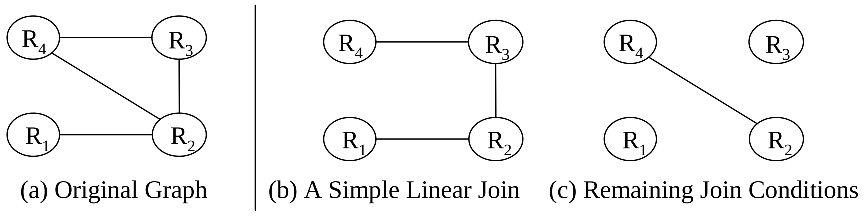


Figure 11.2: Divide View Definition Graph

Any existing view maintenance algorithms that work for a simpler view graph could be applied here. In this work, we choose the grouping maintenance algorithm introduced earlier (Section 11.1.2) to illustrate the basic idea of this transformation-based approach. We choose the grouping algorithm because (1) it has been shown to be more efficient than the typical batch maintenance in a majority of cases for views defined as a linear form such as $R_1 \bowtie R_2 \ldots \bowtie R_n$, and (2) the grouping maintenance as discussed in [69] cannot handle views other than simple linear join views.

Thus the transformation-based approach for maintaining general join views can be abstracted in the following two steps: (1) Find a path in the view graph that goes through all nodes once and apply the grouping algorithm for the part of view defined by the selected path. (2) Apply the remaining join conditions (if any) to the result calculated in the first step. Thus, we end up with $2(n-1)+1$ maintenance queries to calculate ΔV since all the remaining join conditions on the result of the first step can be evaluated locally at the view manager.

Given multiple paths may exist in one view graph, we will also have multiple ways of execution when applying the transformation-based approach using the grouping maintenance. For example, both Formulae 11.4 and 11.5 are possible choices when using the grouping strategy for the view defined in Figure 11.1. Formula 11.4 chooses the path $R_4 \rightarrow R_3 \rightarrow R_2 \rightarrow R_1$, while Formula 11.5 uses the path $R_1 \rightarrow R_2 \rightarrow R_4 \rightarrow R_3$. The superscript in the formulae represents the remaining join condition(s) that need to be evaluated locally at the view site after we get the maintenance result generated by the grouping maintenance. Similarly, we refer to each such execution choice, i.e., Formula 11.4 or 11.5, as a *grouping maintenance plan* since it also specifies how to compute ΔV logically. Not surprisingly, such different grouping maintenance plans may exhibit rather distinct maintenance performance due to variations in join conditions and network costs.

$$\begin{pmatrix} \Delta R_4 \bowtie_{43} R_3 \bowtie_{32} R_2 \bowtie_{21} R_1 \\ R_4' \bowtie_{43} \Delta R_3 \bowtie_{32} R_2 \bowtie_{21} R_1 \\ R_4' \bowtie_{43} R_3' \bowtie_{32} \Delta R_2 \bowtie_{21} R_1 \\ R_4' \bowtie_{43} R_3' \bowtie_{32} R_2' \bowtie_{21} \Delta R_1 \end{pmatrix}^{\bowtie_{24}} \tag{11.4}$$

$$\begin{pmatrix} \Delta R_1 \bowtie_{12} R_2 \bowtie_{24} R_4 \bowtie_{43} R_3 \\ R_1' \bowtie_{12} \Delta R_2 \bowtie_{24} R_4 \bowtie_{43} R_3 \\ R_1' \bowtie_{12} R_2' \bowtie_{24} \Delta R_4 \bowtie_{43} R_3 \\ R_1' \bowtie_{12} R_2' \bowtie_{24} R_4' \bowtie_{43} \Delta R_3 \end{pmatrix}^{\bowtie_{23}} \tag{11.5}$$

In a view graph such as a star-shaped one, there may not be no one single path that goes through all the data sources. Again, we could use the same partition and transform approach that described in Section 10.3.2 to address this. The basic idea is to select a *subgraph* of the view graph such that it contains a path going through all its nodes exactly once. Then, for those data sources and the corresponding join conditions included in the subgraph, we apply the same strategy as described above. This way, we compute the delta result for this subgraph. We then transform the view graph by replacing the subgraph by a single node. We consider the delta result of the subgraph as the delta change to that single node. Thereafter, we can recursively apply this technique to the reduced view graph. Given a connected join graph, such transformation guarantees to terminate with one node in the graph with its delta representing exactly the final ΔV.

11.3 Cost-Based VM Optimization Framework

11.3.1 Cost-Based Analysis

Given the above two maintenance strategies, namely, incorporating extra join conditions (extended batch maintenance) and transforming the view graph (grouping maintenance with a smaller number of maintenance queries), and the possibly large number of available maintenance plans in each strategy, the optimization question arises how to generate an efficient maintenance plan tuned to given environmental settings, i.e., a particular view graph, data source processing capabilities and network costs. We thus propose a cost-based optimization framework to generate optimized maintenance plans.

Cost Factors and Cost Functions

We first enhance the view graph by incorporating the relevant cost information to estimate the cost of maintenance plans in terms of total processing times. We annotate each node in the view graph with cost factors that describe the basic information about the data source. In the remainder, we work with the following two factors, (1) $|R_i|$: the cardinality of the source relation R_i, and (2) $|A_i|$: the number of attributes in the source relation R_i. Other cost information related to a data source could similarly be added into our model, such as the average tuple length or the number of used disk blocks. While additional factors may result in a more precise cost model, it is often not exposed by data sources. As we will illustrate in Section 11.4, we have found these two factors which are most easily available about remote sources to be already effective in estimating the cost of a view maintenance plan.

We also attach cost functions to each edge in the view graph to estimate the cost of a maintenance query (join). In a view maintenance context, the view manager (as shown in Figure 9.1) composes the maintenance query based on the delta change (either source delta or intermediate results from previous maintenance queries). Thus, for each edge ($\bowtie_{ij}$) in the view graph, the cost of having the left operand ready in the view manager and evaluating the join at R_j may differ with having the right operand ready in the view manager and evaluating it at R_i. For example, for join edge $\bowtie_{23}$ in Figure 11.3, the cost of $\Delta \bowtie R_3$ and the cost of $\Delta \bowtie R_2$ may be different even we have exactly the same Δ and the same join condition $\bowtie_{23}$. Correspondingly, we associate two cost estimation functions with each edge. Moreover, a selectivity estimation function σ_{ij} is also necessary for each edge $\bowtie_{ij}$ in the view graph. In summary, the following two types of functions will be associated with each edge in the view graph:

- τ_{ij} (or τ_{ji}) estimates the processing time for evaluating the join condition $\bowtie_{ij}$ at the data source R_j (or R_i) and returning the result back to the view manager.
- σ_{ij} estimates the selectivity of the join operation $\bowtie_{ij}$ between data sources R_i and R_j.

As an example, the view graph described in Figure 11.1 extended with appropriate cost factors and functions is depicted in Figure 11.3. Here, we assume the selectivity of each join edge σ_{ij} is known to the cost model. We apply linear regression techniques [82, 121] to build cost functions for each edge (details provided in Section 11.3.1). In general, any cost estimation methods can be used to build and improve cost functions, and they could be easily plugged into our maintenance optimization framework. Clearly, knowing more information about the view definition and the data sources could help us to build better cost functions. For example, knowing the dependency among join conditions of the view definition could help us to improve the join selectivity estimation function σ, while knowing the particular join method being used for maintenance queries could improve the cost function τ (and the regression model). However, we aim to provide a high level cost model of the view maintenance process in this work.

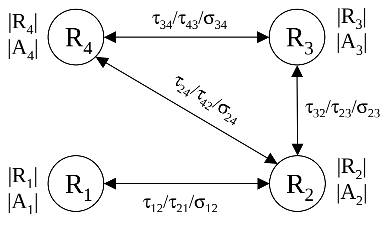

$|R_4|$ $|A_4|$
R_4
$\tau_{34}/\tau_{43}/\sigma_{34}$
R_3
$|R_3|$ $|A_3|$
$\tau_{24}/\tau_{42}/\sigma_{24}$
$\tau_{32}/\tau_{23}/\sigma_{23}$
$|R_1|$ $|A_1|$
R_1
$\tau_{12}/\tau_{21}/\sigma_{12}$
R_2
$|R_2|$ $|A_2|$

Figure 11.3: Cost-Enhanced View Graph

Cost Function Regressions

The basic idea of the regression analysis is to derive a cost model based on observed costs of several sample queries. A major benefit of using the regression model is its local autonomy. That is, we do not require the knowledge of any details regarding the remote data sources to estimate the cost. This is practical in a distributed environment where we have no control over remote data sources. And in fact, it may simply not be possible to get internal information about a source.

As described in Section 11.3.1, a cost function τ_{ij} with two basic input variables (parameters) is used to estimate the maintenance query processing time when evaluating a maintenance query against the data source R_j. Here we use $\mathcal{C}_i$ to represent the cardinality of the operand table and $\mathcal{A}_i$ to denote its number of attributes. We propose the following basic formula, which includes other potential derived variables based on $\mathcal{A}_i$ and $\mathcal{C}_i$, to model the processing time of a maintenance query against a source R_j. [1]

$$\begin{aligned}\tau_{ij} &= B_0 + B_1 * \mathcal{C}_i * \mathcal{A}_i + B_2 * \mathcal{C}_i + B_3 * \mathcal{A}_i + \\ &\quad B_4 * \sqrt{\mathcal{C}_i} + B_5 * \sqrt{\mathcal{A}_i} + B_6 * \sqrt{\mathcal{C}_i * \mathcal{A}_i}\end{aligned}$$

This model can be explained based on existing join query cost models for a DBMS. The coefficient B_0 can be interpreted as the initialization cost. While the combination of B_1, B_2, ..., B_6 and their corresponding variables can be interpreted as the estimations of processing all tuples in the delta table on the source relation incorporating the effect of the number of attributes in each tuple on the total cost.

We run a set of sample queries for each τ_{ij} defined in the view graph in our environment to measure the actual query costs on different inputs (a combination of a different number of tuples and attributes). Based on the observed values and the basic cost model, we apply the *least squares fit* and the *stepwise selection* [82] to find the suitable variables and corresponding

[1]We can build other cost models for different join edges in the view graph, for simplicity but without loss of generality, we only describe one model to illustrate the overall process.

coefficients for each τ_{ij} of a join edge. For example, $B_0 + B_2 * \mathcal{C}_i + B_6 * \sqrt{\mathcal{C}_i * \mathcal{A}_i}$ can be the actual cost function selected for a particular join edge.

Cost of a Maintenance Plan

In this section, we now elaborate on how we have extracted cost expressions of maintenance plans given the annotated cost factors and functions as identified in the above sections. Note that the cost model we developed here is a refined model based on Section 10.4.

The cost of a maintenance plan, the total processing time, can be estimated based on its computation process. For simplicity, we use C_i to represent the cardinality of a source delta ΔR_i, while we use A_i to denote the number of attributes of ΔR_i (1 $\leq i \leq$ n).

The cost of a batch maintenance plan (T_b) can be described as the sum of its maintenance step costs. For ease of explanation, we assume that k_1, k_2, $\ldots$, k_{n-1} denotes the sequence (the order) that the view manager will use to evaluate the maintenance step k (having the source delta ΔR_k, $1 \leq k \leq n$). k_i in the sequence means that the view manager will visit the data source R_{k_i} at the *i*-th maintenance query. This sequence actually is one of the permutations of data source indices without k. More formally, it has the following two properties: (1) $\forall i$, $1 \leq i \leq n-1$, $k_i \neq k$ and $1 \leq k_i \leq n$. (2) $\forall i, j$, $1 \leq i, j \leq n-1$, $i \neq j \iff k_i \neq k_j$.

Given that, the number of attributes in the i-th maintenance query for the maintenance step k is denoted by $\mathcal{A}_i^k$ = A_k + $\sum_{s=2}^{i} |A_{k_{s-1}}|$. While the number of tuples in the delta for the i-th query is described as $\mathcal{C}_i^k$ = $C_k \cdot \prod_{s=1}^{i-1}(\sigma_{k_{s-1},k_s} \cdot |R_{k_s}|)$. The cost of the maintenance step k will be the summation of its n-1 maintenance queries, as described by $T_b^k = \sum_{i=1}^{n-1} \tau_{k_i,k_{i+1}}(\mathcal{C}_i^k, \mathcal{A}_i^k)$. The cost of a batch maintenance plan can be represented as the sum of the cost of n maintenance steps, that is, $T_b = \sum_{i=1}^{n} \mathcal{T}_b^i$.

The cost of a grouping maintenance plan (T_g) can be described as the summation of the cost of the *scroll up* phase (T_u), the *scroll down* phase (T_d), and the cost of applying the remaining join conditions (if any) (T_r). For simplicity, we assume k_1, k_2, $\ldots$, k_n is the path chosen in the view graph. Thus, the cost can be expressed by the following formula.

$$T_u = \sum_{i=1}^{n-1} \tau_{k_i,k_{i+1}} (\sum_{j=1}^{i} (C_{k_j} \cdot \prod_{s=j}^{i} \sigma_{k_{s-1},k_s} \cdot |R_{k_s}|), \sum_{j=1}^{i} |A_{k_j}|)$$

$$T_d = \sum_{i=n}^{2} \tau_{k_i,k_{i-1}} (\sum_{j=i}^{n} (C_{k_j} \cdot \prod_{s=i}^{n} \sigma_{k_{s-1},k_s} \cdot |R_{k_s}|), \sum_{j=i}^{n} |A_{k_j}|)$$

The cost of T_r is denoted by a cost function $Cost_r$ on the result delta from the scroll up and scroll down phases. That is, $T_r = Cost_r(\sum_{i=1}^{n}(C_{k_i} \cdot \prod_{j=1}^{n-1} \sigma_{k_j,k_{j+1}} |R_{k_{j+1}}|), \sum_{i=1}^{n} |A_{k_i}|)$. While the cost of a grouping maintenance plan T_g as a whole equals to T_u + T_d + T_r. Note that the cost of view graphs that do not have a path can be estimated by the sum of individual subgraphs as described in Section 11.2.2.

11.3.2 Generate Optimized Maintenance Plans

As described in Section 11.2, various combinations of join edges and different paths or even different subgraphs can be chosen from a view graph. This would lead to multiple maintenance plans. Such maintenance plans may have rather distinct performances (in terms of total processing times).

In our optimization framework, we reduce the problem of finding the overall optimal maintenance plan to the problem of getting the optimal maintenance plan of each approach (either extended batching or a grouping maintenance plan). Given that, we can simply choose the better one from the most efficient maintenance plan for each individual approach. Note that the maintenance optimization framework can be naturally extended to support a global search based on the whole problem space, i.e., mixing the batching and grouping maintenances.

Select Optimal Batch Plan

One way to generate the optimal batch maintenance plan for views with a small number of data sources is via enumeration. As discussed in Section 11.3.1, the cost of a batch maintenance plan is the sum of the cost of its maintenance steps. Thus, the cost for the batch maintenance plan reaches its minimum given the minimal cost for each maintenance step.

Algorithm 8 sketches the enumeration algorithm that generates all instances of the maintenance step i that contains the source delta ΔR_i. Initially, *vNodes* only contains R_i, *cEdges* has all edges that start from R_i, while *lEdges* has all edges except those in $cEdges$. As described in Section 11.2.2, each instance starts from the source delta relation since that delta is much smaller than the data sources. We prefer to combine as many join edges as possible in each maintenance query. In line 8 of Algorithm 8, we combine those edges that have the same ending node with the start nodes of the edges having been visited. This is because such a combined edge reduces the number of accesses to a data source. Such reduction usually results in a performance improvement in a distributed environment. Thus this enumeration algorithm does not consider any execution instance that starts from a data source other than the source delta or evaluates join conditions separately one by one. Both would in general lead to sub-optimal solutions.

Algorithm 8 EnumerateStep($vNodes$, $cEdges$, $lEdges$)

```
/* vNodes: Nodes visited so far. cEdges: Edges selectable for next step.
lEdges: Edges not yet processed. */
1: while $cEdges \neq \emptyset$ do
2:     Get an edge $c$ from $cEdges$ and remove $c$ from $cEdges$
3:     $n\_vNodes \leftarrow vNodes$; $n\_cEdges \leftarrow cEdges$; $n\_lEdges \leftarrow lEdges$
4:     Put new node (from step 2) into $n\_vNodes$
5:     $nCands \leftarrow$ All new candidate edges
6:     $n\_cEdges \leftarrow n\_cEdges + nCands$ /*add new candidate edges into n_cEdges*/
7:     $n\_lEdges \leftarrow n\_lEdges - nCands$ /*remove new candidate edges from n_lEdges*/
8:     Combine edges in $n\_cEdges$ if applicable
9:     EnumerateStep($n\_vNodes, n\_cEdges, n\_lEdges$)
10: end while
11: if sizeof($vNodes$) = number of graph nodes then
12:     Record the instance and compute its cost
13: end if
```

For example, Figure 11.4 shows all available instances of the maintenance step 4 that contains ΔR_4 for the view defined in Figure 11.1. The number on each edge indicates the execution order. For simplicity, we use the list of data source indices to represent the instance. For example, the instances of the maintenance step illustrated in Figure 11.4 are represented by *4-3-2-1*, *4-2-3-1* and *4-2-1-3* respectively. Thus a batch maintenance plan can be represented by the collection of such lists of instances with one instance per maintenance step.

The cost of an instance can be estimated whenever the instance is found by the enumeration algorithm based on the cost model given in Section 11.3.1 (line 10 in Algorithm 1). For edges that can be combined, we simply

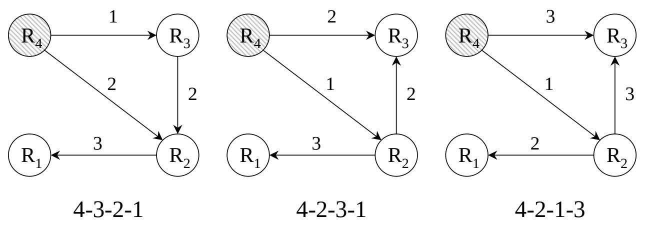


Figure 11.4: Enumerations of Maintenance Step 4

use the average of each individual estimated edge cost. For the combined join selectivity, we choose the product of individual ones as the estimation. Other refined cost estimation strategies could be applied for these to be combined join edges. Such changes on the cost estimation model would not change the overall search strategy. The optimal maintenance step instance can be simply found by choosing the one with minimal cost from all instances.

The complexity of finding an optimal instance of a maintenance step is similar to that of the join ordering optimization problem encountered in traditional distributed query processing, which has been proven to be NP-hard [113]. Thus, efficient algorithms cannot be found to solve such problems in general. We design a controlled dynamic programming algorithm (Algorithm 9) to control the tradeoff of the optimization cost with the quality of the solution. Here, the parameter k is used to indicate how many intermediate plans are kept in the $pruneQueryPlan$ process. For example, the user can set k equal to 1 to prune all other intermediate plans except the most promising one in each recursive call. Thus, the algorithm is reduced to a greedy algorithm with maximal search efficiency for identifying an optimized solution. Or the user can set k to $infinite$ to have the algorithm keep all the intermediate plans. In that case, the algorithm then becomes the full enumeration algorithm that produces the optimal plan. In algorithm 9, the $totalNumnerOfNodes$ denotes the total number of nodes in the view graph, while the $currPlans$.getNumberOfNode() returns the number of nodes have been included in the intermediate maintenance plans.

Algorithm 9 controlledDP($currPlans, k$)

```
/* currPlans: plans have been built so far. Initially, only $\Delta R_i$ in currPlans.
k: the parameter to adjust the pruneQueryPlan. */
if $currPlans$.getNumberOfNodes() $<$ $totalNumberOfNodes$ then
    $currPlanLists$ = $currPlans$.getCurrPlanLists()
    $newPlans$ = $\emptyset$ /* to store new query plans */
    for each $p \in currPlanLists$ do
        $candEdges \leftarrow$ All possible candidate edges for plan $p$
        for each $edge \in candEdges$ do
            Compose new plan $np$ from $p$ and $edge$
            Store $np$ in $newPlans$
        end for
    end for
    $newPlans \leftarrow$ pruneQueryPlan($newPlans, k$)
    controlledDP($newPlans, k$)
end if
if $currPlans$.getNumberOfNodes() = $totalNumberOfNodes$ then
    Compute cost of each plan in $currPlans$
    return the best plan
end if
```

Select Optimal Grouping Plan

The first step to generate a grouping maintenance plan is to find and choose a path in the view graph. However, finding a path (visiting all nodes in the graph once) with minimal cost is equivalent to the **Hamiltonian Path** optimization problem, which is known to be NP-complete [38]. Thus, similar to the selection of the optimal batch maintenance plan, we build an enumeration algorithm to generate all possible paths for a view graph. The algorithm is similar to Algorithm 8. As an example, Figure 11.5 shows all possible paths illustrated by the enumeration algorithm for the view defined in Figure 11.1. Once one path has been selected, the corresponding maintenance plan is also decided accordingly. For simplicity, we again use the list of source node indices in the path to represent a grouping plan. For example, the sequence 1-2-3-4 denotes the first plan listed in Figure 11.5. The dashed line denotes the remaining edge(s) that need to be processed after the completion of the scroll up and scroll down phases of the grouping maintenance (Section 11.1.2).

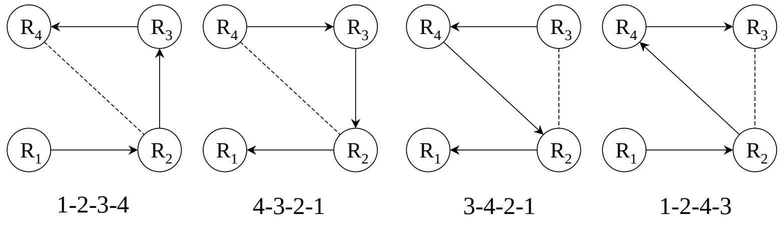


Figure 11.5: Enumerations of Grouping Plans

The cost of a grouping maintenance plan can be estimated as described in Section 11.3.1. The optimal grouping plan will be the one with the smallest estimated cost. Given the number of nodes in a view graph is usually not large, such enumeration-based algorithms are still acceptable for most practical cases.

11.4 Experimental Studies

To verify the feasibility and effectiveness of our view maintenance strategies and our corresponding optimization framework, we have implemented the proposed optimization strategies and the corresponding searching algorithms within a working view maintenance system (TxnWrap) [24]. We deploy join relations across distributed data sources. Each relation has 1,000,000 (1M) tuples. Each tuple has about 80 bytes. Here, join column values in each relation are uniformally distributed integers, while other columns are randomly generated characters. If not specified explicitly, the cardinality of one maintenance query result is similar in size to the delta. That is, if we have 10 tuples in the source delta, we expect to see roughly 10 tuples in the final view delta. Each source delta will have an equal number of insert and delete tuples. These tuples are generated randomly within the join column data value range. For the delete tuples, we make sure the generated join column values are already in the current data source. Two actual maintenance queries, one is insert and the other is delete, are involved in each single logical maintenance query such as $\Delta R_i \bowtie R_j$.

We deploy the view manager and the corresponding materialized view on a Oracle 8i server with dual 2.4GHz Xeon CPUs and 1G main memory. Data sources are deployed on different machines with various machine configurations (different CPU speed and memory size). Thus, different data sources have different processing capabilities. The data sources and the view manager are connected through a local 100M ethernet.

The first set of experiments is conducted based on the configuration shown in Figure 11.6. We employed four data sources with one relation each, denoted by R_1, R_2, ..., R_4. Three join views V_1,V_2 and V_3 are defined upon these four data sources. Note that the view includes all attributes of the underlying join relations in our experimental setup. In this setup, one index has been built on R_4 along the join condition between R_3 and R_4, and one on R_1 based on the join condition between R_2 and R_1.

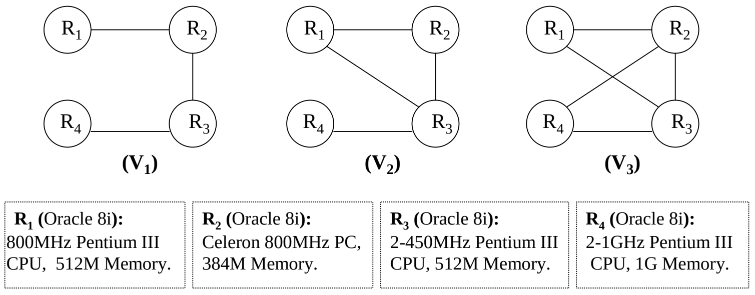

R1
R2
R4
R3
(V1)
R1
R2
R4
R3
(V2)
R1
R2
R4
R3
(V3)
R1 (Oracle 8i): 800MHz Pentium III CPU, 512M Memory.
R2 (Oracle 8i): Celeron 800MHz PC, 384M Memory.
R3 (Oracle 8i): 2-450MHz Pentium III CPU, 512M Memory.
R4 (Oracle 8i): 2-1GHz Pentium III CPU, 1G Memory.

Figure 11.6: Experimental Configuration

11.4.1 Diversity of Maintenance Plans and Costs

Table 1 shows the number of maintenance plans for batching and grouping that are possible for the views defined in Figure 11.6 (V_1, V_2 and V_3). As expected, having more join edges in the view graph dramatically increases the number of maintenance plans available in the maintenance process.

Views	Batching plan #	Grouping plan #
V_1	9	2
V_2	108	4
V_3	576	12

Table 11.1: Number of Available Plans

Not surprisingly, more maintenance plans bring diversity in the view maintenance performance. Figure 11.7 shows the best (and the worst) maintenance plans of both batching and grouping for views V_1, V_2 and V_3 when maintaining a total of 2000 source updates. Each source delta (ΔR_1, ΔR_2, ..., ΔR_4) experiences around 500 updates mixed with both insert and delete tuples. Here Best.B (or Worst.B) represents the least (or the most expensive) total processing time from all available batching maintenance plans, while Best.G (or Worst.G) denotes the best (or the worst) of grouping plans. As can be seen, the addition of more join edges results in more available maintenance plans. This in turn results in a larger gap in the view maintenance performance. Such diversity motivates the needs of the proposed view maintenance optimization.

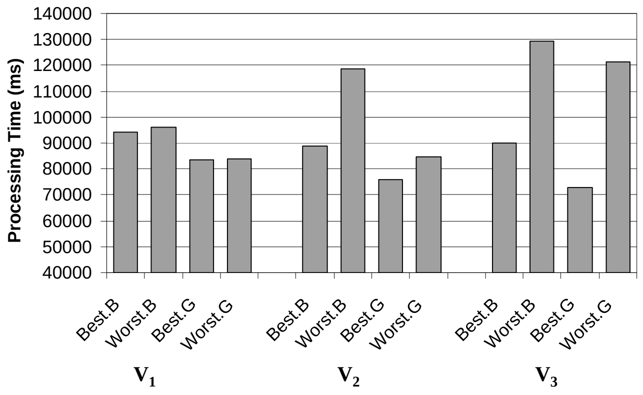


Figure 11.7: Diversity of Maintenance Costs

11.4.2 Cost-Based Optimizations

We now choose V_2 as an example to further explore the cost-based optimizations. Other view graphs we have worked with result in similar conclusions as we describe below.

Cost Function Regression

We first have to establish the cost functions for each edge (join condition) defined in the view graph to estimate the query processing time for a given maintenance plan. Figure 11.8 shows the regression function we built for R_3 along the join condition between R_2 and R_3 using least squares fit and the stepwise selection as discussed in Section 11.3.1. The fitted model (variables) and its coefficients (using *SAS* 8.0) are: $B_0 + B_1 * \mathcal{C}_i * \mathcal{A}_i + B_2 * \mathcal{C}_i + B_3 * \mathcal{A}_i$ with B_0 = 8840.55, B_1 = 0.0242, B_2 = 0.0208 and B_3 = 16.26. Other cost functions are omitted here due to the space constraints. The three solid lines in Figure 11.8 record the observed query processing times (on the y-axis) for sample maintenance queries with 2, 4 and 6 attributes respectively. The number of tuples in the source delta that the maintenance query is generated upon changes from 100 to 1000 (on the x-axis). The three dashed lines illustrate the estimated query processing time given the same input parameters (the number of attributes and the number of tuples) using the fitted function. As seen in Figure 11.8, the cost function adequately captures the basic trends of the actual query processing costs. [2]

Cost Estimation of Maintenance Plans

Figures 11.9 and 11.10 show the cost estimation of various maintenance plans. Figure 11.9 shows both the estimated and measured costs of batch-

[2]Further examination of observed values can help to find a better fitted cost function. However, this is not the main focus of this work, we thus only choose a reasonable good function which reflects the trend of the cost.

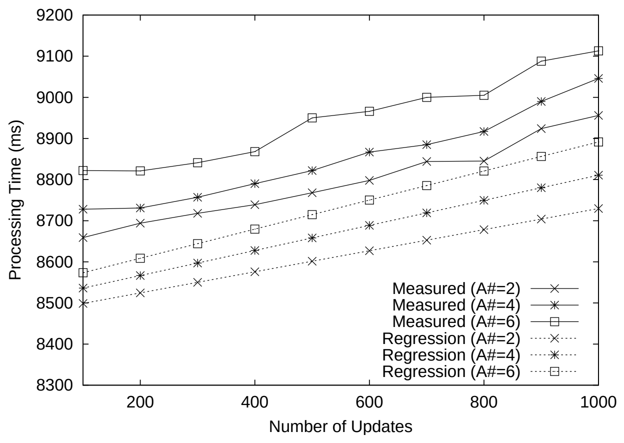


Figure 11.8: Cost Function Regression

ing maintenance plans. The line 'Worst Measured' records the total maintenance query processing time measured for the maintenance plan with the highest estimated cost generated by the search algorithm. The number of source updates ranges from 200 to 2000 (on the x-axis). [3] While the line 'Best Measured' records the corresponding observed processing times for the maintenance plan with the lowest estimated cost. The two dashed lines in Figure 11.9 show the corresponding estimated total query processing times for these two plans. We note that the estimated cost reflects the measured cost trends. However, it only accounts for around 80% of the real cost. This is because: (1) the accumulated errors caused by each individual join cost function, and (2) the measured cost includes all extra processing cost in the view manager such as converting the query results and composing maintenance queries which have not been incorporated into our cost

[3]We assume all the updates are evenly distributed among data sources. Thus, in this experiment configuration, each delta has k/4 updates where k is the total number of updates (tuples) since we have 4 source relations.

model for simplicity reasons.

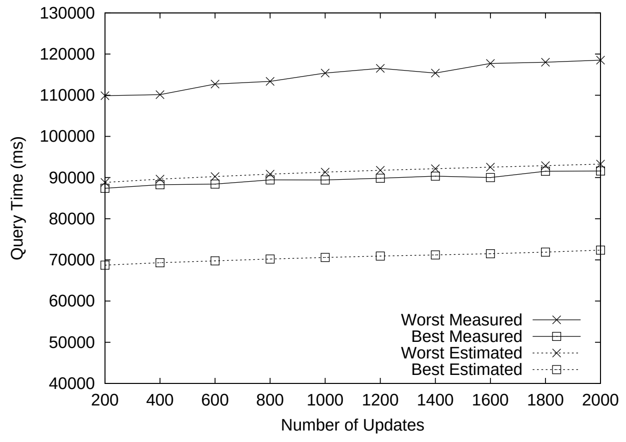


Figure 11.9: Cost Estimation of Batch Plans

Figure 11.10 illustrates all four grouping maintenance plans available for V_2. Each grouping plan is represented by its corresponding path as discussed in Section 11.3.2, i.e., 1-2-3-4. Four solid lines record the measured costs and the dashed lines show the corresponding estimated costs. For the same reasons as those for the batch maintenance cost estimation, the estimated grouping maintenance plan cost is also below that of the measured cost. However, it again indicates the trends well and thus supports the usage of our proposed cost estimation approach.

Batching vs. Grouping

In the above cases (i.e., Figures 11.9 and 11.10), the cost of the best batch maintenance plan is still worse than the worst grouping maintenance plan. This is because the grouping maintenance plans have a much smaller number of accesses to the remote data sources. However, this is not true in

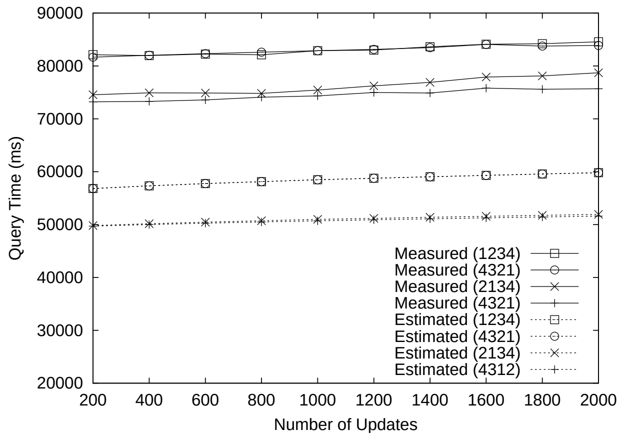


Figure 11.10: Cost Estimation of Grouping Plans

general. To illustrate this, we remove the index on R_1, and assume 2000 updates both in R_1 and R_2 respectively, 10 updates in R_3 and R_4. We set up the join conditions between R_1-R_3 and R_2-R_3 to have an approximate 50 *join ratio* which will return 50 times the number of input tuples as the join results. While the join ratios of other edges is set to be around 1.

Figure 11.11 illustrates the costs of five different batch maintenance plans given the above settings and the cost of four grouping maintenance plans. Note that many other batch plans are available. Here we only select a small subset of them as illustrating examples. The selected plans are listed in Table 2. As identified above, each maintenance plan is denoted by the list of maintenance steps for batching plan or the selected path for grouping plan.

Seen from Figure 11.11, at least two batch maintenance plans are more efficient than even the best grouping maintenance plan. This is because in each grouping maintenance plan, we have to include one of the high join

B(1): 2134,1234,4312,3124		B(2): 2314,1324,4321,3142	
B(3): 2341,1342,4321,3412		B(4): 2341,1342,4321,3241	
B(5): 2341,1342,4321,3214			
G(1): 4312	G(2): 4321	G(3): 1234	G(4): 2134

Table 11.2: Selected Maintenance Plans

factor edges (either R_1-R_3 or R_2-R_3) in the grouping path. Thus, the intermediate result will be amplified by the high join factor of such a join edge. For example, if we use the grouping path 1-2-3-4, then the second query of the scroll up phase may return 4000*50 tuples. However, in the batch maintenance plan, we may defer such joins to avoid unnecessary processing with such a large number of tuples. In this case, a batch maintenance plan having a large number of maintenance queries (in this example, it has 12 queries) will be still more efficient than a grouping maintenance plan (with 7 queries). This is because the large maintenance query results in the grouping plan overtake the benefits gained by the smaller number of maintenance queries.

Impact of the Network Cost

In the above experiments, the difference between the best and the worst maintenance plans is around 40% of the total cost. In a more diverse environment, the cost difference between maintenance plans may vary more dramatically. The following experiment illustrates the impact of the network delay on the total maintenance cost.

To evaluate the impact of different data transfer rates of the network, we

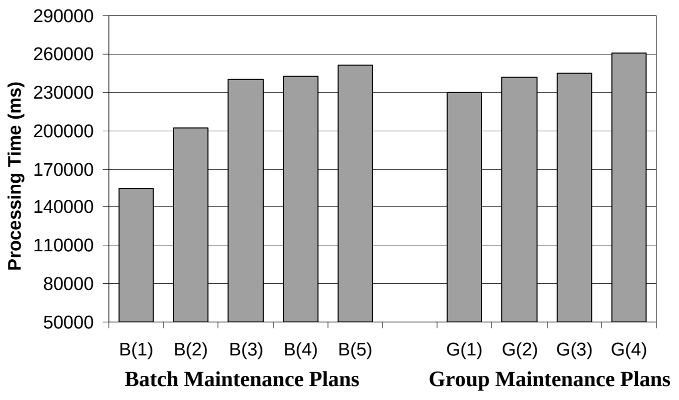


Figure 11.11: Batching Vs. Grouping

insert delay factors before evaluating each maintenance query. The delay is generated based on the average time to transfer one tuple. For example, if we assume that the average time to transfer a tuple with 2 attributes is t, then it takes 100*2*t to transfer one delta with 100 tuples with 4 attributes per tuple. Figure 11.12 shows the batch maintenance plan processing time given t equal to 5 ms while all the other settings are the same as in the experiment in Figure 11.11. Seen from Figure 11.12, the difference between the maintenance plans can be more than 300% of the total processing time. This is because the slower the network, the more the effect of processing extra intermediate query results due to a bad maintenance plans will become apparent.

11.4.3 Complex Join Views and Large Batches of Updates

We now report on some of the experiments we have conducted for complex join graphs with large batches of updates. The view is defined on 6 join relations over 4 data sources as shown in Figure 11.13. In this setup, one

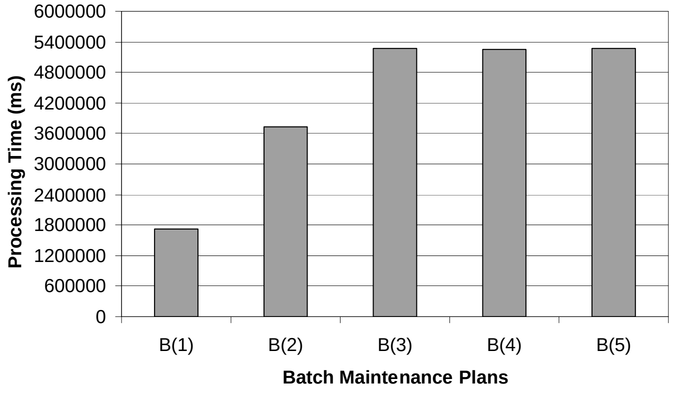
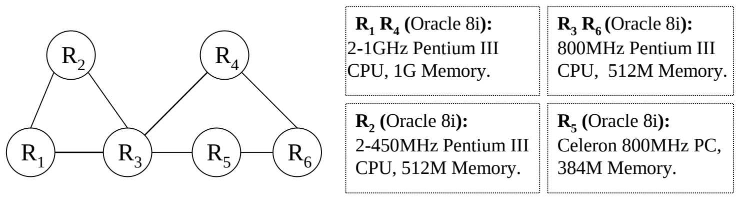


Figure 11.12: Batch Plans with Network Delay

index has been built on R_1 along the join condition between R_2 and R_1, and one on R_6 based on the join condition between R_5 and R_6.



R_2
R_4
R_1
R_3
R_5
R_6
R_1 R_4 (Oracle 8i): 2-1GHz Pentium III CPU, 1G Memory.
R_3 R_6 (Oracle 8i): 800MHz Pentium III CPU, 512M Memory.
R_2 (Oracle 8i): 2-450MHz Pentium III CPU, 512M Memory.
R_5 (Oracle 8i): Celeron 800MHz PC, 384M Memory.

Figure 11.13: A 6-Relation Join View Configuration

We change the number of source updates from 1,000 to 10,000. We compare the best batch maintenance plans with that of the best grouping plans along with their cost estimations. Figure 11.14 shows the performance changes. Again, we see that the cost estimation continues to capture the actual performance difference. We also see that the best grouping plans in this case are almost 3 times better than the best batch maintenance plans. This is because the grouping plan has a lot less maintenance queries (joins over distributed data sources) to process. This reduction of the number of maintenance queries helps to improve overall maintenance performance.

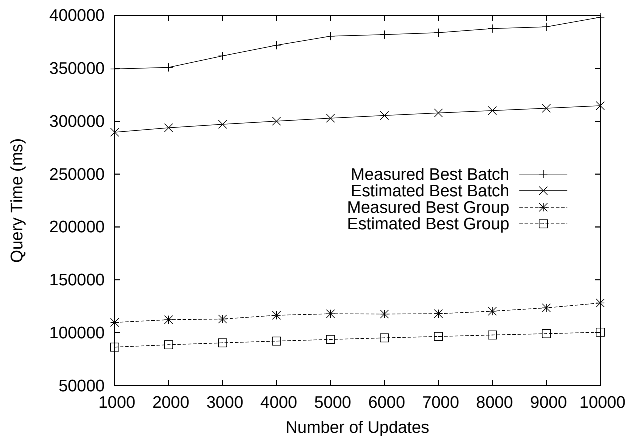


Figure 11.14: Batch vs. Group Plans Given Large Update Batches

11.4.4 Optimization Overhead

Due to the inherent complexity of the optimization problem, there is no 'efficient' algorithm guaranteed to find the optimal plan. The cost of the enumeration algorithm is small when the number of data sources is not large. For example, the batch enumeration algorithm for the four node view graph (V_2) defined in Figure 11.6 takes less than 30 ms in our test environment. The grouping path search algorithm takes less than 15 ms to enumerate the paths. Such optimization cost is almost negligible compared with the total maintenance cost.

As expected, when the number of nodes in the view graph increases, the enumeration time also increases dramatically. A view graph with 9 nodes and 12 join edges needs 12,228 ms to enumerate using the batch enumeration algorithm (Algorithm 8) in our current settings. Such enumeration algorithms are acceptable primarily in cases when the number of nodes in a view graph is not large. Also the actual view maintenance time is usually much larger than that of the optimization time.

Parameters	1 (greedy)	5	20	all (enumeration)
Opt. Time (ms)	51	100	271	11313
Plan Cost (ms)	357465.8	307108.8	293901.7	292246.8

Table 11.3: Optimization Cost vs. Quality of Solution

The proposed controlledDP algorithm (Algorithm 9) can be used to trade the optimization time with the quality of a solution. Here, we use the same 9 nodes and 12 join edges view graph. We now vary the setup of $pruneQueryPlan$ to keep the best 1, 5, 20, and all intermediate plans when building instances of each maintenance step. Seen from Table 11.4.4, the greedy approach only takes 51 ms seconds to find a maintenance plan. However, the plan is not as efficient compared with the optimal one. In this setup, we see that if we keep the best 20 intermediate plans in each recursive call of $controlledDP$, the cost of the result maintenance plan is close to the optimal one. While it only takes 271 ms, almost 50 times less than the enumeration approach, to generate this plan.

Chapter 12

Related Work

Maintaining materialized views under source updates is one of the important issues in information integration given the dynamic nature of the data sources [122]. This is because stale view extents may not help or even mislead user applications. Early work has studied incremental view maintenance assuming no concurrency [27, 73]. In approaches that need to send maintenance queries to the data sources, especially in a environment with autonomous data sources, concurrency problems can arise. Maintenance strategies such as [5, 18, 23, 24, 122, 123] have focused on handling anomaly problems due to concurrent updates among data sources.

Many algorithms have been proposed to date to maintain materialized views incrementally by issuing maintenance queries to the data sources [5, 11, 122, 123]. From both a resource and performance perspective, incrementally maintaining batches of updates is of particular interest. That is, changes to the sources can be buffered and propagated periodically to maintain the view extent. [27, 63, 80, 88, 93] propose algorithms to maintain materialized views incrementally using source-based batching. [93] proposed an asynchronous view maintenance algorithm using delta changes of data sources. [63] proposed a batch maintenance algorithm which can be applied to maintain a set of views. In our previous work [66], we have proposed a batch view maintenance strategy that works even when both data and schema changes may happen on data sources. However, all these existing approaches are only concerned with batching updates from the same data source. [64] introduces a delta propagation strategy that also reduces the number of maintenance queries to data sources. It is close to our proposed adjacent grouping approach. However, none of above have considered how to group heterogenous deltas to further reduce the number of maintenance queries. This is exactly the approach where we observed a major performance gain.

Parallel view maintenance strategies [65, 120] have also been investigated to improve the view maintenance performance. These works are orthogonal to our current research focus since they take the approach to have multiple maintenance plans run in parallel to improve view maintenance performance.

Similar to [63], Posse [84] introduced a view maintenance optimization framework. This work only focuses on the order in which these source deltas are to be installed (to be maintained). While in our work here, we now explore the optimizations at an even lower level. That is, given delta changes, we study how to order and compose maintenance queries to data sources to calculate the maintenance results more efficiently. Recent work [49] proposes a maintenance strategy with a response time constraint by exploiting the asymmetry among different components of the maintenance cost. While our approach in this work focuses on exploiting variations in the maintenance logic (i.e., reducing the number of maintenance queries), view definitions and environmental settings to improve view maintenance performance.

Moreover, all above solutions tend to focus on maintaining acyclic join views. They do not explore the diversity of the view definitions nor the dynamic nature of the environment to further optimize the view maintenance process.

Distributed query processing and the optimization has widely been studied for distributed environments [57]. For example, R* [74] extends System R [97] algorithm to optimize distributed queries. Garlic [44] focuses on optimizing queries across diverse data sources. Mariposa [103] applies an economic paradigm to optimize distributed queries. These works apply common search strategies such as enumeration and dynamic programming to generate efficient query plans due to the inherit complexity in optimizing complex queries.

However, in a view maintenance context, the processing model is slightly different than assumed in traditional distributed query processing. Here, each maintenance query is created by the view manager based on the source delta (or the intermediate results from previous maintenance queries). The results of each maintenance query are typically returned back to the view manager instead of directly communicating with other sources. We also prefer to combine join conditions whenever it is possible to reduce the accesses to distributed data sources. These heuristics help to reduce the overall search space explored by the optimization. Moreover, the grouping maintenance plans cannot be generated by the existing distributed query optimizers such as [44, 74, 103]. This is because the building block, the grouping maintenance logic, is developed in the view maintenance context by combining heterogeneous deltas to reduce the number of join queries. This grouping maintenance has not been investigated in the distributed query processing (nor its cost models).

Various optimization techniques, i.e., finding common sub-expressions, have been studied in the context of multiple query optimization [98, 90]. However, this does not apply in the context of view maintenance. For example, the common sub-expression such as $R_3 \bowtie R_4 \bowtie \ldots \bowtie R_n$ for the first two maintenance steps in Figure 10.3(a) is too expensive to evaluate. This is because each data source may be huge compared to the deltas.

Recent works on adaptive query processing [54, 55, 83, 110] aim to optimize distributed query processing by dynamically monitoring an execution plan and identifying points of sub-optimal performance. This is orthogonal to this work. Here we resort to cost estimation and employ a static cost-based optimizer to find an optimized maintenance plan. We borrow ideas from [121] on developing cost models for data sources, yet other cost models could be easily applied in our framework.

Part IV

Conclusions and Future Work

Chapter 13

Conclusions of This Dissertation

Materialized view computation and maintenance are two very basic services that need to be addressed to achieve the benefits of applying materialized views such as efficient access, reliable performance and high availability. These two services face scalability concerns due to (1) large number of data sources, (2) increasing size in each data source and (3) high volumes of source updates.

In this dissertation work, we aim to provide scalable solutions to these two services. We have proposed parallel and adaptive integration view computation strategies. We have provided novel grouping view maintenance algorithms as well as view maintenance optimization framework to generate optimized view maintenance plans. The conclusions of this dissertation work are listed below.

In part I, we have revisited the common assumption that has been taken by practically all prior work in the literature, namely, to pursue maximal pipelined parallelism when processing multi-join query processing in parallel. We have shown both experimentally and via a cost analysis that the introduction of independent parallelism at the cost of reducing the pipeline can greatly impact the parallel performance. A new type of parallel multi-join query processing strategy, namely, the segmented bushy processing strategy, has been proposed. A heuristic-driven optimization algorithm for generating the segmented bushy processing strategies incorporating independent parallelism and yet controlling its dependencies has been proposed in this part of the dissertation work.

A working distributed query engine called *PETL* has been implemented for this part of dissertation work. Extensive experimental studies have been conducted on a 10 high performance PC cluster connected by a local gigabit ethernet. Experimental studies confirm the effectiveness of our proposed processing strategy. As shown in the Section 3.4, the segmented bushy processing has an average of 50% improvement in terms of total processing time compared to the existing solution with a fully pipelined processing. This confirms our claim that maximal pipelined parallelism is not always the best.

The observation we made in this work also sheds some light on how best to optimize pipelined query plans in general given the optimization function is related to the total processing time. This optimization is bound to get increasing attention due to new and growing research areas such as continuous query processing [3].

In part II, we have extensively studied the tradeoffs and policies of adapting operator states of complex non-blocking multi-input operators to overcome run-time main memory overflow. We have proposed two state level adaptation strategies, namely, lazy-disk and active-disk, that both integrate the state spill and state relocation in memory constrained environments. That is, in environments where the aggregated main memory of the distributed system is still not sufficient for the query processing. Note that such integrations have not been carefully studied in the literature, yet, it is necessary in practical environment since the main memory of a distributed system remains limited. We have shown that run-time state relocation improves query throughput given sufficient overall main memory resource. This is because state relocation helps to maximally utilize available memory resource. We also have shown that active-disk strategy outperforms the lazy-disk one. This is because the active-disk strategy tends to have more productive states remain in main memory in the adaptation.

We also have investigated the dependency problem when adapting operator states for query plans with multiple state-intensive operators. Note that such query plans are common in a data integration context since the integration queries (views) are complex and stateful in nature. We have proposed two global state spill strategies, namely, global output and global output with penalty, that are designed to improve the run-time query throughput. We have shown that these two global adaptation strategies greatly outperform other solutions that do not consider the dependency among operators. All proposed state-level adaptation strategies have been designed and implemented in the D-Cape system [70].

In part III, we have taken a fresh new look at how to restructure a batch view maintenance plan to optimize the view maintenance performance when maintaining a large batch of source updates. This optimization is achieved by dramatically reducing the number of maintenance queries to remote data sources. A series of novel grouping maintenance strategies, namely, adjacent grouping and conditional grouping, have been proposed and implemented in a TxnWrap system [24]. Our experimental studies illustrate that maintenance performance can be significantly improved by having a smaller number of maintenance queries. In particular, our conditional grouping strategy is almost four times faster compared with the typical batch maintenance in a majority of the cases.

The state-of-the-art view maintenance literature tends to focus on the maintenance of acyclic join views [5, 63, 66, 93]. They do not exploit characteristics of the view definitions and the dynamic nature of the environmental settings to further optimize the view maintenance process. In this part of dissertation work, we have proposed view maintenance strategies to handle general join views by extending batching and grouping maintenance techniques. We have proposed a cost-based view maintenance optimization framework that is capable of generating optimized maintenance plans tuned to view definitions as well as particular environmental settings. We have shown that the optimization does make a difference when maintaining complex join views since (1) a lot more maintenance plans are available, and (2) different maintenance plans have rather different performance. We also see that grouping maintenance performs much better than batching maintenance in most of the cases we have considered. This is because a grouping maintenance plan requires much less maintenance queries than a batching plan. In other words, current distributed query optimization techniques if applied to view maintenance would lead to inferior results.

Moreover, this part of dissertation work also brings two independent areas one step closer, namely, the fusion of distributed query optimization and the area of view maintenance optimization. As one possible future work, a more fine-grained adaptive view maintenance optimizer, i.e., one that provides adaptivity at run-time within one single maintenance process by mixing batch and grouping techniques, could be designed. To support this, we need to collect run-time cost statistics in a more timely fashion. We also would need to devise strategies on how to migrate from a partially executed batch maintenance plan to a grouping plan, or vice verse. Also, the adaptation policies would need to consider when to adapt the maintenance process, and so on.

Chapter 14

Ideas for Future Work

14.1 State Spilling for Window Join Queries

The solutions for run-time operator state adaptations in this dissertation are primarily optimized for non-blocking queries with long running yet finite input streams. However, these solutions can also be extended to the continuous query processing context with possibly infinite input streams with window semantics [40, 56]. Note that the major difference here is how to handle the window constraints imposed in the operator when spilling states into disks.

For example, the states of a window join operator are usually not monotonically increasing. This is because the states beyond the window constraint are purged from the operator [40, 56, 102]. However, if we start spilling operator states into disks, then purging of outdated states cannot be performed purely based on the window constraints. This is because partial operator states may have been pushed into disks and temporarily inactive. Thus, missing query results can be generated from the main memory resident states that qualified to be purged based on the window constraint and the spilled states.

One solution for handling window constraints is to also push these out-of-window states into disks whenever there are operator states from the same partition that are disk resident at that time. The following research issues need to be addressed:

- Design solutions to incorporate the window constraints in the state spill process. That is, we need to coordinate among state purge, state spill and state cleanup processes based on the window semantics.
- Given the fact that operator states may not be monotonically increasing, it is necessary to design policies to perform run-time state cleanup processes depending on the memory availability. The adaptation policies may need to be designed to incorporate various factors such as the stream input rate, operator window size, memory usage, partition productivity.

14.2 Pair-Wise Adaptation or Diffusion

Two extreme cases that can be considered when deciding how to adapt states in the state relocation process are the pair-wise and the diffusion approaches. To illustrate, we briefly discuss these two approaches using the sample load bar chart depicted in Figure 14.1. In Figure 14.1, L_i represents the current load of machine m_i. The x-axis represents the machines, while the y-axis denotes the difference $L_i - \overline{L}$. Here, a positive number denotes that the load is higher than the average, while a negative number represents that the load is lower than the average.

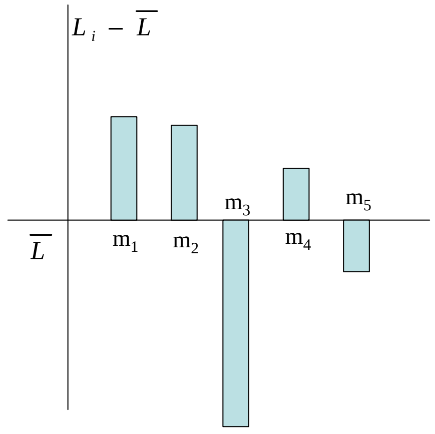


Figure 14.1: Load Representation Example

- **Pair-Wise Approach**: We take the pair-wise approach in this dissertation (see Chapter 6). Note that many other run-time adaptation works also apply this type of approach, for instance, Flux [99] and Borealis [118]. The basic idea in this approach is to choose a pair of processors, one with the largest load, and the other with the least load. The operator states will be moved from the most loaded processor to the least loaded one. This type of approach usually has one round of balancing in each adaptation process. It only involves two processors that is likely to have only a limited effect on the overall processing. However, this may not result in a balanced load in each adaptation even if the environment stabilize, it still takes several rounds to get a balanced load distribution. For example, we may pair m_1 with m_3 as shown in Figure 14.1. As can be seen, the load after this adaptation would still not be balanced.

- **Diffusion Algorithm:** A diffusion-like algorithm [89] usually has no restriction on how the states are moved among processors. That is, many processors may be involved in each adaptation process. For example, load in m_1 can be moved to all the other nodes except m_1. The diffusion algorithm can lead to a better distribution of load in a short time, i.e., in one optimization cycle. However, it may incur high adaptation costs since multiple processors are involved in each adaptation.

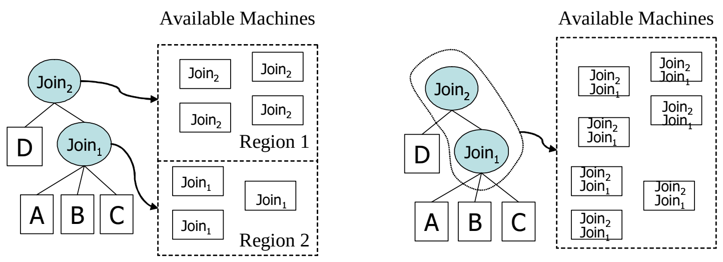
Such tradeoffs can be explored in the future between the adaptation overhead vs. the quality of the adaptation process. In general, the following two issues can be investigated. (1) One could add certain restrictions to the diffusion algorithm to speed up the adaptation process. For example, we can divide the processing nodes into two categories; one category contains all overloaded processors with $L_i > \overline{L}$, while the other one contains all underloaded ones with $L_i < \overline{L}$. Then, the load movement could only appear between these two categories. (2) Even a perfect balance may not be good enough in some cases. For example, the processors with high load may experience a high load quickly again even if the adaptation achieves a balanced load distribution. Thus, we may even consider to distribute the load (states) even the system may experience unbalanced load temporarily to plan for future.

14.3 Distributed Query Plan Adaptation

14.3.1 Impact of Initial Distribution

Given a query plan with multiple state-intensive operators, various ways of distributing the query plan among available machines exist. For example, as shown in Figure 14.2(a), we partition all available machines into non-overlapping regions. Then we divide the query plan according to the state-intensive operators. We distribute one state-intensive operator only into one region of machines.

On the other hand, as shown in Figure 14.2(b), we can have all the state intensive operators evenly distributed into all available machines. In this case, each machine has the same amount of state intensive operators activated with each processing only partial inputs.



(a) Regional Partitioning (b) Mixed Distribution

Figure 14.2: Various Initial Distribution Methods

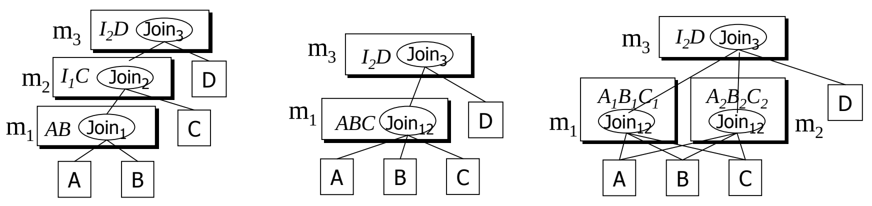
Clearly, many other initial query plan distribution algorithm can be designed. Different initial distribution algorithms may exhibit different performance due to differences in intercommunication among different machines and the overhead of the split operator (related to optimal degree of parallelism). On the other hand, different initial distribution algorithm may even impact the selection of possible adaptation methods.

14.3.2 Transforming Distributed Query Plans

The distributed query plan itself may also need to be optimized for certain metrics by rewriting the query plan shape itself. This is an as of now unexplored area of research in stream processing context. Besides typical query plan rewriting strategies which consider reordering of operators [50], and deciding between m-way joins and binary join trees, we can focus on exploiting opportunities of adapting query plans across inter-operator and intra-operator parallelism boundaries. Figure 14.3 illustrates the basic idea of this distributed query plan restructuring. For example, Figure 14.3(a) shows a binary join tree with each join allocated to one machine. Here, the letters AB, I_1C, and I_2D represent the operator states that have to be stored in the join operators, while I_1 and I_2 denote the intermediate results, and m_1, m_2, and m_3 represent available machines. As shown in Figure 14.3(b), we would merge $Join_1$ and $Join_2$ into a 3-way join $Join_{12}$ if we observe that large intermediate results are transferred from m_1 to m_2 and then stored in m_2. After that, state ABC becomes the only state that needs to be stored. However, the state ABC may no longer fit into one machine. Thus, we need to partition the state into multiple machines as shown in Figure 14.3(c).

Note that such transformation of a distributed query plan can be achieved by a set of basic state relocation services as discussed in Chapter 6.

The policy of adapting a distributed query plan also needs to be de-



(a) Binary Query Plan (b) M-way Query Plan (c) Partitioned Query Plan

Figure 14.3: Distributed Query Plan Restructuring

signed carefully. For example, (1) what composes a good plan given current statistics information? (2) what is the adaptation cost (i.e., states to move, intermediate results to drop or build) for transforming the current distributed query plan to the new one. Moreover, the cost of statistics gathering and maintaining also needs to be considered in this adaptation process.

Several interesting questions that need to be further explored for this distributed query plan adaptation are listed below.

- **Parallel Adaptation:** In a fine-grained adaptation (moving states), multiple adaptations in different machines can be done in parallel. Thus, the overall adaptation cost can be shared.

- **Ordered Adaptation:** Given a pipeline query tree with a regional partitioning approach, it may make a difference in terms of performance which regions we choose to adapt first. This may be similar to the interdependency in spilling operator states among pipelined operators as discussed in Chapter 7.

- **Adaptation Decision:** The adaptation decision is one major issue that still need to be further studied. For example, questions such as investigating cost models and choosing criteria to initialize adaptation are not trivial to address in this partitioned parallel environment.

14.4 Half-Baked Thoughts

In this section, we provide several high-level thoughts on possible future work ideas. Note that those ideas have not been carefully investigated.

Various System Load Measurements. The solutions we have provided for run-time adaptations are primarily based on system main memory usage having the assumption of enough system CPU processing capability. However, system CPU and main memory usage are closely related. Thus, issues such as how to combine these two factors into one system load measurement function and how the changing (spilling/relocating) of operator states relates to the system load need to be further investigated. These may lead to the design of new run-time operator state adaptation policies.

System Load Predictions. In this dissertation, we basically use the average to represent the current system load (see Chapters 6 and 7). We also use this average to indicate the future behavior of the system. However, the current (and the prediction) system load can be described more precisely based on a time series of previous system loads. Techniques in time series analysis [100] can be applied to design a better prediction function. This in turn may affect the designing of adaptation policies such as when to adapt.

Mix Load Shedding and Spilling. Previous work in stream processing resorts to load shedding [107] to handle run-time system resource shortage. In this dissertation, we instead propose state spill, which only temporarily push states into disks. However, it may be useful to combine these two techniques. The combination of shedding and spilling can result in an increasing accuracy of query results by allowing out-of-order results. This may be better than pure shedding or spilling in certain applications.

Semantic Spilling. Instead of simple productivity value that is based on query throughput, we can identify the usefulness of operator states based on certain semantic values related to applications. Thus, the run-time spilling of operator states can be based on their semantic values such as values related to the QoS requirements. The issues here are (1) how to define such requirements, and (2) how to relate such requirements to each individual partition or even each tuple.

Other Parallel Architecture. This dissertation mainly focuses on a shared-nothing architecture, in particular, a local cluster with high-performance PCs connected by a high-speed network. In this architecture, each machine in the system has its own main memory, CPU, and disks. The communication among different machines is achieved by a high-speed network. However, other parallel architectures such as shared-memory and shared-disk exist. The impact of these architectures on proposed parallel computation and adaptation strategies is one interesting future work idea to be investigated.

Relating View Maintenance with Non-Blocking Join Processing. Incremental batch view maintenance process (see Chapter 9) is similar to the non-blocking symmetric m-way hash join processing. Thus, the grouping view maintenance algorithm we have described in Chapter 10 may be applied to the non-blocking join processing. That is, instead of probing and hashing per input tuple, we can defer the join processing using deltas. New optimization techniques may exist even in this pure main memory based join processing.

Views with Other State-Intensive Operators. In this dissertation work, we basically focus on computing and maintaining integration views (multi-join queries). However, there are other types of view definitions, i.e., views with aggregation operators. How the parallel computation, adaptation, and grouping maintenance techniques be extended can be another promising future work direction.

Appendix A

General Partitioned M-Way Join Processing

In Chapter 5, we assume a partitioned m-way join query that has the same join column among different join conditions. Here, we lift this restriction and discuss how to support the processing of a general partitioned m-way join query.

Note that given an m-way join query for which join columns among different join conditions are not the same, simply applying one partition function per input stream does not work well. For example, we have a three way join that is defined $A.A_1 = B.B_1$ and $B.B_2 = C.C_1$, as illustrated in Figure A.1. Here A, B, and C represent input streams (join relations) while A_1, B_1, B_2, and C_1 are join columns from A, B, and C respectively.

Note that input stream B has two different join columns involved in the join conditions. Thus, if we partition input tuples from stream B based on

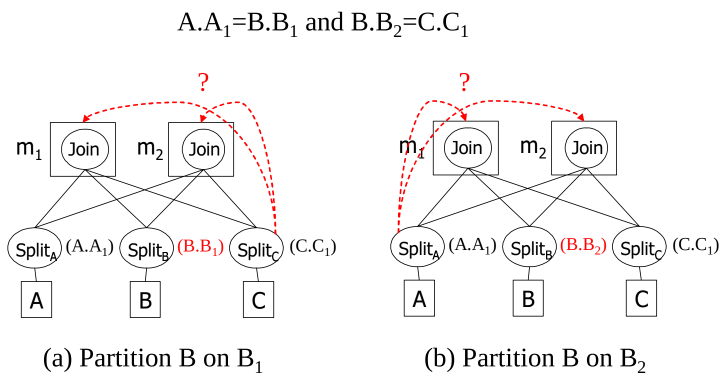


Figure A.1: Partitioning General M-way Joins

join column B_1, as shown Figure A.1 (a), then tuples from input stream C, which are partitioned based on C_1, cannot find the corresponding partitions of input stream B. For example, one tuple t_c from C may have been partitioned into machine m_1 using $C.C_1$, while state partitions of stream B in m_1 are organized based on B_1. However, t_c does not have information about the partitioning join column $B.B_1$, which helps t_c to identify the corresponding matching tuples from input stream B. Thus, tuple t_c cannot found the corresponding matching tuples from B in this case. Note that even a whole scan of all state partitions of B in m_1 still does not address the problem. This is because the matching tuples from B for t_c can be in any other machines, in this case, in machine m_2. Similarly, if we partition B based on B_2, then the tuples from A cannot find the matching tuples, as illustrated in Figure A.1 (b). Clearly, we do not want to scan all state partitions of stream B in all machines whenever we need to probe B since that is a rather expensive process.

For simplicity, we refer to input streams with multiple join columns involved in the join conditions as *m-streams*. We propose two alternate solutions to perform the above partitioned general join query processing: (1) Replicate m-stream inputs to all available machines, or (2) partition m-stream inputs applying multiple join column partition functions.

Replicating Input Streams. The basic idea of this approach is illustrated in Figure A.2. That is, instead of partitioning m-streams (input stream B in this example), we instead send input tuples from B into all machines with partitioned states. In that case, both input tuples from A and C can find the matching tuples from B by a full scan. Note that in such environment, run-time repartitioning of intermediate results is necessary. For example, after we get the intermediate result of $B \bowtie C$, we have to apply the $Split_A$ function on B_1 of the intermediate results to find the corresponding partition of A. For each tuple in the intermediate result set, this partition can A be in a different machine. Thus, the intermediate results may be redirected to other machines at run time.

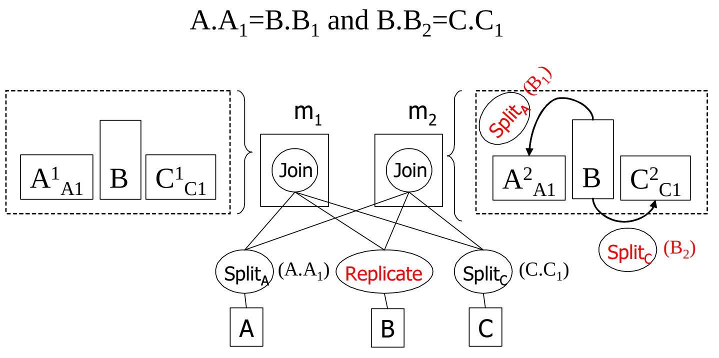


Figure A.2: Replicating m-streams

Generating Multiple Partitions. The second approach is to partition m-streams based on all join columns that involved in the join conditions, as illustrated in Figure A.3. In this example, each input tuple from B is partitioned twice. We first apply partition function based on the value of B_1 and send the tuple to the corresponding partitions. After that, we again apply the partition function that based on B_2 and send it to the corresponding machines. Note that we may have different partition IDs for one single tuple of B, and even if we have the same partition ID, the tuple can still be sent to different machines depending on the mapping functions of partition IDs and machines. Given two types of state partitions of input stream B in each machine, we can always find the corresponding matching tuples. As in the replication approach, run-time repartition and redirection of intermediate results also have to be applied.

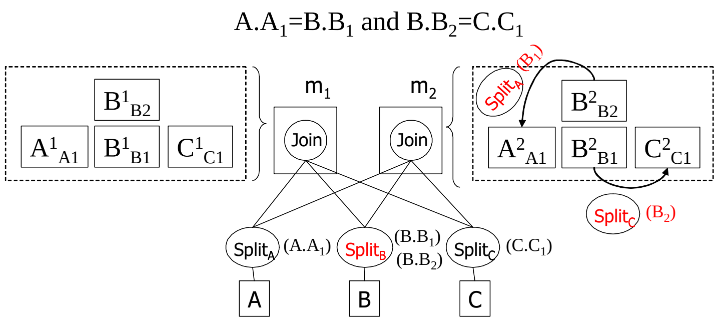


Figure A.3: Generating Multiple Partitions

Note that when multiple join columns in the m-stream are correlated, then we may be able to organize multiple partitions of the m-stream to save storage space in this multiple partitions approach compared with the replication approach.

Bibliography

[1] *Exegesis of DBC/1012 and P-90 - Industrial Supercomputer Database Machines*, 1992.

[2] D. Abadi, Y. Ahmad, and et. al. The design of the borealis stream processing engine. In *Proceedings CIDR*, page to appear, 2005.

[3] D. J. Abadi, D. Carney, and et al. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, 2003.

[4] F. N. Afrati, C. Li, and J. D. Ullman. Generating efficient plans for queries using views. In *SIGMOD*, pages 319–330, 2001.

[5] D. Agrawal, A. E. Abbadi, A. Singh, and T. Yurek. Efficient View Maintenance at Data Warehouses. In *Proceedings of SIGMOD*, pages 417–427, 1997.

[6] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated Selection of Materialized Views and Indexes in SQL Databases. In *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases*, pages 496–505. Morgan Kaufmann, 2000.

[7] I. Ahmad and Y.-K. Kwok. On exploiting task duplication in parallel program scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 9(9):872–892, 1998.

[8] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proceedings of ACM PODS*, pages 1–16, 2002.

[9] B. Babcock, S. Babu, R. Motwani, and M. Datar. Chain: operator scheduling for memory minimization in data stream systems. In *ACM SIGMOD*, pages 253–264, 2003.

[10] T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. In *Scientific American*, May 2001.

[11] J. A. Blakeley, P.-A. Larson, and F. W. Tompa. Efficiently Updating Materialized Views. In *Proceedings of SIGMOD*, pages 61–71, May 1986.

[12] H. Boral, W. Alexander, L. Clay, G. P. Copeland, S. Danforth, M. J. Franklin, B. E. Hart, M. Smith, and P. Valduriez. Prototyping bubba, a highly parallel database system. *IEEE TKDE*, 2(1):4–24, 1990.

[13] L. Bouganim, D. Florescu, and P. Valduriez. Dynamic load balancing in hierarchical parallel database systems. In *The VLDB Journal*, pages 436–447, 1996.

[14] L. Bouganim, O. Kapitskia, and P. Valkuriez. Memory-adaptive scheduling for large query execution. In *Proceedings of the seventh international conference on Information and knowledge management*, pages 105–115. ACM Press, 1998.

[15] S. Chandrasekaran and M. J. Franklin. Streaming queries over streaming data. In *proceedings of VLDB*, pages 203–214, 2002.

[16] S. Chaudhuri and U. Dayal. An Overview of Data Warehousing and OLAP Technology. *SIGMOD Record*, 26(1):65–74, 1997.

[17] C. Chekuri. *Approximation Algorithms for Scheduling Problems*. PhD thesis, Stanford University, Aug 1998.

[18] J. Chen, S. Chen, and E. A. Rundensteiner. A Transactional Model for Data Warehouse Maintenance. In *ER'02*, pages 247–262, Sep 2002.

[19] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. Niagaracq: a scalable continuous query system for internet databases. In *ACM SIGMOD*, pages 379–390, 2000.

[20] J. Chen, X. Zhang, S. Chen, K. Andreas, and E. A. Rundensteiner. DyDa: Data Warehouse Maintenance under Fully Concurrent Environments. In *Proceedings of SIGMOD Demo Session*, page 619, 2001.

[21] M.-S. Chen, M.-L. Lo, P. S. Yu, and H. C. Young. Using segmented right-deep trees for the execution of pipelined hash joins. In *VLDB*, pages 15–26, 1992.

[22] M.-S. Chen, P. S. Yu, and K.-L. Wu. Scheduling and processor allocation for parallel execution of multi-join queries. In *Proceedings of ICDE*, pages 58–67, 1992.

[23] S. Chen, J. Chen, X. Zhang, and E. A. Rundensteiner. Detection and Correction of Conflicting Sources Updates for View Maintenance. In *ICDE 2004*, pages 436–448, Apr 2004.

[24] S. Chen, B. Liu, and E. A. Rundensteiner. Multiversion Based View Maintenance over Distributed Data Sources. *ACM Transactions on Database Systems (TODS)*, 29(4):675–709, 2004.

[25] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. Zdonik. Scalable distributed stream processing. In *2003 CIDR Conference*, 2003.

[26] R. Chirkova, A. Y. Halevy, and D. Suciu. A formal perspective on the view selection problem. *The VLDB Journal*, 11(3):216–237, 2002.

[27] L. S. Colby, T. Griffin, L. Libkin, I. S. Mumick, and H. Trickey. Algorithms for Deferred View Maintenance. In *Proceedings of SIGMOD*, pages 469–480, 1996.

[28] R. C. Correa, A. Ferreira, and P. Rebreyend. Scheduling multiprocessor tasks with genetic algorithms. *IEEE Transactions on Parallel and Distributed Systems*, 10(8):825–837, 1999.

[29] L. Crum. University of Michigan Digital Library Project. *Communications of the ACM*, 38(4):63–65, April 1995.

[30] Y. Cui and J. Widom. Lineage tracing for general data warehouse transformations. In *Proceedings VLDB*, pages 471–480, 2001.

[31] D. DeWitt and J. Gray. Parallel database systems: the future of high performance database systems. *Communications of the ACM*, 35(6):85–98, 1992.

[32] D. J. DeWitt, S. Ghandeharizadeh, D. Schneider, A. Bricker, H.-I. Hsiao, and R. Rasmussen. The gamma database machine project. *IEEE TKDE*, 2(1):44–62, 1990.

[33] D. J. DeWitt, J. F. Naughton, D. A. Schneider, and S. Seshadri. Practical skew handling in parallel joins. In *Proceedings of VLDB*, pages 27–40, 1992.

[34] L. Ding, N. Mehta, E. Rundensteiner, and G. Heineman. Joining punctuated streams. In *Proceedings of the EDBT*, pages 587–604, 2004.

[35] S. Ganguly, W. Hasan, and R. Krishnamurthy. Query optimization for parallel execution. In *Proceedings of ACM SIGMOD*, pages 9–18. ACM Press, 1992.

[36] H. García-Molina, W. Labio, J. L. Wiener, and Y. Zhuge. Distributed and Parallel Computing Issues in Data Warehousing . In *Symposium on Principles of Distributed Computing*, page 7, 1998.

[37] H. García-Molina, W. J. Labio, and J. Yang. Expiring data in a warehouse. In *Proc. 24th Int. Conf. Very Large Data Bases, VLDB*, pages 500–511, 24–27 August 1998.

[38] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. WH Freeman and Co., 1979.

[39] M. N. Garofalakis and Y. E. Ioannidis. Multi-dimensional resource scheduling for parallel queries. In *Proceedings of ACM SIGMOD*, pages 365–376. ACM Press, 1996.

[40] L. Golab and M. Tamer. Processing sliding window multi-joins in continuous queries over data streams. In *Proceedings of VLDB*, pages 500–511, 2003.

[41] G. Graefe. Encapsulation of parallelism in the volcano query processing system. In *Proceedings of ACM SIGMOD*, pages 102–111, 1990.

[42] A. Gupta and I. Mumick. Maintenance of Materialized Views: Problems, Techniques, and Applications. *IEEE Data Engineering Bulletin*, 18(2):3–19, 1995.

[43] H. Gupta. Selection of views to materialize in a data warehouse. In *Database Theory - ICDT '97, 6th International Conference*.

[44] L. M. Haas, D. Kossmann, E. L. Wimmers, and J. Yang. Optimizing Queries Across Diverse Data Sources. In *Proceedings of VLDB*, pages 276–285, 1997.

[45] A. Y. Halevy. Answering queries using views: A survey. *VLDB Journal: Very Large Data Bases*, 10(4):270–294, 2001.

[46] J. Han and M. Kamber. *Data Mining Concepts and Techniques*. Morgan Kaufmann, 2001.

[47] W. Hasan. *Optimization of SQL Queries for Parallel Machines*. PhD thesis, Stanford University, Dec 1995.

[48] W. Hasan and R. Motwani. Optimization algorithms for exploiting the parallelism-communication tradeoff in pipelined parallelism. In *Proceedings of VLDB*, pages 36–47, 1994.

[49] H. He, J. Xie, J. Yang, and H. Yu. Asymmetric Batch Incremental View Maintenance. In *Proceedings of ICDE*, pages 106–117, 2005.

[50] J. M. Hellerstein, M. J. Franklin, S. Chandrasekaran, A. Deshpande, K. Hildrum, S. Madden, V. Raman, and M. A. Shah. Adaptive query processing: Technology in evolution. *IEEE Data Engineering Bulletin*, 23(2):7–18, 2000.

[51] W. Hong and M. Stonebraker. Optimization of parallel query execution plans in xprs. In *Proceedings of PDIS*, pages 218–225, 1991.

[52] J.-J. Hwang, Y.-C. Chow, F. D. Anger, and C.-Y. Lee. Scheduling precendence graphs in systems with interprocessor communication time. *SIAM Journal of Computing*, 18(2):244–257, 1989.

[53] W. Inmon. *Building the Data Warehouse*. John Wiley and Sons, 1996.

[54] Z. G. Ives, D. Florescu, M. Friedman, A. Y. Levy, and D. S. Weld. An Adaptive Query Execution System for Data Integration. In *Proceedings of SIGMOD*, pages 299–310, 1999.

[55] N. Kabra and D. J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, pages 106–117. ACM Press, 1998.

[56] J. Kang, J. F. Naughton, and S. Viglas. Evaluating window joins over unbounded streams. In *Proceedings of ICDE*, pages 341–352, 2003.

[57] D. Kossmann. The State of the Art in Distributed Query Processing. *ACM Computing Surveys (CSUR)*, 32(4):422–469, 2000.

[58] D. Kossmann and K. Stocker. Iterative dynamic programming: a new class of query optimization algorithms. *ACM Trans. Database Syst.*, 25(1):43–82, 2000.

[59] Y.-K. Kwok. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys (CSUR)*, 31(4):406–471, 1999.

[60] Y.-K. Kwok and I. Ahmad. Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 7(5):506–520, 1996.

[61] Y.-K. Kwok and I. Ahmad. Fastest: A practical low-complexity algorithm for compile-time assignment of parallel programs to multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 10(2):147–159, 1999.

[62] W. J. Labio, J. L. Wiener, H. Garcia-Molina, and V. Gorelik. Efficient resumption of interrupted warehouse loads. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 46–57. ACM Press, 2000.

[63] W. J. Labio, R. Yerneni, and H. García-Molina. Shrinking the Warehouse Updated Window. In *Proceedings of SIGMOD*, pages 383–395, June 1999.

[64] K. Y. Lee, J. H. Son, and M. H. Kim. Efficient Incremental View Maintenance in Data Warehouses. In *CIKM'01*, pages 349–356, November 2001.

[65] B. Liu, S. Chen, and E. A. Rundensteiner. A Transactional Approach to Parallel Data Warehouse Maintenance. In *DAWAK'02*, pages 307–317, Sep 2002.

[66] B. Liu, S. Chen, and E. A. Rundensteiner. Batch Data Warehouse Maintenance in Dynamic Environments. In *CIKM'02*, pages 68–75, Nov 2002.

[67] B. Liu and E. A. Rundensteiner. Revisiting Parallel Multi-Join Query Processing via Hashing . Technical Report WPI-CS-TR-05-05, Worcester Polytechnic Institute, February 2005.

[68] B. Liu and E. A. Rundensteiner. Revisiting Parallel Multi-Join Query Processing via Hashing . In *Proceedings of VLDB*, to appear, 2005.

[69] B. Liu, E. A. Rundensteiner, and D. Finkel. Restructuring View Maintenance Plans for Large Update Batches. Technical Report WPI-CS-TR-03-29, WPI, 2003.

[70] B. Liu, Y. Zhu, and et. al. A Dynamically Adaptive Distributed System for Processing Complex Continuous Queries. In *VLDB Demo*, to appear, 2005.

[71] M.-L. Lo, M.-S. S. Chen, C. V. Ravishankar, and P. S. Yu. On optimal processor allocation to support pipelined hash joins. In *Proceedings of ACM SIGMOD*, pages 69–78, 1993.

[72] H. Lu, K.-L. Tan, and M.-C. Sahn. Hash-based join algorithms for multiprocessor computers with shared memory. In *Proceedings of VLDB*, pages 198–209, 1990.

[73] J. J. Lu, G. Moerkotte, J. Schue, and V. S. Subrahmanian. Efficient Maintenance of Materialized Mediated Views. In *SIGMOD*, pages 340–351, 1995.

[74] L. F. Mackert and G. M. Lohman. R* optimizer validation and performance evaluation for local queries. In *Proceedings of ACM SIGMOD*, pages 84–95, 1986.

[75] S. Madden and M. J. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *ICDE*, pages 555–566, 2002.

[76] S. Madden, M. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *ACM SIGMOD*, pages 49–60, 2002.

[77] T. P. Martin, P.-A. Larson, and V. Deshpande. Parallel hash-based join algorithms for a shared-everything. *IEEE TKDE*, 6(5):750–763, 1994.

[78] M. Mehta and D. J. DeWitt. Data placement in shared-nothing parallel database systems. *The VLDB Journal*, 6(1):53–72, 1997.

[79] M. Mokbel, M. Lu, and W. Aref. Hash-merge join: A non-blocking join algorithm for producing fast and early join results. In *ICDE*, page 251, 2004.

[80] I. Mumick, D. Quass, and B. Mumick. Maintenance of Data Cubes and Summary Tables in a Warehouse. In *Proceedings of SIGMOD*, pages 100–111, May 1997.

[81] B. Nag and D. J. DeWitt. Memory allocation strategies for complex decision support queries. In *Proceedings of the seventh international conference on Information and knowledge management*, pages 116–123. ACM Press, 1998.

[82] J. Neter, M. Kunter, C. Nachtsheim, and W. Wasserman. *Applied Linear Statistical Models*. Times Mirror Pub., 1996.

[83] K. W. Ng, Z. Wang, R. R. Muntz, and S. Nittel. Dynamic query re-optimization. In *Statistical and Scientific Database Management*, pages 264–273, 1999.

[84] K. O'Gorman, D. Agrawal, and A. E. Abbadi. Posse: A framework for optimizing incremental view maintenance at data warehouse. In *Data Warehousing and Knowledge Discovery*, pages 106–115, 1999.

[85] H. H. Pang, M. J. Carey, and M. Livny. Managing memory for real-time queries. In *Proceedings of the 1994 ACM SIGMOD international conference on Management of data*, pages 221–232. ACM Press, 1994.

[86] G.-L. Park, B. Shirazi, and J. Marquis. Dfrn: A new approach for duplication based scheduling for distributed memory multiprocessor systems. In *Proceedings of International Conference of Parallel Processing*, pages 157–166, 1997.

[87] D. Quass, A. Gupta, I. S. Mumick, and J. Widom. Making Views Self-Maintainable for Data Warehousing. In *Conference on Parallel and Distributed Information Systems*, pages 158–169, 1996.

[88] D. Quass and J. Widom. On-Line Warehouse View Maintenance. In *Proceedings of SIGMOD*, pages 393–400, 1997.

[89] T. Rotaru and H.-H. N&#228;geli. Dynamic load balancing by diffusion in heterogeneous systems. *J. Parallel Distrib. Comput.*, 64(4):481–497, 2004.

[90] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and Extensible Algorithms for Multi Query Optimization. In *Proceedings of SIGMOD*, pages 249–260, 2000.

[91] E. A. Rundensteiner, L. Ding, and et. al. Continuous Query Engine with Heterogeneous-Grained Adaptivity. In *VLDB Demo*, pages 1353–1356, 2004.

[92] Sagent Technology. http://www.sagent.com.

[93] K. Salem, K. S. Beyer, R. Cochrane, and B. G. Lindsay. How To Roll a Join: Asynchronous Incremental View Maintenance. In *SIGMOD*, pages 129–140, 2000.

[94] D. A. Schneider and D. J. DeWitt. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. In *Proceedings of ACM SIGMOD*, pages 110–121, 1989.

[95] D. A. Schneider and D. J. DeWitt. Tradeoffs in processing complex join queries via hashing in multiprocessor database machines. In *Proceedings of VLDB*, pages 469–480, 1990.

[96] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. pages 82–93, 1988.

[97] P. G. Selinger, M. M. Astrahan, and etal. Access path selection in a relational database management system. In *Proceedings of SIGMOD*, pages 23–34, 1979.

[98] T. K. Sellis. Multiple-query Optimization. *ACM Transactions on Database Systems (TODS)*, 13(1):23–52, 1988.

[99] M. A. Shah, J. M. Hellerstein, and et. al. Flux: An adaptive partitioning operator for continuous query systmes. In *ICDE*, pages 25–36, 2003.

[100] R. H. Shumway and D. S.Stoffer. *Time Series Analysis and Its Applications*. Springer, 2000.

[101] J. Srivastava and G. Elsesser. Optimizing multi-join queries in parallel relational databases. In *Proceedings of the 2nd PDIS*, pages 84–92, 1993.

[102] U. Srivastava and J. Widom. Memory-limited execution of windowed stream joins. In *Proceedings of VLDB*, pages 324–335, 2004.

[103] M. Stonebraker, P. Aoki, A. Pfeffer, A. Sah, J. Sidell, C. Staelin, and A. Yu. Mariposa: A Wide-area Distributed Database System. *VLDB Journal*, 5(1):48–63, 1996.

[104] T. Sutherland, B. Liu, M. Jbantova, and E. Rundensteiner. D-CAPE: Distributed and Self-Tuned Continuous Query Processing. In *CIKM Poster*, to appear, 2005.

[105] T. Sutherland and E. Rundensteiner. D-CAPE: A Self-Tuning Continuous Query Plan Distribution Architecture. Technical report, Worcester Polytechnic Institute, Dept. of Computer Science, July 2004.

[106] K.-L. Tan and H. Lu. Processing multi-join query in parallel systems. In *Proceedings of ACM Symposium on Applied computing*, pages 283–292, 1992.

[107] N. Tatbul, U. etintemel, S. B. Zdonik, M. Cherniack, and M. Stonebraker. Load shedding in a data stream manager. In *VLDB*, pages 309–320, 2003.

[108] TPC. TPC-H Benchmark Standard Specification. *http://www.tpc.org/tpch/*.

[109] T. Urhan and M. J. Franklin. Xjoin: A reactively-scheduled pipelined join operator. *IEEE Data Engineering Bulletin*, 23(2):27–33, 2000.

[110] T. Urhan, M. J. Franklin, and L. Amsaleg. Cost-based query scrambling for initial delays. In *Proceedings of the ACM SIGMOD*, pages 130–141, 1998.

[111] S. Viglas, J. F. Naughton, and J. Burger. Maximizing the output rate of multi-way join queries over streaming information sources. In *VLDB*, pages 285–296, 2003.

[112] Vision Solutions. The Need for Real Time Data Warehousing. http://www.visionsolutions.com/docs/data_warehouse.pdf.

[113] C. Wang and M.-S. Chen. On the Complexity of Distributed Query Optimization. *IEEE TKDE*, 8(4):650–662, 1996.

[114] J. Widom. Research Problems in Data Warehousing. In *Proceedings of International Conference on Information and Knowledge Management*, pages 25–30, November 1995.

[115] A. N. Wilschut and P. M. G. Apers. Dataflow query execution in a parallel main-memory environment. *Distrib. Parallel Databases*, 1(1):103–128, 1993.

[116] A. N. Wilschut, J. Flokstra, and P. M. G. Apers. Parallelism in a main-memory dbms: The performance of prisma/db. In *Proceedings of VLDB*, pages 521–532, 1992.

[117] A. N. Wilschut, J. Flokstra, and P. M. G. Apers. Parallel evaluation of multi-join queries. In *Proceedings of ACM SIGMOD*, pages 115–126, 1995.

[118] Y. Xing, S. Zdonik, and J.-H. Hwang. Dynamic load distribution in the borealis stream processor. In *Proceedings of ICDE*, pages 791–802, 2005.

[119] T. Yang and A. Gerasoulis. Dsc: Scheduling parallel tasks on an unbounded number of processors. *IEEE Transactions on Parallel and Distributed Systems*, 5(9):951–967, 1994.

[120] X. Zhang, E. A. Rundensteiner, and L. Ding. Parallel Multi-Source View Maintenance. *VLDB Journal*, 13(1):22–48, 2004.

[121] Q. Zhu, Y. Sun, and S. Motheramgari. Developing Cost Models with Qualitative Variables for Dynamic Multidatabase Environments. In *Proceedings of ICDE*, pages 413–424, 2000.

[122] Y. Zhuge, H. García-Molina, J. Hammer, and J. Widom. View Maintenance in a Warehousing Environment. In *Proceedings of SIGMOD*, pages 316–327, May 1995.

[123] Y. Zhuge, H. García-Molina, and J. L. Wiener. The Strobe Algorithms for Multi-Source Warehouse Consistency. In *Parallel and Distributed Information Systems*, pages 146–157, 1996.

[124] M. Ziane, M. Zat, and P. Borla-Salamet. Parallel query processing with zigzag trees. *The VLDB Journal*, 2(3):277–302, 1993.