2006-08-23

Dynamic Optimization and Migration of Continuous Queries Over Data Streams

Yali Zhu
Worcester Polytechnic Institute

Dynamic Optimization and Migration of Continuous Queries Over Data Streams

by

Yali Zhu

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 17, 2006

APPROVED:

Professor Elke A. Rundensteiner Advisor	Professor George Heineman Committee Member
Professor Murali Mani Committee Member	Dr. Volker Markl IBM Almaden Research Center External Committee Member
Professor Michael Gennert Head of Department	

Abstract

Continuous queries process real-time streaming data and output results in streams for a wide range of applications. Due to the fluctuating stream characteristics, a streaming database system needs to dynamically adapt query execution. This dissertation proposes novel solutions to continuous query adaptation in three core areas, namely dynamic query optimization, dynamic plan migration and partitioned query adaptation.

Runtime query optimization needs to efficiently generate plans that satisfy both CPU and memory resource constraints. Existing work focus on minimizing intermediate query results, which decreases memory and CPU usages simultaneously. However, doing so cannot assure that both resource constraints are being satisfied, because memory and CPU can be either positively or negatively correlated. This part of the dissertation proposes efficient optimization strategies that utilize both types of correlations to search the entire query plan space in polynomial time when a typical exhaustive search would take at least exponential time. Extensive experimental evaluations have demonstrated the effectiveness of the proposed strategies.

Dynamic plan migration is concerned with on-the-fly transition from

one continuous plan to a semantically equivalent yet more efficient plan. It is a must to guarantee the continuation and repeatability of dynamic query optimization. However, this research area has been largely neglected in the current literature. The second part of this dissertation proposes migration strategies that dynamically migrate continuous queries while guaranteeing the integrity of the query results, meaning there are no missing, duplicate or incorrect results. The extensive experimental evaluations show that the proposed strategies vary significantly in terms of output rates and memory usages given distinct system configurations and stream workloads.

Partitioned query processing is effective to process continuous queries with large stateful operators in a distributed system. Dynamic load redistribution is necessary to balance uneven workload across machines due to changing stream properties. However, existing solutions generally assume static query plans without runtime query optimization. This part of the dissertation evaluates the benefits of applying query optimization in partitioned query processing and shows dramatic performance improvement of more than 300%. Several load balancing strategies are then proposed to consider the heterogeneity of plan shapes across machines caused by dynamic query optimization. The effectiveness of the proposed strategies is analyzed through extensive experiments using a cluster.

Acknowledgments

This dissertation is the result of help, encouragement and pressure from a group of people. Some I have known forever, and others I feel thankful to have come to know in the last few years.

First, I want to thank my advisor, Professor Elke Rundensteiner, for all her support during the process of making this dissertation. I feel truly lucky to have her as my advisor in every possible way. And I feel truly happy to have known her as a person. I thank Professor Murali Mani for his insightful inputs on my research and his lightheartedness that brings joy to the research group. I thank Professor George Heineman for his valuable inputs, discussions and joined meetings on this research work. I thank Dr. Volker Markl for his inspiring comments and valuable discussions on my research.

I thank all previous and current DCAPE members for their hard work on building a system that we can share together as a team, including Luping Ding, Bin Liu, Song Wang, Brad Pielech, Timothy Sutherland, Rimma Kaftanchikova, Brad Momberger, Mariana Jbantova, Nishant Mehta, Hong Su and Jinghui Jian. My thanks also go to all members in the DSRG group,

who together make DSRG a great research group to be part of.

I thank the wonderful professors in the CS department for both their serious lectures and casual chattings. I thank the system support staff in our department and from the school for providing a well-maintained computing environment and utilities for our research needs. I am thankful for the financial supports I have received from my advisor, the department and the school during my study and research in WPI.

I thank my father for all his care and support, and for always over-believing in me. Last, I want to thank my husband, Boyou Chen, for all his encouragement, cooperation and patience. His support is ultimately what makes this dissertation possible.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Research Motivation

1.1.1 General Concepts of Continuous Query Processing

Recent years have witnessed a rapidly increasing attention on continuous query processing in streaming database systems [MWA+03, BBD+02b, ACC+03, AH00, DTW00, VN02b, ILW+00, AAB+05]. Many applications share the same needs for processing streaming data in a continuous fashion, including sensor networks, online financial tickers and medical monitoring systems.

Continuous queries significantly differ from traditional static queries in several aspects. Figure 1.1 depicts the fundamental differences between the two types of queries. For traditional static queries, the data are usually stored on disks. This means that the system has the complete data set available before the query starts. Users can submit one-time queries, after

which the query engine will fetch data from disks and process the queries against the data. Usually the complete results are output all at once. On the other hand, as shown on the right of Figure 1.1, the data set that a continuous query needs to process cannot be all available before the query starts, but rather it arrives continuously as runtime streams. Users submit a set of queries which will be maintained in the query engine. When data arrives at runtime, the queries will process the data and output the update of results at runtime (instead of the complete results as in static query processing). New user queries can be added to the system at runtime while existing queries can be deleted from the system.

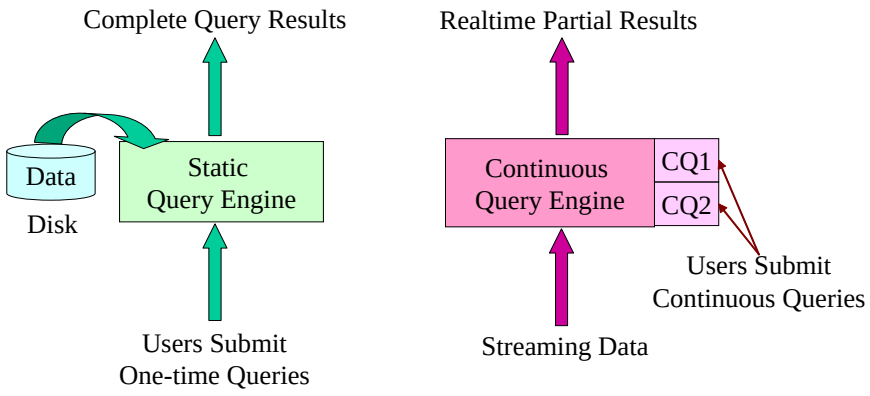


Figure 1.1: Static Query Processing vs. Continuous Query Processing.

In summary, a streaming database stores a collection of continuous queries and the data arrives on the fly, while a static database stores a collection of data sets and queries are posed by database users on the fly. These differences make it unfitting to apply the traditional database design, especially query adaptation techniques, directly to a streaming database, as the for-

mer was not initially designed to deal with on-the-fly real-time data. This calls for a new set of methodologies and algorithms tailored for streaming database technologies to process with continuous queries.

1.1.2 Motivation for Query Adaptation at Runtime

Query optimization is one of the most critical techniques for improving query performance in any database system. In a static database system, such techniques are generally applied at the optimization stage before the query plan starts being executed. This is feasible because, as mentioned earlier, all the data is present before the query plan starts, so the system can collect relatively comprehensive statistics information. Thus static query optimization is able to make reasonably efficient optimization decisions even before the query plan starts.

However, in a streaming database system, data is not stored beforehand, rather it is streaming in as time goes by. At the time when a continuous query is issued, the system may predict the characteristics of the incoming data, but it is generally not possible for the system to gather reasonably accurate statistics before it receives the actual data at runtime. As illustrated in Figure 1.2, several important parameters listed below may change during the usually long execution of a continuous query:

- First, the data characteristics of the incoming streams may change, including stream arrival rates (inter-arrival time), arrival patterns, and data value distributions. Therefore, if any predictions are made regarding the data characteristics before the query starts, they can be

highly inaccurate. Even if they are reasonably accurate, they would hardly hold true throughout the query execution. So the initial query plan needs to be adapted *at runtime* when accurate statistics are being collected or when statistics have changed.

- Second, the system resources available for executing the query, including both CPU and memory resources, may change over time. Different query plans, even for executing the same query, may have different resource requirements. Some may consume more CPU while others have higher memory requirements. Therefore, to prevent system overflow, we need to be able to choose a query plan that can best fit the current available system resources at runtime.

- Third, as the continuous queries are generally long running, during their execution, the Quality of Service (QoS) requirements of users may change as well. For example, the user may switch from requiring the highest output rates to consuming the lowest memory. Based on the requirement of the user, different query plans may need to be chosen accordingly.

- Furthermore, even the collection of continuous queries registered in the system is not stable. A streaming database usually processes multiple continuous queries at the same time. New queries may be registered into the system while old queries may be de-registered if they are no longer useful or required.

In summary, the unpredictable and unstable nature of the streaming

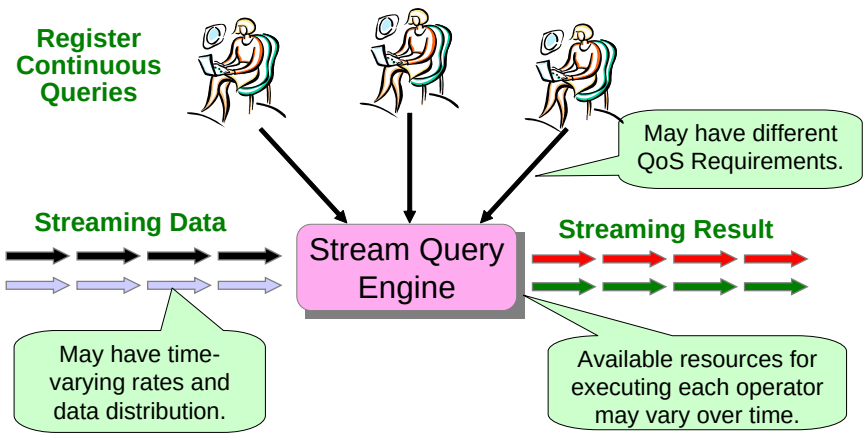


Figure 1.2: Necessity of Runtime Adaptation.

environment poses many new challenges and requirements on query processing in a streaming database system. A query plan that is selected at the start of execution may become sub-optimal or even very inefficient due to these changes in the streaming environment. In order to produce results quickly and efficiently, the streaming database system has to have the ability to constantly collect the data statistics and dynamically adapt the execution of the queries plans based on these realtime statistics.

1.1.3 Existing Query Adaptation Techniques

Dynamic continuous query adaptation can take several forms. Listed below are the most commonly used adaptation techniques in currently continuous query systems [MWA+03, BBD+02b, ACC+03, AH00, DTW00, VN02b, ILW+00, AAB+05]:

- *Intra-operator Adaptation*: An individual operator can have the abil-

ity to react to changing stream characteristics and adapt its execution process accordingly. The XJoin operator [UF99] proposed by Urhan and Franklin is an example of applying intra-operator adaptation techniques. The XJoin operator has three execution modes: memory-to-memory join, memory-to-disk join and disk-to-disk join, and adaptively chooses one of the execution modes based on streams' arrival rates. *Punctuations* [TMSF03a, DMRH04a], which are metadata describing the characteristics of data in arriving streams, can also be used to invoke intra-operator adaptation. As an example, the punctuation can carry information such as "all data arriving after this will have its value larger than 100". The system can take advantage of the punctuations and adapt the execution of individual operators [DMRH04b].

- *Run-time Operator Re-scheduling*: Other researchers have proposed adaptation of continuous query processing at the *scheduling level*: based on the current system load, the adaptive scheduler can dynamically choose the next operator to run and compute its corresponding workloads. The scheduler may make different scheduling decisions based on different optimization goals [WW94, MWA+03, CCea03]. Our own earlier work [SZDR05] also proposes an adaptive framework to choose between different scheduling algorithms at runtime. Changing the scheduling algorithm incurs relatively low overhead, which makes it suitable for run-time adaptation of continuous queries.

- *Load-shedding*: In the case of bursty input streams that exceed the cur-

rent system resource limitations, some research has proposed to do *load shedding* [TCZ+03, BDM04] in order to decrease the workload that the system needs to handle. The basic idea is to drop some workload that has the minimal possible impact on the quality of the outputs. As a side effect, this introduces approximation into the results that the continuous query processing engine produces, which by itself is another important research area in the streaming database field. For a streaming database application that needs to get exact results, approximation techniques such as load shedding are not applicable.

- *Query Plan Re-optimization*: The query optimizer can dynamically alter the order of operators and the shape of the query plans according to the gathered system statistics. This technique, although it may be more costly than operator rescheduling, works the best when fundamental changes such as significant changes of operator selectivities occur. This technique is used both in static query systems as well as in continuous query systems. [1]

- *Query Re-distribution*: Another technique dealing with a large amount of streaming data is to execute the query plans using multiple machines, meaning to de-centralize the query processing and allocate the workload onto different machines in a cluster or a grid structure. A distributed system can absorb a larger amount of incoming data by distributing the workload across potentially all participating machines. Thus it scales better to the amount of stream data and the

[1] In static systems, such runtime query re-optimization is usually caused by inaccurate statistics gathering or false assumptions before the query starts [SLMK01].

number of continuous queries as compared to utilizing only a centralized system.

The first three techniques mentioned above, namely *intra-operator adaptation*, *runtime operator rescheduling*, and *load-shedding*, all try to improve the performance of the query plan without changing the query plan itself. They achieve such performance improvement basically by releasing occupied resources to make more resources available.

Dynamic query plan optimization, another key adaptation technique listed previously, takes the query adaptation into a deeper level by actually changing the shape of the query plan in order to improve the efficiency of the query plan. When the current running query plan degrades too severely, the system would be able to detect such degradation of plan quality and modify the shape of the query plan at run time with the aim to achieve the level of adaptation that the previous three adaptation techniques cannot achieve.

The *distributed query processing* is not only used for faster results, but also a much needed solution for the correct execution of continuous queries. A streaming query engine may take several input streams and execute multiple continuous queries at the same time. The workload such a system needs to deal with can be tremendous. The system resources on a single machine including memory and CPU can be consumed quickly. A continuous query engine that does not have enough system resources to handle the query execution may have to apply load shedding, which incurs inexact query results, or push some data to disk for later processing, which can fur-

ther delay the query results. Hence a streaming system needs to scale well in regards to its potentially very large workload, which generally cannot be achieved by a centralized system with a single machine.

1.2 Research Focus of This Dissertation

The overall research goal of this dissertation is to build a continuous query engine that can dynamically apply query optimization in both a centralized and a distributed environment, as depicted in Figure 1.3. The query executor in the middle executes the query plan and collects runtime statistics, which are input to the *Query Optimization* component. The *Query Optimization* component is then invoked to analyze the given statistics and optimize the query plan if a performance decline has been detected. As a result, a new query plan may be generated by the query optimization component. The system then needs to dynamically transfer the currently running query plan to the newly generated query plan. This process is accomplished by a *Plan Migration* component as shown in Figure 1.3. This component is necessary because the optimization of continuous queries happens at a point during execution, at which time the query execution may have already accumulated intermediate results in the query operators, which needs carefully designed algorithms to safely migrate these data to the new query plan. A valid plan migration strategy needs to guarantee that this migration process does not result in any loss, duplicate or incorrect query results. The *Query Optimization* and the *Plan Migration* are the two adaptation components for the *centralized dynamic query optimization*. Finally, the new query

plan can be executed either in a centralized system or a distributed environment. For the latter case, we also need a *Distributed Adaptation* component to take care of adaptation problems specific only to a distributed system, such as runtime load relocation and dynamic query optimization across multiple machines.

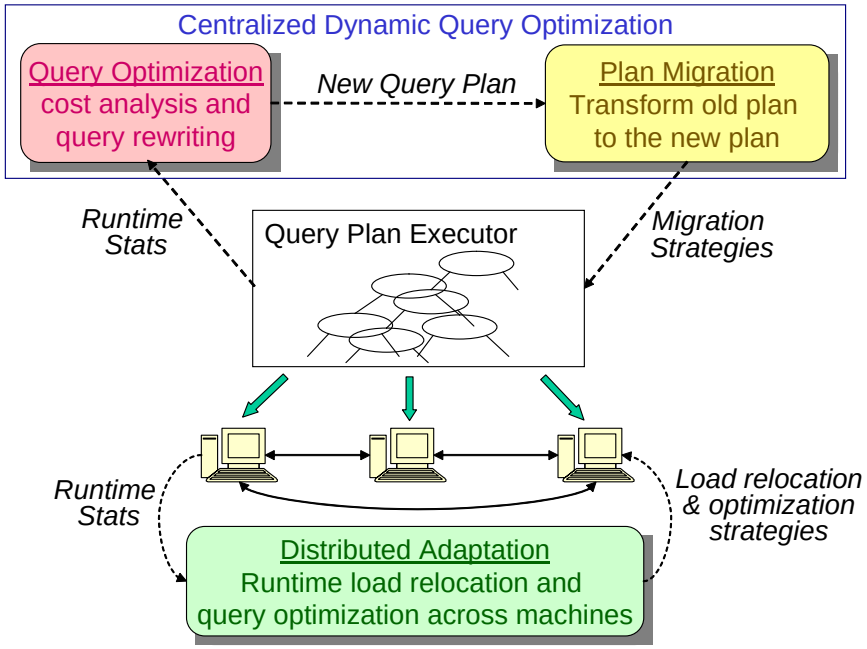


Figure 1.3: Overall Research Focuses.

As shown in Figure 1.3, the main adaptation technologies focused on in this dissertation are query optimization and distributed adaptation. The query optimization includes two components, the optimization component to optimize a query and the plan migration component to dynamically transfer current query plan to a new plan. Therefore, this dissertation focuses on studying the new problems and designing novel solutions for

the three important adaptation components, including dynamic query optimization, dynamic plan migration and distributed adaptation, in order to form a framework to apply runtime adaptations in a continuous query system.

1.2.1 Overview of the DCAPE System

Based on the above research goals and design philosophy, we have built a prototype continuous query system named DCAPE [RDS+04, LZJ+05] at WPI as a team effort to serve as the testbed for our research designs for continuous query adaptations. DCAPE stands for **D**istributed **C**ontinuous **A**daptive **P**rocessing syst**E**m). The DCAPE system is a prototype streaming database system to effectively evaluate continuous queries in highly dynamic stream environments. The system has been demonstrated in VLDB 2004 conference [RDS+04] and VLDB 2005 conference [LZJ+05]. DCAPE adopts a novel architecture that enables adaptive services at all levels of query processing, including reactive operator execution, adaptive operator scheduling, runtime query plan reoptimization and across-machine plan redistribution. All the proposed strategies and algorithms in this dissertation are built into the DCAPE system to equip it with the ability of dynamic query optimization and migration both on a single machine and across multiple machines.

The DCAPE system architecture is depicted in Figure 1.4. The system is built to be run on a single machine as well as across multiple machines. Each machine (processor) can run an instance of the query engine named *CAPE engine*. If the system is run on multiple machines, a distributed man-

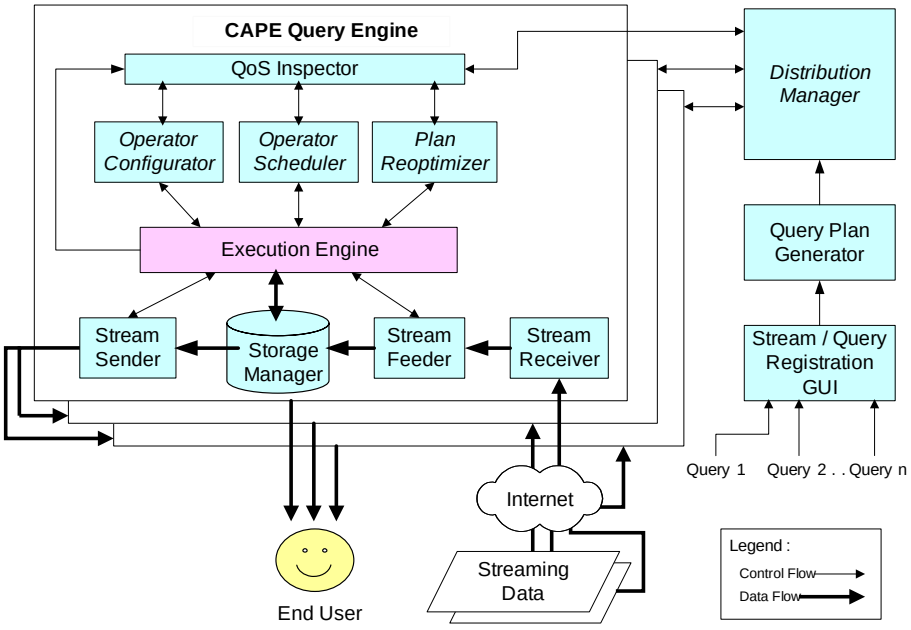


Figure 1.4: D-CAPE System Architecture.

ager overlooks these multiple CAPE query engines and collects statistics from all of them to make system-wide adaptation decisions. The key adaptive components in DCAPE are Operator Configurator, Operator Scheduler, Plan Reoptimizer and Distribution Manager. Once the Execution Engine starts executing the query plan, the QoS Inspector component, which serves as the statistics monitor, will regularly collect statistics from the Execution Engine at each sampling point. This run time statistics gathering component is critical to continuous query processing, as any adaptation technique relies on the statistics gathered at run time to make informed decisions.

The two adaptive components in the DCAPE architecture that are di-

rectly related to this dissertation are Plan Reoptimizer and Distribution Manager, which are in charge of the centralized query optimization and distributed query optimization respectively. As mentioned earlier, this dissertation focuses on investigating three adaptation technologies, including dynamic query optimization, dynamic plan migration and distributed query adaptation. The former two together correspond to the Plan Reoptimizer component in the DCAPE architecture shown in Figure 1.4. The distributed adaptation relies on the coordination between the Distribution Manager and the local adaptation components. If particular machines are detected to be overloaded or underloaded, the Distribution Manager will redistribute one or multiple query plans among the given cluster of machines. All new designs and algorithms in this dissertation are implemented and experimented within the DCAPE system.

1.2.2 Continuous Query Optimization

New Challenges in Dynamic Query Optimization

Query plan optimization has remained at the core of database research for over two decades [MS79, IK84, KBZ86, SI93, IK91, SAC+79]. In a static database, the quality of a query plan is often judged by its total processing time measured in terms of CPU processing and disk I/O costs [SMK97]. However, the optimization of continuous query processing [MWA+03, MSHR02, CCC+02] differs from traditional query optimization in several aspects.

First, the quality of a continuous query plan is typically judged by its runtime output rate [VN02a]. As observed in [AN04], a continuous query

plan produces the optimal output rate as long as the *CPU cost per unit time* required by the plan is less than the system CPU capacity. In this case, the output rate of the query plan is determined *solely* by the stream arrival rates. Therefore, a continuous query optimizer should generate query plans with their required CPU usages below the system CPU resource constraint.

Second, continuous queries are usually assumed to be main-memory-resident due to the often rather stringent real-time output requirements [MWA+03, MSHR02, CCC+02]. Due to the existence of stateful operators, such as join or group-by, which may store large amount of tuples in operator states, continuous query processing tends to be memory-intensive. In case of memory overflow, we have to either spill in-memory data to disk [LZR06, UF99, VNB03], which can further delay the processing, or we could apply load shedding [TCZ+03] to delete data, which incurs approximate results. Clearly, for applications that favor accurate realtime results, the query optimizer instead should aim to generate query plans with their memory cost below the system memory resource constraint.

Lastly, what complicates matters significantly in the streaming context is that the statistics of the streams are usually unknown before a query starts. In fact they may continue to change during the query execution. A query plan that is currently optimal can become sub-optimal at a later time. Therefore, runtime optimization is needed. It is imperative to adopt *efficient* optimization algorithms, as otherwise the cost of the optimization process may outweigh its potential savings.

Therefore, efficient runtime optimization algorithms are required to gen-

erate continuous query plans with both CPU and memory consumptions beneath the respective CPU and memory resource constraints. Continuous query plans that satisfy these dual resource constraints are henceforth called *qualified plans*. Finding a qualified plan is a multi-objective optimization problem [PY01]: the optimizer needs to find a query plan that satisfies both resource constraints. As in other multi-objective optimization work [PY01, HM94, SAL+96], in order to achieve both objectives, we need to characterize the relationship among the determining cost factors.

It is clear that CPU and memory costs are often positively correlated. Intuitively, a query that has less data (less memory) to process needs less CPU for processing the data. This fact is being utilized by most current work on continuous optimization [GO03, VNB03, BMM+04, VN02a]. The main goal of these approaches, which parallels the optimization work in static query processing [MS79, IK84, KBZ86, SI93, SAC+79], is to minimize the amount of intermediate results with the assumption that this also reduces CPU costs.

However, these two resources can also be negatively correlated when processing continuous multi-join queries. This observation has largely been ignored by existing continuous query optimization work. Let us consider the two common methods for executing continuous joins, namely binary join trees (*bjtree*) [VN02a] and multi-way join operators (*mjoin*) [GO03, VNB03, BMM+04, HAE03]. A *bjtree* is a query plan composed of binary join operators. It keeps all intermediate results in operator states, thus saves CPU cost on recomputing these intermediate results but requires high memory costs. On the contrary, an *mjoin* does not keep any intermediate results,

thus saves memory but requires extra CPU for recomputation.

Existing optimization work has focused on minimizing the intermediate results of either an mjoin or a bjtree, which decrease both memory and CPU usages. However, since memory and CPU can be both positively and negatively correlated, the dual resource constraints cannot always be satisfied within the mjoin or the bjtree solution space. Instead, we need to extensively yet efficiently explore the search space both within and beyond the mjoin and the bjtree solution space, while considering both types of correlations between memory and CPU.

Dissertation Contributions to Dynamic Query Optimization

This dissertation proposes two polynomial-time optimization strategies, namely *mjoin-init* and *bjtree-init*, that generate continuous multi-join plans meeting these dual resource constraints. The proposed strategies explore the multi-join solution space, considering mjoin, bjtree, and the tree structures in-between as solution candidates. Each strategy utilizes both positive and negative correlations between CPU and memory. They are thus able to find qualified query plans when existing strategies cannot. The effectiveness of the proposed strategies is thoroughly analyzed and compared through experiments in a prototype continuous query system.

The problem of dynamic continuous query optimization contains two sub-problems:

- First, we need to design online efficient optimization algorithms that find qualified continuous query plans meeting dual resource con-

straints.

- Second, we need strategies to dynamically transfer the currently running query plan to the newly generated query plan without affecting the runtime query results.

The second sub-problem, referred to as *dynamic plan migration* will be addressed in the second part of this dissertation, which will be introduced in Section 1.2.3. In this task of the dissertation, we focus on the first sub-problem.

For the two proposed optimization strategies, I have designed four novel optimization algorithms, two for each proposed strategy. Within each strategy, the first algorithm utilizes the positive correlation to decrease both memory cost and CPU costs of the query plan, while the second utilizes the negative correlation to further exploit the trade-off between the two resource usages.

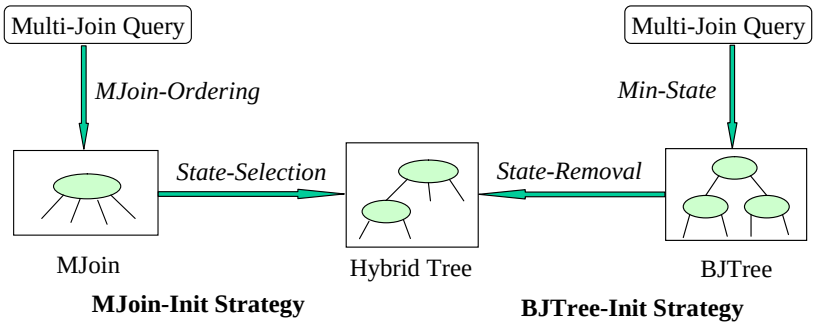


Figure 1.5: Two Alternative Optimization Strategies

As illustrated in Figure 1.5, the *mjoin-init strategy* first applies the *mjoin-ordering algorithm* to find optimal mjoin orderings to minimize both mem-

ory and CPU costs. If the mjoin is still not qualified, the *state-selection algorithm* then carefully selects intermediate states to store in memory in order to save CPU costs. As a result, a *hybrid tree* in between an mjoin and a bjtree may be generated as the qualified plan. On the other hand, the *bjtree-init strategy* first applies the *min-state algorithm* to minimize both memory and CPU costs by generating optimal bjtrees. If the bjtree is not qualified, the *state-removal algorithm* then generates a qualified hybrid tree by removing selected intermediate states. This saves memory while increases CPU costs.

This part of the dissertation work contributes to research in continuous query optimization in the following ways:

- First, two novel optimization strategies are proposed, which contain four efficient algorithms, to generate qualified continuous multi-join plans. To the best of my knowledge, this work is the first in continuous query optimization to 1) consider both resource constraints, 2) utilize both positive and negative correlations between the two resources, and 3) explore the multi-join solution space, considering mjoin, bjtree, and the tree structures in-between as candidate solutions.

- Secondly, each of the four polynomial-time algorithms by itself is an advance of the state-of-art of continuous query optimization. The *mjoin-ordering*, extending the classic *IK algorithm* [IK84], finds *optimal* mjoin orderings for acyclic join graphs. It is an improvement to current solutions on optimizing mjoins [GO03, VNB03, BMM+04]. The *min-state* finds *optimal* bjtree solution for star joins, which to the best

of our knowledge is yet to be achieved by existing solutions on optimizing continuous bjtrees [VN02a, BMM+04]. The *state-selection* and *state-removal* algorithms are algorithms for solving this new problem of generating hybrid trees. To the best of my knowledge, they are the first such algorithms in the current literature.

- The proposed strategies and algorithms are implemented in the DCAPE continuous query system [RDS+04, LZJ+05] introduced in Section 1.2.1. A thorough experimental evaluation is conducted in the DCAPE system. The experimental results show that both proposed optimization strategies are efficient in finding qualified query plans. We also compare the performance of the two optimization strategies in terms of resource consumption of their generated query plans and make recommendations as when to use which optimization method.

1.2.3 Dynamic Plan Migration

New Challenges in Dynamic Plan Migration

Dynamic plan migration is concerned with the on-the-fly transition from one continuous query plan to a semantically equivalent yet more efficient plan. Migration is important for stream monitoring systems where long-running queries may have to withstand fluctuations in stream workloads and data characteristics.

A migration strategy must guarantee that it will not alter the results produced by the system during as well as after the plan transition. Correctness here implies that results are neither missing nor contain erroneous or du-

plicate tuples. Traditionally, a dynamic plan migration strategy [CCC+02] takes the following steps: 1) pause the execution of the current query plan, 2) drain out all existing tuples in the current query plan, 3) replace the current plan with the new plan, and restart the execution. We refer to this traditional approach as the *pause-drain-resume* strategy. The purpose of the draining step is to clean up the intermediate tuples in the query plan to prevent any missing output tuples.

The *pause-drain-resume* migration strategy may be adequate to dynamically migrate a query plan that consists of only *stateless* operators, such as select and project. A *stateless* operator does not need to maintain intermediate data nor other auxiliary state information in order to be able to generate complete and correct results. Intermediate tuples in such a stateless query plan exist only in intermediate queues and can be cleaned completely by the drain step during the migration process.

On the contrary, a *stateful* operator, such as join or group-by, must store tuples that have been processed thus far so to be able to generate future results. For a long-running query as in the case of continuous queries, the number of tuples stored inside a stateful operator, such as a join or a group-by, can potentially be infinite. Several strategies have been proposed to limit the number of intermediate tuples kept in operator states by purging unwanted tuples, including window-based constraints [KNV03, CCC+02, NWAea02, HFAE03] and punctuation-based constraints [DMRH04a, TMSF03b]. In all the above strategies the purge of the old tuples inside the state is driven by the processing of either new tuples or new punctuations from input streams.

It is important to note that for a query plan that contains such *stateful* operators, intermediate tuples may exist in both the intermediate queues and in the operator states. As noted above, the purge of tuples in the states relies on the processing of new data. However, in the *pause-drain-resume* migration strategy described above, before embarking on the drain step, as the very first step the execution of the query plan is paused so that no new tuples beyond the intermediate tuples are being processed until the migration is over. This creates a *deadlock in the migration process*: the migration is waiting for all old tuples in operator states to be purged from the old plan, while the old tuples in those states are waiting for new tuples to be processed in order to be purged.

In this dissertation, we are the first to develop new solutions for online plan migration for continuous plans with stateful operators, or plans with mixture of stateful and stateless operators.

Dissertation Contributions to Dynamic Plan Migration

This part of the dissertation proposes two plan migration strategies for continuous queries over streaming data, namely the *moving state strategy* and the *parallel track strategy*. The first strategy exploits reusability of existing stream states and the second employs parallel query execution to seamlessly migrate between continuous join plans without affecting the results of the query.

The *moving state strategy* first pauses the query plan or part of the query plan that is being optimized and drains out tuples inside the intermediate queues, similar to the above *pause-drain-resume* approach. However,

to avoid loss of any useful data residing in states, it then carefully identifies and moves over all relevant tuples in the states of the old query plan to their corresponding location in the new query plan. Beyond that, to assure correctness, selectively certain intermediate tuples are then recomputed. Lastly, the execution of the query plan is then resumed with the new plugged-in plan.

The second migration strategy, called the *parallel track strategy*, migrates a query plan in a more gradual fashion by continuing the delivery of output tuples even during migration. Instead of moving tuples to the new query plan and discarding the old query, it plugs in the new query plan and starts executing both query plans in parallel. Algorithms are developed to eliminate potential duplicates and maintain the appropriate order of output tuples. Once the old plan is found to be "antiquated", it can simply be disconnected and the migration stage is then over.

In summary, this part of the dissertation makes the following contributions to plan migration of continuous queries at runtime:

- Two migration strategies, namely the moving state strategy and the parallel track strategy, are designed for migrating query plans that are composed of stateful operators. The proposed migration strategies cover query plans that consists of stateful operators, such as join and group-by, as well as query plans with a mixture of stateful operators and stateless operators, such as select and project.

- Cost models are developed to analyze and compare the costs of the two proposed migration strategies.

- Various execution models adopted in existing continuous query systems [CCC+02, NWAea02, MSHR02, RDS+04] are identified and categorized to illustrate how different execution model can affect the runtime plan migration strategy. Each identified execution model has its unique properties on tuple execution order and operator scheduling. Changes made to the proposed migration strategies to support each execution model are also introduced in this dissertation.

- The proposed migration strategies are implemented in the DCAPE system. Experimental evaluations have been conducted to compare their performances. The experimental results demonstrate performance improvements in the order of magnitude by dynamically applying the migration strategies in the middle of query processing in a variety of system settings.

1.2.4 Distributed Query Adaptation

New Challenges in Distributed Query Adaptation

A continuous query system may easily run out of resources due to high stream input rates or cost-intensive query operations. Distributed continuous query processing over a shared nothing architecture, i.e., a cluster of machines, has been recognized as one of the prevalent methods for solving this scalability problem [AAB+05, CBB+03, MJSM03, LZJ+05, DH04]. Distributing the query workload across multiple machines can greatly improve the system performance due to having aggregated resources and parallel processing capabilities. However, uneven workload among ma-

chines may occur due to (1) the lack of initial cost information when distributing the queries, and (2) the potentially fluctuating nature of the incoming stream data. This imbalance of workloads may impair the benefits of distributed processing. Thus, *dynamic load balancing*, which deals with the problem of re-distributing workload across machines in the cluster, has become one of the most crucial technologies for a distributed continuous query system [MJSM03, AAB+05, LZJ+05, DH04].

In existing distributed continuous query systems [AAB+05, CBB+03, DH04], the basic unit of workload to be moved among machines during load rebalancing tends to be whole operators. This assumes that each operator is small enough to fit on one machine. Such operator-level adaptation is a good choice for for query plans containing only stateless operators or stateful operators with fairly small states. However, the operator-level adaptation is not always practical for stateful operators with huge states. For such cases, operator-level adaptation can be inefficient, if not impossible.

Partitioned parallelism is a general query plan distribution strategy, which has been routinely applied to traditional query processing [Has95, Gra90]. The Flux system [MJSM03] is the first to apply partition-level load redistribution to continuous queries and has demonstrated promising performance.

However, all existing partition-level load redistribution solutions in the literature implicitly assume that all query instances installed on machines have the same query shapes and remain so throughout the query execution [MJSM03, AAB+05, CBB+03]. They have not considered the situation

that the query optimizer restructures the shape of the query plan residing on individual machines. Therefore, existing work on partitioned continuous query processing has not considered the benefits of integrating the load balancing with query optimization. Consequently, the effects of query optimization and its impact on load rebalancing strategies remain an open issue to date.

This however is clearly a major limitation, as runtime query optimization has been shown to be critical for streaming systems in the existing literature [MWA+03, BBD+02b, ACC+03, AH00, DTW00, VN02b, ILW+00] as well as by the research conducted in the first two parts of this dissertation (as introduced in Sections 1.2.2 and 1.2.3). Load balancing strategies typically just move workload from one machine to another, while the total resource consumption in the system as a whole is not decreased. On the other hand, plan optimization may be able to decrease the resource consumption on each machine, therefore also decreasing the overall resource consumption in the distributed system. For example, a plan optimization may dynamically switch two join operators in a plan in the face of changing statistics. This can reduce the intermediate results, which leads to less CPU and memory costs on this machine and thus to less overall resources required to process this query in the distributed system.

Local query optimization however complicates load rebalancing strategies. Traditionally, partition-level load rebalancing algorithms assume that the shapes of the query plan are the same across machines and remain so throughout the query execution. Balancing load among machines in such a stable environment can be achieved by moving some load (parti-

tions) from over-loaded machines to under-loaded machines with matching query plan shapes. However, if local query optimization is applied on individual machines, at any given time, the shapes of the query plan on different machines can be distinct from one another.

This problem of integrating partition-level load rebalancing with query optimization remains an unaddressed problem to date. Clearly, advanced load balancing strategies are needed to collaborate with the query optimization strategies and take the heterogeneity of plan shapes on difference machines into account.

Dissertation Contributions to Distributed Query Adaptation

Part III of this dissertation makes contributions in both operator-level and partition-level distributed adaptations, with a focus on the latter adaptation. The main focus of this dissertation part is on partition-level adaptation, which solves the new problems of integrating query optimization with the partition-level runtime load balancing for continuous queries. For the partitioned query adaptation, the first research goal is to study the effects of adding plan optimization to distributed continuous query processing. As the second research goal, we propose to design, implement and evaluate advanced load rebalancing strategies which take the heterogeneity of query plan shapes on difference machines into account. For the operator-level adaptation, the dissertation extends the centralized migration strategies proposed in Part II to parallel query processing environment.

In particular, this research on distributed continuous query adaptation makes the following contributions:

- I propose operator-level plan migration protocols for distributed continuous query processing. These protocols extend the plan migration strategies described in Part II of this dissertation to distributed system.

- I relax the assumption of unchanged plan shapes made by state-of-art load balancing adaptation. I design two new load balancing strategies, namely parallel track load balancing (PTLB) and moving state load balancing (MSLB), and their corresponding protocols that can balance workload while seamlessly handling the complexity caused by local plan changes. The PTLB strategy is a general load balancing strategy that requires no detailed knowledge of the underlying query plans, such as types of operators and shapes of query plans. I then propose the more plan-aware MSLB strategy, which rebalances the workload by comparing the detailed shapes of the query plans among different machines.

- I have implemented the two new load balancing strategies in the DCAPE system. I have experimentally evaluated the effects of query optimization as well as load rebalancing for partitioned continuous query processing on an actual cluster. The extensive experiments show that the combination of query optimization and load balancing has superior performance than applying each adaptation technique alone. Between the two load balancing strategies, the MSLB is shown to be more efficient than the PTLB in many situations, while the PTLB can win under certain conditions.

1.3 Dissertation Organization

The rest of this dissertation is organized as follows: The three research topics are discussed in detail in Part I, Part II and Part III in this dissertation respectively. The discussions of each of the three research topics include the relevant research motivation, problem introduction, background, solution description, experimental evaluation and discussions of related work. Chapter 25 concludes this dissertation and Chapter 26 discusses possible future work.

Part I

Continuous Query Optimization with Resource Constraints

Chapter 2

Introduction and Research Outline

2.1 Continuous Query Optimization

Query plan optimization has remained at the core of database research for over two decades [MS79, IK84, KBZ86, SI93, IK91, SAC+79]. In a static database, the quality of a query plan is often judged by its total processing time measured in terms of CPU processing and disk I/O costs [SMK97]. However, the optimization of continuous query processing [MWA+03, MSHR02, CCC+02], a recently emerging research topic, differs from traditional query optimization in several aspects.

First, continuous queries over streaming data are usually long running and theoretically can be even infinite. Thus the total processing time is no longer a suitable criteria to measure the quality of a query plan. Instead,

the quality of a continuous query plan is typically judged by its runtime output rate [VN02a], i.e., how fast it can produce real-time query results. As observed in [AN04], a continuous query plan produces the optimal output rate as long as the *CPU cost per unit time* required by the plan is less than the system CPU capacity. In this case, the output rate of the query plan is determined *solely* by the stream arrival rates. Therefore, a continuous query optimizer should generate query plans with their required CPU usages below the system CPU resource constraint.

Second, continuous queries are usually assumed to be main-memory-resident due to the often rather stringent real-time output requirements [MWA+03, MSHR02, CCC+02]. Some operators need to keep *states* in order to be non-blocking [MWA+03, CCC+02]. For example, a join operator needs to put tuples processed so far from one input stream into a join state in order to join them with future incoming tuples from the other stream. In case of high stream arrival rates or large processing windows, the size of the operator states kept in main memory can be huge. Therefore, continuous query processing tends to be memory-intensive. When memory overflows, we have to either spill in-memory data to disk [LZR06, UF99, VNB03], which can further delay the processing, or we could apply load shedding [TCZ+03] to delete data, which incurs approximate results. Clearly, for applications that favor accurate results, the query optimizer instead should aim to generate query plans with their memory cost below the system memory resource constraint.

Lastly, what complicates matters significantly in a streaming context is that the statistics of the streams are usually unknown before a query starts.

In fact they may continue to change during the query execution. A query plan that is currently optimal can become sub-optimal later. Therefore, runtime optimization is needed. It is imperative to adopt *efficient* optimization algorithms, as otherwise the cost of the optimization process may outweigh its potential savings.

In summary, efficient runtime optimization algorithms are required to generate continuous query plans with both CPU and memory consumptions beneath the respective CPU and memory resource constraints. Continuous query plans that satisfy these dual resource constraints are henceforth called *qualified plans*.

2.2 Relationships Between Resource Usages

Finding a qualified plan is indeed a multi-objective optimization problem [PY01]: the optimizer needs to find a query plan that satisfies both resource constraints. As in other multi-objective optimization work [PY01, HM94, SAL+96], in order to achieve both objectives, we need to characterize the relationship among the determining cost factors.

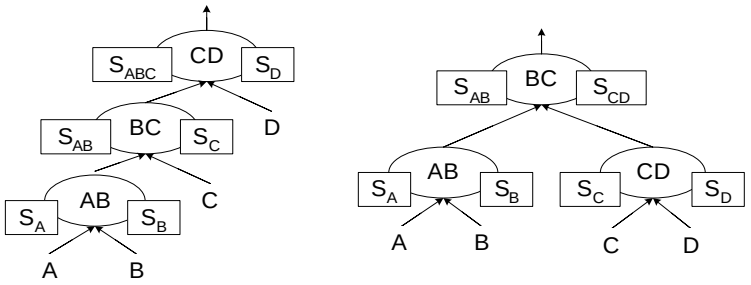
It is clear that CPU and memory costs are positively correlated. Intuitively, a query that has less data (less memory) to process needs less CPU for processing the data. This fact is being utilized by most current work on continuous multi-join optimization [GO03, VNB03, BMM+04, VN02a]. The main goal of these approaches, which parallels the join ordering work in static query processing [MS79, IK84, KBZ86, SI93, SAC+79], is to minimize the amount of intermediate results with the assumption that this also

reduces CPU costs.

However, these two resources can also be negatively correlated when processing continuous multi-join queries. This observation has largely been ignored by existing continuous query optimization work. Let us consider the two common methods for executing continuous joins, namely binary join trees (*bjtree*) [VN02a] as shown in Figure 2.1 and multi-way join operators (*mjoin*) [GO03, VNB03, BMM+04, HAE03] as shown in Figure 2.2.

A *bjtree*, as shown in Figure 2.1 in two different shapes, is a query plan composed of binary join operators. It is a direct extension from the typical query plans used in static query processing [SAC+79, IK84, KBZ86]. Figure 2.1 shows two sample binary join trees. The one on the left is a *linear tree*, in which one of the two inputs for each join operator is a stream input, except for the leaf, which has two stream inputs. The bjtree on the right is a *bushy tree*, in which both inputs of a join operator can be intermediate results from the join operator below it. Each binary join operator applies symmetric join algorithm [WA93a, HH99] (which will be illustrated in Section 3) and keeps two states that stores tuples that the operator has received so far. Some states, such as state S_A in Figure 2.1, keep the stream input tuples. Other states, such as S_{AB} and S_{ABC}, keep intermediate join results.

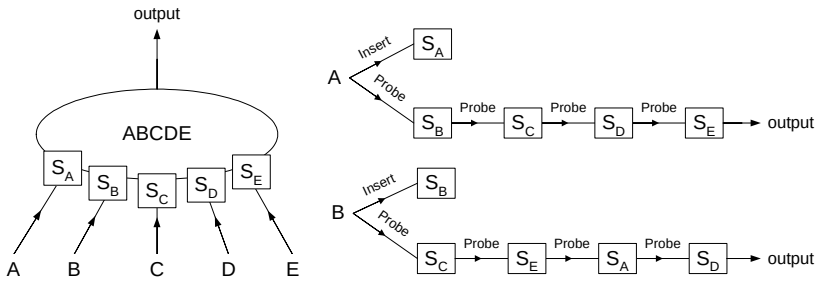
Different from a bjtree, an mjoin use a single multi-way join operator that takes in all joining stream inputs and outputs the joined results from this join operator. Figure 2.2(a) shows the basic data structure of a mjoin operator in a continuous query that contains a four-way join $A \bowtie B \bowtie C \bowtie D$. The operator takes in four input queues and outputs the joined tuple ABCD. Four states are kept in the operator, each associated with one



(a) A left-deep binary join tree (b) A bushy binary join tree

Figure 2.1: BJTree

input queue. Suppose now the multi-way join operator takes one tuple A from input queue A. It would first insert this tuple A into the state S_A, then it uses this tuple A to purge and join with all other remaining states in a certain order. The processing of new tuples from other input queues follows the same procedure, except that they may join with remaining states in a different order. Figure 2.2(b) shows possible join orders for tuples from input queue A and input queue B.



(a) A 5-way Mjoin Operator (b) Sample join orderings for input A and B

Figure 2.2: MJoin Operator

As we can see a bjtree keeps all intermediate results in operator states,

thus saves CPU cost on recomputing these intermediate results but requires high memory costs. On the contrary, an *mjoin* does not keep any intermediate results, thus saves memory but requires extra CPU for recomputation.

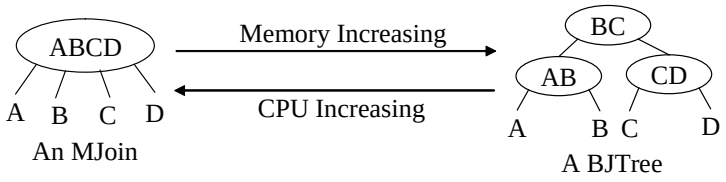


Figure 2.3: MJoin and BJTree

As shown in Figure 2.3, from one join method to the other, we save one resource by increasing the other: From mjoin to bjtree the memory cost increases while the CPU cost decreases. For the opposite direction the memory cost decreases while the CPU cost increases. Most optimization strategies for multi-join queries use only one of these two join methods [VN02a, GO03, VNB03, BMM$^+$04, HAE03]. Hence they can at most exploit the positive correlation between CPU and memory within each join method, but cannot utilize the two resources' negative correlation that arises when crossing different join methods.

For example, consider the scenario that an optimizer may find the best bjtree with the least possible memory and CPU costs. However, the memory needed for storing all intermediate results may still exceed the available system memory. Another possible scenario is that an optimizer may generate the optimal join orderings for an mjoin and thus guarantee minimized CPU cost. However, due to requiring possibly large amounts of recomputations, the CPU cost may still exceed available CPU resources. In either

case, the overflowing cost factor cannot be further reduced without taking advantage of the negative correlation between CPU and memory that can be exploited by crossing the boundary between the two join methods. At this point, the memory cost may not be further reduced without using the negative correlation between memory and CPU. However, the CPU may not be further reduced without taking advantage of the negative correlation between CPU and memory.

Therefore, the existing solutions in the literature miss the important opportunity for trading-off between the two resources. This severely limits the optimizer's search space. It fact, it simplifies the optimization to a one-dimensional problem. Such optimization strategies may fail to find a qualified continuous multi-join plan that satisfies both resource constraints, even though a qualified plan may indeed exist.

2.3 Proposed Strategies in This Dissertation

In this first part of the dissertation, I propose two efficient polynomial-time optimization strategies, namely the *mjoin-init* and the *bjtree-init strategy*, that exploit both the positive and the negative correlations between CPU and memory to generate qualified plans for continuous multi-join queries. The *mjoin-init strategy* optimizes starting from an mjoin while the *bjtree-init strategy* starts from a bjtree. To the best of our knowledge, our work is the first to explicitly consider both CPU and memory resource constraints and their relationships while optimizing a continuous multi-join query. I will show that our optimizer can find qualified query plans when other exist-

ing techniques in the literature cannot.

The problem of dynamic continuous query optimization contains two sub-problems:

- First, we need efficient online optimization algorithms that find qualified continuous query plans meeting dual resource constraints.
- Second, we need strategies to dynamically transfer the currently running query plan to the newly generated query plan without affecting the runtime query results.

The second sub-problem, referred to as *dynamic plan migration*, will be addressed in the Part II of this dissertation. In this part of the dissertation, I focus on the first sub-problem.

I have designed four novel optimization algorithms, two for each proposed optimization strategy. Within each strategy, the first algorithm utilizes the positive correlation to decrease both memory cost and CPU cost of the query plan, while the second utilizes the negative correlation to further exploit the trade-off between the two resource usages.

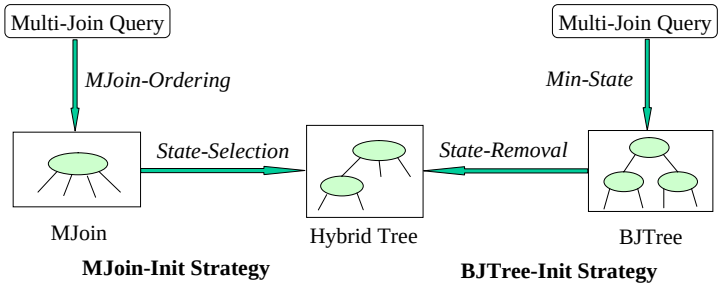


Figure 2.4: Two Alternative Optimization Strategies

As illustrated in Figure 2.4, the *mjoin-init strategy* first applies the *mjoin-ordering algorithm* to find optimal mjoin orderings to minimize both memory and CPU costs. If the mjoin is still not qualified, the *state-selection algorithm* then carefully selects intermediate states to store in memory in order to save CPU costs. As a result, a *hybrid tree* in between an mjoin and a bjtree may be generated as the qualified plan. On the other hand, the *bjtree-init strategy* first applies the *min-state algorithm* to minimize both memory and CPU costs by generating optimal bjtrees. If the bjtree is not qualified, the *state-removal algorithm* then generates a qualified hybrid tree by removing selected intermediate states. This saves memory while increases CPU costs.

Besides the two efficient optimization, I also designed an exhaustive search strategy using dynamic programming, which searches the whole multi-join search space to find a qualified query plan. This exhaustive search strategy guarantees that a qualified plan can be found if there exist one. However, since the strategy, as any of the classic dynamic programming used in static databases [SAC+79], takes exponential time and space, it is only useful when the number of joins in the query is relatively small.

This part of the dissertation work contributes to research in continuous query optimization in the following ways:

- First, I propose two novel optimization strategies, which contain four efficient algorithms, to generate qualified continuous multi-join plans. To the best of my knowledge, this work is the first in continuous query optimization to 1) consider both resource constraints, 2) utilize both positive and negative correlations between the two resources,

and 3) explore the multi-join solution space, considering mjoin, bjtree, and hybrid tree in-between as candidate solutions.

- Secondly, the four polynomial-time algorithms each by itself is already an advance of the state-of-art of continuous query optimization. The *mjoin-ordering*, extending the classic *IK algorithm* [IK84], finds *optimal* mjoin orderings for acyclic join graphs. It is an improvement to current solutions on optimizing mjoins [GO03, VNB03, BMM+04]. The *min-state* finds *optimal* bjtree solution for star joins, which to the best of our knowledge is yet to be achieved by existing solutions on optimizing continuous bjtrees [VN02a, BMM+04]. The *state-selection* and *state-removal* algorithms are algorithms for solving this new problem of generating hybrid trees. To the best of our knowledge, they are the first such algorithms in the current literature.

- I extend the classic left-deep exhaustive search optimization algorithm to now cover the entire hybrid-tree search space.

- A thorough experimental evaluation is conducted in the CAPE continuous query system [RDS+04]. I compare the two polynomial-time strategies with the exhaustive strategy. The experimental results show that both proposed optimization strategies are as reliable in finding qualified query plans as the exhaustive search strategy, while taking much less time and space than the exhaustive strategy. I also compare the performance of the two optimization strategies in terms of resource consumptions of their generated query plans and make recommendations as when to use which optimization method.

2.4 Road Map

The rest of this part of the dissertation is organized as follows: Chapter 3 introduces the background information for this research, including stateful operators and window constraints. Chapter 4 analyzes costs and correlations between CPU and memory. The two alternative optimization strategies are described in Chapters 5 and 6 respectively. The exhaustive search algorithm is described in Chapter 7. Experimental results are reported in Chapter 8. Chapter 9 discusses related work.

Chapter 3

Background

3.1 Stateful Operators in Continuous Queries

Continuous queries generally require real time responses. Query results received after a certain time period may no longer be needed by the end user. This requires that all operators in the query plans need to be operated in a pipelined fashion: the operator needs to be able to generate partial results based on the data that it has received so far. This promotes the usage of stateful operators. A *stateful* operator, such as join or group-by, must store all tuples that have been processed thus far from one input stream so to be able to join them with future incoming tuples from the other input stream. For a long-running query as in the case of continuous queries, the number of tuples stored inside a stateful operator can potentially be infinite. Several strategies have been proposed to limit the number of intermediate tuples kept in operator states by purging unwanted tuples, including applying window-based constraints [KNV03, CCC+02, NWAea02, HFAE03] and

punctuation-based constraints [DMRH04a, TMSF03b]. On the contrary, a *stateless* operator, such as Select and Project, does not need to maintain intermediate data nor other auxiliary state information so to be able to generate complete and correct results.

Join is one of the most important stateful operator in continuous query processing, and is the focus of the research in this part of the dissertation. As commonly used by continuous query plan in most streaming database systems [KNV03, CCC+02, NWAea02], in this dissertation I adopt a symmetric window-based binary join algorithm [WA93a, HH99] for join processing. A sample query plan for the query $A \bowtie B \bowtie C \bowtie D$ that consists of three join operators with input streams A, B, C and D is depicted in Figure 3.1(a). The join operator $B \bowtie C$ in Figure 3.1(b) has two input queues Q_{AB} and Q_C, two *states* S_{AB} and S_C, one associated with each input queue, and one output queue Q_{ABC}. Each *state* stores the tuples that fall within the current window frame (either a certain time period or a certain number of tuples) from its associated input queue. For each tuple AB from Q_{AB}, the join involves three steps: 1) purge – AB is used to purge tuples in state S_C that are outside of the window frame, 2) join – AB is joined with the tuples left in S_C, and 3) insert – AB is inserted into state S_{AB}. The same process applies similarly to any tuple from Q_C. This 3-step process is referred to as the *purge-join-insert* algorithm.

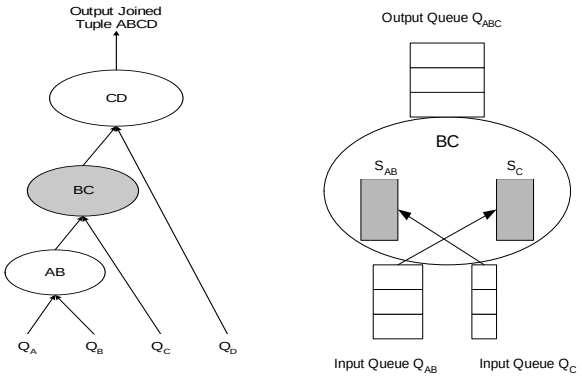


Figure 3.1: Join Operators and Their States

3.2 Window Constraints

Without any constraints, the states of a stateful operator can grow infinitely, and the system can eventually grows out of memory. To solve this problem, streaming databases usually adopts sliding window constraints to limit the size of states. A sliding window-based constraint [KNV03, CCC+02, NWAea02] can be used to purge unwanted tuples stored in the state. Usually two kinds of window constraints can be posed over an operator: time-based [KNV03] and count-based [NWAea02]. For a time-based window, the window size is represented as a time frame, while the count-based window is described as the number of tuples in the window. For the rest of this dissertation, unless otherwise noted, I will be using the symmetric join operator with time-based window constraints as the illustrating stateful operator in continuous queries.

A time-based window constraint W_{AB} posed over streams A and B indicates that two tuples from streams A and B respectively can be joined

only if their timestamps are within W_{AB} from each other. The most commonly used window constraint is *global window constraint* in which all pairs of streams has the same window constraint. For the example illustrated in Figure 3.2, a query $A \bowtie B \bowtie C \bowtie D$ has a global window constraint means that $W_{AB} = W_{BC} = W_{CD} = W_{AD} = W_{AC} = W_{BD}$. Theoretically it is possible that the window constraints among join pairs may be different or even unconstrained, these type of window constraints are referred to as *local window constraints*. If these situations do occur, the window constraint between any pair of streams that do not have a pre-defined window constraint can be treated as having a window constraint equal to the *shortest path* between the pair in a window constraint relationship graph as the one in Figure 3.2.

A time-based window constraint requires that each newly arriving tuple has a timestamp. Only tuples with timestamps that are within the current time window can be processed by the operator. A tuple has a single timestamp when it first arrives in the stream, referred to as a *singleton tuple*. Within each stream entering the query engine, the singleton tuples are assumed to be ordered by their timestamps [KNV03, CF02, NWAea02]. When two tuples are joined together, the timestamp for the joined tuple is an array that concatenates the timestamps from both joining tuples, as indicated in Figure 3.3. Both timestamps are kept because either of them might be used by other join operators in the query plan to purge tuples. Such a tuple with a combined timestamp is referred to as a *combined tuple*.

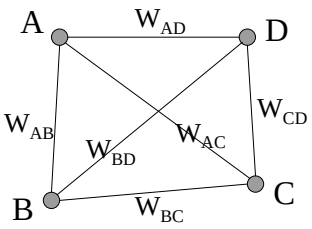


Figure 3.2: Graph on Window Constraints

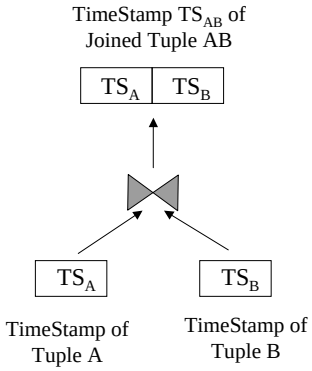


Figure 3.3: Combined Timestamp

Chapter 4

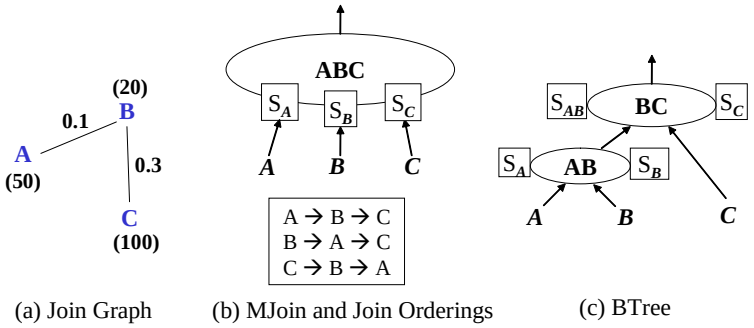
Cost Analysis for Continuous Multi-Join

In this chapter I illustrate cost models for computing CPU and memory costs of mjoin and bjtree respectively.

4.1 Cost Analysis for MJoin

To describe the methods of estimating CPU and memory costs, we use the example query represented as a *join graph* in Figure 4.1(a). A vertex represents an input stream and is marked by the stream name and arrival rate per unit of time. An edge indicates a join predicate between the two streams and is marked by the join selectivity. The join selectivities are assumed to be independent. Given this join graph, Figures 4.1(b) and (c) depict the best mjoin and the best bjtree, respectively. The widely adopted symmetric nested loop join [WA93a] with time-based window constraints

[KNV03, CCC+02, MWA+03] is used for the cost analysis. Each join has monotonically increasing results. Note that the proposed algorithms are general and are not restricted by join algorithms and window constraints used. For ease of exposition, all join predicates are assumed to have the same window size though our technique is not restricted to this case. Each join operator keeps one *state* per input, and each state stores tuples in one window frame from the corresponding input so to join them with future incoming tuples from other inputs.



(a) Join Graph (b) MJoin and Join Orderings (c) BTree

Figure 4.1: Example Query With Two Joins

Figure 4.1(b) shows the optimal join orderings for each input of the mjoin. The ordering $B \rightarrow A \rightarrow C$ indicates that tuples from input B are first inserted to state S_B, and then are joined with tuples in state S_A and S_C in that order. The join orderings minimize the amount of intermediate results so the CPU costs are kept minimal. This implies a positive correlation between CPU and memory.

To estimate the CPU costs of an mjoin, I apply the commonly adopted per-unit-time cost metric [KNV03], in which the CPU cost is the CPU pro-

cessing time required to process all tuples arriving in one time unit. The result can be treated as the amount of CPU per unit time that the system needs in order to keep up with the incoming tuple rates without any processing delay and without tuple accumulation. Terms used in our cost models are explained in Table 4.1. For the mjoin in Figure 4.1(b), the CPU costs consist of the cost spent on processing A, B and C tuples. According to the join orderings in Figure 4.1(b), a new tuple A is first inserted into state S_A (at cost C_i), causing the existing tuples that are now outside the window frame to be deleted from S_A (at cost C_d per tuple). The same tuple A then joins with tuples in state S_B and the joined AB tuples are used to join with tuples in state S_C. A similar process applies to tuples from B and C. The formula to estimate the CPU cost for input A in a unit time is $CPU_A = \lambda_A(C_i + C_d) + \lambda_A|S_B|\sigma_{AB}C_j + \lambda_A|S_B||S_C|\sigma_{ABC}C_j$, in which $\sigma_{ABC} = \sigma_{AB}\sigma_{BC}$, and $|S_B| = \lambda_B W$. So we have:

$$\begin{aligned} CPU_A &= \lambda_A(C_i + C_d) + \lambda_A\lambda_B\sigma_{AB}WC_j + \lambda_A\lambda_B\sigma_{AB}\lambda_C\sigma_{BC}W^2C_j \\ &= \lambda_A(C_i + C_d) + (\lambda_A\lambda_B\sigma_{AB}W + \lambda_A\lambda_B\lambda_C\sigma_{ABC}W^2)C_j \end{aligned} \tag{4.1}$$

By applying the same equation for inputs B and C, the total CPU processing cost for mjoin can be estimated as:

$$\begin{aligned} CPU_{mjoin} &= CPU_A + CPU_B + CPU_C \\ &= (\lambda_A + \lambda_B + \lambda_C)(C_i + C_d) + \lambda_B\lambda_C\sigma_{BC}WC_j \\ &+ 3\lambda_A\lambda_B\lambda_C\sigma_{ABC}W^2C_j + 2\lambda_A\lambda_B\sigma_{AB}WC_j \end{aligned} \tag{4.2}$$

Table 4.1: Terms Used in Cost Models

Term	Meaning
C_i	Cost of inserting a tuple to a state
C_d	Cost of deleting a tuple from a state
C_j	Cost of joining a pair of tuples
λ_A	Average input rate from stream A
λ_B	Average input rate from stream B
λ_C	Average input rate from stream C
σ_{AB}	Selectivity of join $A \bowtie B$
σ_{BC}	Selectivity of join $B \bowtie C$
W	Sliding time-based window constraint
$\|S_A\|$	Number of tuples in state S_A

The memory cost of mjoin can be conveniently estimated as the total number of tuples in all states. The actual memory cost may fluctuate a bit as intermediate tuples may temporarily exist. This temporary cost is being minimized by choosing the optimal join orderings. The total state size is relatively stable and usually accounts for the majority of the memory cost. For simplicity, all tuples are assumed to be of the same size. However, the cost model can be easily extended to take different tuple sizes into account. Overall the estimated memory cost for the mjoin in Figure 4.1(b) is:

$$MEM_{mjoin} = |S_A| + |S_B| + |S_C| = \lambda_A W + \lambda_B W + \lambda_C W \qquad (4.3)$$

4.2 Cost Models For BJTree

The CPU cost of a bjtree also consists of the CPU costs for processing each input stream. In Figure 4.1, a new tuple A is first inserted to the state S_A and on average one old tuple from S_A is deleted from the state. The new tuple then joins with tuples in state S_B. The joined tuples are inserted into the intermediate state S_{AB} and old tuples are being deleted from S_{AB}. These joined tuples finally join with tuples in state S_C. Tuples from input B follow similar steps. Tuples from input C have a shortcut to directly join with tuples in state S_{AB}. By keeping intermediate states, the bjtree saves the CPU of recomputing intermediate results. The cost models to compute the unit CPU costs for inputs A (or B) and C are given as follows:

$$CPU_A = \lambda_A(C_i + C_d) + \lambda_A|S_B|\sigma_{AB}(C_j + C_i + C_d) + \lambda_A|S_B||S_C|\sigma_{ABC}C_j$$

and $CPU_C = \lambda_C(C_i + C_d) + \lambda_C|S_{AB}|\sigma_{BC}C_j$.

Given that $|S_{AB}| = \lambda_A\lambda_B\sigma_{AB}W^2$ and $|S_B| = \lambda_B W$, the total CPU is:

$$\begin{aligned} CPU_{bjtree} = {} & (\lambda_A + \lambda_B + \lambda_C)(C_i + C_d) \\ & + 3\lambda_A\lambda_B\lambda_C\sigma_{ABC}W^2C_j + 2\lambda_A\lambda_B\sigma_{AB}W(C_j + C_i + C_d) \end{aligned} \tag{4.4}$$

The memory cost is again estimated by number of tuples in all states:

$$MEM_{bjtree} = |S_A| + |S_B| + |S_C| + |S_{AB}| = MEM_{mjoin} + \lambda_A\lambda_B\sigma_{AB}W^2 \tag{4.5}$$

The first two terms in Equation 4.4 remain the same for any shape of the bjtree. The third term, which is equivalent to $2|S_{AB}|(C_j + C_i + C_d)$, is

join-order-dependent. Choosing a better join ordering lowers the size of intermediate states, which decreases the memory cost as indicated by Equation 4.5, and also lowers the CPU cost as indicated by Equation 4.4. This shows that within the bjtree method, CPU and memory costs are positively correlated.

4.3 Comparing the Cost Models

As indicated by Equations 4.3 and 4.5, MEM_{bjtree} is always larger than MEM_{mjoin} because the bjtree needs to store extra intermediate states. So the negative correlation between CPU and memory exists when the CPU cost of the bjtree is smaller than the CPU cost of the mjoin. At the first look, this seems to always hold, because without storing intermediate results, the mjoin requires extra CPU cost to recompute any intermediate result used more than once. However, the bjtree also needs extra CPU cost to maintain the intermediate states, that is, for inserting/deleting tuples to/from these states. This is shown when comparing Equations 4.2 and 4.4. After cancelling out common terms, both equations have one extra term left:

$CPU_{mjoin\_diff} = \lambda_B \lambda_C \sigma_{BC} W C_j$, and

$CPU_{bjtree\_diff} = 2\lambda_A \lambda_B \sigma_{AB} W (C_i + C_d)$

It is easy to see that CPU_{mjoin_diff} calculates the CPU cost of recomputing intermediate results BC, while CPU_{bjtree_diff} calculates the cost of maintaining the intermediate state AB. So the CPU and memory have a negative correlation only when $CPU_{mjoin\_diff} > CPU_{bjtree\_diff}$, meaning the cost of recomputing the intermediate results is larger then the cost of

maintaining the intermediate state if these results were stored in the mentioned state.

In general, an optimizer could calculate the CPU costs of both the mjoin and the bjtree using the above cost models. We know for sure that the negative correlation between CPU and memory exists if the CPU cost of mjoin is greater than the CPU cost of the bjtree.

Chapter 5

The MJoin-Init Strategy

5.1 Finding Join Orderings For MJoin

For an mjoin operator, the best join orderings imply the least intermediate results and therefore the least CPU cost. This is the same problem as finding the optimal join ordering for each mjoin input, which is known to be NP-complete [IK84]. Existing join ordering algorithms proposed for continuous mjoin are all heuristics-based greedy algorithms [VNB03, GO03, BMM+04] and thus cannot guarantee optimality.

This section describes the proposed *mjoin-ordering algorithm*, which can find *optimal* join orders for each input of mjoin in polynomial time, when given an acyclic join graph. Our algorithm extends the classic *IK algorithm* [IK84], which has first been proposed for static query optimization to generate *optimal* join orders in polynomial time for acyclic join graph. It requires the precondition that the cost model satisfies the Adjacent Sequence Interchange (ASI) property [MS79]. We first prove that the cost model for

mjoin in Section 4.1 also satisfies the ASI property given an acyclic join graph. This proves the applicability of the concept of the IK algorithm to our continuous mjoin context. We then show how to apply the algorithm to our problem. Lastly, we add a greedy pre-selection step when the join graph contains cycles.

5.1.1 Finding Optimal Join Ordering For Acyclic Joins

We now show that the cost model for computing CPU cost of an mjoin described in Section 4.1 also satisfies the ASI property given an acyclic join graph, therefore proving the applicability of the concept of the IK algorithm to our problem. According to the cost model in Equation 4.1, given n input streams, an acyclic join graph and a join sequence $S = (R1, R2, ..., R_n)$ starting from input R_1[1], the total CPU cost of such a join sequence can be computed as:

$C'(R_1) = \lambda_{R_1}(C_i + C_d) + C_j \sum_{i=2}^{n} \prod_{j=2}^{i} (\sigma_{R_j} \lambda_{R_j} W)$.

In this equation, σ_{R_j} denotes the join selectivity between R_j and inputs in $(R_1, ..., R_{j-1})$. Since the join graph is acyclic and the join sequence starts at R_1, it must be true that R_j only connects to one of the inputs in $(R_1, ..., R_{j-1})$. Since the first term $\lambda_{R_1}(C_i + C_d)$ and the constant C_j are order-independent, they are ignored in the following analysis. The order-dependent part of $C'(R_1)$ is denoted as $C(R_1)$, which is estimated as following:

[1]The join sequence must confirm to the partial order defined in the given join graph. This means that in a rooted tree of the acyclic join graph with root R_1, if R_i is the parent of R_j, then R_i must be placed in front of R_j in the join sequence.

$$C(R_1) = \sum_{i=2}^{n} \prod_{j=2}^{i} (\sigma_{R_j} \lambda_{R_j} W) \tag{5.1}$$

The above equation can be defined recursively as follows:

$C(\Lambda) = 0$	For the null sequence Λ.
$C(R_1) = 0$	For starting input stream.
$C(R_i) = \lambda_{R_i} \sigma_{R_i} W$	For single input $R_i (i > 1)$.
$C(S_1 S_2) = C(S_1) + T(S_1) C(S_2)$	For subsequences S_1 and S_2 in join sequence S.

where T(*) is defined by:

$T(\Lambda) = 1$	For the null sequence.
$T(R_1) = \lambda_{R_1}$	For starting input stream.
$T(R_i) = \sigma_{R_i} \lambda_{R_i} W$	For single input $R_i (i > 1)$.
$T(S_1) = \prod_{k=i}^{j} (\sigma_{R_k} \lambda_{R_k} W)$	For any subsequence $S1$ = $(R_i, ..., R_j)$.

The ASI property is stated in [IK84] by Lemma 5.1 below.

Lemma 5.1 *For arbitrary sequences A and B, and nonnull sequences S_1 and S_2 such that AS_1S_2B and AS_2S_1B are both compatible with the given order constraints, we have $C(AS_1S_2B) \leq C(AS_2S_1B)$ if and only if $rank(S1) \leq rank(S_2)$, where the rank function is defined for any nonnull sequence as $rank(S) = (T(S) - 1)/C(S)$.*

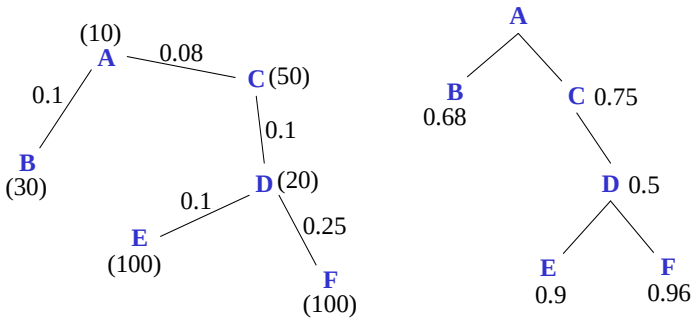
Proof: From the recursive definition, we have: $C(AS_1S_2B) - C(AS_2S_1B) =$

$T(A)[C(S_2)(T(S_1)-1) - C(S_1)(T(S_2)-1)] = T(A)C(S_1)C(S_2)[rank(S_1) - rank(S_2)]$. So the lemma follows directly from this equation. □

Since the cost model for mjoin satisfies the ASI property, the IK algorithm [IK84] can find the *optimal* join ordering for each input of an mjoin in polynomial time, given an acyclic join graph. *Optimal* here means that the total number of intermediate results is the smallest.

Next, we use a query example to describe how to apply the IK algorithm to continuous queries. We illustrate intuitively why it can find an optimal join ordering while existing greedy algorithms cannot. Figure 5.1(a) gives an example acyclic join graph, with each edge marked by its join selectivity and each vertex marked by the stream input rate. The join selectivity is computed as $\frac{\#\ of\ output\ tuples}{\#\ of\ possible\ output\ tuples}$. It falls in the range of [0, 1]. The corresponding tree rooted at input A is depicted in Figure 5.1(b). Each node is marked by the $rank$ computed by $rank(R_i) = (T(R_i) - 1)/C(R_i)$ for an input stream R_i. Here for simplicity, we set the window constraint W to 1. The process of applying the IK algorithm to find the optimal join ordering for the input stream A is illustrated in Figure 5.2.

Starting from the rooted tree in Figure 5.1, the algorithm traverses the tree bottom up. It finds the first node that has more than one child (node D in this example), and checks to see if all its children branches are ordered by non-decreasing ranks. If so, the children branches are merged into one sequence sorted by ranks, shown as the left most tree in Figure 5.2. The next node with more than one child is the root node A. Note that the right branch below node A is not ordered. The algorithm then merges nodes C and D and recomputes the rank for the merged node CD. This is



(a) An acyclic join graph (b) Rooted tree for input A

Figure 5.1: Join Graph and Rooted Tree

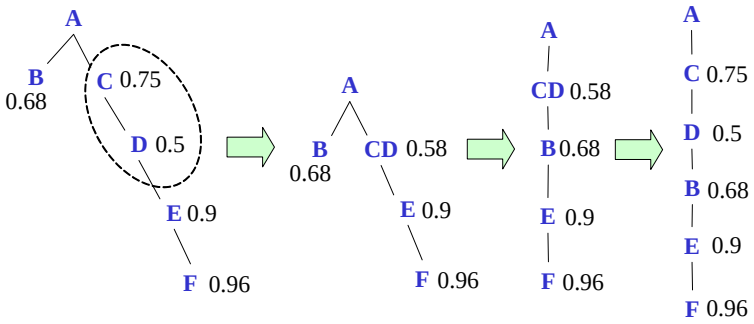


Figure 5.2: Finding the Optimal Join Ordering

repeated until the branch is in order. The two branches under node A are then merged into one. As a final step, the algorithm *un-merges* all combined nodes back to their original orders. The final sequence is the optimal join sequence for input stream A. The same procedure can be applied to find the optimal join sequences for all other input streams.

A greedy algorithm, as commonly assumed in the literature [VNB03, GO03, BMM+04], would choose the next input in the join sequence differently from the IK algorithm. It would select the next input to join if

it produces the smallest amount of intermediate results. During the join ordering process, if the current amount of intermediate results is I, then $I * C(R_i)$ in fact computes the amount of intermediate results generated by joining current chosen inputs and input R_i. Since $C(R) = T(R)$ and $rank(R) = (T(R) - 1)/C(R) = 1 - 1/C(R)$, an input R with smaller $C(R)$ would have a lower $rank$. Therefore a greedy algorithm always selects the input with the lowest $rank$ as the next in the join sequence. Looking back at the left most tree in Figure 5.2, at this step a greedy algorithm would have chosen the node B to be the next to join instead of the subsequence CD. A greedy algorithm makes locally optimal decisions, while the IK algorithm is able to look forward and make globally optimal decisions by selecting the next sequence of inputs with the lowest $rank$.

5.1.2 Heuristic-based Join Ordering for Cyclic Joins

The above algorithm finds the optimal join orders for acyclic join graphs. For a cyclic join graph, the problem is again NP-complete [IK84]. [KBZ86] proposes an algorithm to first find a minimum spanning tree of the cyclic join graph before applying the IK algorithm [IK84]. The weight of each edge is its selectivity, based on the heuristic that smaller selectivity is more likely to result in smaller amount of intermediate results. However, for a continuous query, the amount of intermediate results is not determined by the selectivity alone, but is also significantly affected by the stream input rates. So in my work, I consider both selectivity and input rates when finding the minimum spanning tree. The weight of an edge connecting two vertices A and B is computed as $\lambda_A \lambda_B \sigma_{AB}$, instead of σ_{AB}. The IK algo-

rithm is then applied to the generated min spanning tree.

5.1.3 Considering Cartesian Product

Cartesian product has been excluded by many query optimization algorithms, especially the ones for continuous queries, due to its potential high cost. However, some research [VM96, OL90, CM95] have shown that considering cartesian product can be helpful to improve the performance of the optimized query plan.

Figure 5.3 shows an example when considering cartesian product can benefits the query plan. Given the join graph in Figure 5.3 (a), for input tuples from B, the only possible joining sequence is $B \rightarrow A \rightarrow C$. However, in this example since input B and C both have much smaller input rate than input A. So we can add one edge connecting B and C and assign the edge a selectivity of 1 to form a cartesian product between B and C, as shown in Figure 5.3 (b). This enables the join order $B \rightarrow C \rightarrow A$, which generates a smaller amount of intermediate results than using the order of $B \rightarrow A \rightarrow C$. To be more exact, assuming W = 5, using the cost model defined in Equation 5.1, the join order $B \rightarrow A \rightarrow C$ needs 750 join operations, while the join order $B \rightarrow C \rightarrow A$ only needs 550 join operations.

So we can further improve the join ordering algorithm described previously to take into consideration the benefits of cartesian product. This can be done by adding new edges with selectivity of 1 to the given join graph. Notice that by adding these new edges, a previously acyclic join graph will become cyclic. However, this now cyclic join graph will enable the join ordering algorithm to find join orders that are at least as good as

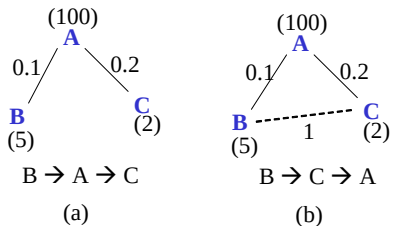


Figure 5.3: With or without Cartesian Product

the optimal join orders from the original acyclic graph. This is because if an edge of cartesian product is not beneficial, it would have larger weight than its neighboring edges and thus would not be picked to be in the minimum spanning tree of the cyclic graph.

5.1.4 Overall Join Ordering Algorithm.

The overall join ordering algorithm and its helper functions are described in Algorithm 1. $Find\_MinSpanning\_Tree()$ generates a minimal spanning tree. The algorithm then calls function $Merge\_Branches()$ to generate join order for each vertex in the join graph. Function $Un\_Merge()$ then unmerges previously merged nodes back to their original orders. For a join graph with n vertices, finding a minimum spanning tree among the total $n(n-1)/2$ edges takes $O(n^2log(n))$ time. The procedure of generating the join sequence for each input stream takes $O(nlog(n))$. Thus generating the join orders for all n input streams takes $O(n^2log(n))$. The overall algorithm has $O(n^2log(n))$ time complexity.

Algorithm 1 The MJoin-Ordering Algorithm

```
//Input: joinGraph
//Output: Orders, array of join orders for each vertex.

Add_CartesianProduct_Edges(joinGraph);
Find_MinSpanning_Tree(joinGraph);

for each vertex v in joinGraph do
  rootNode = Generate_Rooted_Tree(v);
  joinOrder = Merge_Branches(rootNode);
  Un_Merge(joinOrder);
  Add joinOrder to Orders.
return Orders;
/* helper functions: */
Merge_Branches(snode) {
  for each child node cnode of snode do
    Normalize_Branch(cnode);
  Sort all children of snode by increasing rank;
  return joinOrder for rootnode;
}
Normalize_Branch(snode) {
  if (snode has no child)   return;
  if snode has more than one child
    Merge_Branches(snode);
  Get child node cnode of snode;
  if rank(cnode) < rank(snode)
    Merge the two into cnode and compute its new rank;
  Normalize_Branch(cnode);
}
```

5.2 Generating Hybrid Tree from MJoin

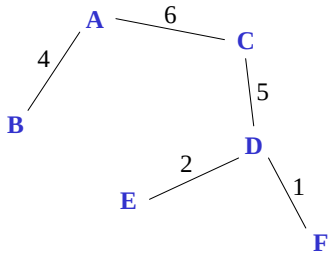
When the optimized mjoin produced by the above *mjoin-ordering algorithm* is not a qualified plan, we then further tune the balance between CPU and memory utilizing their negative correlation. The goal is to increase the memory cost by selecting intermediate results to keep so that the CPU cost of recomputation can be saved. In this section, we describe our *state-selection algorithm* to iteratively select intermediate results to keep in order to generate a qualified hybrid tree.

Selecting intermediate states can be viewed as selecting edges in a join graph. Given a join graph and the join orderings, we consider two performance factors: *edge frequency* and *edge weight*. The *edge frequency* is defined as how many times an edge appears in the join sequences. The *edge weight* connecting vertices X_1 and X_2 is defined as $X_1 X_2 \lambda_{X_1 X_2}$. This is proportional to the estimated size of its intermediate state. Heuristically, the higher the edge frequency, the more likely that storing the intermediate results for this join can save CPU cost. Based on this heuristic, the state selection algorithm chooses the edges with the largest frequency/weight ratio.

Using the join graph in Figure 5.1, the computed join sequences for all input streams are shown in Figure 5.4(a). In Figure 5.4(b), each edge in the join graph is marked by its edge frequency. The algorithm each time selects an edge with the largest frequency/weight ratio as the candidate state. However, it is not guaranteed that doing so will assure a decrease in the CPU cost, even for an edge that appears in all join sequences. This

A → C → D → B → E → F
B → A → C → D → E → F
C → A → D → B → E → F
D → C → A → B → E → F
E → D → C → A → B → F
F → D → C → A → B → E

(a) Computed Join Sequences



(b) Frequency Counts

Figure 5.4: Counting Edge Frequencies

is because as discussed in Section 4.3, the negative correlation exists only when the CPU cost spent on recomputing a intermediate state is larger than the cost spent on maintaining the state. Such condition is not guaranteed to be satisfied by choosing the edge with the largest ratio.

Thus each time an edge XY is selected as a candidate state, the state selection algorithm checks if it is beneficial to keep this intermediate state. It does so by calling the join ordering algorithm as described in Section 5.1, while treating the vertices X and Y as one merged vertex with the input rate now set to $\lambda_X \lambda_Y \sigma_{XY}$. If the cost of the new join sequences is smaller than the cost of the original sequences before merging X and Y, the state S_{XY} is then selected and the two connecting vertices are merged. The same state selection procedure is applied iteratively to the new join graph. The algorithm terminates when one of the following conditions holds:

- None of the remaining intermediate states is beneficial, or (2) the first time M'_m surpasses M_a, with M_a denoting the total memory in the system, and M'_m denoting the memory cost of the current hybrid tree.

- $M'_m > M_a$ and $C'_m < C_a$.

Although the generated hybrid tree may not be optimal, this algorithm guarantees that it is better than the mjoin because it has less CPU cost. The overall memory cost is guaranteed to be less than the system memory constraint.

Figure 5.5 depicts how an hybrid tree is generated during the state (edge) selection process. Given a 6-way mjoin and its join graph, suppose $D - E$ is the next edge to be selected based on the state selection algorithm described earlier, the 6-way mjoin is then broken into a 5-way join and a binary join, as shown in the middle of Figure 5.5. The intermediate results of the newly added join $D \bowtie E$ would now be kept in one of the states in the mjoin on top of it. The join graph is updated by merging the edge $D - E$ into one node. The output rate of the new join DE is computed as $\lambda_D \lambda_E \sigma_{DE} W$. The same process is then repeated when edge $A - B$ is selected.

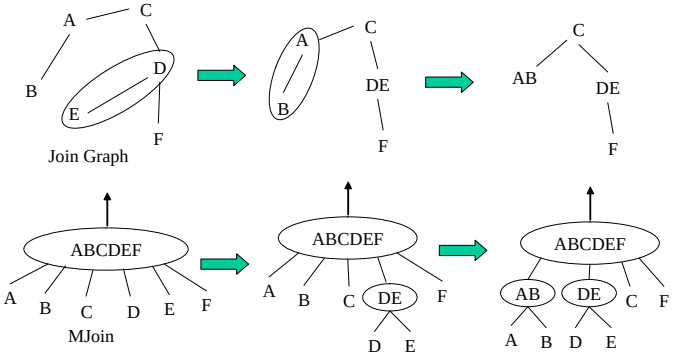


Figure 5.5: Generate Hybrid Tree By State Selection

The above process add only binary join operators into the hybrid tree.

This limits the optimizer's search space. To solve this problem, we add operator merging steps to merge several binary joins into one bigger mjoin operator. This is applied only when doing so can save both memory and CPU costs. Consider the hybrid tree in Figure 5.6(a) when edge $C - DE$ is selected next. A join operator CDE is created for this edge selection, as shown in Figure 5.6(b). At this point, we try to merge the new join operator with its children join operators, in this case operator DE. This would create a new join tree as depicted in Figure 5.6(c). If the new hybrid tree has both less CPU and less memory costs than the hybrid tree in Figure 5.6(b), it is then kept as the current hybrid tree.

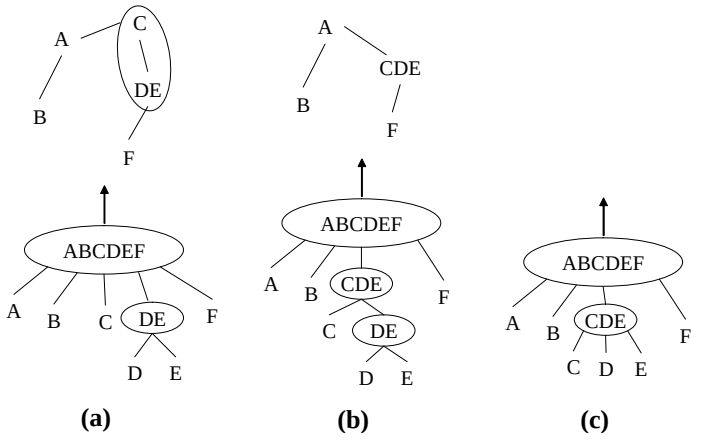


Figure 5.6: Operator Breaking and Merging

The overall algorithm is shown in Algorithm 2. For a join graph with n input streams, at most $(n-1)n/2$ edges exist in the join graph. Each time an edge is selected as a candidate, the algorithm needs to recompute the join sequences. This takes $O(n^2 + n^2 log(n))$ time. Therefore the overall worst case complexity of the algorithm is $O(n^4 log(n))$.

Algorithm 2 The State Selection Algorithm

```
//Input: joinGraph and joinOrders;
//Output: newGraph, representing a hybrid tree.
//C_m and M_m are CPU and memory costs of joinOrders;
//C_a and M_a are available system CPU and memory.
while (C_m > C_a)&&(M_m < M_a) do
  edgeSet = Set of candidate edges;
  while (edgeSet != null) do
    Choose edge with max freq/weight in edgeSet;
    Merge two vertices of edge, get newJoinGraph;
    newOrders = MJoin_Ordering(newJoinGraph);
    new_C_m = CPU cost of newOrders;
    if (new_C_m < C_m) then
      joinOrders = newOrders; C_m = new_C_m;
      M_m = Memory cost of joinOrders;
      if(M_m > M_a) return newGraph;
      newGraph = newJoinGraph;
      Merge edge in newGraph with each child if this decreases both
      resource usages;
      break; // from inner while loop.
    end if
  end while
  if (edgeSet.size() == 0) break; //from outer while loop.
end while
return newGraph;
```

Chapter 6

The BTree-Init Strategy

6.1 Generating BJTree

The best bjtree is the one that has the least memory cost (minimal intermediate states) and hence the least CPU cost (thanks to positive correlation) among all semantically equivalent bjtrees. Finding such an optimal bjtree is NP-complete [IK84]. In order to reduce the complexity, query optimizers usually limit their search spaces to linear trees (often left-deep trees) [SAC+79]. However, limiting the search space can miss some potentially good query plans. In fact, research on continuous queries has shown that bushy trees in many cases have better performance than linear trees because join operators in bushy trees have more symmetrically balanced inputs [VN02a, VNB03].[1]

In this section, we describe our *min-state algorithm* that finds a bjtree

[1] In a linear bjtree each join has one stream input and one intermediate input, except for the leaves. In a bushy both inputs to a join operator can be intermediate results.

with small total intermediate states in polynomial time. For star joins, the algorithm guarantees to find an *optimal bjtree*. In general, the algorithm can generate either linear or bushy trees, and it can be applied to both acyclic and cyclic join graphs.

The min-state algorithm is illustrated using an example of a 5-way join in Figure 6.1. The input to the algorithm is a join graph. The weight of an edge X_1X_2 is computed as $\lambda_{X_1}\lambda_{X_2}\sigma_{X_1X_2}$. Starting from the original join graph in Figure 6.1(1), the algorithm each time picks the current smallest weighted edge and forms a join. In this example, edge AD is picked first and forms the join $A \bowtie D$. The edge is then merged into a single vertex AD. Its input rate is updated as the weight of the original edge AD. The input B has a join predicate with both A and D. So the selectivity between vertex B and the new vertex AD is also updated by multiplying the selectivities on the original edges BA and BD. The algorithm then continues to pick the smallest weighted edge from the updated join graph shown in Figure 6.1(2). This time the edge CE is picked and a new join $C \bowtie E$ is formed. The two involved vertices are merged and the graph is updated accordingly. Since the new join ($C \bowtie E$) and the existing join ($A \bowtie D$) have non-overlapping inputs, this indicates that the algorithm will eventually produce a bushy tree. The same procedure is repeated in Figure 6.1(3). The final output bjtree is depicted in Figure 6.1(4) at the bottom. When the algorithm is over, the original join graph would be merged into a single vertex.

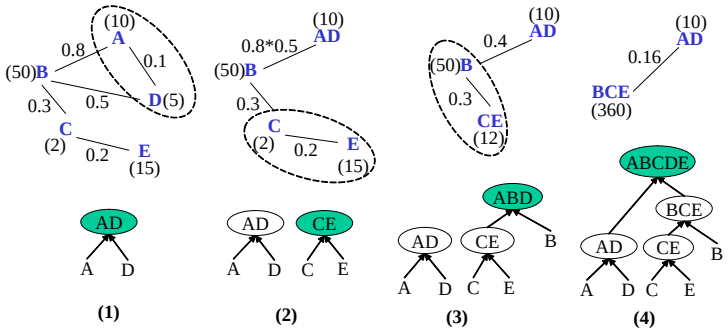


Figure 6.1: Min-State Algorithm Walkthrough

6.1.1 Considering Cartesian Product

Same as the join ordering algorithm discussed in Section 5.1, the min-state algorithm can also take advantage of cartesian products. Figure 6.2 shows the benefits of considering cartesian product to reduce the size of the intermediate states. The most number of cartesian product edges that can be added to the join graph is the total number of edges minus the total number of existing edges in the given join graph.

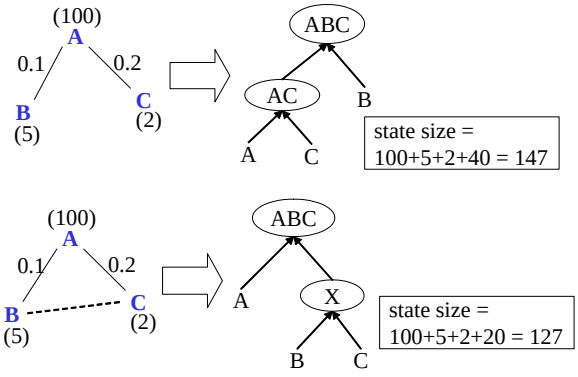


Figure 6.2: Potential Benefits of Cartesian Products

6.1.2 The Overall Min-State Algorithm

Given a join graph with n vertices, the largest possible number of edges is $n(n-1)/2$. So the overall algorithm complexity is $O(n^2 log(n))$.

Algorithm 3 The Min-State Algorithm

```
//Input: initial joinGraph
//Output: the generated bjtree
Add_CartesianProduct_Edges(joinGraph);
while (there are edges left in the joinGraph) do
  Select smallest weighted edge;
  Merge selected edge;
  Update selectivities of affected edges;
  Update arrival rates of affected vertices;
end while
```

6.1.3 Optimality of Generated BJTree.

The pseudo-code of the min-state algorithm is shown in Algorithm 3. It is a greedy algorithm and does not guarantee to always find the optimal bjtree with the smallest state size.

Figure 6.3 depicts a case when the min-state does not generate the optimal query plan. The upper half of Figure 6.3 shows the tree that generated by the min-state algorithm, and the lower half shows the optimal tree.

However, for a star join, as depicted in Figure 6.4, the min-state algorithm always generates an optimal bjtree. Intuitively, this can be explained as follows: When any edge in the star join graph is picked, the edge is merged into the node in the center of the join graph which connects to all the other nodes. So the update on the input rate of the central node affects the rest of the edges in the join graph equally. Therefore in the star join

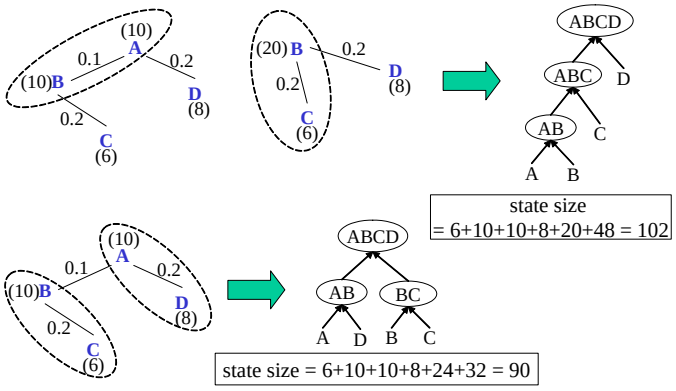


Figure 6.3: An example that the algorithm generates sub-optimal plan.

graph, the local minima is also the global minima. For other types of join graphs, this is no longer true because one node is not guaranteed to connect to all the other nodes. Therefore updating one node may have different impacts on different edges in the join graph.

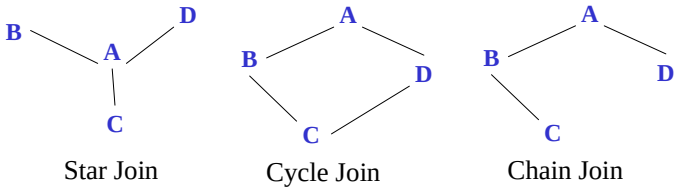


Figure 6.4: Different Types of Join Graph.

When the query plan generated by the min-state algorithm is sub-optimal, it may affect the precision of the optimization strategy in two ways, namely *false negative* and *false positive*. *False negative* here means that the optimizer fails to find a qualified plan, while in fact a qualified plan does exist given the allocated system resources (C_a and M_a). *False positive* means a qualified plan is found, but in fact a qualified plan does not exist given C_a and

M_a. It is clear that the imprecision caused by a sub-optimal bjtree is always of the false negative type. This in fact makes the optimizer more conservative, because the optimizer may end up requiring the system to allocate more resources than the query actually needs. However, the false positive would here gives the resource allocator the wrong assurance that the allocated resources are sufficient, while in fact they are not. Finding a good plan quickly versus finding an optimal plan is very often a tradeoff between efficiency and precision. The min-state algorithm is chosen for its efficiency, which is much needed by continuous query processing.

6.2 Generating Hybrid Tree from BJTree

If the bjtree generated by the min-state algorithm above is not a qualified plan, we then utilize the negative correlation to further balance the memory and the CPU costs. The optimizer now aims to save memory by removing intermediate states and merging existing join operators. This can increase CPU costs because the removed states now need to be recomputed. In this Section, we describe our *state-removal algorithm* to select intermediate states to remove in order to generate qualified hybrid tree.

The process of state removal is done iteratively, with each step removing one intermediate state and merging two join operators. Figure 6.5 shows an example of applying state-removal to a bjtree. The initial bjtree generated by the min-state algorithm for a 6-way join query is shown in Figure 6.5 (a). Four intermediate states exist in this bjtree, each represented by a small rectangle at the sides of the three operators on the top. Now

let us assume that the circled intermediate state at join operator BC is selected to be removed. This is achieved by merging join operators AB and BC into a single mjoin operator ABC, which results in the hybrid tree in Figure 6.5(b). We then need to apply the mjoin-ordering algorithm, as described in Section 5.1, to find optimal or good join orderings for the new mjoin operator ABC. The same process is then repeated until a qualified hybrid tree is found, or all the candidate intermediate states are removed, which indicates that the query plan has been merged to a single mjoin.

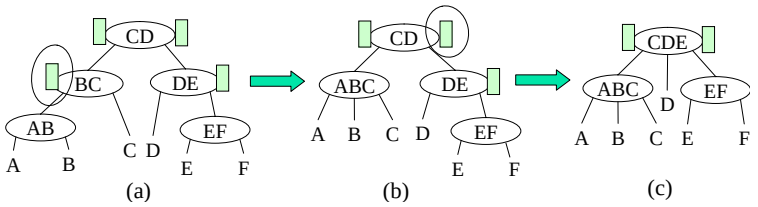


Figure 6.5: Removing State by Merging Joins.

The important remaining problem now is how to pick intermediate states to remove in order to quickly lead to a qualified query plan. For each candidate intermediate state, we need to consider two factors: the memory saved by removing the state and the possible CPU increase due to recomputing the intermediate results. Suppose the CPU costs of the old plan and of the new plan after operator merging are C_{old} and C_{new} respectively, the CPU increase can be computed as $CPU_{increase} = C_{new} - C_{old}$. The memory saving is in fact the size of the removed intermediate state.

Intuitively, each time we should remove an intermediate state that has the smallest ratio of $CPU_{increase}/state\_size$. This ratio is referred to as the *state quality ratio*. The optimizer removes the intermediate state with the

smallest state quality ratio by merging two corresponding join operators into a larger mjoin operator. The first operator is the one that contains the selected state. The second operator inputs tuples to the first operator, and the input tuples also would be inserted to the to-be-removed state. After two operators are merged into a larger mjoin operator, the join orderings of the new operator are recomputed. Note that sometimes merging two operators into a larger mjoin may save both CPU and memory. This is when the maintenance cost of the intermediate state overpasses the cost of recomputing such a state, as discussed in Section 4.3. In this case, the $CPU_{increase}$ factor is actually a negative number.

Algorithm 4 illustrates the state removal algorithm. For a join query with n input streams, there are at most $n-1$ intermediate states in a query plan. Therefore, in the worse case the state selecting and operator merging process of the algorithm may be repeated $n-1$ times. Since the join ordering algorithm takes at most $O(n^2log(n))$, the total running time is bounded by $O(n^2log(n))(n-1)(n-2)/2 = O(n^4log(n))$.

6.3 Discussion on Qualified Plans

So far all qualified query plans for the same query are considered to be equal because using any of them will result in the same query output rate, as discussed in Chapter 2. However, among all qualified query plans, some may require less memory cost or less CPU cost than others. The plan that consumes less resources can be more resistant to the changing stream characteristics. Therefore, between two qualified plans, an optimizer should

Algorithm 4 The State Removal Algorithm

```
//Input: A BJTree generated by min-state algorithm.
//Output: The generated HybridTree
HybridTree = BJTree;
i_states = set of intermediate states in HybridTree;
while (i_states is not empty) do
  old_cpu = CPU cost of HybridTree;
  min_ratio = largest number;
  selected_state = null; selected_tree = null;
  for (each s in i_states) do
    op1 = join operator contains s;
    op2 = join operator with output tuples inserted to s;
    Merge op1 and op2 into new_op and get newTree;
    Compute join orderings of new_op;
    new_cpu = CPU cost of newTree;
    ratio = (new_cpu − old_cpu)/(size of state s);
    if (ratio < min_ratio) then
      min_ratio = ratio; selected_state = s;
      selected_tree = newTree;
    end if
  end for
  HybridTree = selected_tree;
  Remove selected_state from i_states;
  return HybridTree if it is a qualified plan;
end while
return HybridTree;
```

favor the one with both less memory cost and less CPU cost.

To compare among continuous query plans, we use the notions of *dominating plans* and *non-comparable plans*. Given two continuous query plans, p1 and p2, if p1 has both less memory cost and less CPU cost than p2, we say p1 is the *dominating plan* between the two. However, if p1 has less memory but more CPU than p2, or p1 has more memory but less CPU than p2, we say that p1 and p2 are *non-comparable plans*. In Figure 6.6, if each dot represents a qualified query plan, the dots connected by the line are all the dominating query plans that are non-comparable to each other. Among all qualified query plans, we are interested in the set of maximally dominating query plans D, meaning for a plan p in D, no other plans exist in D that dominate p. Clearly, all plans in the set of maximally dominating plans are non-comparable with one another. Figure 6.6 depicts such a scenario. Assuming each dot represents a qualified query plan, the darker dots connected by a line are the maximally dominating query plans.

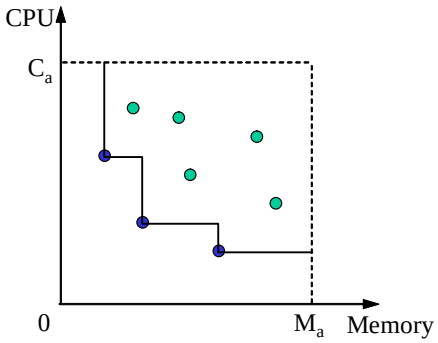


Figure 6.6: Dominating Plans.

The concept of the dominating plans parallels the problem of *skyline*

queries[2][BKS01] or the problem of finding *Pareto curve* in multi-objective optimization [PY01][3].

The four algorithms proposed in Chapters 5 and 6 are all designed to return the first encountered qualified plan without further exploration. This saves optimization time but the returned qualified plan may not be the best that can be found by the optimizer in terms of plan dominance. This is a trade-off between plan quality and optimization time. A viable solution is to assign a certain exploration time to each algorithm, which allows the algorithm a chance to possibly explore further. When the time expires, if the algorithm has found multiple qualified plans, it can choose to return one or all maximally dominating query plans found in the given time. The dominating query plan chosen to return may depend on which resource factor (memory or CPU) is the more ample one in the current system or which one is expected to become more critical in the near future.

[2]Skyline queries concerns database queries to find data points that satisfy multiple conditions simultaneously.

[3]Similar concepts also exist in the larger research family of multi-objective optimization [Ste86], which is an important problem that exists in many other research fields besides computer science.

Chapter 7

The Exhaustive Search Strategy

7.1 The Multi-Join Search Space

To find a qualified query plan, an exponential-time solution is for the optimizer to use an exhaustive search algorithm to cover query plan search space. Exhaustive search is valued for its thoroughness but is also hindered by its high costs. Dynamic programming is an efficient method to implement an exhaustive search compared to full enumeration and thus has been widely adopted in existing database systems [SAC+79]. To decrease the cost, the search space is usually restricted to left-deep trees [SAC+79]. This restriction implies that optimal query plans may be missed by the search algorithm. In this section, we present a bottom-up dynamic programming algorithm to find a qualified plan for multiple join continuous queries. The proposed dynamic programming algorithm differs from traditional algorithm in several aspects, including search space used, cost models applied and termination criteria. Theoretically, if the search space covers all pos-

sible query plans, the exhaustive search method is guaranteed to find the desired plan if there exists one.

Instead of the left-deep tree search space, we ideally would need to adopt the search space that contains all mjoin trees, i.e., the search space that contains all the possible continuous multi-join query plans. It includes linear trees and bushy trees, and each join operator in the tree can be either a binary or an mjoin. The search space covers all possible mjoin trees (including mjoin and btrees) and thus guarantees that the exhaustive search is able to find a qualified plan, if one exists. The complete search space contains a larger number of possible query plan shapes as compared to the left-deep tree search space. On the other hand, the use of pipelining symmetric join algorithms helps to remove some query plans from the search space. For example, a binary join $A \bowtie B$ is equivalent to a binary join $B \bowtie A$ in continuous queries due to its symmetric input processing. Therefore, the two join trees depicted in Figures 7.1 (a) and (b) are treated as equivalent in continuous query optimization, although they are usually considered as two different query plans in static query optimization.

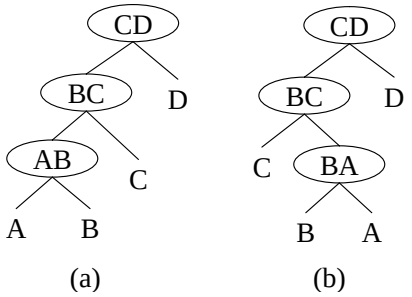


Figure 7.1: Two Equivalent Multi-Join Trees

The cost models described in Chapter 4 are used in our algorithm to compute memory and cpu costs of partial joins (sub-plans) generated during the bottom-up dynamic programming process. The memory and cpu costs are used to identify a qualified plan. The two costs of a plan can also be used to detect the subplans that have both their memory cost and cpu cost beyond the system thresholds and thus are impossible to lead to a qualified query plan. These subplans can be filtered out early on in the dynamic programming process to decrease the optimization cost.

In static query optimization, a dynamic programming process terminates when it finds the optimal solution, which is the plan that has the smallest cost in the given search space. For continuous query optimization, however, the search process can be terminated once a qualified plan is found. Note that given two qualified query plans P1 and P2, it is possible that P1 may have both less memory cost and less cpu cost than P2, and therefore P1 should be "favored" by the optimizer over P2.

7.2 Bottom-up Dynamic Programming

We now illustrate the main steps of our bottom-up dynamic programming algorithm to find a qualified query plan. Given an n-way join query, the algorithm takes as input the system statistics measured at run time. These statistics include system resource consumptions and availabilities, stream input rates and selectivities of join predicates between join columns. The algorithm builds a table of all possible multiple join plans with the number of input streams ranging from 2 to (n-1). It starts from two-way joins (n=2)

and gradually constructs larger joins based on previously computed partial query plans. Finally the algorithm constructs query plans for n-way joins and outputs a qualified plan based on the given data statistics.

Figure 7.2 shows the process of building the query plan table using bottom-up dynamic programming for a 4-way join among streams A, B, C and D. The first column indicates the partial joins by the names of their input streams. To generate a new join, we use a previously existing partial join and add a new stream to it. The second and the third columns indicate the previously existing partial joins and the new stream that is being used to construct the join in the first column. The last column lists all the possible ways to construct the join, each corresponding to a distinctive query plan. For each query plan, we compute its memory and cpu costs. The last row of Figure 7.2 is the final 4-way join. The algorithm terminates once if finds a qualified plan to execute the 4-way join.

JoinEntry	Origins	QueryPlans			
A	Ø + A	A			
B	Ø + B	B			
C	Ø + C	C			
D	Ø + D	D			
AB	A + B	(AB)			
AC	A + C	(AC)			
AD	A + D	(AD)			
BC	B + C	(BC)			
BD	B + D	(BD)			
CD	C + D	(CD)			
ABC	AB + C	((AC)B)	(A(BC))	(ABC)	((AB)C)
ABD	AB + D	((AD)B)	(A(BD))	(ABD)	((AB)D)
ACD	AC + D	((AD)C)	(A(CD))	(ACD)	((AC)D)
BCD	BC + D	((BD)C)	(B(CD))	(BCD)	((BC)D)
ABCD	ABC + D	(((AD)C)B)	((A(CD))B)	...	

Figure 7.2: Bottom-up Dynamic Programming for 4-way Join

One important problem is how to avoid duplicate computations during the query building process. It is possible that a join of input length p can be constructed by different partial joins of length $p-1$. For example, join ABC can be constructed by adding stream C to partial join AB, or it can also be constructed by adding stream B to partial join AC. To avoid such duplicate computation, we assign an ID to each stream. Each time a new stream is added to an existing partial join, the stream must have larger ID than all the input streams in the partial join. For example, for partial join AC, we can only add stream D to it to construct join ACD, but stream B cannot be added to partial join AC because it has smaller ID than stream C. We also apply early filtering to further save the computation costs. Early filtering here means that if a query plan has both memory cost and cpu cost beyond system thresholds, it is discarded right away and will not be used to construct any new query plans.

Because the search space of the algorithm contains all possible mjoin trees, there are many possible ways of combining an existing join plan and a new stream input. The possible combinations can be classified into two categories: 1) the new stream can be merged with an existing join operator in the plan, and 2) the new stream can be merged with a queue in the query plan to create a new binary join operator. Figure 7.3 shows the two types of combinations when a query plan (AB) is merged with a new stream C to create all possible query plans for join ABC.

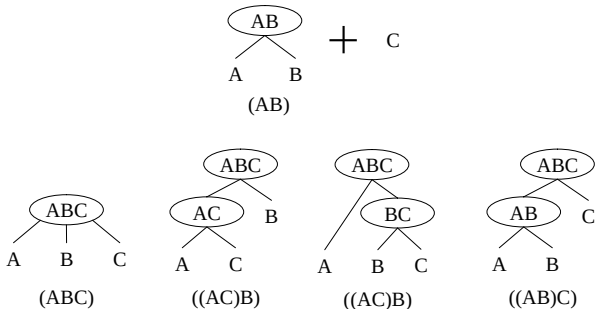


Figure 7.3: Generate New Query Plans by Merging

7.3 The Overall Exhaustive Search Algorithm

Algorithm 5 describes the pseudo-code of the above dynamic programming algorithm to exhaustively search for a qualified query plan given the mjoin tree search space.

Algorithm 5 Bottom-up Dynamic Programming.

```
//Input: N-way join query
//Output: A qualified query plan or null.
QueryPlans; //Arraylist of query plans starting as empty
i = 0;        //points to current position of QueryPlans.
//initialize QueryPlans.
Assign streamID starting from 0 to each stream.
Insert each of the N input streams to QueryPlans.
index = 0; //pointer to current partial plan in QueryPlans
While (index < size of QueryPlans) do
{
 Get query plan qp at position index in QueryPlans;
 largestStream = stream with largest streamID in qp;
 for (each stream S with ID larger than largestStream)
 {
  Generate set of plans newplans by merging qp and S;
  for (each plan P in newplans)
  {
    memory = memory cost of P;
    cpu = cpu cost of P;
   if (P has N inputs and is a qualified plan)
    return P.
   if (either memory or cpu is less than the system threshold)
     Add P to QueryPlans.
  }
 }
index++;
}
```

Chapter 8

Experimental Evaluation

I have implemented the proposed optimization strategies in the DCAPE continuous query system [RDS+04, LZJ+05]. A large number of experiments have been conducted to thoroughly examine the ability of the proposed optimization algorithms to generate qualified query plans under various resource availabilities. In this section, I present two main aspects of the experimental results. First, I verify the cost analysis in Chapter 4 by comparing the performances of mjoin, bjtree and hybrid tree under different system resource availabilities. These query plans are all generated using the optimization algorithms proposed in this paper. Secondly, I compare the optimization effects of the two optimization strategies, namely mjoin-init and bjtree-init strategies. In particular, I compare their optimization time, ability of finding qualified query plans and the resource consumptions of the qualified plans generated by the two methods. The experimental results for these two aspects are reported in Sections 8.1 and 8.2 respectively.

The data generator in the query system generates tuples with arrival patterns modeled as Poisson process. The mean inter-arrival delay between two consecutive tuples is exponentially distributed to model the Poisson arrival pattern. All implementation is done in Java. All experiments are conducted on a machine running windows XP with a 1495MHz processor and 512MB main memory.

8.1 Verifying Cost Analysis

I used various sets of experiments to compare the performances of the bjtree, mjoin and hybrid tree in various system resource settings. The three types of query plans are generated by using the optimization algorithms proposed in Chapters 5 and 6.

The experiments are categorized into four different sets based on the availabilities of memory and cpu:

- In set 1, I apply sufficient memory and CPU for executing both the bjtree and the mjoin. Therefore both are qualified plans according to our cost analysis. The distributions of CPU/memory consumptions of the mjoin and bjtree are depicted in Figure 8.1 (a).

- In set 2, I use high stream rates and selectivities to make the query CPU-intensive. Since bjtree generally requires less CPU than mjoin, bjtree is more likely to be qualified than mjoin (Figure 8.1 (b)).

- In set 3, I use lower stream rates and selectivities. More importantly, I restrict the system memory to a much lower value. Since mjoin gen-

erally requires less memory than bjtree, mjoin is more likely to be qualified than the bjtree (Figure 8.1 (c)).

- In set 4, I use high stream rates and selectivities, while restrict the system memory. Both the bjtree and the mjoin are likely not qualified. Instead a hybrid tree is generated as a qualified plan (Figure 8.1 (d)).

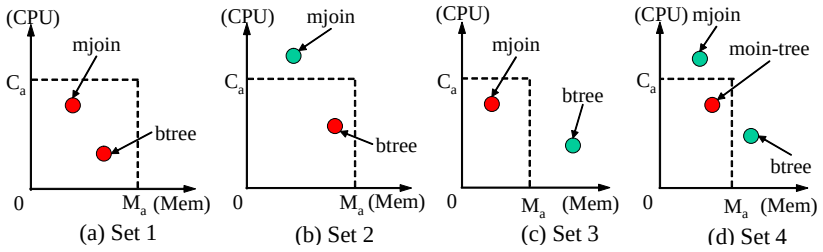


Figure 8.1: Experimental Sets

The amount of available memory is controlled by setting the initial and maximum java heap sizes using the command "java -Xms(size) -Xmx(size)". Amount of available CPU (C_a) remains the same for all experiments since they are all run on the same machine. The effects of different CPU availabilities are achieved by increasing/decreasing stream and query parameters, including stream arrival rates, window sizes and join selectivities.

Table 8.1 lists the parameter configurations in the four sets of experiments. The parameters include stream arrival rates, join selectivities, window sizes and allocated java heap sizes. λ_A is the input rate of stream A in tuples/second. σ_{AB} denotes the selectivity between streams A and B. W stands for the window size in milliseconds (ms). The java heap size (M_a) is in MB.

Table 8.1: Parameter Configurations in Experiments

Set	M_a	λ_A	λ_B	λ_C	λ_D	σ_{AB}	σ_{BC}	σ_{CD}	W
1	300	20	20	20	—	0.05	0.5	—	5000
2	300	20	20	50	—	0.02	0.5	—	15000
3	30	10	10	10	10	0.1	0.15	0.1	15000
4	30	20	20	20	20	0.02	0.2	0.05	50000

Experiment Set 1: The results for set 1 are shown in Figures 8.2 and 8.3. In this set both the mjoin and bjtree are qualified plans according to our cost analysis. We can see that both plans indeed have similar throughputs in Figure 8.2. Sufficient CPU ensures that new tuples can be processed quickly so the number of tuples accumulated in input queues are close to 0. This is true for both mjoin and bjtree, shown as the bottom two lines in Figure 8.3. The major performance difference is the state size. Since the bjtree needs to store intermediate results, its total state size is much larger than that of mjoin. The top two lines in Figure 8.3 clearly display this trend.

Experiment Set 2: In this experiment, the bjtree is qualified but the mjoin is not. Figure 8.4 shows that the bjtree has much faster throughput (more than 100% improvement) than the mjoin. Since the mjoin is not a qualified plan, newly arriving tuples cannot be processed right away and thus accumulate quickly in stream input queues, as shown in Figure 8.5, while the bjtree processes new tuples right away and keeps the input queues small.

Experiment Set 3: In this experiment, the query is given enough CPU resources so the CPU costs of mjoin and bjtree are both under the allocated threshold. The two plans have similar throughputs at the beginning of

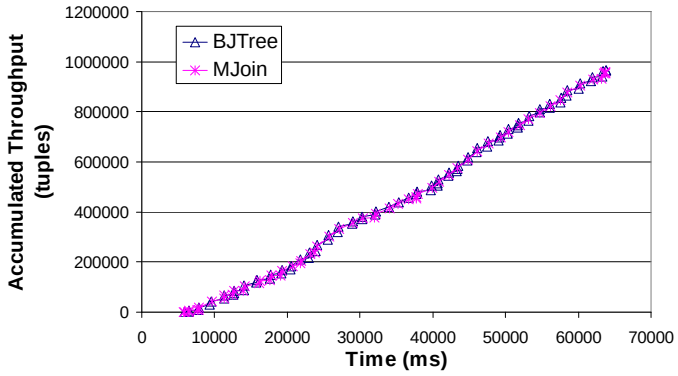


Figure 8.2: Accumulated Throughput (Set 1)

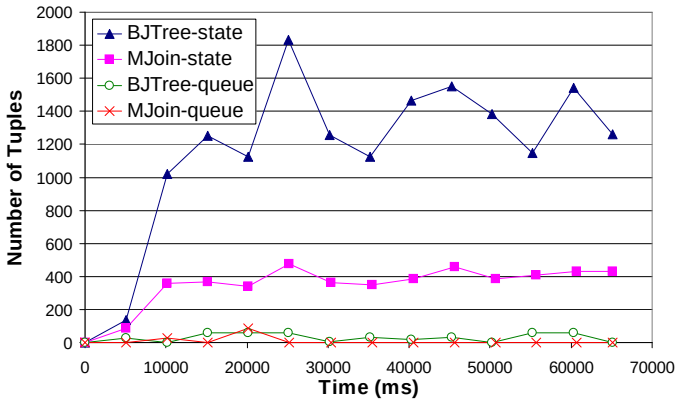


Figure 8.3: Tuples in States/Queues (Set 1)

the experiment, as shown in Figure 8.6. Figure 8.7 displays the measured memory consumptions. The memory of the bjtree keeps on accumulating and quickly reaches the system memory threshold (M_a=30MB) at around 50,000ms. The memory consumed by the mjoin is smaller and averages around 12MB after the start-up stage. The fluctuations of the memory usage in the mjoin are due to the temporary memory consumed by the intermediate results.

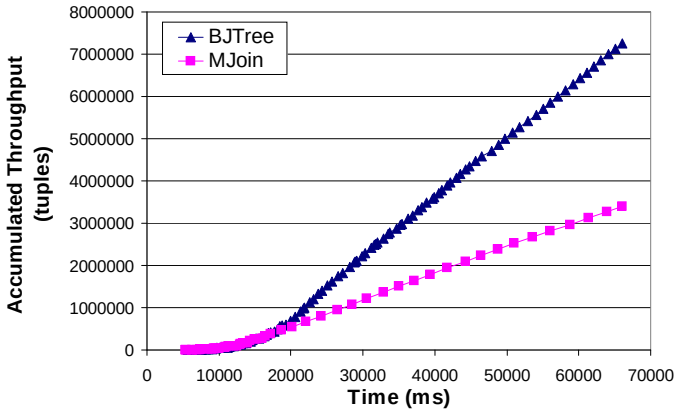


Figure 8.4: Accumulated Throughput (Set 2)

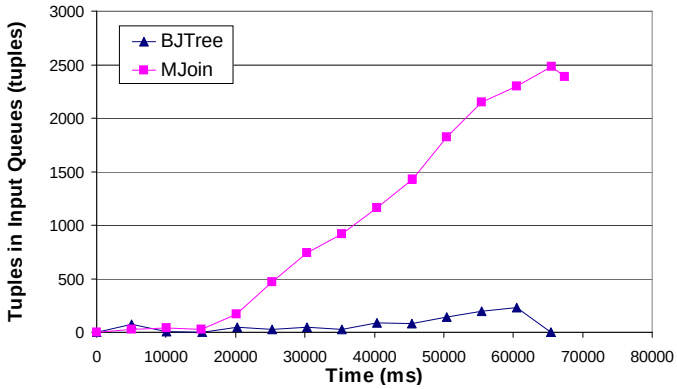


Figure 8.5: Tuples in Input Queues (Set 2)

Experiment Set 4: In this experiment, neither mjoin nor bjtree are qualified plans. Instead, an hybrid tree is generated as a qualified plan. Figures 8.8 and 8.9 compare the accumulated throughput and memory consumptions of the three plans. The bjtree has the least CPU cost and is shown to have faster throughput at the beginning. However, it quickly runs out of memory at around 80,000ms. The hybrid tree, although has slower throughput

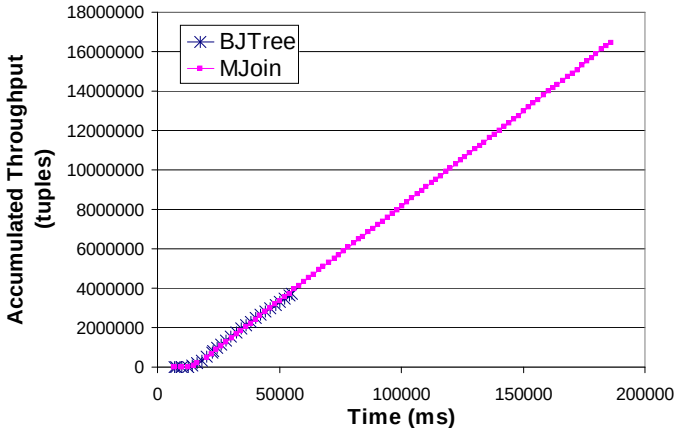


Figure 8.6: Accumulated Throughput (Set 3)

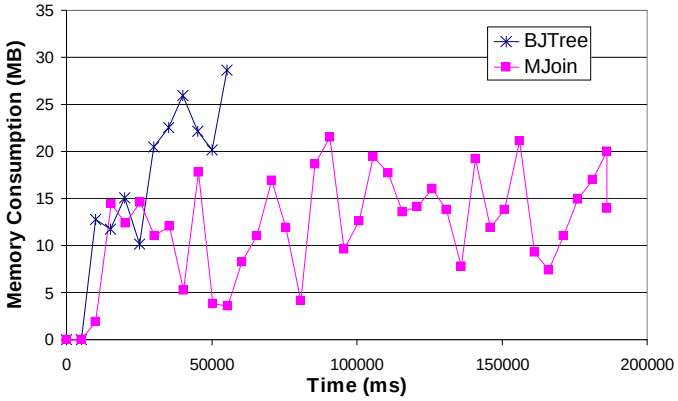


Figure 8.7: Memory Consumptions (Set 3)

than the bjtree, outputs results faster than the mjoin. And it does not runs out of memory because it requires less memory than the bjtree.

In summary, our experimental results confirm our cost analysis, and demonstrate that the proposed optimization framework is able to pick a qualified query plan under various system resource availabilities.

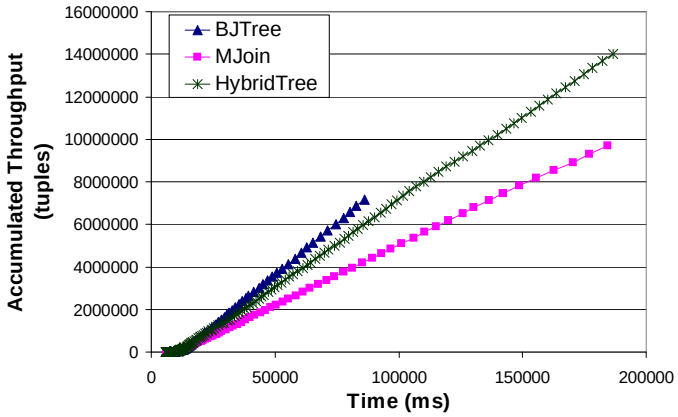


Figure 8.8: Accumulated Throughput (Set 4)

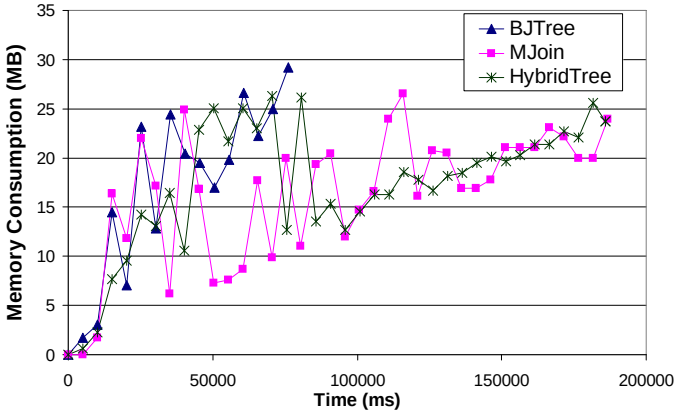


Figure 8.9: Memory Consumptions (Set 4)

8.2 Comparing Optimization Strategies

Next I compare the performances of the two alternative optimization strategies proposed in this paper, namely the mjoin-init strategy and the bjtree-init strategy. The experiments are designed to compare their effectiveness at finding qualified query plans and their average optimization time. I also compare the memory and CPU costs of the qualified plans generated by

the two optimization strategies.

The first set of experiments compares the *optimization effectiveness* of the two optimization strategies, meaning how often the algorithms can find a qualified query plan given various experiment settings. In our experiment, each setting is characterized by the number of input streams, the arrival rates of streams, the joins among streams and the join selectivities. Changing any one of the parameters would result in a different experimental setting. To quantify the concept of optimization effectiveness, I use the term *qualified percentage* defined as below: Given N different experimental settings, if an algorithm finds qualified query plans in M ($M <= N$) settings, the *qualified percentage* of the optimization algorithm is M/N.

In order to demonstrate how effective the proposed optimization methods are, I compare their performance against an exhaustive search algorithm for finding qualified query plans. The exhaustive search is implemented as a bottom-up dynamic programming (DP) algorithm, which is a direct extension to the classic dynamic programming algorithm in static query optimization [SAC$^+$79]. However, instead of searching in the left-deep join tree space, our DP algorithm searches in the entire hybrid tree space. To save processing time, the DP algorithm returns once the first qualified query plan is found. The DP exhaustive algorithm is guaranteed to find a qualified query plan if one exists. So it has the highest possible qualified percentage in any experiment.

Our experiments are set up as follows: during each experiment, the number of input streams N varies from 3 to 20. For each N, we go through the following setup process: 1) the input streams are randomly generated

within the range of [1, 100] tuples/second, 2) the joins among input streams are randomly selected, and 3) the corresponding join selectivities are randomly generated within the range of (0, 1) [1]. Such setup process is repeated 100 times for each distinctive N. Therefore, totally (20-3 + 1) * 100 = 1800 different parameter settings are applied in each experiment. This produces a sufficiently large sample set to illustrate the performance trends of the examined optimization algorithms.

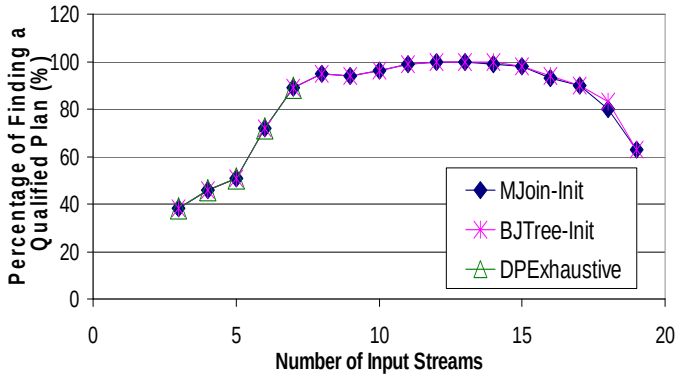


Figure 8.10: Qualified Percentage

Figure 8.10 depicts the experimental results of the *qualified percentage*. Because even the most effective dynamic programming runs in exponential time and space, the results denoted by *DPExhaustive* end as soon as the number of input streams (N) reaches to 8. This is when the DP algorithm runs out of memory and is unable to produce a result. We can see from Figure 8.10 that the two proposed optimization strategies, denoted by "MJoin-Init" and "BJTree-Init" respectively, have very similar qualified

[1]Note that the join selectivity here is defined as $(num\ of\ outputs)/(num\ of\ possible\ outputs)$. Therefore, a join with selectivity less than 1 can still produce more output tuples than the input tuples it takes in.

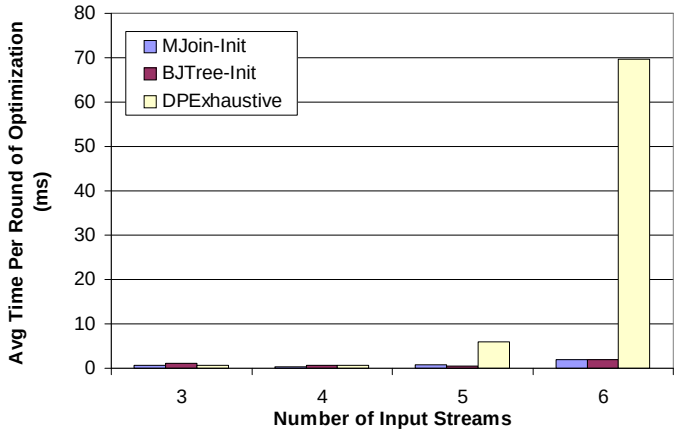


Figure 8.11: Average Optimization Time

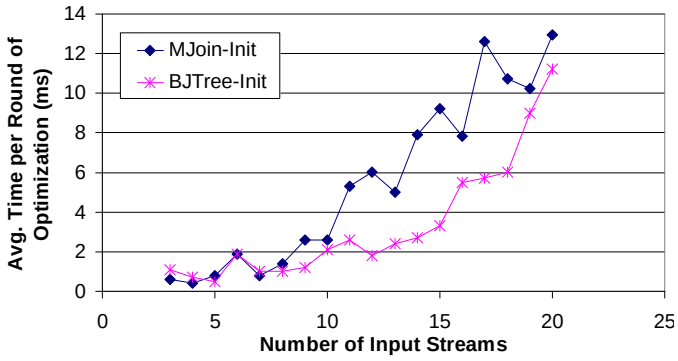


Figure 8.12: Comparing Avg. Optimization Time (II)

percentage given various N. This means that among the 100 optimizations processed for each distinctive N, the number of qualified plans found by the two strategies are very similar. Furthermore, both have almost the same qualified percentage as the "DPExhaustive". This confirms that both optimization strategies are highly effective in finding qualified query plans.

Figures 8.11 depict the experimental results of comparing runtime opti-

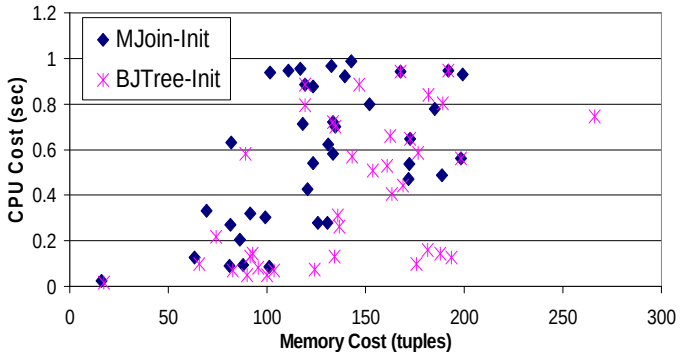


Figure 8.13: Distribution of Qualified Plans (n=3)

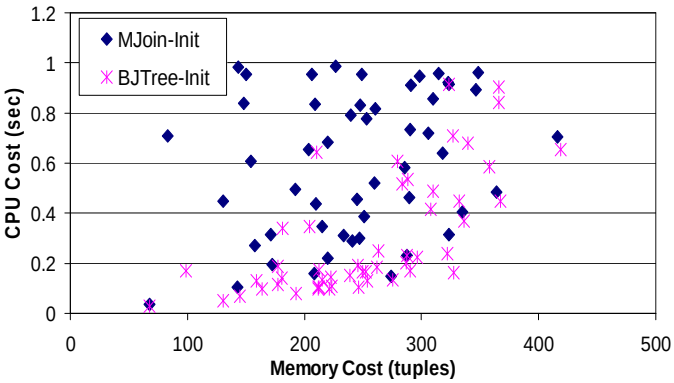


Figure 8.14: Distribution of Qualified Plans (n=5)

mization time (average over 100 runs) for different number of input streams (N). Both proposed optimization strategy takes polynomial time, while the DP exhaustive search algorithm takes exponential time. It is clear from Figure 8.11 that DPExhaustive becomes too slow as compared to the polynomial-time strategies even for very small N.

Lastly I compare the memory and cpu costs of the qualified query plans generated by the two optimization strategies. The results are shown in Fig-

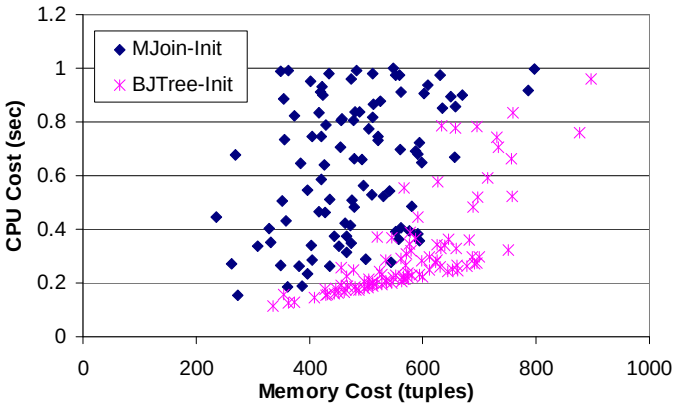


Figure 8.15: Distribution of Qualified Plans (n=10)

ures 8.13, 8.14 and 8.15. We observe the trend that the qualified plans generated by the mjoin-init strategy generally have smaller memory costs but larger cpu costs, as compared to the qualified plans generated by the bjtree-init strategy. Figures 8.13, 8.14 and 8.15 depict the distributions of memory and cpu for all the qualified query plans found for N equals to 3, 5 and 10, respectively. We can see that this trend becomes more apparent as N increases. In Figure 8.13 when N equals to 3, the qualified plans from both optimization strategies are mixed on the plot. As N increases, the qualified plans generated by the mjoin-init strategy tend to be located more at the upper-left area, while the qualified plans generated by the bjtree-init strategy tend to be located more at the lower-right area. The two sets of qualified plans are clearly separating from each other. This trend is caused in part due to the differences in the starting points of the two optimization strategies. Since an mjoin usually has smaller memory costs but larger CPU costs than a bjtree generated for the same query, optimizing from each start-

ing point has the tendency to reach qualified query plans that are closer to that starting point.

In summary, our experiments demonstrate that both proposed optimization strategies are highly effective in finding qualified query plans. They both run in polynomial time and therefore are suitable for runtime optimization for continuous queries. The qualified query plans generated by the mjoin-init strategy tend to have lower memory but higher cpu costs as compared to the qualified query plans generated by bjtree-init strategy. Therefore, an runtime optimizer may choose which optimization strategy to apply based on the current resource availabilities in the query system.

Chapter 9

Related Work

The problem of optimizing multiple joins is a core research area [MS79, IK84, KBZ86, IK91, SI93, SAC+79]. The majority of results focus on optimizing multiple joins in static query systems, in which the query plan is in the shape of a left-deep [SAC+79, IK84, KBZ86], a right-deep [SMK97, LVZ93], or a bushy [LVZ93] binary join tree.

The idea of using an mjoin operator for continuous query processing is first discussed in [VNB03, GO03, HAE03]. [BMM+04] proposes heuristics-based join ordering algorithms for mjoin that consider dependent join selectivities. However, compared to bjtree, an mjoin may need extra CPU time to recompute intermediate results. [VNB03] confirms this analysis by showing experiments that a bjtree has better performance than an mjoin when the amount of recomputations is large.

[VN02a] proposes a heuristic algorithm named FastLeaves to optimize continuous multiple joins. It is however very different from our proposed algorithms in terms of optimization goal and algorithm design. FastLeaves

optimizes multiple joins for the purpose of achieving high output rate, while our algorithms find query plans with both CPU and memory under system resource constraints by utilizing the correlations between the two resources.

[MSHR02] introduces the Eddy approach of adaptively executing a query by routing tuples among operators. Eddy's always-adapting solution makes it suitable for a highly dynamic environment. The SteMs structure [RDH03] further enhances the flexibility of an Eddy operator. Eddy together with SteMs can be considered to be a more flexible version of the mjoin approach. However, like mjoin, this approach also needs to recompute all intermediate results. [DH04] proposes solutions for dynamically optimize a query plan containing Eddy operators to decrease the negative impacts caused by tuple routing history. However, this solution does not consider both CPU and memory resource constraints when optimizing a plan.

The A-caching algorithm [BMW05] optimizes a single continuous mjoin operator by adding or removing temporary caches for selected intermediate results. It relies on value-based hashing for detecting a cache hit/miss and only considers *non-overlapping* caches, meaning two caches cannot have common joins. This limits the search space for the possible query plans. Our solution can be applied to a query plan instead of just one operator and does not have the above restrictions on search space, thus significantly broadens the range of intermediate results that can be simultaneously stored.

The problem of considering multiple resources has been widely studied in parallel query optimization and task scheduling in a distributed system

[HS91, HM94, SAL+96, PY01, GI96, EHJ+96]. The parallel query optimization in [HS91] considers both CPU costs and buffer sizes. The approach is to minimize the CPU cost of the query at compile time and delay the decision made based on buffer sizes to runtime by adding "choose" operators. In my solutions I consider both cost factors when optimizing the query. The Mariposa project [SAL+96] optimizes a parallel query plan based on a concrete user-defined cost-delay curve. In my work, I do not assume that the relationship between CPU and memory is given a priori as this concrete curve is hard to capture in reality. [HM94, GI96] propose solutions for the problem of distributing query operators to several processors while exploring the trade-off among multiple system resources, including CPU, memory, disk I/O and network communication. In these works, query plans are given as inputs. Our work instead finds the query plan that satisfies multiple resource constraints.

Part II

Dynamic Plan Migration for Continuous Queries

Chapter 10

Introduction

10.1 Motivation for Migrating Continuous Queries at Run Time

Many applications require the monitoring of data streams using standing queries, including sensor networks, stock and medical monitoring systems [CCC+02, BBD+02a, NWAea02, MSHR02, CF02, RDS+04, LZJ+05]. In those systems, data may stream in from several often distributed network locations, with unpredictable fluctuations in arrival rates and in value distributions. Queries posed over such streaming data are usually long-running. Hence an originally well tuned query plan may later become sub-optimal or even exhibit poor performance due to these changes. A stream query engine must cope with such changing characteristics of the streaming environment.

On-the-fly query re-optimization, one critical technology addressing

this problem, has attracted much recent research attention [BBD+02a, CCC+02, NWMS98, VN02a, CDN02, IHW02]. Such a solution usually takes two steps. First, the optimizer dynamically selects a new more efficient yet semantically equivalent query plan based on system statistics gathered at run time. This is referred to as the *dynamic query optimization*. Then the system needs to be migrated from the query plan that it is currently running to the new plan that the optimizer has chosen. We refer to the latter process as *dynamic plan migration*.

10.2 Limitations of Existing Migration Approaches

A migration strategy must guarantee that it will not alter the results produced by the system during as well as after the plan transition. Correctness here implies that results are neither missing nor contain erroneous or duplicate tuples. Traditionally, a dynamic plan migration strategy [CCC+02] takes the following steps: 1) pause the execution of the current query plan, 2) drain out all existing tuples in the current query plan, 3) replace the current plan with the new plan, and restart the execution. We refer to this traditional approach as the *pause-drain-resume* strategy. The purpose of the draining step is to clean up the intermediate tuples in the query plan so to prevent any missing output tuples.

The *pause-drain-resume* migration strategy may be adequate to dynamically migrate a query plan that consists of only *stateless* operators, such as select and project. A *stateless* operator does not need to maintain intermediate data nor other auxiliary state information in order to be able to generate

complete and correct results. Intermediate tuples in such a stateless query plan exist only in intermediate queues and can be cleaned completely by the drain step during the migration process.

On the contrary, a *stateful* operator, such as join or group-by, must store tuples that have been processed thus far so to be able to generate future results. For a long-running query as in the case of continuous queries, the number of tuples stored inside a stateful operator, such as a join or a group-by, can potentially be infinite. Several strategies have been proposed to limit the number of intermediate tuples kept in operator states by purging unwanted tuples, including window-based constraints [KNV03, CCC+02, NWAea02, HFAE03] and punctuation-based constraints [DMRH04a, TMSF03b]. In all the above strategies the purge of the old tuples inside the state is driven by the processing of either new tuples or new punctuations from input streams.

It is important to note that for a query plan that contains such *stateful* operators, intermediate tuples may exist in both the intermediate queues and in the operator states. As noted above, the purge of tuples in the states relies on the processing of new data. However, in the *pause-drain-resume* migration strategy described above, before embarking on the drain step, as the very first step the execution of the query plan is paused so that no new tuples beyond the intermediate tuples are being processed until the migration is over. This creates a **deadlock in the migration process**: *the migration is waiting for all old tuples in operator states to be purged from the old plan, while the old tuples in those states are waiting for new tuples to be processed in order to be purged*.

10.3 My Research on Run Time Migration

In my research, I observe that the problem of dynamic migration can be abstracted using a notion of *migration box* and then propose migration strategies based on this notion. We use the term *migration box* to refer to the plan or sub-plan selected for migration. Each *box* consists of a set of operators that together represent a connected query sub-plan. It can be as large as the complete plan or as small as one operator. Each box can have several *box root* operators, each associated with a *box output queue*, and several *box leaf* operators, each associated with a *box input queue*. *Box intermediate queues* connect operators inside a box.

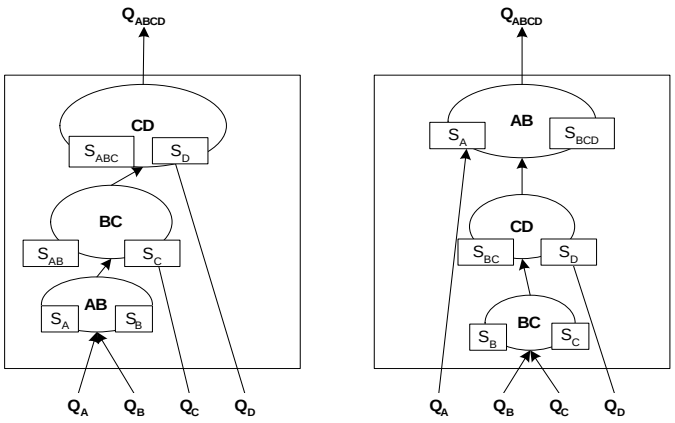


Figure 10.1: Two Exchangeable Query Boxes

Using the notion of migration box, the migration problem can then be defined as the process of transferring an old box containing the old query plan to a new box containing the new query plan. As shown in Figure 10.1, the old and the new query plans must be semantically equivalent to each other, indicating that the old and new boxes have the same sets of box input

and output queues.

In my work, I propose two plan migration strategies for continuous queries over streaming data, namely the *moving state strategy* and the *parallel track strategy*. The first strategy exploits reusability of existing stream states and the second employs parallel query execution to seamlessly migrate between continuous join plans without affecting the results of the query.

The *moving state strategy* first pauses the query plan or part of the query plan that is being optimized and drains out tuples inside the intermediate queues, similar to the above *pause-drain-resume* approach. However, to avoid loss of any useful data inside states, it then carefully identifies and moves over all relevant tuples in the states of the old query plan to their corresponding location in the new query plan. Beyond that, to assure correctness, selectively certain intermediate tuples are then recomputed. Lastly, the execution of the query plan is then resumed with the new plugged-in plan.

The second migration strategy, called the *parallel track strategy*, migrates a query plan in a more gradual fashion by continuing the delivery of output tuples even during migration. Instead of moving tuples to the new query plan and discarding the old query, it plugs in the new query plan and starts executing both query plans in parallel. I develop algorithms to eliminate potential duplicates and maintain the appropriate order of output tuples. Once the old plan is found to be "antiquated", it can simply be dis-connected and the migration stage is then over.

In this part of the dissertation, I first present the basic ideas of the two

migration strategies by focusing on joins only and by assuming a particular system execution model. I then generalize and significantly extend the existing migration strategies along several dimensions, including to cover all common types of operators (and not just joins), all main system execution models and timestamp representations common in the current stream literature. I describe how to apply these migration strategies to query plans that contain Select, Project and Join (SPJ) operators, and Group-by and Aggregate operators.

In summary, I have made the following contributions on plan migration of continuous queries at runtime:

- I first design and give cost analysis of the two migration strategies, namely the moving state strategy and the parallel track strategy, for migrating query plans that are composed of stateful operators, such as join and group-by, and query plans with a mixture of stateful operators and stateless operators, such as select and project.

- I extend the basic migration strategies to apply to query plans with a mixture of stateful operators, such as joins, and stateless operators, such as select and project. I illustrate the new migration problems of mixing the two types of operators and the design changes that need to be made to the basic migration strategies.

- I also propose migration strategies to cover the migration of group-by with aggregates. I propose new methods to migrate a continuous query plan with both joins and group-bys, the two common classes of stateful operators in continuous queries.

- I identify and categorize the various execution models adopted in existing continuous query systems [CCC+02, NWAea02, MSHR02, RDS+04]. I illustrate how the execution models can affect runtime plan migration, and then present critical changes that need to be made to our migration strategies in order to support these execution models. In particular, I analyze the problem of synchronization during migration, one of the key steps to guarantee the correctness of run time migration for a system to be able to adopt alternative execution models. I present several algorithms to achieve such synchronization efficiently.

- I have implemented both strategies within the CAPE system [RDS+04], a prototype continuous query system, and have conducted experimental evaluations of the two proposed migration strategies and compared their performances. I present the performance improvements by dynamically applying the migration strategies in the middle of a query processing in a variety of system settings.

10.4 Road Map

The rest of this part of the dissertation is organized as follows. Chapter 11 establishes the foundations, including describing the two stateful operators, join and group-by, and the concept of the system execution model. In Chapter 12, I describe the basic ideas of the two migration strategies to migrate query plans containing only joins. Chapter 13 extends the migration strategies to support the migration of Select-Project-Join (SPJ) query plans.

In Chapter 14 I discuss the migration of query plans containing both group-bys and joins. In Chapter 15 I categorize the execution models in existing stream query systems and generalize my proposed migration strategies to support different execution models. Chapter 16 is devoted to the experimental results, followed by a discussion of related work in Chapter 17.

Chapter 11

Background

11.1 Operator States and Window Constraints

As described in Section 10.1, dynamic query plan migration aims to smoothly change the shape of a query plan in the middle of its running. A valid migration must guarantee that the migration process does not alter the results of the running query. This means that the migration should not cause any missing results, duplicate results or incorrect results.

For a valid migration strategy, the key to guarantee correct results of the migrated query is to preserve and utilize *all* useful information kept in *operator states* during migration. *Operator state* is simply some data structure inside stateful operators, such as joins and group-bys, that stores tuples received so far for future processing. This data structure is necessary since a continuous query over streams must continuously produce results, requiring all operators to be operating in a non-blocking fashion. Any operator needs to output partial results based on the already received tuples. To

make traditionally blocking operators, such as joins or group-bys, become non-blocking, we can store tuples received so far in this *state* data structure. For example, for a join operator, tuples that arrive later can join with tuples in the join states and produce more join results.

An operator state stores tuples received so far for future processing. Since a continuous query can theoretically be infinite, that is, without any restriction the states could grow arbitrarily large. *Window constraints* can be used to limit the number of tuples stored in each state. A window constraint can be either *time-based* or *count-based*. A *time-based* window constraint indicates that only tuples that arrived within the last window timeframe are useful and need to be stored in states. A *count-based* window constraint indicates that only the most recent certain number of tuples need to be kept in states.

Window constraints are common in user-defined continuous queries. For example, given three input streams A(a1, a2), B(b1, b2) and C(c1, c2), a user may submit the following query with window constraints:

```
SELECT    A.a1, B.b1, SUM(C.c2)
FROM      A [range 30 min], B [range 30 min], C [range 30 min]
WHERE     A.a1 = B.b1 and B.b2 = C.c1
GROUP BY  C.c1
```

The above query is defined using the continuous query language (CQL) proposed in [ABW03]. The time range after each stream defines the time-based window constraint on that stream. The query contains two joins and one group-by with aggregate SUM. In this example, all operators are eval-

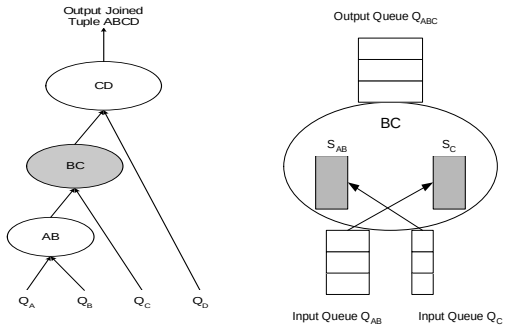
uated using the same time window of 30 minutes. One result set is output for each of the latest 30-minutes window. By using a sliding window, a result set is output whenever new tuples of the next time unit (one minute in this example) have arrived.

11.2 Stateful Join Operator

An essential part of the migration process is to properly preserve the useful information (tuples) found in states, and to discard useless tuples from states. I now show how the states are being managed in the two most common stateful operators in continuous queries, namely join and group-by, given window constraints. The state operations of a join operator are introduced below, while the stateful group-by operator is introduced in Section 11.3.

We use an example of the symmetric binary join [WA93b, HH99], commonly used for implementing joins in continuous queries [KNV03, CCC+02, NWAea02], to show how tuples are being processed. Without loss of generality, we apply time-based sliding window constraints in our description. Applying the concepts discussed in this section to count-based window constraints is straight-forward. A sample query plan for the query $A \bowtie B \bowtie C \bowtie D$ that consists of three join operators with input streams A, B, C and D is depicted in Figure 11.1(a). For instance, the join operator $B \bowtie C$ in Figure 11.1(b) has two input queues Q_{AB} and Q_C, two *states* S_{AB} and S_C, one associated with each input queue, and one output queue Q_{ABC}. Each *state* stores the tuples that fall within the current time window

from its associated input queue.



(a) (b)

Figure 11.1: Join Operators and Their States

Given the above example, for each new tuple AB from Q_{AB}, the join involves three steps: 1) purge – AB is used to purge tuples in state S_C that are outside the window frame from AB, 2) join – AB is joined with the tuples left in S_C, and 3) insert – AB is inserted into state S_{AB}. The same process applies to tuples from Q_C. Using this join algorithm, tuples from either input to the join operator can be used to probe the tuples stored in the state corresponding to the other input. Therefore the join operator is being executed in a non-blocking fashion.

11.3 Stateful Group-by Operator

Group-by operator is another commonly used operator that is also stateful. A group-by operator is usually done for the purpose of applying an aggregate function on each group of tuples. In this part of the dissertation, if not otherwise noted, the group-by operator is assumed to also contain an

aggregate function.

The group-by is usually a blocking operator in static query processing. By adding the window constraint, a group-by with aggregates generates a set of output tuples per window frame, with one tuple for each group. Thus it becomes non-blocking because it is able to continuously output results on a per window base. Figure 11.2 depicts an example of applying a group-by over a stream named C with window constraint W. Each tuple in the stream has two columns (c1, c2). The group-by column is on $C.c1$, and an aggregation function $SUM(C.c2)$ is applied to each group of tuples with a distinctive $C.c1$ value within a window frame. As we can see the operator generates a set of output tuples for the most recent W time window. Specifically, in the recent W time frame, the sum of all $C.c2$ values among tuples with C.c1=1 is 10. For tuples with C.c1=2, the sum of column value C.c2 is 22.

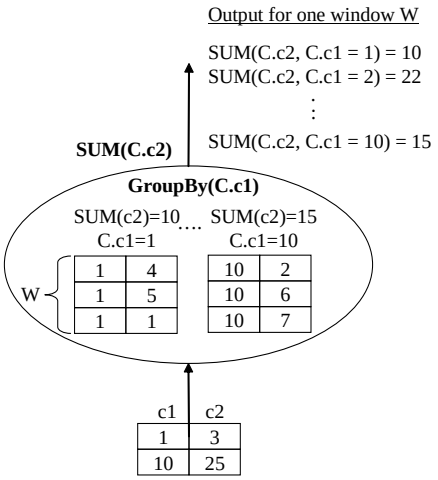


Figure 11.2: Stateful Group-by and Aggregate (SUM) with Window Constraint

For a window-based group-by operator, it is often not sufficient to just store an aggregate result for each tuple group in its state. Instead, the operator may need to keep in its state all the tuples processed in the most recent W window frame. To show why this is the case, let us again use the example depicted in Figure 11.2. Suppose after outputting results for the most recent window, two new tuples arrive in the input queue. Since the $C.c1$ column value of the first tuple (1, 3) equals to 1, it would be grouped with tuples of the same C.c1=1 value. This group is stored in the left most table inside the group-by operator. Since the group-by is sliding window-based, it only generates one tuple for each group per window. The new tuple (1, 3), along with other tuples in the same group that are less than W away from this new tuple, would be contributing to the new result for the now most recent window. Any tuple outside the window should not affect the result in any way and can be purged from the state. Suppose that the first tuple (1, 4) in the left table is more than a window away from the new tuple (1, 3), its effect on the aggregate result for this group should be cancelled. The aggregate result, SUM(C.c2), for this group of tuples in the previous window is 10. The new aggregate result for the new most recent window should now become 10 - 4 + 3 = 9. This shows that in order to get the new aggregate result, we need information from three aspects: the previous aggregate result, the new tuple and the old tuple that is outside the window frame from the new tuple. Therefore it is necessary to store tuples in the most recent window frame in the group-by operator state.

A *winid* solution has been proposed in [LMT$^+$05] recently, in which a group-by operator does not need to store multiple tuples, but instead it

stores multiple aggregate results, one for each active window. This only works for certain aggregate functions. In this part of the dissertation, I thus focus on the more general and also traditional approach of storing the tuples within the latest window frame inside the group-by operator.

Also note that if *negative tuples* [HMA+04] are used in the query processing, it is possible a group-by operator does not need to store all the tuples in the most recent windows. However, using negative tuple also has the drawback of doubling the workload. In my dissertation work, I focus on the cases of query processing that do not need negative tuples.

11.4 Tuple Arrival Order and Execution Order

Applying time-based window constraints requires that each newly arriving tuple has a *timestamp*. When tuples first arrives at the query engine, they each carries a single timestamp. We refer to these tuples as *singleton tuples*. They are usually assumed to be ordered by their timestamps [KNV03, CF02, NWAea02].

The algorithm to purge a state by a singleton tuple is straightforward. For a join operator $A \bowtie B$ with window size W, since singleton tuples from stream A are strictly ordered non-descendingly, a B tuple in state S_B is purged by an A tuple if and only if $(TS_A - TS_B) > W$.

Operators may combine two or more tuples into one complex tuple in an operator, such as by a join operator. We refer to such a tuple as a *combined tuple*. The original singleton tuples forming this combined tuple are called the *sub-tuples*. A combined tuple may need to keep timestamps from all

its sub-tuples in order to preserve all the timestamp information critical for correct purging in the purge-join-insert process described above.

However, the combined tuples output from a join operator may not be ordered by any column's timestamps. Figure 11.3 depicts such a scenario. In Figure 11.3, a binary join operator takes two inputs, named A and B respectively. The input tuples each is marked by its arrival timestamp. The output tuples combine the two timestamps of their sub-tuples. Assume the input tuples arrive at the system from remote resources in the order of $A1, A2, B1, B2, A3$. This order is referred to as *tuple arrival order*. We refer to the order that tuples are actually being processed by the join operator as *tuple execution order*. The tuple execution order is typically determined by the system scheduler. As a result, it can be different from the tuple arrival order. In Figure 11.3, we assume that the tuples execution order is $A1, B1, A2, B2, A3$, which is different from the tuples' arrival order. We can see that the output tuples generated by this execution order are not ordered by any of the sub-tuples' timestamps: timestamps from A sub-tuples are ordered as $A1, A2, A1, A2, A3, A3$, while timestamps from B sub-tuples are ordered as $B1, B1, B2, B2, B1, B2$.

As a summary, if the tuple execution order inside an operator is not the same as the tuple arrival order, it is likely that the combined tuple output from this operator is not ordered by any of its sub-tuples' timestamps. This complicates the window-based tuple purging in the subsequent operators in the query plan.

To solve this problem, in the literature it has been proposed to apply certain model that enforces the execution order of the tuples inside the query

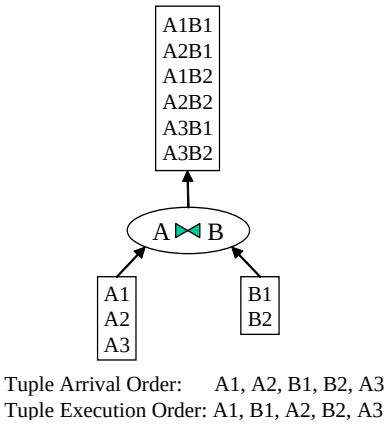


Figure 11.3: Tuple Arrival Order and Execution Order

plan. We referred to such a model that a continuous query system adopts to control tuple execution order as *system execution model*.

In Section 11.5, I introduce the most restricted execution model, named *total synchronized execution model* and the corresponding tuple orderings and timestamp representation. The discussions of other execution models adopted by current existing continuous query system are delayed to Chapter 15. As will be discussed in Chapter 15, execution models other than the total synchronized model introduce a new critical problem during the migration process. This requires a carefully designed synchronization process in order to guarantee correct plan migration.

11.5 Total Synchronized Execution Model

The *total synchronized execution model* executes tuples in a completely synchronized fashion [KNV03, VNB03, ZRH04]. This enforces two properties

on the tuple processing. First, it enforces that the tuple execution order is the same as the tuple arrivel order in the query processing system. This guarantees that a singleton tuple with a smaller timestamp is always being executed earlier than a singleton tuple with a larger timestamp. Secondly, when a new tuple arrives into the system, the tuple is being processed by all the operators in the query tree from bottom up until the final results, if there are any, are being output from the root of the query tree. After this complete execution, the next new tuple would then be processed following the same procedure. This in fact restricts the functionality of the scheduler in the system, because operators are invoked in the order from the leaf operator, where the new tuple is inserted, all the way to the root operator.

The total synchronized model is a highly restricted and thus simplified execution model. Therefore it is commonly adopted by theoretical studies on cost models in continuous query processing [KNV03, VNB03, ZRH04]. However, it does not give flexibility, such as scheduling flexibility, and thus is typically not adopted in most actual systems for processing continuous queries.

Using the total synchronized execution model has an impact on the timestamp ordering of the combined tuples output by a join operator. Figure 11.4 depicts such a scenario. Here the tuple arrival order is the same as the tuple execution order. Although the combined tuples output by the join operator are not ordered by either of the sub-tuples' timestamps, they are indeed ordered by the larger timestamp within each combined tuple. The larger timestamp in each combined tuple is underlined in Figure 11.4.

Taking advantage of this observation, I define the timestamp represen-

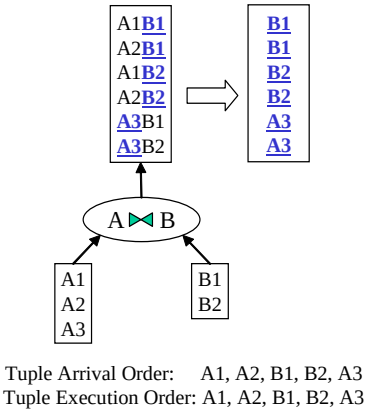


Figure 11.4: Timestamps Order When Applying Total Synchronized Execution Model

tation of a combined tuple to be the maximum timestamp among all its sub-tuples. As a result, the combined tuples would now be ordered by their newly defined single timestamps, as depicted in Figure 11.4. Because the tuples in any input queue feeding an operator in the query plan are now ordered by their singleton timestamps, the straightforward state purging algorithm by a singleton tuple, as described in Section 11.4, can now be applied to any operator in the query plan. In Chapter 15, I will define other common execution models and their associated timestamp representations in existing continuous query system, and will extend our migration solutions as needed to address newly arising challenges when adopting these execution models.

The migration strategies introduced in the following three chapters, including Chapters 12, 13 and 14, assume that the *total synchronized execution model* is used by the query execution system. This is an important assump-

tion and is rather restricted. In Chapter 15, I categorize execution models in existing continuous query systems and then generalize our proposed migration strategies to support all the execution models.

Chapter 12

Migrating Join Query Plans

In this section, I describe the two migration strategies to migrate continuous query plans with one or more joins. To focus on the key ideas, I first make the simplifying assumption that the system adopts the total synchronized execution model (see its definition in Section 11.5). This indicates that (1) a joined tuple only keeps its maximal timestamp among all its subtuples and (2) tuples in any queue of the query plan are ordered by their timestamps. Later in Chapter 15 this assumption will be relaxed.

Several terms need to be made clear before introducing our proposed migration strategies. I denote the time period of each online plan migration process as *migration stage*, with the migration start time as T_{M_start} and the migration end time as T_{M_end}. During the migration stage, I refer to the states in the old box as *old states*, and states in the new box as *new states*. All tuples existing in the old box at T_{M_start} are called *old tuples*, and any tuple entering old and new boxes after that time point are called *new tuples*. That is, it is not the system time that determines a tuple's old or new status, but

rather the location of the tuple at T_{M_start}. If a tuple enters the old box any time during the migration stage, although it has arrived in the system or has been generated before T_{M_start}, it is still treated as a new tuple by the old box. A combined tuple that has any of its sub-tuples marked as *old* is referred to as an *old* tuple, since it still has some contents that had existed in the old box at T_{M_start}. A combined tuple is considered a *new* tuple only if all its sub-tuples are *new*.

12.1 Moving State Strategy

The basic idea of the *moving state strategy* is to safely move old tuples in old join states directly into the join states in the new box without losing any useful data. In this section, I detail the necessary steps of the moving state strategy, including *state matching*, *state moving* and *state recomputing*.

In the moving state migration strategy, I first pause the execution of the operators inside the old box. If the old box contains a sub-plan of a complete query plan, then the rest of the operators in the query plan that are outside the old box can still be processed as usual. After pausing the execution of the old box and before any of the state operation can be carried on, we should first clean the tuples accumulated in intermediate queues inside the old box. This is the same as the "drain" step in the pause-drain-resume strategy discussed in Chapter 10.

12.1.1 State Matching

State matching determines the pairs of join states, one in the old and one in the new box, between which tuples can be safely moved. When the query plan only contains join operators, we can match states by their *state schema*. I define a state's schema to be the same as the schema of all its tuples. A tuple's schema is defined as the combination of all its column IDs. Each column ID is composed of the stream ID it belongs to, a dot and the name of the column itself.

If two states have the same state schema, we say that those two states are *matching states*. In Figure 12.1, states (S_A, S_B, S_C, S_D) exist in both boxes and are matching states. States (S_{BC}, S_{BCD}) appear in the new box only, and states (S_{AB}, S_{ABC}) appear in the old box only. These are thus unmatched states.

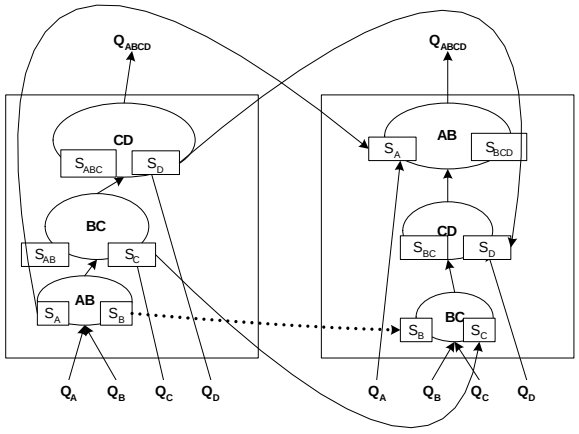


Figure 12.1: Moving State Strategy

This simple schema-based state matching assumes that when two tuples are joined, all columns are kept in the joined tuple. This is an reason-

able assumption when the query *only* contains multiple joins. For queries that contain other types of operators, such as select and project operators, the schema-based state matching is no longer sufficient. In Chapter 13, I illustrate the problem of state matching in a query plan that contains other types of operators and then describe the general techniques of state matching for such query plans.

12.1.2 State Moving

After the *state matching*, the *state moving* step then moves tuples between all pairs of matching states. Conceptually, for all matching states, I directly move the tuples from the old state to its matching new state. This method, although correct, is a waste of both time and storage. Thus in our CAPE system [RDS+04], I instead use an improved method of state sharing by utilizing the queue sharing technique.

In our system, a queue inside a query plan can have multiple operators as its producers that append new tuples to the end of the queue, and multiple operators as consumers that fetch tuples from the top of the queue. I refer to such a queue as *shared queue*. A shared queue stores one cursor for each consumer that points to the position of the tuple that this consumer would fetch next. Figure 12.2 shows the scenario where one input queue is shared by two operators. Each has a different cursor pointing to its next tuple in the shared queue.

For such shared queues, state moving can be achieved by simply creating a new cursor for each matching new state that points to the first tuple in its matching old state. This indicates that all tuples in the old state are now

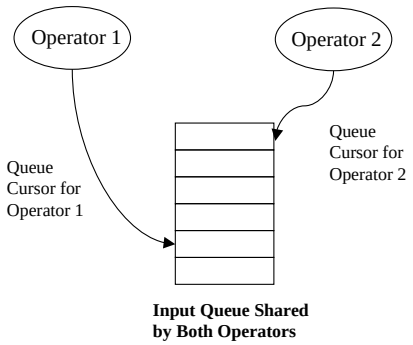


Figure 12.2: One Input Queue Shared by Two Operators

shared by both matching states. The cursors for the old matching states are then dereferenced to complete this state moving process.

12.1.3 State Recomputing

Two questions remain regarding the unmatched states in both old and new boxes: 1) Can we leave the unmatched states in the new box empty? 2) Can we throw away the old tuples inside the unmatched states in the old box?

To answer the first question, we need to determine whether or not the complete set of results can be generated if the unmatched states in the new box are left empty. We again use the migration example shown in Figure 12.1, with the old box on the left and the new box on the right. Each tuple in the output queue Q_{ABCD} can be treated as a combination of four *sub-tuples* A, B, C and D, originally from Q_A, Q_B, Q_C, and Q_D respectively. We divide all the possible outcomes of the tuples in queue Q_{ABCD} based on the *old/new* status of their sub-tuples. Figure 12.3 lists all 16 possible cases with their case #.

Case #	*A*	*B*	*C*	*D*
1	Old	Old	Old	Old
2	Old	Old	Old	New
3	Old	Old	New	Old
4	Old	New	Old	Old
5	New	Old	Old	Old
6	Old	Old	New	New
7	Old	New	Old	New
8	New	New	Old	Old
9	New	Old	Old	New
10	Old	New	New	Old
11	New	Old	New	Old
12	Old	New	New	New
13	New	Old	New	New
14	New	New	Old	New
15	New	New	New	Old
16	New	New	New	New

Figure 12.3: Possible Old/New Combinations for Tuples in Output Queue ABCD

I now show that by leaving the unmatched states in the new box empty, tuples in some of the 16 cases may be lost. Figure 12.4 depicts the status of the new box right after the state matching and moving steps. I show each tuple inside the states and input queues by its sub-tuples' *old/new* status. All tuples in the input queues are *new*. And all tuples in the matching states are *old* because they are copied over directly from the old box. The two unmatched states S_{BC} and S_{BCD}, both empty, are shaded grey.

Assume that now we discard the old box and start executing the new query plan with the unmatched states being empty. In the join operator $B \bowtie C$ in Figure 12.4, only *new* B tuples can be joined with *old* or *new* C tuples in S_C.[1] Also, only *new* C tuples can be joined with *old* or *new* B tuples in S_B. Hence only the combined BC tuple with its two sub-tuples' old/new status as (new, old), (old, new) or (new, new) can be generated by the join

[1] In Figure 12.4 S_C only contains *old* tuples. However, each *new* C tuple inserted into S_C may have been joined with B tuples, and after a while the state S_C may contain both *old* and *new* tuples.

operator $B \bowtie C$ and later be inserted into state S_{BC}. The combination (old, old) would never be generated and inserted into S_{BC}. This means that among the 16 cases in Figure 12.3, cases #1, #2, #5 and #9 cannot be generated by the query plan after migration, because those cases all require that both sub-tuples B and C are *old*. The same kind of problem occurs when leaving the other unmatched state S_{BCD} empty.

By leaving unmatched states in the new box empty, we lose the *all-old* combinations of sub-tuples in these states. This leads to the loss of some result tuples as shown in the example above. So before restarting the execution of the query plan, some computations need to be undertaken first for the unmatched states in the new box in order to gain back those *all-old* combinations. I refer to this step as *state recomputing*. I have designed a recursive algorithm shown in Algorithm 6, recompute_unmatched_states(), to compute the unmatched states in the new box. It is described for binary join operators to keep it simple, but could easily be modified to suit multiple-input join operators as well.

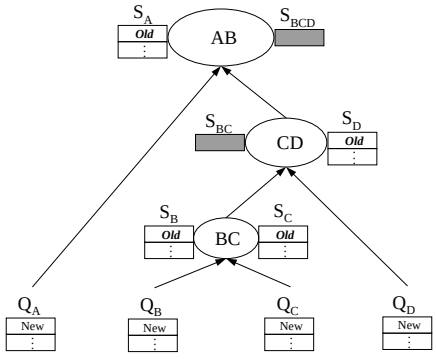


Figure 12.4: Empty Unmatched States in the New Box

Algorithm 6 Recompute_Unmatched_States

During *state matching* step, mark a state as "matched" if it has a matching state.
To start, set *current_op* $= new\ box\ root\ operator$
Recompute_Unmatched_States(*current_op*)
{
while *current_op* has more state **do**
  get the next state S_i of *current_op*;
  get the child operator *child_op* that has its output queue associated with S_i;
  if *child_op* is not new box operator **then**
    continue;
  end if
  if S_i is unmatched **then**
    get *child_op_l_state*;
    get *child_op_r_state*;
    if either state of *child_op* is unmatched **then**
      Recompute_Unmatched_States(*child_op*);
    end if
    $S_i = window\_join$(*child_op_l_state*, *child_op_r_state*);
    mark S_i as "matched";
  end if
  Recompute_Unmatched_States(*child_op*);
end while
}

12.1.4 Safe State Discarding

Now we need to address the question if it is safe to discard the old tuples inside those unmatched states in the old box. As for the example in Figure 12.1, we have to determine if we can discard the old tuples in states S_{AB} and S_{ABC} inside the old box on the left. To answer this question we need to know if any of those *old* tuples in the unmatched old states may have the potential to join with any *new* tuples.

Taking the unmatched old state S_{AB} in the old box in Figure 12.1 as an example, clearly it can be discarded if the following condition holds: All sub-tuples A and B of the AB tuples in S_{AB} also exist in states S_A and S_B respectively. This is because the states S_A and S_B are already shared by the new states in the new box. This way no data would be lost by discarding the unmatched old state S_{AB}. However, we can show that the above condition cannot be guaranteed. For example, inside the join operator $A \bowtie B$ in Figure 12.1, some tuples A and B in S_A or S_B may have already been purged by newer tuples from the input queue Q_B and Q_A. Before these tuples are being purged from S_A and S_B, they may have already joined with other B and A tuples and the joined AB tuples may have already been inserted into S_{AB}. Hence *not* all sub-tuples A and B in S_{AB} are necessarily present in S_A and S_B. After the state matching, moving and recomputing state, if we discard the unmatched old state S_{AB}, some tuples in state S_{AB} that may still be able to join with a new tuple C may then be lost. Then the results of the query plan may be incomplete.

To decide if an unmatched old state can be safely discarded, I first define

the *old state closure property*. If this property is satisfied, an old unmatched state can be safely discarded during the migration process without losing any useful information.

Old State Closure Property: *For a tuple in an unmatched old state, if one of its sub-tuples does not exist in any of the matched old states, then it is invalid to join this tuple with any future incoming tuples due to window constraints.*

It is apparent that if such a property is satisfied, discarding the old states will not lose any useful tuples. This is because all tuples in the unmatched states contain data that either already exist in the old matched states or is useless for future processing.

I now show that by using the totally synchronized execution model, this property is guaranteed to hold. Thus the old unmatched states can always be safely discarded without losing any useful data. I use the example in Figure 12.1 to illustrate this point. I use T_{purged_max} to denote the largest timestamp among the sub-tuples A and B in the state S_{AB} that do not exist in S_A and S_B, respectively. These tuples must have been purged by tuples from either Q_A or Q_B with timestamps larger than $T_{purged\_max} + W$. So if we know that all C tuples in Q_C at that point have a timestamp larger or equal to $T_{purged\_max} + W$, then they are not able to join with any tuples in S_{AB} that contain sub-tuples A and B that do not exist in S_A and S_B. This indeed can be guaranteed by the totally synchronized model. When using this execution model, the tuple arrival order is the same as the tuple execution order. If tuples with a timestamp larger than $T_{purged\_max}+W$ have been processed from Q_A or Q_B, then all tuples in Q_C must have timestamps

larger than $T_{purged\_max} + W$.

Therefore when using the totally synchronized execution model, it is safe to discard the unmatched states in the old box, given that tuples accumulated in intermediate queues in the old box are being cleared at the beginning of the migration process.

However, I will show in Chapter 15 that other types of execution models exist in the literature, for which the state closure property does not hold. For these other execution models, an extra synchronization process is needed to make sure that all useful information in the unmatched states is also contained in the matched old states before the unmatched states can be discarded.

12.1.5 Overall Moving State Algorithm

Putting all the pieces together, I now show the complete algorithm for our moving state strategy in Algorithm 7.

Algorithm 7 Moving State Migration

$clean\_accumulated\_tuples()$;
connect input and output queues of old and new boxes;
$match\_states(old\_box, new\_box)$;
$move\_states(old\_box, new\_box)$;
$recompute\_unmatched\_states(root\_op\_of\_new\_box)$;
disconnect old box from current query plan;
start executing query plan with new box;

Once the moving state migration starts, after clean_accumulated_tuples(), no new results are produced until the steps of matching, moving and recomputing states are finished. The length of this output silence is closely

related to the amount of tuples that need to be moved or recomputed during the migration stage. This duration of output silence may be less desirable for applications that are in favor of a more steady output rate. To solve this problem, I design the second migration strategy, the parallel track strategy, to continuously deliver outputs even during the migration stage.

12.2 Parallel Track Strategy

The basic idea for the *parallel track migration strategy* is that at the migration start time, the input queues and output queue are connected and shared between the old box and the new box, using the *queue sharing* technique depicted in Section 12.1.2. Both boxes are then being executed in parallel, as shown in Figure 12.5, while waiting for all old tuples in the old box to be gradually purged. During this process, new outputs are continually being produced as well by the query plan.

When the old box contains only *new* tuples, it is safe to discard the old box. This is because all old tuples have finished their duty in terms of contributing to the generation of output results from the old box. Since we have been executing the new box in parallel with the old box when the migration first starts, all the new tuples now in the old box exist in the new box as well. So if the old box is discarded at this time, no useful data will be lost.

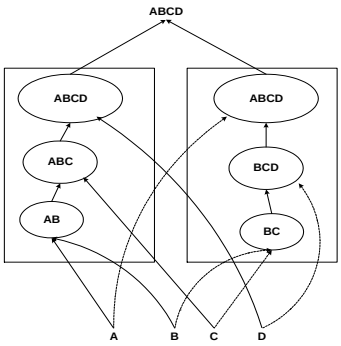


Figure 12.5: Parallel Track Strategy

12.2.1 Correctness of the Results

Correctness of the results involves two aspects: (1) the outputs are complete and (2) the outputs do not contain duplicates. I use the example in Figure 10.1 to show that by going through the parallel track migration to transfer the query plan from the left to the right, all 16 possible sub-tuple combinations of any output tuple ABCD, as listed in Figure 12.3, can still be obtained. In our parallel track strategy, both old and new boxes are running in parallel until all the tuples with *old* status are purged from the old box. By this time, the output tuples that contain any old sub-tuple, as in the cases #1-#15, have already been generated by the old box, either before (case #1) or during the migration stage (cases #2-#15). Since the new box starts its execution right after T_{M_start}, its states are initially all empty, and all the *new* tuples fed into the old box are also being processed by the new box. All output tuples from the new box will have all their four sub-tuples marked as *new*, reflecting case #16 in Figure 12.3. Thus all 16 cases are covered by either the old box or the new box.

12.2.2 Duplicate Elimination

We must also ensure that no duplicate tuples are being generated. If we use the parallel track strategy described above, although the old box will cover all 15 cases consisting of at least one *old* sub-tuple, it may also generate the all-new sub-tuple combination belonging to case #16 in Figure 12.3, duplicate to the output results from the new box.

To solve this duplication problem, a naive approach would be to discard from the old box *any* tuples with all-new sub-tuples. However, this method is too aggressive and will lose some must-have tuples. For example in the join operator $B \bowtie C$, we cannot discard any combined tuple AB from input queue Q_{AB} with both sub-tuples A and B marked as *new*, because this AB tuple may still be able to join with an *old* C tuple in state S_C, and generate output tuples that belong to either case #8 or case #14 in Figure 12.3. Even if the AB tuple ends up joining with a *new* C tuple, the joined tuple ABC, with all its sub-tuples marked as *new*, may still join with an *old* D tuple in state S_D. So the final joined tuple ABCD belongs to case #15, which can only be generated by the old box.

Thus the root join operator of the old box is the only safe place to eliminate duplicates. This is done by preventing a *new* tuple from joining with another *new* tuple. Hence if two tuples that are about to be joined are both *new*, we simply skip the join step in the regular purge-join-insert symmetric join algorithm. The purge and insert steps are however still undertaken as usual.

12.2.3 Timestamp Order Preservation

As described in Section 11.4, correct timestamp order must be preserved to ensure that the correct results are being generated. During the parallel migration stage, both the old and the new box share the same output queue into which both will insert output tuples. Keeping the timestamp order of the tuples in the output queue requires that both the old and the new box coordinate with each other to output tuples in the proper order.

Two characteristics of our parallel migration strategy assist in developing a valid method for preserving timestamp order. First, since each box is executed as a valid sub-query plan, the timestamp order among the output tuples from each box is preserved. Second, any output tuple from the old box will be guaranteed to have at least one sub-tuple being *old* (arrived earlier), and all output tuples from the new box will have all sub-tuples as *new* (arrived later). This means that any tuple generated by the new box will have a larger timestamp than any tuple generated by the old box.

Taking advantage of those two characteristics, I develop an easy yet effective method to preserve the timestamp order in the parallel track strategy. During the migration stage while both boxes are executing, we only output tuples generated by the old box into the shared output queue. Any output tuples generated by the new box are instead held in a temporary buffer. When the old box is removed, all output tuples held in the temporary buffer are then inserted into the output queue all at once.

12.2.4 Overall Parallel Track Algorithm

As described above, although join operators in both boxes are executed in parallel during the migration stage, besides the regular join operation, they may have other tasks to finish: The old box root operator needs to avoid joining two *new* tuples to prevent duplicate results, and the new box root operator needs to hold any results during the migration stage in a temporary buffer to preserve the timestamp order. We use the $W\_Join()$ method for the regular purge-join-insert symmetric window join algorithm described in Chapter 11. The methods used by the operators in the old box and the new box are referred to as $W\_Old\_Join()$ and $W\_New\_Join()$ respectively.

Algorithm 8 Parallel Track Strategy

```
Pause execution of old box at T_M_start;
Connect input and output queues of old and new boxes;
Start a separate thread to run Monitor_Old_Box();
while No signal from thread Monitor_Old_Box() do
   Old operators run W_Join_Old();
   New operators run W_Join_New();
end while
Disconnect old box from current query plan;
Operators in new box resume running W_Join();
```

To determine when to finish the migration, each operator has an IF_FINISHED flag initialized to be false. During the migration stage, each operator in the old box checks periodically to see if all *old* tuples have been purged from its states. Once this is the case the operator sets its IF_FINISHED flag to true. The system also runs a light-weighted monitor method called $Monitor\_Old\_Box()$ in a separate thread to check at intervals all the IF_FINISHED

flags of the operators in the old box. If within a scan all the flags are detected as true, the monitor method sends a signal to the main thread to tell it to finish the migration by disconnecting the old box from the current query plan. The pseudo-code for the overall parallel track migration strategy is shown in Algorithm 8.

12.3 Cost Analysis

In this section, I describe cost models for estimating the migration length and the system processing time required by each migration strategy.

12.3.1 Analysis of Moving State Strategy

To estimate how long it takes to finish a moving state migration, we need to add up the time spent on each migration step, including clean accumulated tuples, state matching and moving, and state recomputing. The cost model utilizes the binary nested-loop join algorithm with time-based window constraint. For simplicity, all join operators in the query plan are assumed to have the same window size. The cost models can however easily be extended to cover other join algorithms and different window sizes. I also assume that the system has enough computing power and memory resources to keep up with the query processing without much delay given the incoming data load.

Given the sufficient-system-resources assumption, new tuples are generally being processed immediately without being accumulated in the input queues. So the time spent on the clean_accumulated_tuples() method

Table 12.1: Terms Used in Cost Model

Term	Meaning
N	Number of operators in the old box
M	Number of operators in the new box
T_m	Time spent for each string comparison
T_c	Time spent to create a new cursor
λ_A	Average tuple input rate from Q_A
λ_B	Average tuple input rate from Q_B
σ_{AB}	Reduction factor of join operator $A \bowtie B$
W	Global time window constraint
T_j	Time spent to join a pair of tuples
T_s	Time spent to insert one tuple into a state
$\|S_A\|$	Number of tuples in state S_A
$\|S_B\|$	Number of tuples in state S_B

is likely to be small compared to other migration steps and is thus not counted in the model. The time spent on state matching and moving is related to the total number of states in both boxes. State matching is basically a string matching between two lists of state IDs. Moving a state is creating a new cursor to a state so to enable its sharing between two matching states. Thus its costs are minimal.

A list of terms and their meanings used in our model are listed in Table 12.1. The time spent on state matching T_{match} and state moving T_{move} can be calculated as below. Here I use the minimum of N and M to estimate the upper bound on number of matching state pairs.

$T_{match} = 4NMT_m$ and $T_{move} = 2min(N, M)T_c$

In order to estimate the time spent on the state recomputing step, I de-

velop a general model to estimate the time to recompute a single state. This model can then be applied to each state that needs to be recomputed to get the total recomputation time. Assume we have a join operator $A \bowtie B$ with two input queues Q_A and Q_B, two states S_A and S_B, and one output queue Q_{AB}. Without loss of generality, the tuple A and B each can be either a singleton or a combined tuple. Suppose that the state S_{AB} needs to be recomputed. This is done by joining tuples from S_A and S_B using the purge-join-insert symmetric join algorithm but skipping the purge step. The time spent on this recomputing process can be formulated as: $T_{S_{AB}} = T_j|S_A||S_B| + T_s|S_A||S_B|\sigma_{AB}$.

Given the time window W and input rates from inputs A and B, the state sizes of S_A and S_B, represented as $|S_A|$ and $|S_B|$, can be estimated as: $|S_A| = \lambda_A W$, and $|S_B| = \lambda_B W$.

Putting the above formulae together, we get the time for recomputing S_{AB} from S_A and S_B as:

$$\begin{aligned} T_{S_{AB}} &= T_j\lambda_A\lambda_B W^2 + T_s\lambda_A\lambda_B W^2\sigma_{AB} \\ &= \lambda_A\lambda_B W^2(T_j + T_s\sigma_{AB}) \end{aligned} \tag{12.1}$$

If another unmatched state above S_{AB} needs to be recomputed, according to Equation 12.1, the output rate λ_{AB} is then required. This rate can be estimated using Equation 12.2.

$$\lambda_{AB} = \lambda_A|S_B|\sigma_{AB} + \lambda_B|S_A|\sigma_{AB} = 2\lambda_A\lambda_B W\sigma_{AB} \tag{12.2}$$

If we use T_S to denote the total time spent on recomputing all unmatched states in the new box, the total migration length of the moving state strategy T_{MS} can be estimated using the following model:

$$T_{MS} = T_{match} + T_{move} + T_S \tag{12.3}$$

12.3.2 Analysis of Parallel Track Strategy

We denote T_{PT} as the length of the migration stage for the parallel migration strategy. For this strategy, all old tuples (tuples with at least one old sub-tuple) need to be purged from the old box in order to finish the migration stage. Suppose that h ($h >= 1$) is the height of the query tree inside the old box. We analyze the time spent on the parallel track migration stage in two cases:

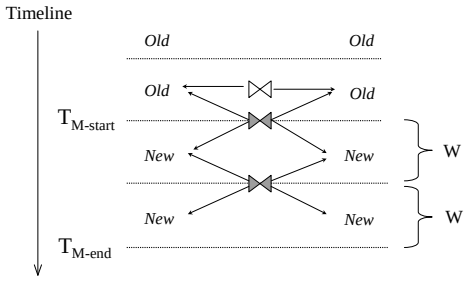


Figure 12.6: 2W to purge all old tuples

1) h = 1. In this case the query tree has only one level of join operators. For a join operator inside the old box to purge all old tuples from one of its two states, the join operator must process new tuples from the input that arrive in the next W time units. Given that the system has enough

computing power, $T_{PT} = W$.

2) $h > 1$. This means that in the old box there is at least one join operator which is above another join operator. Figure 12.6 depicts the old and new tuples along a timeline. The migration start and end time is marked to the right of the timeline. From the figure we can see that when the migration begins, W time window's new tuples from the box input queues are needed to purge old tuples inside the states of box leaf operators. However, as these new tuples are used to purge old tuples, they may also join with some of the old tuples. These joined results in turn are being inserted into the state of the join operators above the box leaf operators. Because the joined tuples contain an *old* sub-tuple, they are treated as *old* tuples and need to be purged as well. In order to do so, the old box needs to process another W time window's new tuples to completely purge these *old* tuples from the old box. So in this case, $T_{PT} = 2W$.

Other even older tuples may exist in the old box when the migration first starts, represented by the first line of "old" in Figure 12.6. These tuples will be purged by the first W new tuples after migration starts. They will thus not be able to join with any of the new tuples.

As a summary, given sufficient system processing power, T_{PM} has a linear relationship with the global window size W. It can be formulated as:

$$T_{PT} = \begin{cases} W & if \quad h = 1 \\ 2W & if \quad h > 1 \end{cases} \tag{12.4}$$

Equation 12.4 shows that in order to complete a parallel track migration,

both old and new boxes need to process at most 2W worth of new tuples. However, this is valid only when the system has enough processing power so that a tuple arrives in the system can be processed immediately. If the system processing power is not sufficient, the actual migration length may be longer than 2W.

I now give the cost model to estimate the cost of processing during the parallel track migration. As in the cost analysis for the moving state strategy, I first develop general cost formulae to estimate the processing cost spent on any one join operator (let us denote it as $A \bowtie B$) in the old box and any join operator (let us denote it as $B \bowtie C$) in the new box. I then apply the general formulae to all operators in the query plan. The total cost is the sum of the cost of each operator.

I first compute T_{AB}, the total cost of processing tuples in 2W timeframe in operator $A \bowtie B$ inside the *old box*. It is easy to see that for each new tuple A, the average number of tuples B that will be purged from state B is $\lambda_b \frac{1}{\lambda_a}$, and vice versa. The same method in Equation 12.2 can be applied to compute the tuple output rate of operator $A \bowtie B$.

$$\begin{aligned} T_{AB} &= Cost\ of\ Purge + Cost\ of\ Insert + Cost\ of\ Join \\ &= 2W[T_s(\frac{\lambda_a}{\lambda_b}\lambda_b + \frac{\lambda_b}{\lambda_a}\lambda_a + \lambda_a + \lambda_b) + T_j(\lambda_a|S_B| + \lambda_b|S_A|)] \qquad (12.5) \\ &= 2W[2T_j\lambda_a\lambda_b W + 2T_s(\lambda_a + \lambda_b)] \end{aligned}$$

One major difference between operators inside the old and the new boxes is that the states of operators inside the new box all start empty. The

sizes of the states keep on increasing with no tuples being purged until the W*th* time unit, after which tuples begin to be purged and the state size on average is limited by the window size W. This leads to different methods of computing processing cost and tuple output rate for a join operator inside the *new box*. These are described in Equations 12.6 and 12.7, respectively.

$$\begin{aligned} T_{BC} &= Cost\ for\ the\ first\ W + Cost\ for\ the\ second\ W \\ &= W[T_s(\lambda_b + \lambda_c) + T_j(\lambda_a \int_0^W \lambda_b t dt + \lambda_b \int_0^W \lambda_a t dt)] \\ &+ W[2T_j\lambda_a\lambda_b W + 2T_s(\lambda_a + \lambda_b)] \end{aligned} \tag{12.6}$$

$$\lambda_{BC} = \begin{cases} \int_0^t 2\lambda_b\lambda_c\sigma_{bc} t dt & if \quad t \leq W \\ 2\lambda_b\lambda_c\sigma_{bc} W & if \quad t > W \end{cases} \tag{12.7}$$

12.4 Comparing the Cost of Migration Strategies

From the cost analysis from the above, we can see that the cost of moving state strategy is determined mainly by the cost of recomputing unmatched state, which is then determined by parameters such as window size, selectivities and tuple arrival rates. Intuitively, the cost of recomputing states does not exceed the cost of processing all the tuples in the states inside the existing query plan. Since each state in the query plan is bounded by one window constraint. So we can estimate that the cost of moving state strategy is roughly the cost of processing one window worth's of new tuples.

However, the parallel track strategy *has* to process *two* window worth's

of new tuples in order to complete the migration process. Although during this process, new outputs can be generated, which cannot be achieved by the moving state strategy, the optimizer usually expects the migration process to be finished as soon as possible so that the advantage of the new query plan can be fully realized. Furthermore, the 2W time frame for the parallel track strategy does not mean that it will take 2W time for the migration to finish. This is only true when the system has enough resources to keep up with the newly arrived tuples without delay. If the system does not have enough resources, the migration process will take longer than 2W. However, the very point of activating runtime optimization is to recover a query plan that is not good enough to keep up with the current workload. So we should expect that when runtime migration is in force, the system resources are almost certain to be not enough, and therefore the parallel track strategy will most likely takes more than 2W time to finish.

So in most cases, the moving state migration strategy is the more efficient one between these two strategies in terms of system processing cost. For applications that prefer low execution cost to smoother output rates, the moving state strategy should be the favored strategy between these two migration strategies.

However, there are still situations when the parallel track strategy would be favored over the moving state strategy. The parallel track strategy was initially designed to solve the problem of output silence that might be experienced by using the moving state strategy. Because the moving state strategy requires to pause the execution of the sub-plan contained in the migration box until the migration is done, it may experience a period of

output silence. For example, if the migration box covers the whole plan, or if it covers the sub-plan that without executing it, the query would not be able to output any results, no output will be generated until the moving state migration is done. Applying the parallel track strategy does not have such a problem because new output will continuous generated even during the migration process. If the smooth output is one of the quality of service that the application wants to have, it may be willing to pay the price of higher migration cost in case of plan migration.

Chapter 13

Migrating Queries with SPJ Operators

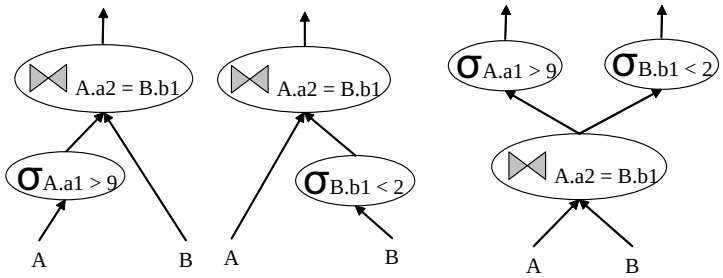
In this chapter, I extend the migration strategies proposed in Chapter 12 to support the migration of query plans with Select, Project and Join (SPJ) operators. Both select and project operators are stateless operators, which means they do not need to store any information in order to produce results continuously. Migrating a query plan that contains solely stateless operators, such as select and project, only involves draining tuples in intermediate queues. So the *pause-drain-resume* strategy discussed in Chapter 10 would be sufficient because no state is involved in the migration process.

However, mixing select and project operators with the stateful join operators can create new optimization opportunities and along with them new migration problems. The optimization is no longer restricted to the cases of switching join orderings, as discussed in Chapter 12. Below I first

describe examples of possible optimization scenarios for SPJ queries. I then describe the new migration problems followed by our proposed solutions.

13.1 Queries with Select and Join

While select is usually applied before join to filter out useless tuples early on, there are scenarios where their orders must be switched. This optimization is particularly useful for multiple join processing, when sharing of common operators can fluctuate from being beneficial to being non-beneficial depending on the data characteristics.



(a) Separate Two Plans (b) Shared Plan

Figure 13.1: Opportunities in Switching Select and Join in Continuous Query Processing.

Figure 13.1 shows an example of such a scenario. Suppose the query engine processes two continuous queries (depicted in Figure 13.1(a)) that have a common join and two distinctive selects. If the two queries are processed separately, each would have the select pushed below the join to reduce intermediate results. On the other hand, it is a common practice for an optimizer to generate a query plan that shares the common join operator as

shown in Figure 13.1(b). For such a shared plan, the two different select operators have to be applied after the join operator. For the two query plans depicted in Figure 13.1, each can be the better one than the other one under certain stream characteristics. For example, when the two select operators have very low selectivities, that is, the two selects can filter out most of the input workload, it may be better to adopt a solution that uses a separate query plan per query. As depicted in Figure 13.1, the two selects can filter out most of workload before it even reaches the join operator. For other instances, sharing of the join computation can be better than executing the two query plans separately.

For continuous query processing, since the stream characteristics can change over time, the optimizer may need to switch between the two plans depicted in Figure 13.1 at run time. The core of such query plan change involves indeed a switch between select and join. Clearly the migration strategies proposed in Chapter 12 now need to support the switching of select and join at run-time.

The parallel track migration strategy, one of the two migration strategies proposed in Chapter 12, is a general approach that does not rely on the details of the state purging in the query plan. So it naturally supports the switch of select and join. No changes need to be made to this strategy.

However, the moving state strategy proposed in Chapter 12 may or may not apply for switching select and join, depending on the direction of the switch. As shown at the top of Figure 13.2, if the optimizer were to migrate from the old plan on the right to the new plan on the left, it would need to pull the select up above the join. In this case the two states in the old

plan (S'_A and S'_B) match with the two states in the new plan (S_A and S_B) respectively. The old states can be copied directly to their corresponding matching states in the new query plan although the tuples in state S'_A have already passed the select filter. This is because the tuples in S_A in the new query plan on the left would eventually have to eventually pass the select operator. Thus copying the contents in S'_A directly to S_A does not alter the results of the query plan in any way.

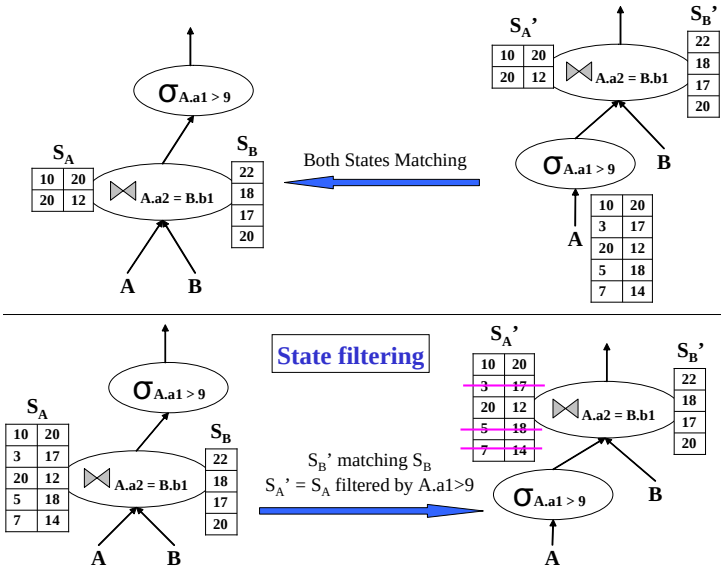


Figure 13.2: State Filtering.

On the other hand, as depicted in the lower half of Figure 13.2, if the optimizer were to migrate the old query plan on the left to the new query plan on the right, it would need to pull the select up through the join operator. State S_B still matches with S'_B and can be copied directly to S'_B. However, although state S_A matches with S'_A by tuple semantics, it *cannot* be directly

copied over to S'_A. This is because the tuples in S_A are yet to be filtered by the select operator, while the tuples in S'_A are expected to have already been filtered by the select at this point. Copying the tuples in S_A directly to S'_A would result in invalid tuples being present in S'_A, thus causing the query plan to create incorrect results.

To solve this problem, we add a new operation, called the *state filtering*, into the moving state migration strategy before the step of moving state. The *state filtering* operation applies the relevant select predicate to a state in the old query plan before copying it over to the corresponding matched state in the new query plan. This can filter out unwanted tuples from the old state and prevent the new query plan from generating incorrect results. As illustrated above, the state filtering is only necessary when the select operator is being pull up through a join operator.

13.2 Queries with Project and Join

For a continuous query plan that contains project and join, a project and a join may also be switched in the context of sharing or un-sharing in the multiple query execution scenario as discussed in Section 13.1. A project may be pushed down or pull up through a join operator as the stream characteristics change. The parallel track strategy again naturally supports this new optimization scenario because it is general and does not depend on the state operations inside both the old and the new migration boxes.

However, necessary changes need to be made to the moving state migration strategy. A project operator takes an input tuple and removes some

of its columns, which means that it changes the semantics of the input tuples. Therefore by switching a project and a join, the semantics of states may be altered as well. Figure 13.3 depicts the two directions of switching a project and a join and the new operations that need to be added to the moving state strategy.

When the project is pulled up through a join, as shown in the top half of Figure 13.3, the contents in the old state S'_B can be copied directly to its matching state S_B. However, the old state S'_A may or may not be copied directly to the new state S_A, depending on the system implementation of the project operator. In some systems such as our CAPE system [RDS+04], the project operator filters columns according to the column positions. This indicates that the project operator in the new plan on the left is expecting the tuples output by the join operator to have three columns, and the second and the third column are therefore kept ($A.a2$ and $B.b1$) after the project. For such an implementation, copying the tuples in S'_A directly to S_A would cause problems for the project operator because tuples in S'_A only have one column instead of two.

To solve this problem, we add another state operation, called the *state stuffing*, which adds a *null* column to the appropriate position of tuples in the old state before copying them over to states in the new query plan. This operation is only necessary when the project operator is implemented as discussed above. Another possible implementation of the project operator is to remove tuple columns based on their semantic IDs. In this implementation, the project operator is able to identify the semantics of each tuple column and keep the correct columns without considering their rela-

tive positions in the input tuple. For such project implementation, it is *not necessary* to apply the state stuffing operation.

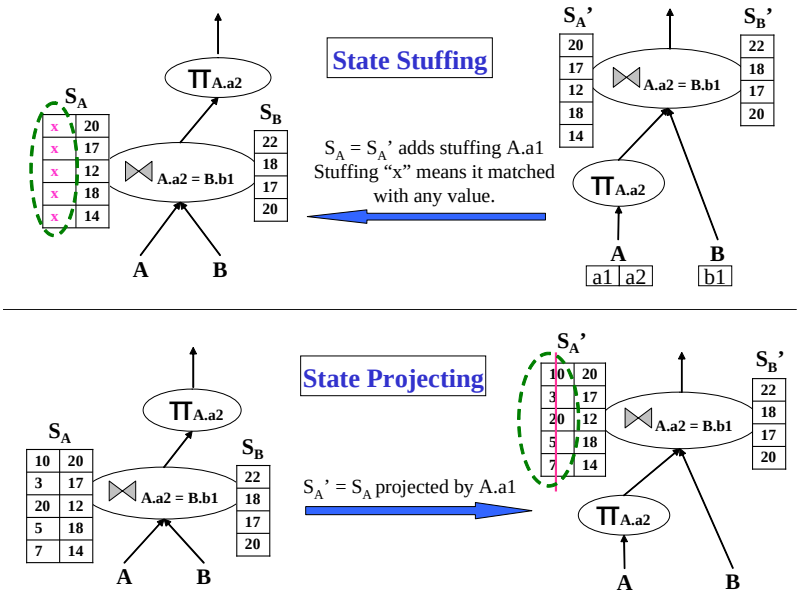


Figure 13.3: State Projecting and State Stuffing.

A project can also be pushed down through a join operator, as illustrated in the bottom half of Figure 13.3. It would be split into multiple project operators as necessary. The old state S_B can be copied directly to the new state S'_B. However, if the old state S_A is directly copied to the new state S'_A, the tuples in the new state S'_A would have two columns, while in fact they should only have one column because the project operator $\Pi_{A.a2}$ should by now already have been applied to these tuples in states S'_A. The same problem applies to S_B and S'_B.

Therefore for the moving state migration strategy, we add a state operation *state projecting* to project redundant tuple columns from states to

guarantee the correctness of the final query results.

13.3 State Matching Methods for SPJ Queries

As discussed in Section 12.1, the state matching step is an important step of the moving state migration. In Section 12.1.1, we have discussed a rather simplistic first state matching strategy based on comparing the schema of state tuples. Two states that have the same schema are matching states and their contents can be copied directly from one to the other during plan migration.

However, this simple schema-based state matching method works only when the query plan contains just join operators. For query plans that also contain select and project operators, this state matching method becomes insufficient. This is because a project operator placed before a join instead of after filters out some columns from a tuple and thus may change the semantics of a join state. This causing two join states (one in the new plan and one in the old plan) that should have been matched become un-matched when attempting to use the simple schema-based state matching method. Furthermore, a select placed before a join instead of after filters out some tuples from a join state. These filtered tuples would have been in the same join state if the select is applied later than the join. Therefore an old join state and a new join state in the new plan may have the same state schema but different sets of tuples. Their contents cannot be copied directly from each other.

We have designed two new state matching methods for SPJ queries.

The two methods are designed for two different runtime optimization approaches. One method is used for *incremental optimization*, while the other can be used for *total re-optimization*. The *incremental optimization* optimizes the query plan step by step starting from its current plan shape [ACC+03]. Each step applies one of the rewriting rules to the existing query plan, changing the plan to one of its neighbors. For example, switching two consecutive join operators or pushing a select operator through a join below it. The *total re-optimization*, usually used in traditional static query optimization, searches the complete or part of the query plan space until a good query plan is found. It does not start with the current shape of the query plan. Below we describe the two different state matching methods designed for the two different optimization approaches respectively.

13.3.1 State Matching for Incremental Optimization

When the incremental optimization approach is applied, we are able to record the changes made to the query plan step-by step. The state matching method can take use of this feature.

We assign a temporary distinctive state ID to each state in the part of the query plan that is to be optimized. During each optimization step, the optimizer records three changes made to the query plan. First, if two joins are switched, the new state created by this switch is given a new distinctive state ID. Secondly, if the rewriting is to push down select or project through a join, the corresponding select or project conditions are recorded in the affected states of the join operators as interpretation instructions. Thirdly, if the rewriting is to pull up a project through a join, the columns that need

to be stuffed are also recorded in the corresponding operator state.

Figure 13.4 shows an example when the rewriting is to push down a select. In this example, the join state affected by this switch is the state S_B and thus the select condition $\sigma(B.b1 > 9)$ is recorded in this state. After the optimization is done, the states in the newly generated plan can be matched with those in the old plan by comparing state IDs. Only an old state and a new state that have the same state ID would be matched with each other. When moving tuples from an old state to a new matching state, according to the migration instructions stored in the join states, the corresponding state operation, including state filtering, state projecting and state stuffing, would be applied to the state before copying it over to the matching new state.

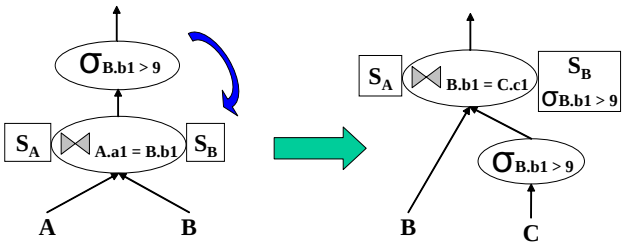


Figure 13.4: State Recording for One Step in Incremental Optimization.

Figure 13.5 depicts another example of applying the migration instruction scheme described above. Note that the query plan shown in this example can be treated as a subplan inside a bigger query network. Suppose the old sub-plan applies a select and project after two join operators. The optimizer decides to push down select and project and also to switch the two joins, according to current data statistics. All join states are each as-

signed a unique state ID in the range of [1, 4] before optimization starts. An incremental optimizer would push the select and the project step-by-step through the two join operators, and switch the two joins in another step. One project may be broken into several projects if the projecting columns exist in multiple inputs. The same applies to the select operator. As we can see from Figure 13.5, when the two joins are switched, the newly generated join state is assigned a unique state ID 5. When the select is pushed through a join, the select predicate is recorded in the affected join states, here states S_{BC} and S_C. The projecting columns are also recorded in an affected join states when a project is pushed down through a join. In Figure 13.5, all the join states are affected when pushing down the project. After the optimizer has generated the new query plan, the states between the old plan and the new plan are matched by comparing their temporary IDs. During state moving, an old state that has a matching new state in the new plan must apply the state operations indicated by the migration instructions recorded in the corresponding matching new state.

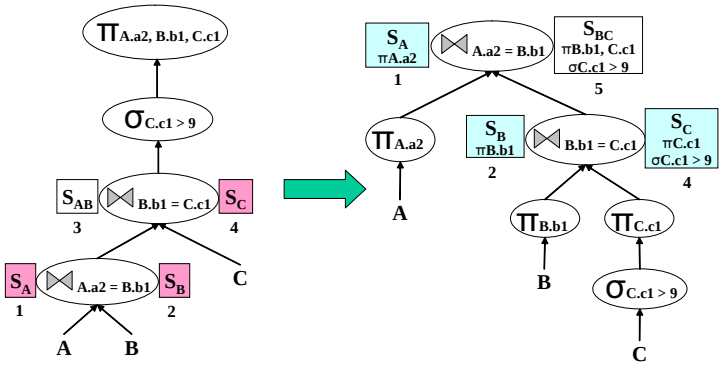


Figure 13.5: State Matching for Optimization by Steps.

13.3.2 State Matching for Total Re-Optimization

Next we describe a new state matching algorithm, referred to as the *bit-map algorithm*, for the total re-optimization method. A state matching method for total re-optimization needs to have two functionalities. First, it needs to find "matching" old states and new states. Second, it needs to apply appropriate state operations to old states in order to get the correct state contents and copy them to the matching new states.

To achieve the two functionalities, the proposed bit-map algorithm first assigns a unique state ID to each join state based on the inputs that tuples in this state come from. For example, as shown in Figure 13.6, state S_A only contains tuples from input stream A, so its ID is S_A. The join states in Figure 13.6 are all marked by their state IDs, denoted as subscripts of the state names.

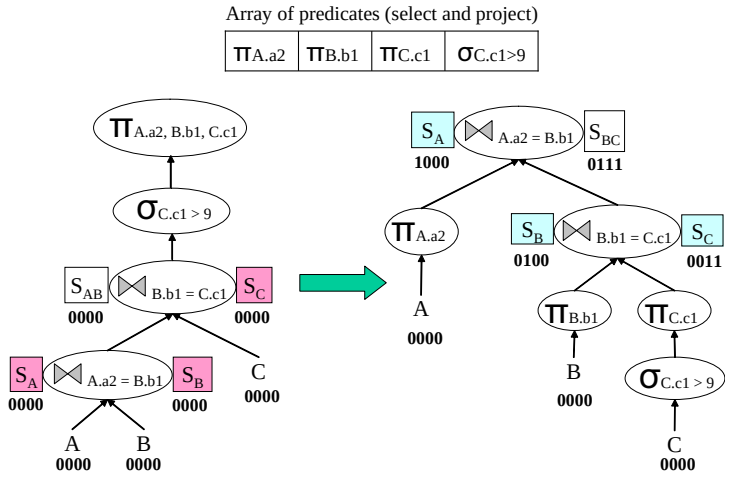


Figure 13.6: State Matching for Traditional Search-based Optimization.

The algorithm then generates an array of predicates. Simple boolean

expressions of all predicates, with each predicate representing a select condition or a single column of a project operator, are collected into an ordered array of predicates. As shown at the top of Figure 13.6, in this example, the array of predicates contains one select and three project predicates.

After the array of predicates has been determined, each join state is then assigned a bitmap with the number of bits equal to the size of the predicate array. A bit "1" indicates that the corresponding predicate in the predicate array has been applied to tuples in this state, while a bit "0" means the opposite. All bit maps are initialized to be 0. Each stream is also assigned an all-zero bitmap.

Now the bit-map algorithm starts to propagate the all-zero bitmaps bottom up starting from each stream until reaching the root operator in both the old plan tree and the new plan tree, using the following two propagation rules:

1) If the next parent operator is a select or a project, set the corresponding bit in the propagating bitmap to "1".

2) If the next operator has a state that stores tuples from the input that the propagating bitmap is currently located, merge the existing state bit map with the propagating bitmap using *bit-wise OR*, and assign the new bitmap to be the state bitmap. The propagating bitmap is also set to be equal to the newly generated bitmap.

The above propagation rules are applied repeatedly until the propagating bitmap reaches the root operator. The same propagation procedure is applied starting from each stream in the query tree. This allows the bitmap to be propagated following all possible tuple flows in the query plan. The

orders of propagating paths do not matter. Therefore, after the complete bitmap propagation procedure, the bitmap in each state records exactly the select or project predicates in the predicate array that have been applied to tuples in this state. Figure 13.6 depicts the bitmaps inside each state at the end of the bitmap propagation procedure. For example, none of the select or project predicates have been applied to tuples in state S_B in the old plan, so the bit-map of that state is marked as 0000. However, the state S_B in the new plan is marked as 0100 because a project has been applied to the state tuples. As another example, the bitmap in state S_{BC} in the new plan is the bit-wise OR between the bitmaps in state S_B and S_C. This is an application of the second propagation rule described above.

After this bottom-up bitmap propagation, the old states and the new states are matched by their state IDs. The bit-maps of a pair of matching states are then compared bit-by-bit. Two combinations of old bit and new bit indicate that additional migration operations need to be applied to the matched states: (1) If a bit in the bit-map of the old state is 0, while the same bit in the new state is 1, then the old state needs to apply the corresponding state filtering or state projection. (2) If an old bit is 1 while a new bit is 0, and if the bit indicates a column projection, the old state needs to apply state stuffing on the predicate column before copying over the tuples to the new state. (3) If an old bit is 1 while a new bit is 0, and if the bit indicates a select condition, it means that the select condition has been applied to all the tuples in the matching old state and therefore no state filtering is necessary. This is explained earlier in the upper half of Figure 13.2 in Section 13.1.

Chapter 14

Migration Queries with Group-by And Aggregates

Group-by is also an important stateful operator in continuous queries. I will illustrate that mixing group-by and join operators creates new optimization opportunities and hence requires corresponding support from the migration strategies.

In traditional query processing, group-by is usually performed after join. However, pushing the group-by below the join can be beneficial under certain conditions. Intuitively, group-by can put multiple tuples into a single group, thus potentially decreasing the number of tuples fed into the subsequent joins. Such optimizations have been studied for static databases in [CS94, YL94]. Clearly, the same optimizations can also be applied to continuous queries.

In this section, I study the migration process of switching group-by and

join at runtime. The optimal order of evaluating group-by and join depends on the statistics of the data and can be determined by cost models, such as those presented in [CS94]. This is an orthogonal problem to the dynamic plan migration problem I am concentrating on. Thus I do not discuss it further here.

14.1 The Migration for Switching Join and Group-by

I first describe a representative example of switching group-by and join. Thereafter I illustrate the migration of such plan change.

The following two streams are used in our examples:

```
AStream(aid, a2)
BStream(bid, aid, b2, b3)
```

For the two streams, aid is the primary key of $AStream$, and bid is the primary key of $BStream$. The second column aid in $BStream$ is a foreign key reference to $AStream$. Given the two streams, consider the following query written in a stream CQL-like language [ABW03]:

```
Query 1:
SELECT    A.aid, A.a2, SUM(B.b3)
FROM      AStream A [range 30 min], BStream B [range 30 min]
WHERE     A.a2 = B.b2
GROUP BY  B.b2
```

This query requires a join (between A.a2 and B.b2) and a group by on column B.b2. An aggregate function *sum* is applied to each group in the

group by. Notice that neither A.a2 and B.b2 are keys nor are they foreign keys to another stream. So the join is a *many-to-many* join between the two input streams. The notation *range* after each stream defines the window constraint. In this example, both streams have window constraints of 30 minutes. This indicates that the join has a sliding window constraint of 30 minutes and the group-by outputs results for each window of the most recent 30 minutes.

Figure 14.1 depicts the changes that need to be made to the query plan when changing the evaluation order of the group-by and join. When the query changes from evaluating the group-by after the join (the plan on the left) to evaluating the group-by before the join (the plan on the right), since the join is a many-to-many join, simply pushing down the group-by through the join is not sufficient. This is because for the new query plan on the right, the lower group-by produces groups based on the value of B.b2. However, each group of each distinctive B.b2 value may still join with multiple A tuples, namely those with the same value of A.a2. So more than one tuple may be produced for each group by that subsequent join. Thus we need another group-by on top of the join to group these tuples together into one final group. Lastly, here we then compute the total sum of B.b3 for that group. This in fact can be considered equivalent to conducting one group-by in two stages. Given certain data statistics, the two-staged group-by can be better than the single group-by because the lower group-by decreases the number of tuples input to the join, and hence may decrease the total processing cost spent on joining tuples. It is clear that such savings only exist under certain cost-based conditions. Discussions on such cost-based

optimization can be found in [CS94, YL94].

On the other hand, group-by and join can also be switched in the opposite direction. As shown in Figure 14.1, the plan on the right evaluates the group-by as early as possible, while the plan on the left evaluates the group-by after the join. Changing from the right plan to the left plan has the effect of merging two group-bys into one group-by operator on the top.

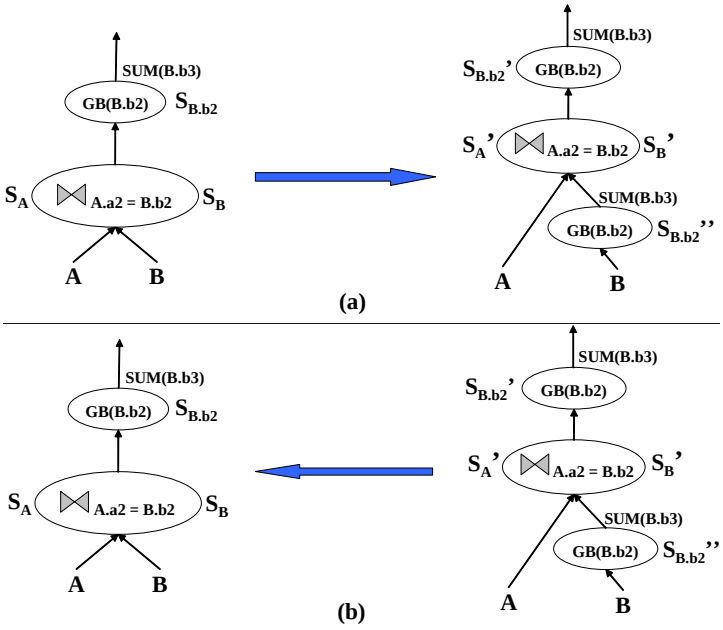


Figure 14.1: Switching Group-by and Join by Moving State Strategy – General Case.

14.1.1 Applying the Moving State Migration Strategy

I now describe how this change of the query plan can be achieved by the moving state migration strategy. The general concept of the moving state strategy still applies here. The only changes must be made are in the state

matching and moving stages. Figure 14.1(a) shows the case when migrating from the old query plan on the left that evaluates the join first to the new query plan on the right that evaluates the group-by first. It is clear that the join states between the old plan and the new plan can still be matched accordingly during the state matching process. In this example, the old join states S_A and S_B are matching states to the new join states S'_A and S'_B, respectively. So the contents in the old join states can be copied over to the new join states.

However, since multiple group-bys now exist in the new query plan while the old plan only has one group-by (as shown in Figure 14.1(a)), the matching between the group-by states does not work the same way as the matching of join states. As analyzed in [CS94], the lower group-by in the right plan in Figure 14.1(a) is semantically redundant but not computational redundant. This means that the lower group-by can be inserted and removed without affecting the correct semantics of the query, but adding or removing it can affect the performance of the query plan. Based on this principle, in Figure 14.1(a) when migrating from the old plan on the left to the new plan on the right, the old group-by state $S_{B.b2}$ can be matched to the new group-by state $S'_{B.b2}$. So the content of $S_{B.b2}$ can now be copied to $S'_{B.b2}$ directly. On the other hand, the group-by state $S''_{B.b2}$ in the lower group-by can start as empty without having been matched to any existing old state. The empty group-by state will build up as it starts processing new input data.

A similar state matching strategy can be applied to the migration case depicted in Figure 14.1(b). For this direction of migration, the old plan on

the right has multiple group-bys while the new plan on the left has only one group-by after the join. The top most group-by states, namely $S_{B.b2}$ in the new plan and $S'_{B.b2}$ in the old plan, can still be matched with each other. However, at this point the group-by state $S''_{B.b2}$ of the lower group-by in the old plan can be non-empty. The tuples stored in state $S''_{B.b2}$ cannot be thrown away and must be preserved in some way to eventually also be processed by the join operator in the new query plan. Otherwise the query may miss some results. To keep the tuples in the unmatched group-by states, we can insert them into the appropriate queues in the new plan. For the example in Figure 14.1(b), the tuples in the unmatched group-by state $S''_{B.b2}$ are inserted into input queue B of the join operator. In this way, the tuples that have been processed by the lower group-by in the old plan will still be processed by the join operator.

The rest of the moving state migration as described in Chapter 12 can be applied as before. As shown in Algorithm 7, the tuples accumulated in the intermediate queues at the migration start time are always cleaned first before state matching and state moving activities are started.

14.1.2 Applying the Parallel Track Migration Strategy

The basic idea of the parallel track migration strategy is to run both the old plan and the new plan concurrently. Once the parallel track migration process starts, both plans are processing the same new data until all the old tuples in the operator states inside the old plan are expired. At that point, the migration process is over. In principle, this strategy works for any operator type. Therefore we can still apply the parallel track strategy to

the problem of switching plans with different evaluation orders of group-by and join.

However, applying the parallel track migration strategy directly to a query plan with both join and group-by may result in missing result tuples. This is because such a query groups tuples together into one combined aggregate tuple. Thus both the old plan and the new plan may each produce two such partially aggregated tuples that must be further combined by incorporating the values from tuples in this group that come from the old plan and the new plan. For a pair of an old plan and a new plan that both contain a group-by, as shown in Figure 14.1(a), tuples that belong to the same group may be partially generated by the join in the old plan, and partially by the join in the new plan. However, the two group-by operators on the top of each plan would form each group separately based on only their partial join results. As an example, a group of a specific B.b2 value may contain a total of 10 tuples in a certain window frame. So the correct results from the group-by for that window frame should be the sum of the B.b3 column of these 10 tuples. However, 3 of the 10 tuples may be generated by the join in the old plan and the other 7 may be generated by the join in the new plan. So the result outputted from the old plan is the sum of B.b3 columns of 3 tuples, while the result from the new plan is the sum of B.b3 columns of the other 7 tuples. Simply put the two aggregate results together won't form the correct final results. The two aggregate outputs have to be once more combined into the same group so that the two partial sums can now be added together to output the correct final result.

To solve this problem, I add a temporary special Group-By operator,

shown as the *SGB operator* in Figure 14.2, on the top of the two concurrently running plans to combine their results together and to put the final correct results into the output queue. This is similar to the concept of the merging nodes in parallel aggregate execution [SN95]. The temporary *SGB operator* serves two purposes. First, if two aggregate tuples belong to the same group (based on the group-by condition) and the same window frame but are output by the two distinctive plans, the SGB operator puts them into the same group. Whether two aggregate tuples belong to the window frame can be easily determined by the two tuples' timestamp. Secondly, the SGB operator can then apply a pre-defined function to combine the two aggregate tuples that belong to the same group into a single output tuple. A combining function may need to be designed for each type of aggregate function used in the queries.

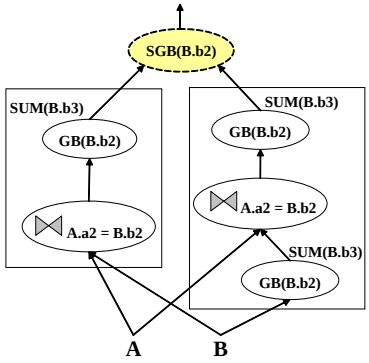


Figure 14.2: Applying Parallel Track Migration Strategy When Switching Group-by and Join.

Only certain types of aggregate functions used in the query plan can be supported by the SGB operator. To be exact, only *distributive aggregate*

functions [GBLP96] can be migrated using an SGB operator.

The definition of a distributive aggregate function is as following: Let $A()$ denote an aggregate function that can be applied to a data set D. Suppose two non-overlapping data sets $D1$ and $D2$ are two subsets of the data set D and $D1 \cup D2 = D$. Suppose $A1$ and $A2$ are the aggregate results generated by applying function $A()$ over $D1$ and $D2$ respectively. We say the aggregate function $A()$ is distributive if we can find a function $F()$ so that:

$$A(D) = F(A1, A2)$$

An example distributive aggregate function is *sum*, because *sum(D) = A1 + A2*. However, the aggregate function *avg* does not have this property because we do not have a function $F()$ such that *avg(D) = F(A1, A2)* without knowing the cardinalities of the two data sets $D1$ and $D2$. The aggregate functions that are currently supported by the SGB operator in our prototype CAPE system [RDS+04] include *count*, *min*, *max* and *sum*. In principle, any distributive aggregate function can be supported by the SGB operator. For queries that need to apply non-distributive aggregate functions, the moving state migration strategy described earlier can be used instead to achieve the runtime plan migration.

14.2 Group-by and Key-to-Foreign-Key Join

In [CS94], a special case for such group-by rewriting has been identified when the joins are *key-to-foreign-key* joins (one-to-many joins) and the group-by column is on the foreign key as well. From now on I refer to this type

of join as *key/fkey join*. In this case, only a single group-by is needed after pushing the group-by down through a key/fkey join operator, as has been discussed in [CS94]. The group-by on the top, as shown in the right plan in Figure 14.1(a), is no longer necessary. This is because when the join is a key/fkey join and the group-by column is on the fkey, each tuple with the fkey column can only join with exactly one tuple. Therefore each group formed in the group-by operator only contains one tuple, no matter in what order the join and the group-by operators are being evaluated. I use the following query example to show such a special case:

```
Query 2:
SELECT    A.aid, A.a2, SUM(B.b3)
FROM      AStream A [range 30 min], BStream B [range 30 min]
WHERE     A.a1 = B.a1
GROUP BY  B.a1
```

The above query requires a join and a group-by operator. A possible query plan for the above query is depicted on the left of Figure 14.3(a). Since $A.a1$ is the key column of stream A and $B.a1$ is the foreign key in stream B referencing A, the join is a key/fkey join. Also the group-by column is on the foreign key column B.a1 of stream B. If the two operators are switched, only a join and a group-by are needed in the new query plan, as shown on the right of Figure 14.3(b). However, we can still put another group-by on the top of the join, as shown on the right of Figure 14.3(a). Placing another group-by on the top is semantically redundant because each input tuple to this group-by ends up forming a unique group.

However, this additional operator would not affect the correctness of the final results.

In fact, the extra group-by on the top is temporarily required during the moving state migration process when migrating from a plan with the group-by on the top of a join to a plan with the group-by being pushed down. Using the example in Figure 14.3(a), the migration is from the plan on the left to the plan on the right. The tuples in the join state S_B can be divided into two types: *key-matched* and *key-unmatched*. A *key-matched* tuple T in S_B has been joined with a tuple from input A. This means that the A tuple with the corresponding key value that matches the foreign key column of the tuple T has already arrived and in fact exists in the same window frame as the tuple T. A tuple in S_B is *key-unmatched* if it hasn't been joined with any tuple yet. Since join state S_B matches with join state S'_B in the new query plan, all tuples are moved from S_B to S'_B after the state matching stage. However, a key-unmatched tuple in S'_B can potentially be joined with a future incoming tuple from input stream A. At that point, it would need to be put into the group of tuples with the same foreign key column and contribute to the final aggregate value of that group. Therefore another group-by/aggregate operator is needed after the join in the new query plan, even when another group-by were to already exist below the join operator. This extra group-by is *not* semantically redundant at this point.

This extra group-by on the top of the k/fk join, as shown in the right plan of Figure 14.3(a), is only temporarily necessary. It will become unnecessary once all old tuples in the join state S'_B (that had been inserted

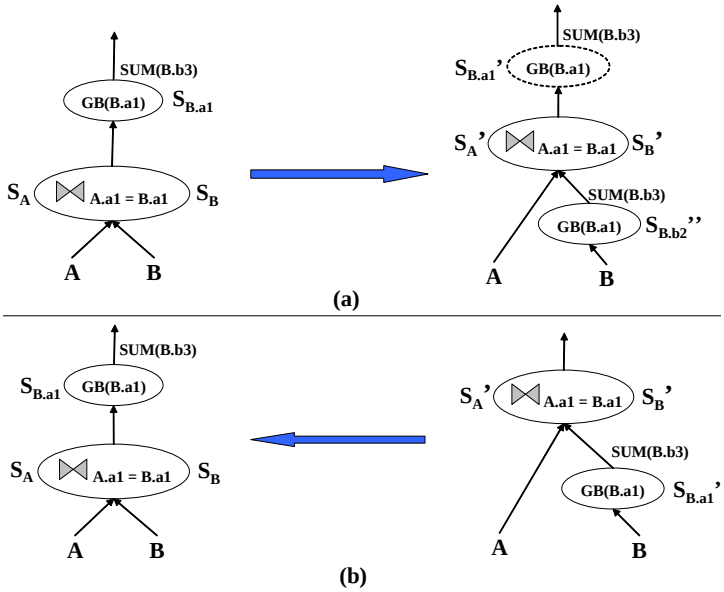


Figure 14.3: Switching Group by and Join – Special Case.

into S_B before the migration started) are eventually being purged from the state. After that, the group-by on the top can be removed from the query. The moving state migration for the opposite direction, as shown in Figure 14.3(b), as well as the parallel track migration for both migration directions, are the same as the general group-by migration strategies described earlier in Section 14.1.

Chapter 15

Execution Models and Generalized Migration Strategies

As described in Chapter 11, continuous query systems may adopt various *execution models* to determine the tuple execution order, thus to ensure correct tuple processing and purging given a window constraint. The execution model affects the tuple order in each intermediate queue in the query plan. It also affects the tuple timestamp representation in the system.

The migration strategies described in previous sections are based on the assumption that the *totally synchronized execution model*, as described in Section 11.5, is being used. This model is the most strict execution model that guarantees that tuples are always executed in the same order as they arrive. This simplifies the migration process and avoids potential migra-

tion problems that may otherwise arise if other less restricted models were being used.

In this section, I generalize the migration strategies proposed in previous sections by relaxing the assumption I made on the execution model. I first categorize the execution models found in existing continuous query systems in the literature and then discuss their corresponding timestamp representations and tuple purging algorithms. These execution models I have identified include the *totally synchronized model*, the *semi-synchronized model* and the *un-synchronized model*. I then describe the changes that need to be made to the migration strategies when used in systems that employ these alternative execution models. In particular, I identify the necessity of applying an additional *synchronization* process during a migration. Lastly, I describe the methods of synchronization that can be applied in the migration process.

15.1 Execution Models

Any continuous query system relies on a clear execution model to carry on proper query processing. An execution model defines the relationship between the order of tuple arrival and the order of tuple processing. The most restricted execution model requires that these two orders are exactly the same, while the most relaxed model poses no restrictions on the relationship of these two orders.

Depending on how strict the execution model is, I categorized existing execution models into three classes: *totally synchronized model*, *semi-*

synchronized model and *un-synchronized model*.

15.1.1 Totally Synchronized Execution Model

As discussed in Section 11.5, this is the most strict execution model for continuous query processing. When using this model, tuples are being processed in *exactly* the same order as they arrive. By applying the complete synchronized execution model, tuples in any queue in the query plan are ordered by their *max* timestamp. Hence for a combined tuple, keeping only the max timestamp is sufficient when the complete synchronized execution model is applied.

By using this model, a tuple $t1$ that has a smaller timestamp than a tuple $t2$ is guaranteed to be processed before $t2$, even if $t1$ and $t2$ are in different input queues. Conceptually, we can consider the system as having a single stream input queue. Whenever a stream tuple arrives, it is placed in this stream queue. All leaf operators in the query plan obtain tuples from this single input queue.

When applying such a model, a scheduling algorithm that would modify the execution order cannot be applied at the same time. The execution order of tuples is instead completely controlled by the execution model.

15.1.2 Semi-Synchronized Execution Model

The *semi-synchronized execution model* is a bit more relaxed than the previous model. This model only enforces that each operator processes tuples in all its input queues in increasing order of their timestamps. Thus when one

of an operator's input queues is empty, the operator cannot be scheduled to run. This guarantees that the tuples are being processed in order by each operator. Such an execution model has been adopted in some existing prototype stream processing systems [ABB+03, CCD+03].

Different from the totally synchronized model, this model *only* enforces the tuple execution order to be the same as the tuple arrival order locally *at each operator*. It does not enforce the tuple execution order across *all* input streams (nor all operators). Although this model is more relaxed in execution order, the combined tuples in each queue are still ordered by their max timestamp because each operator makes sure that tuples in its input queues are executed in the right order.

This model gives more control to the scheduling algorithm than the total synchronized execution model. A scheduler is able to choose which operator to run next as long as that operator does not have an empty queue.

15.1.3 Un-Synchronized Execution Model

The *un-synchronized execution model* does not pose any constraints on the tuple execution order. It is completely up to the scheduler to pick which operator to run next. Inside each operator, the tuples do not need to be executed in order. The benefit of such a model is that the scheduling algorithm does not have any restrictions and can be optimized to achieve the best performance. However, an obvious drawback is that the combined tuples in queues are ordered neither by max nor by min timestamp of sub-tuples. To preserve complete temporal information in order to apply correct state purging based on window constraint, a combined tuple needs to keep all

timestamps of all its sub-tuples. I refer to this type of timestamp as the *combined timestamp*. For example, as depicted in Figure 15.1, after joining tuples A and B, the new timestamp for the joined tuple AB should be a combination of both the timestamp of the sub-tuple A and the timestamp of the sub-tuple B. An state purging algorithm using combined timestamp is described in the following section.

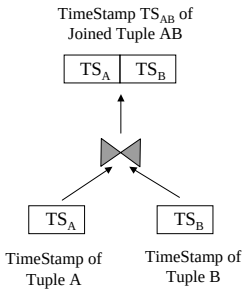


Figure 15.1: Combined Timestamp

As shown above, the execution model can determine the timestamp representation for combined tuples in the system. This is important for state purging and thus may affect the migration process. Worst yet, when tuples are not being executed in the same order as they arrive (as would be the case in the semi-synchronized or un-synchronized models), out-of-order execution is possible. This means that some tuples that arrived earlier (with smaller timestamps) may be executed later than some other tuples that arrived later (with larger timestamps). As I will show in Section 15.3.1, the out-of-order execution creates a problem during the migration process. It must be dealt with by the migration strategies via a process that is termed as *execution synchronization*.

15.2 Timestamp Representation and State Purging

As discussed earlier, by applying the un-synchronized execution model, combined tuples in the same queue may not be ordered by any of the two timestamps from the two inputs. This has been shown in Figure 11.3 in Section 11.4. Therefore it is necessary to keep combined timestamps for combined tuples.

As discussed in Section 11.4, purging a state using a tuple with a singleton timestamp is rather straightforward: For a join operator $A \bowtie B$ with window size W, since tuples from stream A with singleton timestamps are strictly ordered in non-descending order, a B tuple in state S_B is purged by an A tuple if and only if $(TS_A - TS_B) > W$.

Purging a state by a combined tuple with combined timestamp is more complex than purging by a tuple with a singleton timestamp. This is because a combined timestamp has multiple columns and may not be ordered by any of these timestamp columns. By utilizing the same purge algorithm described above, some tuples may be purged by earlier combined tuples even though they may still have the potential to join with future incoming tuples. In the following section, I describe the extended algorithm for purging a state by tuples with combined timestamps.

15.2.1 Purge by Combined Tuples

Although a sequence of tuples with combined timestamps is not strictly ordered by any one of its timestamps, some *Timestamp Order* can still be observed. Since combined tuples are usually only generated by join or union

operators, in this section, I use join to illustrate the timestamp order and our proposed purging algorithm for combined timestamps.

Lemma 15.1 (Timestamp Order Lemma) *Let t and t' be two tuples in the output queue of a binary window join operator. Both tuples have timestamps of size n, represented as* $[TS_1, ..., TS_n]$ *and* $[TS'_1, ..., TS'_n]$ *respectively. If tuple t appears earlier than tuple t' in the queue, then there must exist at least one i* $(1 <= i <= n)$*, such that* $TS_i < TS'_i$.

Proof: I now give a proof by induction on the size of timestamp array n. Suppose that the window join operator has two input queues Q_L and Q_R, two states S_L and S_R, and one output queue Q_{LR}. t and t' are tuples in Q_{LR}.

***Base case*: n = 2.** Let $[TS_1, TS_2]$ and $[TS'_1, TS'_2]$ be the timestamps of tuples t and t' respectively. Tuples with a combined timestamp array of size 2 must be formed by joining two sub-tuples each with a timestamp of size 1. So t is formed by joining t1 with timestamp TS_1 and t2 with timestamp TS_2. And t' is formed by joining t1' with timestamp TS'_1 and t2' with timestamp TS'_2. Without loss of generality, let us assume that t1 and t1' are from Q_L, and t2 and t2' from Q_R. All tuples in Q_L and Q_R are singleton tuples and are strictly ordered by their timestamps, respectively.

Since tuple t comes before t' in Q_{LR}, t must have been generated earlier than t'. When sub-tuples t1 and t2 are about to be joined to generate tuple t, two cases are possible: 1) t1 is the *first* tuple in Q_L and t2 is inside S_R, or 2) t1 is inside S_L and t2 is the *first* tuple in Q_R. At this time, sub-tuples t1' and t2' cannot both be in states. Because otherwise they must have been joined

already and tuple t′ would appear before t in Q_{LR}. So if sub-tuple t1′ with timestamp TS'_1 is not yet in S_L, then it either is or will still arrive in Q_L. In this case we have $TS_1 < TS'_1$ and i = 1. If sub-tuples t2′ with timestamp TS'_2 is not in state S_R, then it is or will arrive in Q_R. So $TS_2 < TS'_2$ and i = 2.

From above I conclude that for base case n = 2, there always exists an i such that $TS_i < TS'_i$.

Inductive Hypothesis: Assume that the timestamp order lemma holds for any tuple sequence with size $n <= k$.

Inductive Step: I now show that the timestamp order lemma also holds for sequences with size $n = k + 1$.

The timestamp array for t with size n = k + 1 can be treated as a combination of two sub-tuples t1 and t2 with timestamp arrays as $[TS_1, ..., TS_j]$ and $[TS_{j+1}, ..., TS_{k+1}]$, respectively. Similarly, t′ can also be treated as the combination of two sub-tuples t1′ and t2′ with timestamp array as $[TS'_1, ..., TS'_j]$ and $[TS'_{j+1}, ..., TS'_{k+1}]$, respectively. Since each array is at least of size 1, it must be true that $j <= k$. So both timestamp arrays have a size of at most k.

Using the same reasoning as in the base case, when sub-tuples t1 and t2 are about to be joined to generate tuple t, at least one sub-tuple t1′ or t2′ does not yet exist in its respective join state. If sub-tuple t1′ with timestamp $[TS'_1, ..., TS'_j]$ is not in state S_L, then it is or will arrive in Q_L. Since t1′ must come after t1 in Q_L, based on *Induction Hypothesis*, we know that there exists an m ($0 < m <= j$) such that $TS_m < TS'_m$. So in the case i = m. If sub-tuple t2′ with timestamp $[TS'_{j+1}, ..., TS'_{k+1}]$ is not in state S_R, then it is or

will arrive in Q_R. Since t2' comes after t2 in Q_R, we can again find *i* = *m* ($j < m <= k + 1 = n$) such that $TS_m < TS'_m$.

So I conclude that the lemma holds for any tuple sequence in a query plan. □

The timestamp order lemma naturally holds for tuple sequences with ordered singleton timestamps. Also for single-input operators, such as select, project and group-by, the order of input tuples is not altered by the operator itself. Therefore the timestamp order lemma still holds for the output tuple sequence from these operators.

By utilizing the timestamp order lemma, I now describe the general purge algorithm to safely purge tuples by either a singleton tuple or a combined tuple. I attach a min-max timestamp pair $[TS_{min}, TS_{max}]$ to each tuple, corresponding to the smallest and largest timestamps in its timestamp array. For a singleton tuple, TS_{min} equals TS_{max}.

Lemma 15.2 (Purging Lemma) *Assuming that timestamp order holds for any tuple sequence including queues and states*[1] *in the query plan, given two tuples t_L (with n timestamps) and t_R (with m timestamps) that have min-max timestamp pairs $[TS_{min_L}, TS_{max_L}]$ and $[TS_{min_R}, TS_{max_R}]$ respectively, if $(TS_{min_L} - TS_{max_R}) > W$, then t_R can be purged from its state by t_L.*

Proof: We need to show that t_R can be safely purged because it can no longer be joined with any tuple that arrives after t_L in that sequence. Because the timestamp order holds for any tuple t'_L arriving after t_L in the

[1] In the case of hash join, tuples belonging to the same hash bucket are assumed to be ordered by their insertion time.

same sequence, there exists an i $(0 <= i <= n)$ such that $TS_{i_L} < TS'_{i_L}$. Since TS_{min_L} is the smallest timestamp in the timestamp array of tuple t_L, we know that $TS_{min_L} <= TS_{i_L}$. Thus $TS_{min_L} < TS'_{i_L}$. Now for t_R, given any j with $(0 <= j <= m)$ we have $TS_{j_R} <= TS_{max_R}$. Since we know that $(TS_{min_L} - TS_{max_R}) > W$, putting above together, we can get $(TS'_{i_L} - TS_{j_R}) > W$. Since the global window constraint is assumed in any join pair, for any tuple t'_L that comes after tuple t_L in the same sequence, it is outside the W window frame from tuple t_R. So we conclude it is safe to purge t_R. □

The above general purging lemma works for both singleton and combined tuples. To our best knowledge, our timestamp order lemma and purging algorithm are the first algorithms to explicitly deal with the purging of a combined tuple with multiple timestamps. In the case of singleton tuples, our purging algorithm essentially reduces to the commonly used purge algorithm [KNV03, CCC+02, NWAea02, MSHR02].

15.3 Generalized Migration Strategies

Thus far, I have described our migration strategies based on the assumption that the continuous query system adopts the total synchronized execution model. This model guarantees that between two tuples with different timestamps, the tuple with a smaller timestamp is always processed earlier than the tuple with a larger timestamp.

However, such execution order property no longer holds for a system that adopts a semi-synchronized execution model or an un-synchronized

execution model. In either models, a tuple with a larger timestamp may be processed before a tuple with a smaller timestamp. This asynchronism between tuple timestamp order and tuple processing order creates new issues for correct state migration and requires changes in the migration strategies.

In fact, most of the changes that need to be made for generalization are only necessary for the moving state migration strategy. This is because the parallel track strategy itself is more general than the moving state strategy: It does not care about the details of how states are being purged, as long as tuples are being purged from states in some way so that the old box can eventually be expired. Therefore, using a different execution model does not impact the functioning of the parallel track strategy. The moving state strategy, on the other hand, relies on the knowledge of how states are maintained in order to re-utilize the useful information in the states of the old box. Therefore it relies on the proper tuple order for correct tuple purging, which is in turn determined directly by the execution model.

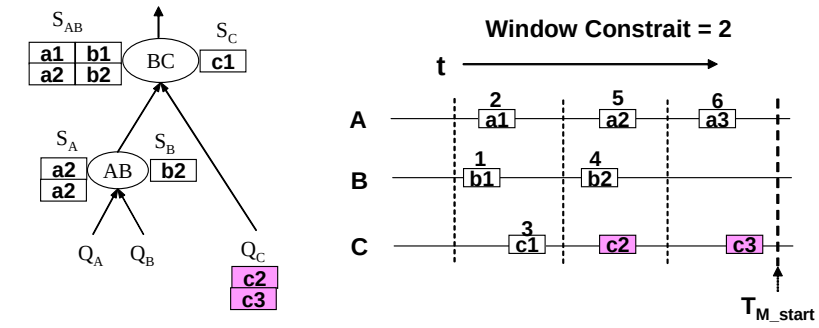
15.3.1 The Problem of Synchronization

Adopting the semi-synchronized or the unsynchronized execution model creates a new problem for the moving state migration process. The problem occurs when discarding the unmatched old states in the old box. When using these two execution models, the tuples may not always be processed in the order of the tuples' timestamps. So the *old state closure property* defined in Section 12.1.4 no longer holds. This means that tuples in unmatched old states may still be useful to future incoming tuples, i.e., they may still be joined with accumulated tuples in the input queues. Therefore the un-

matched old states cannot just be thrown away.

Figure 15.2 shows an example when the unmatched old states contain tuples that are still useful for future processing. Suppose that the query plan with two join operators shown in Figure 15.2(a) is contained in the old box during migration. Also assume states S_A, S_B and S_C are all matched states, while state S_{AB} is an unmatched state. At the migration start time T_{M_start}, the tuples inside each of the states and input queues are also shown in the figure. Here it shows that all tuples in input queues Q_A and Q_B are empty because all tuples inserted to these two queues have been processed at T_{M_start}. However, two tuples accumulated in input queue Q_C. The orders in which tuples arrived in the three input streams A, B and C are depicted in Figure 15.2(b). The window constraint is set to 2 time units in this example and one time unit is the elapsed time between two consecutive vertical bars. The number above each tuple indicates the order in which this tuple was processed. For example, among the tuples shown in Figure 15.2(b), tuple $b1$ is the first tuple being processed by an operator in the system, while tuple $a3$ is the sixth tuple being processed an operator. Tuples $c2$ and $c3$ are not labeled by numbers because they are being accumulated in input queue Q_C and have not been processed at time T_{M_state}.

When using the semi-synchronized execution model, each operator processes tuples strictly based on their timestamps. So the tuple execution order inside the join operator AB is $b1$, $a1$, $b2$, $a2$ and $a3$. The joined tuple from operator AB keeps only the max timestamp of its sub-tuples. Inside the operator BC, the execution order is also strictly based on tuple timestamps. However, the tuple execution order across different operators does



(a) Old Query Plan at T_{M_start} (b) Tuple Arrival Order and Processing Order

Figure 15.2: The Issue of Synchronization

not have to be strictly based on tuple timestamps. As indicated in Figure 15.2(b), tuple $a3$ arrives later than tuple $c2$, but it may be processed earlier by operator AB. This means $a3$ may have purged tuples outside its window from state S_A, such as tuple $b1$. As a result, $b1$ only exists in the old unmatched state S_{AB} but not the matched state S_B. However, $b1$ is still within the window frame from $c2$ so the two may be able to join with each other. The *old states closure property* is not satisfied in this example. So if the unmatched old state S_{AB} is discarded, useful data is lost and some tuples can be missing from the results.

The key reason for this problem is that the tuple processing across operators is not synchronized. Caused by either a scheduling algorithm or the limitations on system resources, tuples may not be processed immediately after they arrive. Rather they may accumulate in the input queues. The execution may be more advanced in regard to the timestamp in one operator compared to in the other operator. For example, as shown in Figure 15.2, at T_{M_start}, tuples in Q_C may have timestamps earlier than T_{M_start}. This can

affect both the state moving and state discarding steps in the moving state strategy.

15.3.2 The Punctuation-based Synchronization Algorithm

To solve this problem, I add an extra synchronization process before the state moving step in the moving state strategy. The goal of this process is to synchronize the execution among operators inside the old box so that they all have processed tuples with timestamp smaller than T_{M_start}. If we want to safely discard any unmatched states in the old box, one practical method is to finish processing all the accumulated tuples in the old box's input queues that have arrived before T_{M_start}. This works fine if all the old box input queues are stream input queues, which means they are input queues to the leaf operators.

However, if the old box contains only a sub-tree of the complete query tree and the box input queues are not the stream input queues of the whole query plan, we need to identify all the queues (from box input queues down to the stream input queues) that may have some contribution in terms of forwarding tuples to the old box. For example, in Figure 15.3, the query plan that is included in the old box is only a subplan of a larger query plan. The stream input queues that contribute to the input queues of this old box in fact include totally six queues, from Q_A to Q_F. The accumulated tuples that arrive before T_{M_start} in the six involved stream queues all need to be processed and pushed up the query tree until reaching the output queue of the old box.[2]

[2]I have further optimized this step by finding the largest timestamp of the first tuple (or

To coordinate this synchronization process among all the involved operators, I developed a *punctuation-based synchronization algorithm*. In each involved stream queue, I insert a synchronization punctuation tuple into the proper position, so that all tuples before this punctuation have timestamps smaller than T_{M_start} and all tuples after it have timestamps larger than T_{M_start}. Figure 15.4(a) depicts the status of the input queues Q_E and Q_F at the migration start time T_{M_start}. Figure 15.4(b) shows the status when a punctuation tuple has been inserted into the two queues respectively. If the queue is empty at T_{M_start}, as Q_F in this example, a punctuation would just be inserted to the first position of the queue. Each involved operator along the path, upon receiving such punctuation from each of its input queues, would propagate this punctuation by inserting a punctuation tuple into its output queue. The synchronization process is complete when the root operator inside the old box has received such a punctuation tuple from all its input queues.

Applying the moving state migration strategy to the un-synchronized execution model creates the same problem of unsynchronized execution among operators as the semi-synchronized execution model. Hence a similar punctuation-based synchronization approach can be adopted before unmatched old states can be discarded.

T_{M_start}, whichever is smaller) in all the involved stream input queues, and push up all accumulated tuples in these queues that have timestamps no later than this largest timestamp.

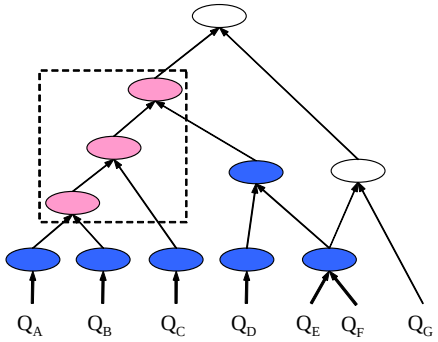


Figure 15.3: Trace Back to Contributing Stream Queues

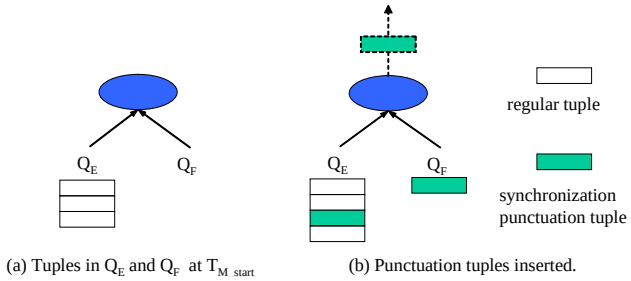


Figure 15.4: Synchronize by Propagating Punctuations

15.3.3 Discussions on Synchronization Methods

The punctuation-based synchronization process described earlier requires us to trace all the way back to the leaf level. This can be a problem if the subplan contained in the old box is high above in the query plan tree. In that case the number of operators involved in the synchronization process can be large. Hence the cost of synchronization may be fairly high. It may even dominate the total cost of the migration process. Ideally, the migration process should only affect the operators inside the old box and should not disturb the normal execution of other operators outside the old box.

To keep the synchronization as a local process to the old box, we can pick out all the sub-tuples in the unmatched states that do not exist in the matched states and insert them back into the corresponding matched states. By using this *localized synchronization algorithm*, all the useful information in the unmatched states will be kept and these unmatched old states can then be safely discarded. Note that one sub-tuple may be joined with several different sub-tuples and thus may appear in a state multiple times. Therefore duplicate elimination needs to be applied to the sub-tuples inserted back to the matched states. Otherwise duplicate results may be generated by the query plan. To ease the duplicate elimination process, a reasonable assumption is that all tuples arriving in the same stream would have different timestamps. So distinctive sub-tuples in a state would have distinctive timestamps. This fact can ease the process of duplicate elimination, because duplicate tuples can now be easily detected by having the same timestamps.

The localized synchronization method described above can be applied in a system using the un-synchronized execution model. However, it may not be sufficient to be applied in a system using the semi-synchronized execution model. This is because the synchronization method requires that tuples are first being drained from intermediate queues. When using this execution model, operators in the query plan need to execute tuples strictly based on the order of the timestamps. This creates a problem of draining tuples from the intermediate queues: If an operator has two input queues, such as a binary join operator, and if one input queue is empty, the operator cannot be processed. That is, it must wait until some tuples appear in the

empty queue. This can block the operator during migration which in turn would hang the migration process itself. So the synchronization among operators inside the old box cannot be achieved by only executing these old operators or utilizing information stored only in the old box.

One possible solution may be to undo the processing of tuples in the output queue of an old operator and put the original tuples (for example, sub-tuples of the joined tuples) back to the operator's input queues. The process can be done recursively from the root operator in the old box to the leaf operators in the old box. However, this method is in fact *moving backward* along the execution line, as doing so may waste work that has already been done. Instead, the approach of tracing back to the leaf level is actually *moving forward* along the execution line and thus none of its computation is wasting any processing.

Chapter 16

Experimental Evaluation

16.1 Experimental Setup

The proposed migration strategies are implemented in the CAPE system [RDS+04] and various experiments have been conducted to compare their performance. I use the query in Figure 10.1 on which the migration is performed to transfer the plan from the left to the right. System parameters such as stream input rates, operator reduction factors and global time window sizes are varied to reflect the changes of workload and data characteristics.

Our stream data generator generates tuples with arrival patterns modeled as the Poisson process. The mean inter-arrival delay between two consecutive tuples is exponentially distributed in order to model the Poisson arrival pattern. In each experiment, the stream generator continuously generates streams for 50,000ms. All query plans are being executed for a time period at least a few times longer than the global window in order to pass

the initial warm-up phase. A migration strategy is activated by the change of system parameters for the running query plan.

All implementation is done in Java. The experiments were run on a machine running windows 2000 with Pentium-III processor at 500MHz and 384M of main memory.

16.2 Length of Migration Stage

In this section, I analyze the experimental results related to the measured length of the migration stage and compare them with the estimation models described in Section 12.3.

Both old and new query plans in Figure 10.1 have a height h = 3. According to the Equation 12.4 in Section 12.3.2, the total length of the migration stage of the parallel track strategy should be $T_{PT} = 2W$.

Given the same query plans, by applying the Equations 12.1, 12.2 and 12.3 from Section 12.3.1, we can estimate the length of the migration stage for the moving state strategy as:

$$T_{MS} = \lambda_B\lambda_C W^2(T_j + T_s\sigma_{BC}) + 2\lambda_B\lambda_C\lambda_D W^3(T_j\sigma_{BC} + T_s\sigma_{BC}\sigma_{BCD}) \quad (16.1)$$

From the above results, we see that T_{PT} grows linearly with W. However, T_{MS} is controlled by several parameters, including input rates from Q_B, Q_C and Q_D, reduction factors σ_{BC} and σ_{BCD}, and the global time window W, with which it has a polynomial relationship.

The above estimations are based on the assumption that the system has enough processing power to handle incoming tuples without much delay. We judge the availability of system processing power in our experimental setup by comparing the total system running time vs. the stream generator running time. In our experiment, the stream generator in each experiment runs for 50,000 ms, generating stream tuples according to the given mean inter-arrival time. The system stops executing the query plan when there are no more tuples to process. If the system finishes at about 50,000ms as well, it implies that the system has enough processing power to keep up with the given parameter configurations.

To verify these estimations on the length of the migration stage, I run three sets of experiments:

1) Set 1: Only W increases linearly, while all other parameters are kept constant.

2) Set 2: I_B, the tuple inter-arrival time of stream B, is decreased (indicating that input rate λ_B is increased) while keeping all other parameters the same.

3) Set 3: W is increased linearly while other parameters are kept the same. The difference from set 1 and 3 is that set 3 has higher configurations with respect to input rates and operators' reduction factors.

Figures 16.1 and 16.2 depict the results of the first experimental set. Figure 16.1 illustrates that T_{PT} has a linear relationship with W and is statistically equivalent to 2W, as is suggested by the Equation 12.4 in the case of $h > 1$. The increasing curve of T_{MS}, marked as "Measured T_{MS} in Figure 16.2, indicates a close to polynomial relationship with W. A polynomial

Table 16.1: Parameter Configurations

Parameters	Section 16.2			Section 16.3	
	set1	set2	set3	set1	set2
W(ms)	vary	1000	vary	1000	2000
$I_A(ms)$	100	50	100	100	50
$I_B(ms)$	100	vary	12	100	50
$I_C(ms)$	100	50	12	100	50
$I_D(ms)$	100	50	12	100	50
σ_{AB}	0.1	0.1	0.1	0.1	0.2
σ_{BC}	0.05	0.05	0.1	0.02	0.05
σ_{CD}	0.02	0.02	0.1	0.02	0.05

trendline marked as "Poly.(Measured T_{MS})" is depicted as well.

Figure 16.3 displays the results of set 2 when increasing the input rate λ_B. It shows that the increase of λ_B has almost no effect on T_{PT}, which is fairly stable. However, when λ_B increases, T_{MS} increases as well.

The results of the experimental set 3 are depicted in Figure 16.4. At small window constraint sizes, the moving state strategy migrates faster because the state sizes limited by the window constraint are small. As the window size increases, the parallel track migration time increases linearly, and stays at about 2W. Since the total migration time for the moving state strategy has a polynomial relationship with the window size, the gap between the two lengths of migration stages is getting smaller. After a certain window size, the parallel track strategy surpasses the moving state and becomes the faster one of these two strategies.

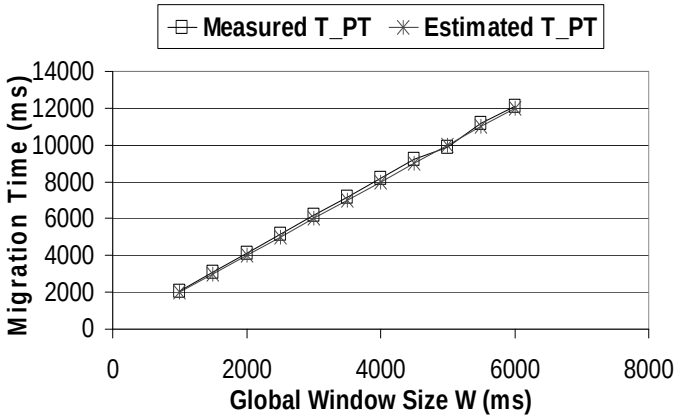


Figure 16.1: T_{PT} vs. W

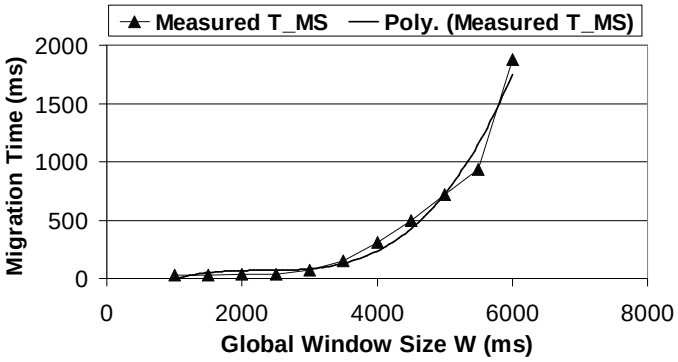


Figure 16.2: T_{MS} vs. W

16.3 Effects on Minimizing Intermediate Data

A common goal for a query optimizer is to minimize a query plan's intermediate data. This is usually achieved by pushing the operators with the smallest reduction factors down the query plan tree. In this section, I study the performance of both migration strategies working with an optimizer that has such an optimization goal.

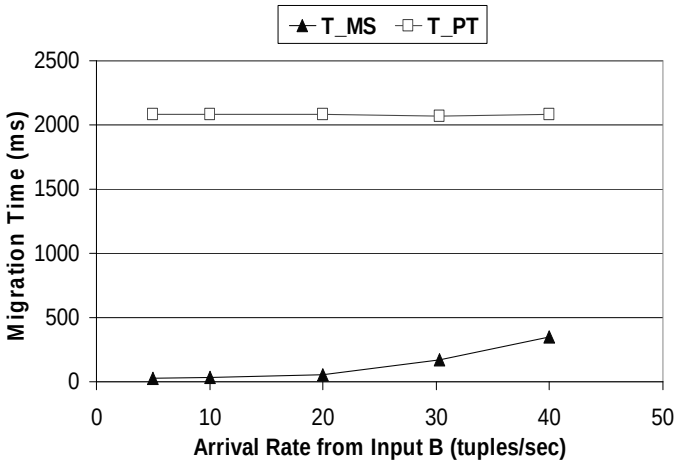


Figure 16.3: T_{MS} and T_{PT} vs. λ_B

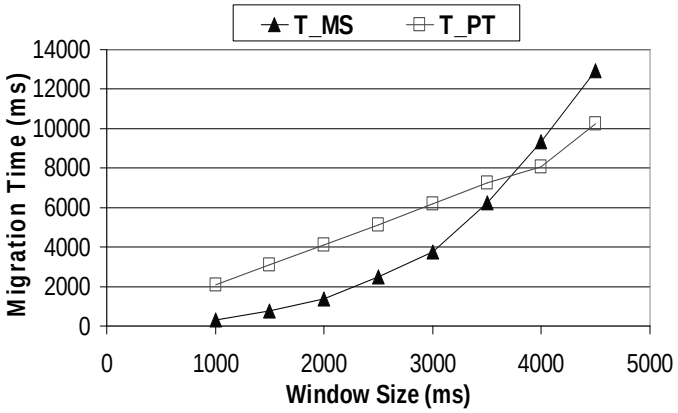


Figure 16.4: Comparison of T_{MS} and T_{PT} vs. W

I have conducted two sets of experiments with the parameters' configurations shown in Table 16.1. Parameters in set 1 are set to be low to create the situation of sufficient system computing resources, while set 2 configures parameters to their high values to model the scenario that the system computational power is not sufficient to process the old query plan. Hence,

a large delay for processing new tuples is expected for the second set. In all the experiments, I start migrating the old plan to the new plan after the old plan has been executed for 10,000ms.

The results of the first experimental set with a low configuration are shown in Figures 16.5 and 16.6. Each graph depicts the results for four different cases: 1) the moving state strategy (MS), 2) the parallel track strategy (PT), 3) the new query plan only (New), and 4) the old query plan only (Old).

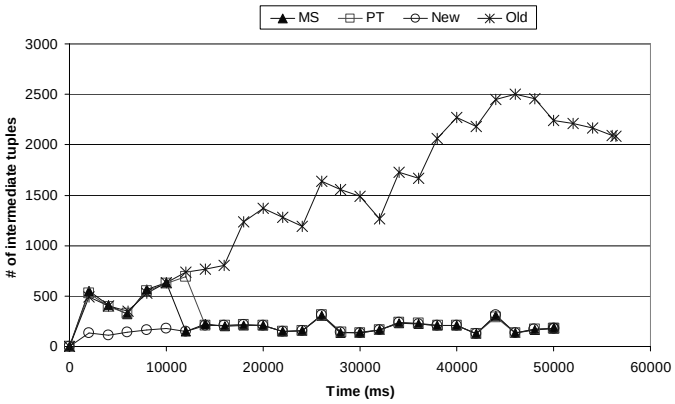


Figure 16.5: Intermediate Tuple Counts - Low Config

Figure 16.5 shows the intermediate tuple counts for the above four cases. The new plan has a much smaller intermediate tuple count than the old plan throughout the experiment. At the first 10,000ms, the three lines overlap each other indicating that they have the same performance. However, starting from around 10,000ms, two plans are migrating to the new plan each using one of the migration strategies. When given sufficient system processing power, which usually indicates a small window size, the mov-

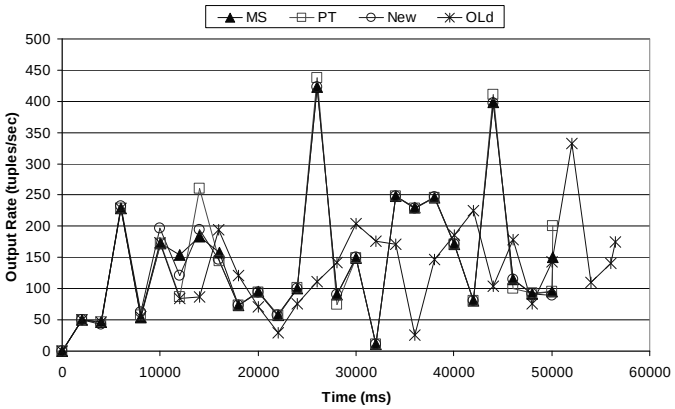


Figure 16.6: Output Rate - Low Config

ing state strategy starts to have the same intermediate tuple count as the new plan case (earlier than the parallel strategy). This is because it usually migrates to the new plan faster given a smaller window size. Both plans going through two different migration strategies eventually have the same intermediate tuple count as the one running the new plan only.

Figure 16.6 depicts the four cases with respect to their output rates. No strong advantage can be observed for either migration strategy. This may be due to the fact that the migration stage under a low configuration is usually short. So even the parallel track strategy keeps on producing new tuples during the migration stage, the plan using the moving state strategy is able to migrate to the new plan faster so the output silence is short enough to be neglected.

The situation changes for the second experimental set with a high configuration. Figures 16.7 and 16.8 show the results of all four cases. Since the system has insufficient processing power to keep up with the old query

plan, the new query plan as well as the query plan with migration both out-perform the old query plan dramatically in all experimental results.

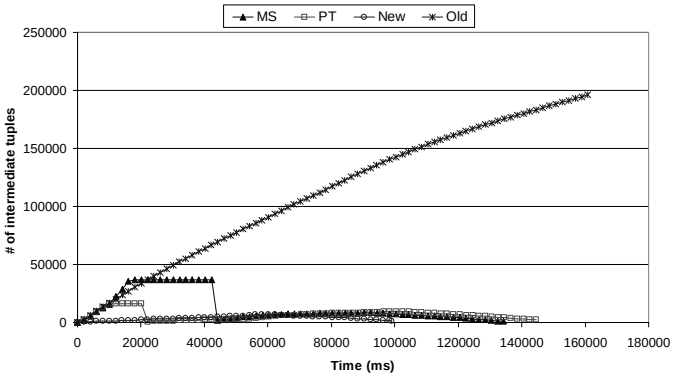


Figure 16.7: Intermediate Tuple Counts - High Config

The parallel strategy has a smaller intermediate tuple count as shown in Figure 16.7 and a higher output rate at the initial stage of migration in Figure 16.8. This is because the state sizes are much larger and thus the migration time is much longer than what we have seen in the case of a low configuration. During the state recomputing of the moving state strategy, tuples in all states cannot be disposed until the migration is over. There is a noticeable output silence between 10,000ms and 20,000ms in Figure 16.8 for the moving state strategy.

On the other hand, the parallel track strategy starts executing both the old and the new plans immediately, so intermediate tuples are being consumed (purged), and some output tuples are being generated while the migration is ongoing. Figures 16.7 and 16.8 show that although the total time for the migration stage is still smaller in the case of the moving state strategy, as it ends the overall execution earlier, during the migration stage,

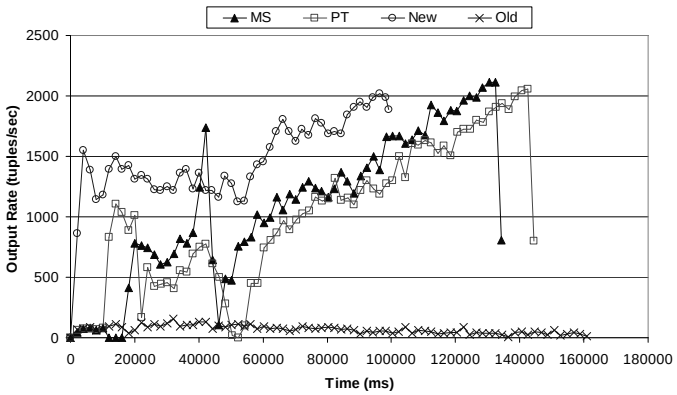


Figure 16.8: Output Rate - High Config

the parallel strategy has a better output rate and a smaller intermediate tuple count.

16.4 Apply Migration at Run Time

I have conducted experiments to measure the performance improvement of queries by applying query migration at runtime. During each experiment, data statistics, such as operator selectivities, are being collected at runtime. The runtime optimizer is invoked periodically to evaluate the current statistics and optimize the running query plan if necessary. The migration process is activated when a new query plan is generated by the runtime optimizer. I use the SPJ query as depicted on the right of Figure 13.5 in these experiments. In this set of experiments, the optimizer applies heuristic-based optimization to decide if the joins need to be switched according to their measured selectivities.

In these experiments, the data characteristics are changed during query

execution so that the query optimization and migration need to be applied at runtime. I here present the experimental results of three representative runtime migration scenarios. They vary on how often the join selectivities are changed at runtime, which then determines how often the migration is invoked to migrate a query plan. The join selectivity in our experiment is estimated as $\frac{\#\ of\ output\ tuples}{\#\ of\ possible\ output\ tuples}$ and is in the range of [0, 1]. The arrival rates of all streams are set to be 20 tuples/sec. I apply time-based sliding window constraint with the window set to 5,000 milliseconds (ms). In each experiment, the optimizer is invoked every 10,000ms. The migration is invoked if a new query plan is generated by the optimizer. I compare the performance of the query when runtime optimization and migration are activated to the performance of the same query but without applying the runtime query optimization and migration.

In the first set of experiments, the join selectivities are kept stable and therefore no migration process needs to be nor will be applied. I hence show the overhead of the optimizer to periodically analyze the collected statistics. As shown in Figure 16.9, the query throughput when the optimizer is disabled and the query throughput when the optimizer is enabled are almost identical. Our experiments confirm that the overhead of constantly invoking the optimizer to analyze data statistics is minimal and does not have much impact on the query performance.

In the second set of experiments, the join selectivities are changed only once during the query execution. Therefore the migration is invoked once at runtime. The performance comparisons on query throughput and memory consumption are shown in Figures 16.10 and 16.11, respectively. The

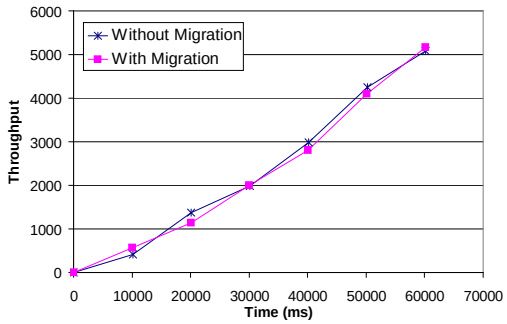


Figure 16.9: Runtime Overhead

starting selectivities for the upper join and the lower join are set to be 0.005 and 0.02 respectively. Shortly after the query starts, the join selectivities of the two joins are switched. When the optimizer is invoked at 10,000ms, it switches the two joins and invokes the query migration process. As shown in Figure 16.10, the query execution with migration has a approximately 40% to 50% better throughput than the same query but without migration after 10,000ms. This continues to be true for the rest of the query execution. Figure 16.11 compares the two query executions in terms of memory consumption. We can see that the memory consumption of the query with migration is decreased after the migration and continue to consume at least 50% less memory than the query without migration.

In the last set of experiments, I switch the selectivities of the two join operators randomly during the query execution. Join operator 1 starts with a selectivity of 0.02 and join operator 2 with selectivity 0.3. After a while the selectivity of join operator 1 become 0.3 and the selectivity of join operator 2 become 0.02. This switch is repeated several times during the query execution. Therefore, it may be necessary to invoke the migration multi-

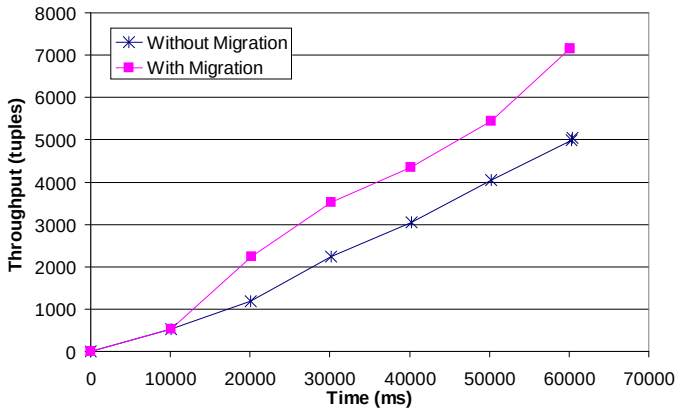


Figure 16.10: Migrate once at runtime - Throughput Comparison

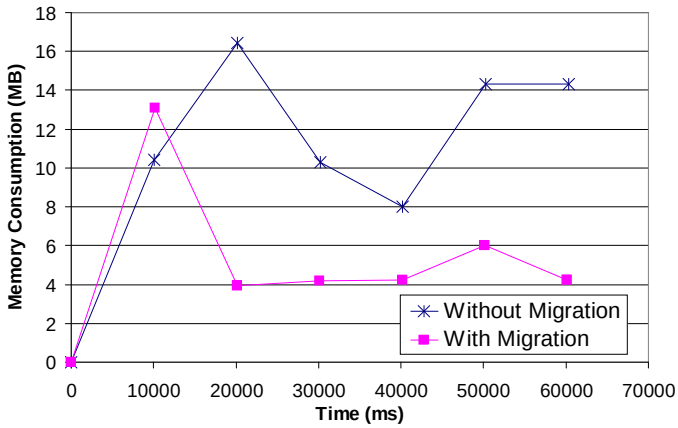


Figure 16.11: Migrate once at runtime - Memory comparison

ple times during the query execution. As shown in Figure 16.12, the query throughput of the query with migration is consistently higher than that of the query without migration. Due to the higher configuration (higher selectivities) than second experiment set, more tuples are accumulated in operator states and queues at the migration start time. Therefore the over-

head of migration increases accordingly and the migration process takes longer. In this experiment, the migration process is recorded to be invoked for four times, which are at approximately 10,000ms, 30,000ms, 70,000ms, and 130,000ms. The memory consumed by the query plan with migration is also shown to be consistently less than the memory consumed by the query without migration.

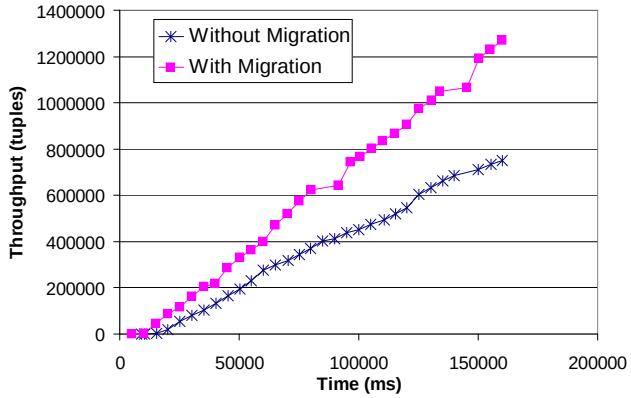


Figure 16.12: Migrate multiple times at runtime - Throughput comparison

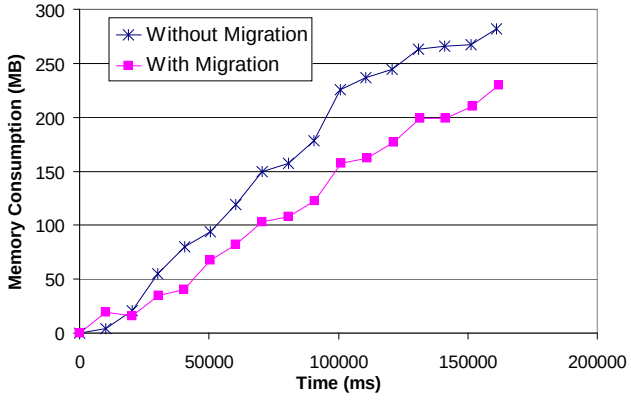


Figure 16.13: Migrate multiple times at runtime - Memory comparison

Chapter 17

Related Work

Although there is a renewed and more pressing need for dynamic query plan optimization and migration for continuous queries in streaming environments, on-the-fly query plan re-optimization has initially been also explored for static databases [KD98, Ant96, GC94, INSS92].

[KD98] utilize a run-time statistics collector and reconfigure only the *unprocessed* portion of the running query plan to improve performance. This solution is not practical for stream processing, because all operators in a long running query plan may have been executing by the time the migration is needed. The dynamic optimization for static databases proposed in [Ant96] only applies to *scan* operators and thus is limited in its usage.

[GC94, INSS92] describe a query plan competing model to dynamically change the running query plan to another plan. The approach requires that before the query starts, several plans have been chosen and will be executed in parallel. After a while the plan that has the best performance thus far will then be running alone with all other plans being discarded. Although this

approach shares some ideas with our parallel track migration strategy, it is technically difficult or almost impossible to come up with the candidates for query plans before continuous queries start running. Furthermore, this dynamic plan migration or re-configuration can be applied only once, and is thus too limiting for a long running query.

The research in [CCC+02] proposes to utilize the pause-drain-resume strategy for dynamic plan migration. I now put forth that this strategy has not explicitly addressed how to handle the case of query plans containing stateful operators such as window joins with intermediate states. [NWMS98] targets the dynamic plan migration in the context of long-running queries in a distributed database system. The proposed migration strategy cannot be undertaken whenever an optimizer has selected a new query plan, but rather it needs to wait until all involved operators have entered their own suspendable point. This extra wait is undesirable in a volatile streaming environment since the new plan may become sub-optimal again before the migration can even start.

Several dynamic query re-optimization techniques by changing the structure of the query plan have been proposed in [VN02a, CDN02]. Most such optimization strategies alter the order of operators inside the query plan to achieve a better performance. These works do not address how to migrate from one plan to another plan at run time, once the optimizer has picked a better plan for the system. This however is the exact problem I am addressing in this part of the dissertation.

A phased query execution model is proposed in [Ive02], which can be applied to query reoptimization and migration of long running static

queries. At the beginning of each phase, the query processor can use the knowledge gained in prior phases to choose a better evaluation plan for the query. The system then performs a phase transition by pausing execution of the old plan in a stable state and starting execution of the new plan over the portions of the data sources that have not yet been consumed. The system needs to maintain all previously computed results (including intermediate results) from all earlier phases to be able to use them later. Finally when all input data is processed, a special cleanup phase is invoked to recover the missing query results by joining all combinations of subresults across all phases. The phased execution model allows arbitrary changes to the executing query plan in mid-stream. However, the missing results by joining tuples across phases can only be obtained at the very end of the query execution. This is not acceptable for continuous queries that may run for an arbitrary long time, may need real-time query output, and may pose requirements on output tuple orders. Furthermore, the migration process in such a phased query execution is actually not complete until the end of the execution, so it is not truly a run time migration strategy and thus cannot be applied repeatedly during a continuous query processing.

[MSHR02] introduces adaptive query plan execution by routing tuples among operators inside a query plan. This novel adaption method is different from the generally adopted query plan re-optimization method, in which tuples follow the same assumed optimal processing path until the structure of the plan is re-optimized. Eddy's always-adapting solution is suitable for a highly dynamic environment. However, the flexibility of Eddy comes at the price of a per-tuple based overhead since extra infor-

mation needs to be carried or computed to make routing decisions for each individual tuple (or at best for subgroups of individual tuples). Furthermore, the Eddy approach has the inherent problem of having to recompute all delta intermediate results in the case of multiple joins. This can cost large amounts of processing time given high stream rates and join selectivities. For a changing environment that is not highly dynamic, the re-optimization and migration method may exhibit a better performance.

Existing research has also shown how to migrate parts of a query plan to other processors (machines) according to current system statistics [NWMN99]. In this case the structure of the query plan itself remains unchanged. This is a different problem from the plan migration problem discussed in this dissertation. Our plan migration targets the situation that the structure of the query plan has changed, yet the execution of the query plan remains on the same processor.

Part III

Distributed Continuous Query Adaptations

Chapter 18

Distributed Continuous Query Processing

Continuous queries have become popular in recent years due to demands of numerous applications, including online auctions, financial analysis, sensor monitoring systems, etc [CCC+02, BBD+02a, NWAea02, MSHR02, CF02]. A continuous query engine takes in real-time streaming data and sends out results in a continuous fashion. High stream input rates and cost-intensive query operations may cause a continuous query system to run out of resources. Distributed continuous query processing over a shared nothing architecture, i.e., a cluster of machines, is a prevalent method for solving this scalability problem [AAB+05, CBB+03, MJSM03, LZJ+05, DH04].

18.1 Distributed Query Adaptation

Distributing the query workload across multiple machines can greatly improve the system performance due to the availability of aggregated resources, including both CPU and memory. However, uneven workload among machines may occur over time due to (1) the lack of initial cost information at the time when first distributing the queries, and (2) the potentially fluctuating nature of the incoming data streams even if the statistics could be measured at runtime. This imbalance of workloads on different machines may impair the benefits of distributed processing. Thus, *dynamic load balancing*, which deals with the problem of re-distributing workload across machines in the cluster, has emerged as a crucial technology for distributed continuous query systems [MJSM03, AAB+05, LZJ+05, DH04].

In existing distributed continuous query systems such as [AAB+05, CBB+03, LZJ+05], the basic unit of moving workload during the adaptation is typically one whole operator, assuming that each operator is small enough to fit on one machine. This adaptation is referred to as *operator-level adaptation*. Such adaptation relies on the assumption that the contents of each operator (including operator states for stateful operators) are small enough to fit on a single machine in the cluster. Therefore, while such adaptation is sufficient for query plans containing only stateless operators or stateful operators with small states, it may not work as well for stateful operators with large states. In case of high stream arriving rates or large processing windows, the size of the states in an operator can be huge [MJSM03]. For such cases, applying operator-level adaptation can be difficult and inefficient.

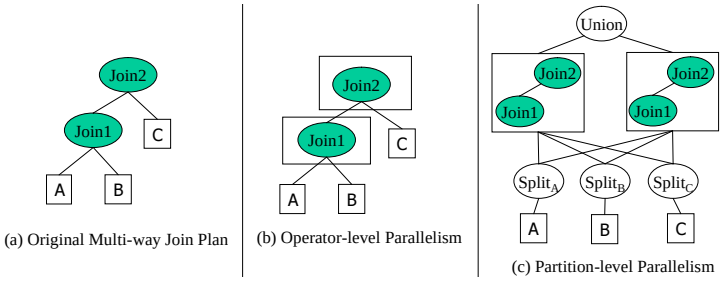
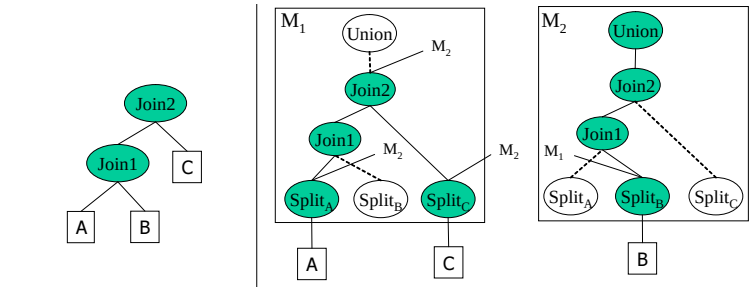


Figure 18.1: Operator-level and Partition-level Parallelism

18.2 Advantages of Partitioned Query Processing

Partitioned parallelism [Has95] is a common method for processing query operators with large states in a distributed system. Instances of each operator will be installed on multiple processors, with the input data being partitioned among these operator instances. The outputs from all operator instances are unioned to form the final output stream. Partitioned parallelism, a general query plan distribution strategy, has been routinely applied for traditional query processing [Has95, Gra90]. It has also been applied for continuous queries [MJSM03].

For example, the continuous query plan with two joins in Figure 18.2(a) can be assigned to two machines as in Figure 18.2(b). Each machine runs instances of both join operators. To partition the data, we add three *split operators* and a *union operator* to the query plan. The *split operators* operate as routers: They apply partition mapping functions, such as a value-based mapping, to divide the streams of input tuples into partitions and direct these partitions to the corresponding machines. The darker shading indicates that the operator is active on that machine.



(a) Original Multi-way Join Plan (b) Distribution of the Partitioned Plan.

Figure 18.2: Distribution of Partitioned Plan

As compared to the operator-level adaptation, the partitioned parallelism makes runtime adaptation more efficient: instead of having to move the operator as a whole unit as in [AAB+05, CBB+03, DH04], we now have the choice of moving only some partitions of an operator state to another machine at runtime. This enables finer-grained runtime adaptation.

This part of the dissertation first discusses my research findings on operator-level plan migration in Chapter 19 as extension to the centralized migration strategies proposed in Part II of this dissertation. The main focus of my research in this part is therefore the *partitioned continuous query processing*, which is described in detail in Chapters 20, 21, 22 and 23.

18.3 Limitations of Existing Strategies

The load balancing strategies currently proposed in the literature for partitioned continuous queries make the implicit assumption that the partitioned query plans on different machines remain identical [MJSM03, LZJ+05]. They have not considered the situation that the query optimizer restruc-

tures the shape of the query plan residing on its machine. Therefore, existing work on partitioned continuous query processing has not considered integrating the load balancing with query optimization. Consequently, the effects of query optimization and its impact on load rebalancing strategies remain an open issue to date.

However, this clearly is a major limitation, as runtime query optimization has been shown to be critical for streaming systems. Load balancing strategies typically just move workload from one machine to another, while the total resource consumption in the system as a whole is not decreased. On the other hand, plan optimization may be able to decrease the resource consumption on each machine, therefore decreasing the overall resource consumption in the distributed system. For example, a plan optimization may dynamically switch two join operators in a plan in the face of changing statistics. This can reduce the intermediate results, which leads to less CPU and memory costs on this machine as well as less overall resources required to process this query in the distributed system. Plan optimization may also reduce the number of times load rebalancing is needed during query execution. Without local plan migration, changes in data characteristics may result in imbalance among machine loads, possibly causing some machines to be over-loaded. Typically this may trigger load rebalancing. However, by applying local optimization on over-loaded machines, it can decrease the load on these machines and thus may prevent unnecessary load rebalancing.

Therefore, it is necessary to develop solutions that can seamlessly integrate query optimization and load balancing within one runtime adapta-

tion system. This is an open problem that is yet to be studied in the literature. It is now the focus of the research in this part of the dissertation.

18.4 New Research Problems

Traditionally, load rebalancing algorithms assume that the shape of the query plan stays the same throughout the query execution. Therefore balancing loads among machines can be simply achieved by moving some load (partitions) from over-loaded machines to under-loaded machines. However, this is no longer valid if local query optimization has been applied. Since each machine can apply its own local optimization separately from other machines, at any given time, the shapes of the query plan on different machines can be distinct from one another.

To illustrate the problem, I use the query example from Figure 18.2. As depicted in Figure 18.3, each join operator instance has two states, with each state containing several partitions (without loss of generality, here I assume value-based partitioning is applied in the split operator). Each partitioned state contains a set of partitions with different partition IDs. In Figure 18.3, we can see that machine M1 processes partitions with IDs 1, 2, 3, and machine M2 processes partitions with IDs 4, 5.

After machine M2 applies the local plan optimization, the two joins on M2 are switched. Now the query shapes on the two machines are distinct from each other. A new partitioned state P_{BC} has been created on M2 as the result of this plan optimization. If at this time the load balancing algorithm decided to move all partitions with ID 2 from M1 to M2, the partition

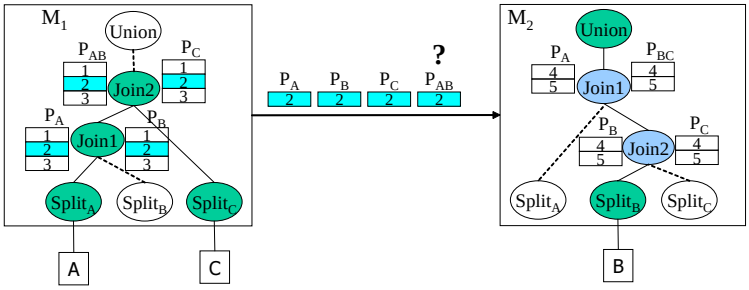


Figure 18.3: Problem with Simple Partition Moving During Load Rebalancing

2 belonging to the partitioned state P_{AB} cannot be put into any join state on M2, because it does not have any matching state on that machine. Simply discarding this unmatched partition would cause loss of data. This problem of integrating load rebalancing with query optimization remains an unaddressed problem to date. Clearly, advanced load balancing strategies are needed that can take the heterogeneity of plan shapes on different machines into account. This is now the focus of our work.

18.5 Research Outline

In this part of the dissertation, I first introduce how to extend the centralized plan migration strategies described in Part II of this dissertation to distributed systems. These operator-level plan migration strategies are discussed in Chapter 19 as extensions to the centralized migration strategies.

For the partition-level adaptation, which is the focus of this part of the dissertation, I solve the problem of integrating query optimization with the partition-level runtime load balancing for continuous queries. For this part

of the research work, I first study the effects of adding plan optimization to distributed continuous query processing. The performance gains of plan optimization versus load balancing in isolation as well as in their integrated forms are shown through experimental studies in an actual stream query system running on a compute cluster (not just a simulation). Secondly, I have designed, implemented and evaluated advanced load rebalancing strategies which take the heterogeneity of query plan shapes on difference machines into account. In particular, my research makes the following contributions:

- I propose operator-level plan migration protocols for distributed continuous query processing. These protocols extend the plan migration strategies described in Part II of this dissertation to distributed system.

- I relax the assumption of unchanged plan shapes made by state-of-art load balancing adaptation. I design two new load balancing strategies, namely PTLB and MSLB, and their corresponding protocols that can balance the workload while seamlessly handling the complexity caused by local plan changes. The PTLB strategy is a general load balancing strategy that requires no detailed knowledge of the underlying query plans, such as types of operators and shapes of query plans. I then propose the more plan-aware MSLB strategy, which rebalances the workload by comparing the detailed shapes of the query plans among different machines.

- I have implemented runtime plan optimization and the two new load

balancing strategies as the three runtime partition-level adaptation strategies in a continuous query system called D-CAPE [LZJ+05].

- I have experimentally evaluated the effects of query optimization as well as load rebalancing for partitioned continuous query processing using the D-CAPE system [LZJ+05] run on an actual cluster. The corresponding experiments show that the combination of query optimization and load balancing has superior performance than applying each adaptation technique in isolation. Between the two load balancing strategies, the MSLB is shown to be more efficient than the PTLB in many situations, while the PTLB can win under certain conditions.

18.6 Road Map

For the remainder of this part of the dissertation, I describe operator-level plan migration protocols in Chapter 19. Chapter 20 describes preliminaries on partitioned query processing, my design policies for the load balancing strategies and conditions for load balancing. The two proposed load balancing strategies and their protocols in a distributed system are described in Chapters 21 and 22, respectively. Chapter 23 shows my experimental evaluations. Chapter 24 discusses related work.

Chapter 19

Operator-Level Distributed Migration

19.1 Distributed Moving State Migration Protocols

In this chapter, I extend the two centralized migration strategies described in Chapter 12 to a distributed system. The basic ideas of the two migration strategies can be re-applied in a straightforward way. When applying a migration strategy in a distributed system, if all operators in the old migration box are active on the same machine, then it is the same as the local migration. Otherwise, coordination and communication among multiple machines are necessary.

I have developed a distributed protocol to achieve moving state migration across machines when operators in the same query are active on different machines. The key steps include the synchronization of the execution

on multiple machines, changing the shape of the query plans on each machine, filling the matched states and recomputing the unmatched state, and finally reactivating the corresponding operators on each involved machine to resume the normal execution.

The protocols for applying a cross-machine moving state strategy are depicted in Figures 19.1, 19.2, 19.3 and 19.4. Here I use an example query plan with four join operators. When the migration process starts, as indicated by the distribution table inside the distribution manager in Figure 19.1, op1 and op3 are activated on machine M1 and op2 and op4 are activated on machine M2. The goal of the migration is to switch the top two join operators, which are now active on different machines. So this is a case of cross-machine migration.

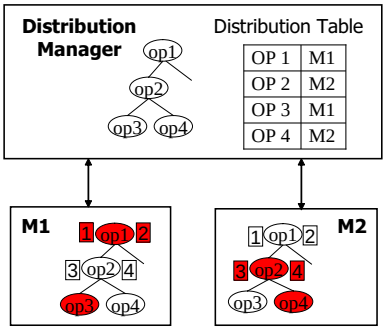


Figure 19.1: Distributed Moving State Protocol: Start of Migration

Figure 19.2 illustrates the first few steps needed to synchronize the execution among multiple machines. This is necessary to prevent any missing or duplicate results. The Distribution Manager (DM) initiates the migration process by sending a "Request SyncTime" to each involved machine. Upon

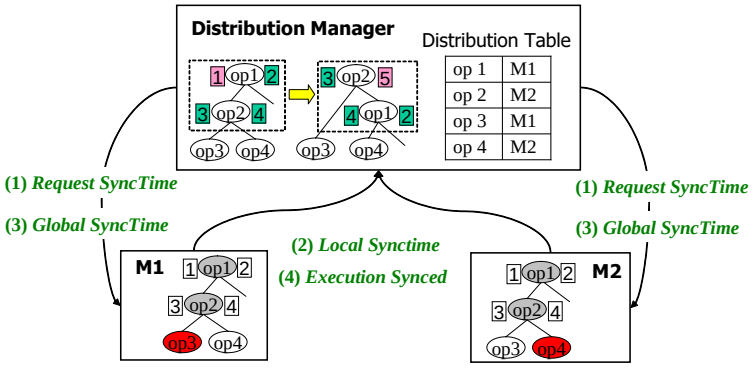


Figure 19.2: Distributed Moving State Protocol: Execution Synchronization

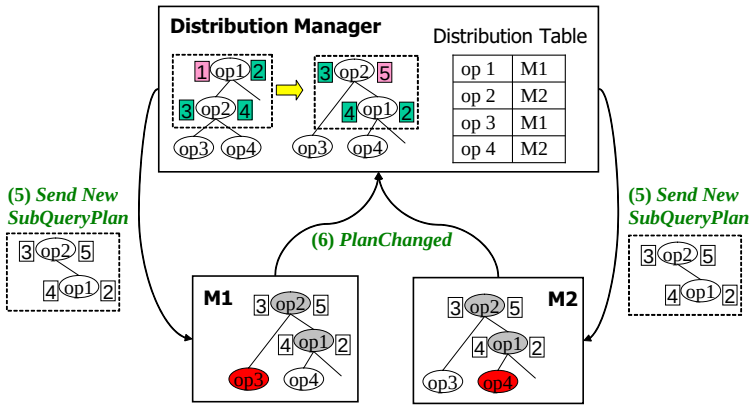


Figure 19.3: Distributed Moving State Protocol: Change Shape of Query Plan

receiving such a message, each machine collects the latest timestamp of tuples in its input queues and sends it back the DM. The DM computes the *global synctime* and sends it to each machine. Each machine then executes the query until all the tuples with a timestamp smaller than the *global synctime* are being executed. This ensures that all machines have processed tuples up to the same timestamp. When the DM receives "Execution Synced"

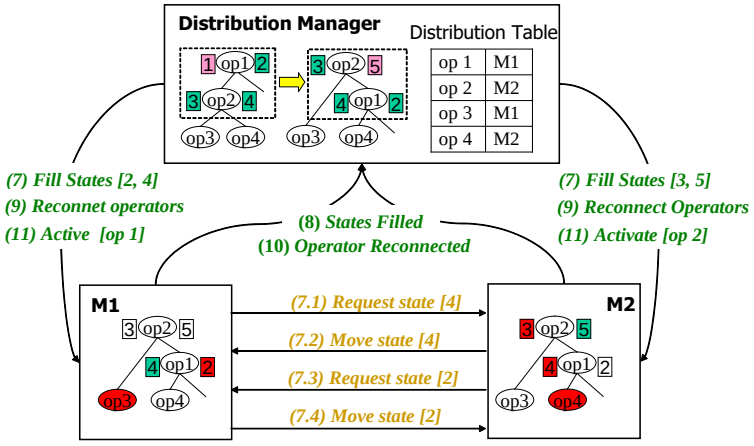


Figure 19.4: Distributed Moving State Protocol: Fill States and Reactivate Operators

message from all involved machines, the synchronization phase is finished. Note that the two operators that would be changed should be deactivated on the respective machine that they each was originally activated on.

It is now safe for the machine to change the shape of the query plan. The DM sends the new sub-plan to each machine, and waits for the acknowledgments from all machines indicating that the shape of the query plan has been changed.

Next, the DM sends the list of states in the migration box to the machines that should have these states after the migration is over. Along with the list of states it also sends the machines that the old matching states are currently on. The computation of which states are matched or unmatched on which machine is done locally inside the DM. When a machine receives the list of operator states to fill, it sends requests to machines if it does not already have those states. If the state is a new state (judging by the state

ID), the machine should recompute the state and send any request to the corresponding machines if it does not have the state to recompute this new state. A machine sends a "State Filled" message back to the DM when all the matching states are filled. After two more handshakes between DM and the participating machines, the operators deactivated at the migration start time are reactivated. After this step the migration is over and normal execution is resumed.

19.2 Distributed Parallel Track Migration Protocols

The protocols for distributed parallel track strategy are depicted in Figures 19.5, 19.6 and 19.7, assuming the same starting state as depicted in Figure 19.1. The synchronization step is unnecessary for the parallel track strategy because the old migration box and the new migration box are both going to read tuples from the same queue.

The first step in this protocol is to deactivate operators inside the migration box, as indicated in Figure 19.5. After the deactivation, the migration box containing the new sub-plan is sent from the Distribution Manager (DM) to the involved machines. Each machine then connects the corresponding box input queues to both the old migration box and the new migration box. Note that some of the connections are cross-machines. After the connecting step, the DM notifies all machines to start the execution. At this time, both migration boxes would be executed at the same time. The same duplicate prevention method, as described in Chapter 12, is applied here to avoid duplicate results being generated from both migration boxes.

Each machine monitors the tuples in the operators inside the old box. If all tuples that had existed before migration start time have been purged, the machine notifies the DM. The migration process is finalized by removing the old box from each machine and then resuming the normal execution, now with only the new sub-plan connected to the input queues.

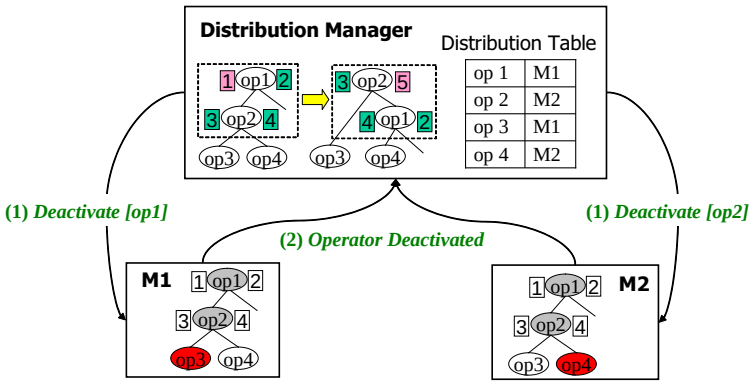


Figure 19.5: Distributed Parallel Track Protocol: Deactivate Operators in Old Box

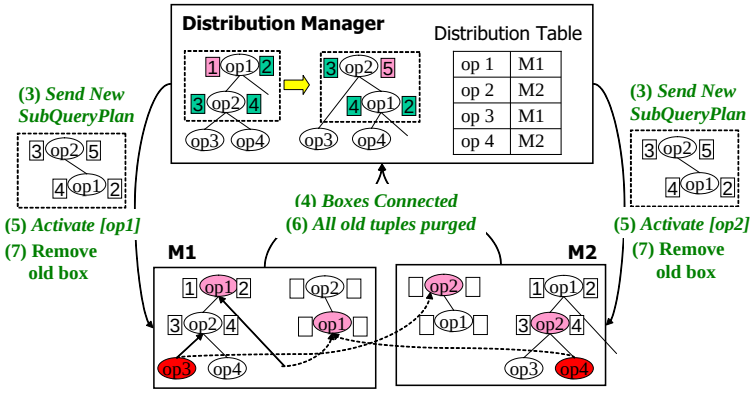


Figure 19.6: Distributed Parallel Track Protocol: Connect and Execute Old and New Boxes

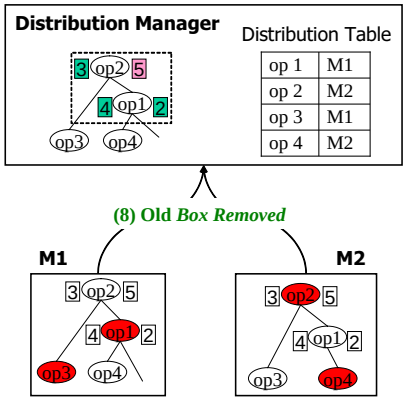


Figure 19.7: Distributed Parallel Track Protocol: Remove Old Box

19.3 Discussion on Distributed Migration Strategies

As discussed in the cost analysis in Section 12.4 of Part II and shown by the experimental results in Chapter 16, the moving state strategy requires less CPU costs than the parallel track strategy in a centralized system. However, in a distributed system, the cost of moving matching states across machines is added to the moving state strategy, while this cost is almost negligible in a centralized system where matching states can be shared right away by creating cursors. Another extra cost for distributed moving state is that when an unmatched state needs to be recomputed, if the machine does not have the states that are needed to recompute this unmatched state, the missing states have to be requested to send from some other machines. Therefore, the total extra cost of the distributed moving state can be at most the communication costs to transfer all the matching states from the old migration box to the new migration box.

For the distributed parallel track strategy, the extra cost is spent sending

the same tuples to both the old box and the new box when they are being executed parallely over the same input data. As analyzed in Chapter 12, this can be estimated as sending 2W worth's of new input tuples as well as sending those intermediate tuples across machines when the producer operator and the consumer operator are located on different machines. In a distributed system with the network connection as the bottleneck, we can compare the size of the intermediate states to be transfered in case of the moving state, to the 2W worth's of new tuples that need to be sent to both machines. Whichever of the above two is larger, the corresponding migration strategy is likely to be more costly. We should then apply the other distributed migration strategy to the global plan migration process.

Chapter 20

Preliminaries on Partition-level Adaptations

20.1 Partitioned Continuous Query Processing

For the research on partition-level adaptation, I adopt partitioned parallel processing as used in Volcano [Gra90] and Flux [MJSM03]. Input streams of operators are partitioned into many smaller partitions based on values of tuple columns. Each operator instance is allocated a number of partitions. The partitioned processing is accomplished in the split operators. We use the example shown in Figure 20.1 to illustrate the partitioning process. The example query plan has two joins and is being processed in parallel on two machines. Each machine executes the same instance of the query plan but processes a different portion (set of partitions) of the input data. The data partitioning process in the split operators will be described shortly. The

query plan has three input streams A, B and C, and the join predicates are defined as $A.A1 = B.B1 = C.C1$. Therefore three split operators are added to the query plan, shown as $Split_A$, $Split_B$ and $Split_C$ in Figure 20.1.

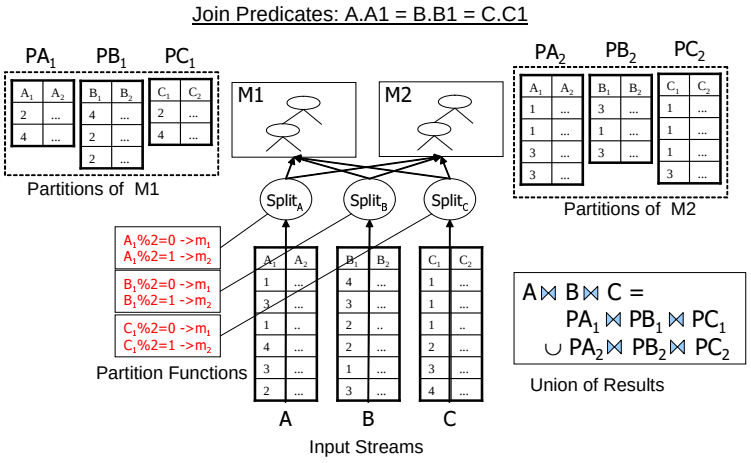


Figure 20.1: Tuple Partitioning in Split Operators

Each split operator stores a partition function and a partitionID-to-machine mapping table between partition IDs and machines. *Partition ID* is an ID assigned to a tuple as the result of applying the partition function to a tuple. Each tuple will henceforth store its ID. When a new tuple arrives, the split operator for that input stream applies the partition function to calculate the partition ID of the new tuple. It then checks its partitionID-to-machine mapping table and figures out to which machine this tuple should be sent to. For example, for input stream A, the split operator $Split_A$ has partition functions based on modular function %2 in our example (Figure 20.1). All the tuples with partition ID "0" are mapped to machine M1 while the other tuples are mapped to machine M2. As the result of this partitioning, all

tuples with an even first-column value are processed on machine M1 and temporarily stored in the corresponding states on machine M1. All the tuples with an odd first-column value are processed and stored on machine M2. The partitions in each state on M1 and M2 are also shown in Figure 20.1. The final result of this query processing is the union of the two non-overlapping sets of results from the two machines.

20.2 Design of Load Balancing Strategies

As described in Chapter 10, the problem of dynamic plan migration is summarized as the problem of replacing one *query box* by the other at runtime. The term *query box* refers to the plan or sub-plan selected for migration. As shown in Figure 20.2, the left box contains the old plan and the right box contains the new plan. The plan migration process can then be defined as the process of transferring an old box containing the old query plan to a new box containing the new query plan. In the second part of this dissertation, two migration strategies have been proposed, namely the parallel track strategy and the moving state strategy, to dynamically migrate continuous query plans that contain stateful joins.

The problem tackled in this part of the dissertation is how to dynamically balance workload among machines with heterogeneous plan shapes. At first glance, the problem of plan migration and the problem of load balancing are two very different problems: The former deals with plan changes in a local machine, while the latter moves workload among machines in a distributed environment. However, if we treat each query box,

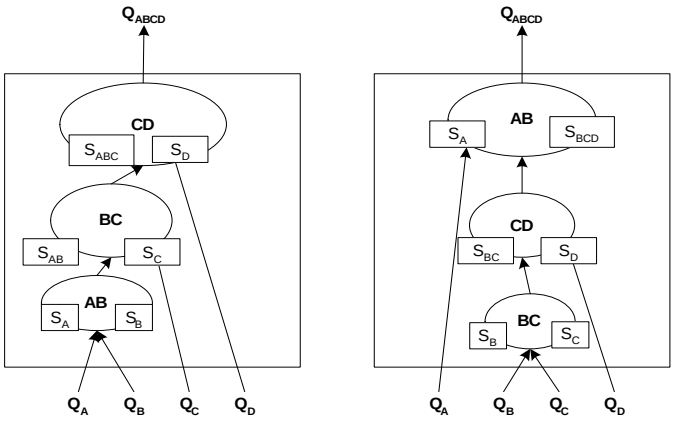


Figure 20.2: Two Exchangeable Query Boxes

shown in Figure 10.1, instead as a machine in a distributed system, the two seemingly very different problems now begin to look similar: in both cases, we are moving some workload from one plan/machine box to another plan/machine box. We observe that the challenges of both problems are similar as well: First, the two boxes contain query plans of different shapes. Second, the whole process needs to be done at runtime without resulting in any lost, duplicate or incorrect query results. This sharing of similar concepts and challenges has inspired us to reuse the ideas of the two existing plan migration strategies as much as possible to solve the problem of load balancing.

Based on these similarities between local plan migration and distributed load balancing, I have designed two strategies, namely the moving state load balancing and the parallel track load balancing, which reuse the respective core ideas of the two local plan migration strategies to achieve the distributed load balancing process with heterogenous plan shapes among

machines. However, although the problem of dynamic plan migration and the problem of runtime load balancing have similarities, they are still different problems. The former changes the currently running query plans with stateful operators on a local machine. On the other hand, the latter moves operator partitions (not the whole operator) among machines with possibly different query plan shapes. Two major differences exist between these two problems:

- First, the local plan migration is conducted on a single machine, while the distributed load balancing process involves the distribution manager, the sender machine and receiver machine. Therefore three machines participate in this process across the network. This requires carefully synchronized coordination among these participating machines to ensure the correct results of the query processing.

- The second major difference is the unit of data on which the migration processing is applied. In local plan migration, all tuples in the same state are treated as one unit. Any migration action applied to one tuple in the state will also be applied equally to all other tuples in the same state. During the moving state plan migration, the state matching, state moving and state recomputing all use the state as the basic operation unit. During the parallel track plan migration, *all* tuples sent to the old box are also sent to the new box. However, in distributed load balancing, we are interested in moving only subset of the partitions from one machine to another machine. So now the unit of operation becomes a partition instead of a state. Partitions in

the same states can be treated differently. The split operators need to distinguish the to-be-moved partitions from the other partitions. While the to-be-moved partitions are being matched, moved or recomputed in the moving state strategy, or duplicated in the parallel track strategy, the other partitions should be processed normally.

The next two chapters describe the two load balancing strategies, namely moving state load balancing and parallel track load balancing, respectively. Especially they show what modifications to the two local migration strategies are needed to resolve the differences described above between local plan migration and distributed load balancing, so that the concepts of the local migration can be reused and applied to load balancing in partitioned continuous query processing. Chapter 21 describes the parallel track load balancing strategy and the protocols designed to apply the method for the purpose of distributed load balancing. This strategy conducts the load balancing process in a gradual fashion by executing the tuples belonging to the to-be-moved partitions on both the sender machine and the receiver machine in parallel. Chapter 22 introduces the moving state load balancing strategy and the designed protocols to apply it to the distributed load balancing problem. This strategy pauses the processing of tuples that belong to the to-be-moved partitions on the sender machine and carefully maps and moves over the to-be-moved partitions in the states of the sender machine to their corresponding locations on the receiver machine.

20.3 Conditions for Load Rebalancing

I have developed the proposed load balancing strategies in the D-CAPE system [LZJ+05]. The load rebalancing process is invoked by the distribution manager (DM). Each processor run on one machine periodically sends its resource usage statistics to DM, who then analyzes these statistics and makes the decisions to apply runtime load rebalancing if it detects a load imbalance among processors. For discussion's sake, let us assume the resource statistics are memory usages. Each load balancing process is between a pair of machines: the DM selects the machine M_l with the lowest memory load C_{min} and the machine M_h with the highest memory load C_{max} and moves $(C_{max} - Cmin)/2$ worth of partitions from M_h to M_l. M_h is referred to as the *sender machine* and M_l is referred to as the *receiver machine*. The DM makes the decision based on three cost-related parameters:

- C_{max}: The load on the most-loaded machine among all machines. If this parameter is smaller than a threshold T_{max}, then no machine is over-loaded in the system. Therefore it is not necessary to balance the load even if the load is unbalanced among machines in the system.

- C_{min}: The load on the least-loaded machine among all machines. If this parameter is larger than a threshold T_{min}, that indicates that even the least loaded machine is system is over-loaded, and moving some load to this machine won't solve the problem. Therefore load rebalancing should not be applied. At this time, other adaptation strategies that can decrease the total amount of load in the sys-

tem should be applied. These strategies include local plan migration [ZRH04], load shedding [TCZ+03, BDM04] or spilling data to disk [LZR06, UF99, VNB03].

- C_{diff}: The difference of the load between the most-loaded machine and the least-loaded machine in the system. If this difference is smaller than a threshold T_{diff}, the load imbalance is not serious enough and therefore the load rebalancing should not be invoked.

Chapter 21

Parallel Partition Load Balancing

21.1 Distributed Parallel Track Load Balancing

The basic idea for the *parallel track load balancing strategy*, which is inspired by the parallel track migration strategy, is that at the load balance start time, the set of tuples belonging to the to-be-moved partitions are send to both the sender and the receiver machines. Both machines then process this same portion of data in parallel, while waiting for all *old* tuples in the to-be-moved partitions to be gradually purged. Here a tuple is *old* if it had already existed in any partition before the load balancing starts. A tuple is *new* if it arrives after the load balancing has started.

When the to-be-moved partitions on the sender machine contain only *new* tuples, it is safe to discard them from the sender. This is because all old

partitions have finished their duty in terms of contributing to the generation of output results from the sender machine. Since we have been feeding the tuples belonging to these to-be-moved partitions to the receiver machine in parallel from the moment the load balancing first started, all the new tuples belonging to these partitions now in the sender machine also exist in the receiver machine as well. So if the old partitions are discarded from the sender machine at this time, no useful data will be lost.

We also must ensure that no duplicate tuples are being generated. In the parallel track strategy described above, although the sender machine will generate all output tuples from the to-be-moved partitions that consist of at least one *old* sub-tuple, it may also generate the all-new sub-tuple combination, which duplicates the output results from the receiver machine.

To solve this duplication problem, we can apply the same duplicate prevention algorithm described in Chapter 12 for local parallel track plan migration strategy. The root join operator of the sender machine can prevent a *new* tuple from joining with another *new* tuple. Hence if the join predicate is evaluated on two tuples that are both *new*, we simply skip the join step in the regular purge-join-insert symmetric join algorithm. The purge and insert steps are however still undertaken as usual.

For this strategy, all old tuples (tuples with at least one old sub-tuple) need to be purged from the to-be-moved partitions. Suppose that h $(h >= 1)$ is the height of the query tree on the sender machine. Again as in the local parallel track plan migration described in Chapter 12, the duration of the parallel track load balancing can be estimated by two cases:

1) h = 1. In this case the query tree has only one level of join operators.

For a join operator on the sender to purge all old tuples in the to-be-moved partitions from one of its two states, the join operator must process new tuples from its second input that arrive in the next W time units. Therefore $T_{PT} = W$.

2) $h > 1$. This means that on the sender there is at least one join operator which is above another join operator. When the load balancing begins, W time window's new tuples from the input queues are needed to purge old tuples inside the to-be-moved partitions of leaf operators on the sender machine. However, as these new tuples are used to purge old tuples, they may also join with some of the old tuples and the results are being inserted into the state of the join operators above the leaf operators. Because the joined tuples contain an *old* sub-tuple, they are treated as *old* tuples and need to be purged as well. In order to do so, the sender machine needs to process another W time window's new tuples to completely purge these *old* tuples from the old partitions. So in this case, $T_{PT} = 2W$.

In summary, the lower bound of time spent on finishing the parallel track process is $2W$ for a query with more than one join operator, given that W is the window size of the query. The lower bound is W if the query contains only one join operator.

The flow chart of the parallel track load balancing strategy is shown and compared to the parallel track plan migration strategy step-by-step in Figure 21.1. As we can see, the two strategies are similar to each other, except that each step of the parallel track load balancing is applied to tuples in to-be-moved partitions only. This distinction between to-be-moved partitions and other partitions is accomplished in the split operators, because

these are where the partitions are formed and distributed to the sender machine and the receiver machine. The detailed protocols are described in the following Section 21.2.

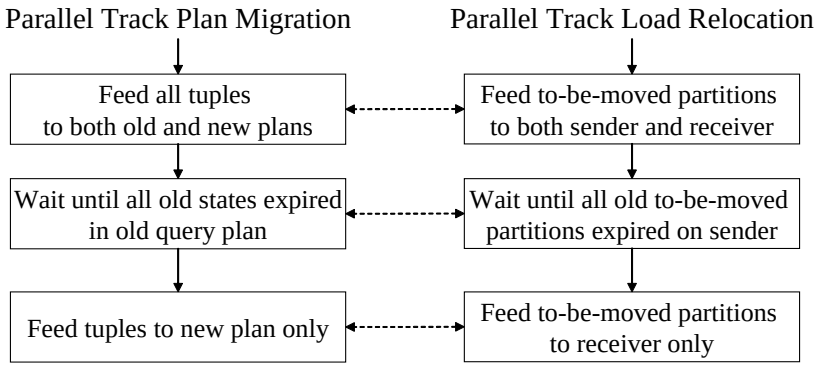


Figure 21.1: Parallel Track Strategy

21.2 Distributed PTLB Protocols

This section describes the distributed Parallel Track Load Balancing (PTLB) protocols I have designed to solve the problem of load balancing among machines with heterogeneous plans in a distributed environment. As mentioned earlier, the key point in PTLB, different from the local parallel track plan migration, is that we now need to distinguish the to-be-moved partitions in the split operators. Only the these partitions are send to both sender and receiver during the PTLB process, while other partitions should be processed as normal without being affected by the PTLB process.

A 5-step communication protocol has been designed to achieve the PTLB once the DM has made the decision to apply load rebalancing. Each step

contains a message passing between the distribution manager (DM) and one of the query processors. I divide the 5 steps into 3 phases, including computing partitions, duplicating partitions and stopping duplication, which corresponds to the three phases in the local parallel track plan migration process. The query example in Figure 18.3 is used here to illustrate the execution of the protocols.

The three phases in the protocols are illustrated in Figures 21.2, 21.3 and 21.4.

PTLB Phase 1: Computing Partitions

During the first phase of the protocol, the to-be-moved partitions that need to be moved from the sender to the receiver are determined by communication between the distribution manager and the sender machine. This phase consists of steps 1 and 2, which together with the corresponding timeline of the PTLB process are depicted in Figure 21.2. When the DM makes the decision to invoke load balancing, it has already calculated three variables used in the load balancing process: the sender machine, the receiver machine and the amount of partitions in terms of memory the sender should send to the receiver. Therefore, in the first step of load balancing, the DM sends a request *computePartitionsToMove* to the sender (assumed to be M1 in Figure 21.2), with the amount of partitions that need to be moved. Upon receiving such a request, the sender machine selects the partitions whose sum is close to the amount of partitions that need to be moved. In step 2, the sender then sends the IDs of the selected partitions, denoted as *partitionsToMove* in Figure 21.2, back to the DM.

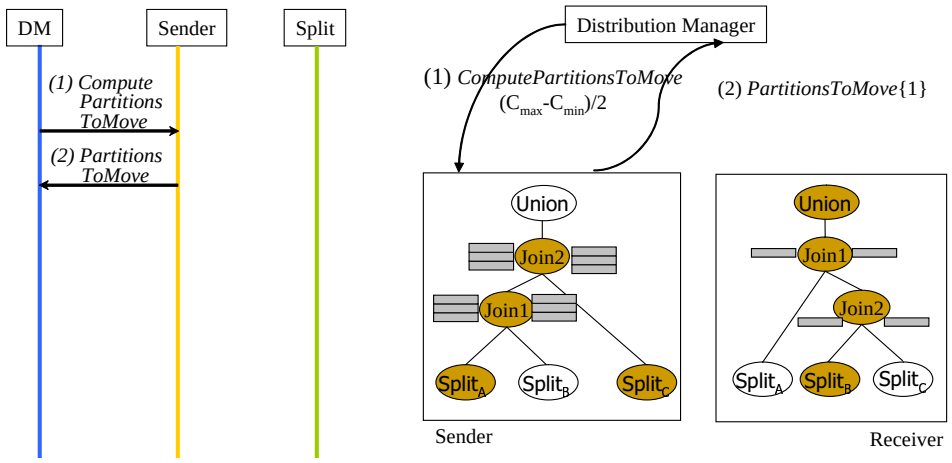


Figure 21.2: PTLB: Compute Partitions to Move.

Each partition ID represents all the partitions on the sender machine with that same partition ID. In fact, each partitioned state on a machine can have a partition with the selected ID. Therefore each partition ID indicates one partition from each state. Our mechanism is to choose the partitions in all the states with the same partition ID as a whole unit to move. This avoids joins across multiple processors. For example, as shown in Figure 18.3, we denote the partition with ID 2 in partitioned state P_A as partition A_2. If we only move partition A_2 from M_1 to M_2, then after the load balancing, the newly coming tuples to partition A_2, which is now located on M_2 would have to probe and join partition B_2, which is still located on M_1. Therefore in our load balancing process, the unit to move between two machines is not a single partition, but a *partition group* that contains all the partitions with the same ID on the sender machine.

PTLB Phase 2: Duplicating Partitions

During the second phase of the PTLB process, the to-be-moved partitions are sent to both the sender and the receiver machines until all old tuples in these partitions are purged on the sender machine. In order to achieve this goal, the split operators should distinguish the to-be-moved partitions (as calculated in the first phase of PTLB) from other partitions by modifying their partition mapping tables: the to-be-moved partitions are mapped to both sender and receiver, while mappings for other partitions remain the same. Similar to the local plan migration, each operator on the sender machine checks to see if all old tuples belonging to the to-be-moved partitions are purged from its states. And if so, the operator sends a message to its parent. This message is propagated bottom up until the top most operator on the sender has received the message from all its children and has purged all old tuples from its to-be-moved partitions.

This second phase contains steps 3 and 4 that exploit parallel processing principles, as shown in Figure 21.3. In Step 3, the DM sends a *DuplicatePartitions* to the sender machine as well as all the machines with active split operators. Upon receiving the message, an active split operator will add entries to the existing partition mapping table, which map each of the to-be-moved partitions to the receiver machine. This allows the split operator to henceforth forward tuples that belong to these selected partitions to both the sender and the receiver machines.

Before actually sending the tuples to both machines, each split operator first sends a meta-data tuple, referred to as *CheckOld* tuple, to its parent

operator that is active on the sender machine. This is shown as the step 3.1 in Figure 21.3. When each operator on the sender machine receives such meta-data tuple from all its children, this operator then enters a checking-old-expired mode. When in this mode, an operator periodically checks to see if all the old tuples in the to-be-moved partitions in its states are purged. The operator also immediately propagates the meta-data tuple to its parent.

After sending the *CheckOld* meta-data tuple, the split operator then immediately starts sending tuples belonging to to-be-moved partitions to both sender and receiver. Whenever a tuple is forwarded to a sender machine, the split operator sets a flag on the tuple as *new*. This indicates that this tuple is also being sent to the receiver machine. The flag on all other tuples, including the tuples being sent to the receiver machine in parallel, are by default set to be *old*.

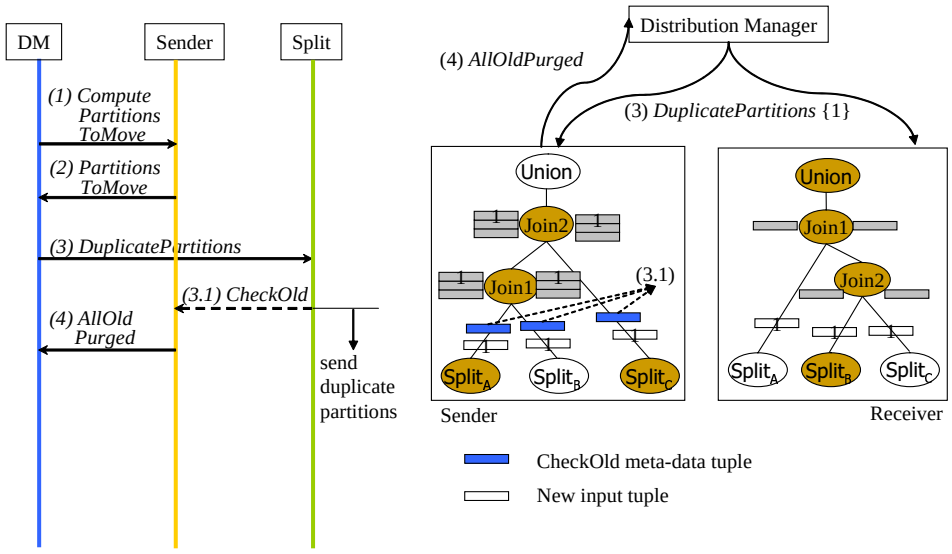


Figure 21.3: PTLB: Send Partitions to Both Machines.

Upon receiving the *DuplicationPartitions*, the join operators on the sender machine process tuples in the following way in order to avoid producing duplicate results from the sender and the receiver.

- For all join operators except the root join operator on the sender machine, a *new* tuple is being treated the same as an *old* tuple. When a joined tuple is outputted from a join operator, the joined tuple is set to be new *only when* all its sub-tuples are *new* as well. Otherwise, the tuple is still set to be *old*.

- At the root join operator, when two tuples are to be joined, if both tuples are marked as new, they are *not* joined together. Instead, the tuples are just used to purge partitions and are then inserted to the corresponding partitions. This is because the new-to-new joins are to be done on the receiver machine.

When all old tuples in the to-be-moved partitions are purged from an operator, it then sends another meta-data tuple, referred to as the *AllOldPurged* tuple, to its parent. This meta-data tuple again is propagated in the same fashion as the *CheckOld* tuple. The same bottom-up propagation of *AllOldPurged* tuple ensures that when the top-most operator emits such a tuple, it indicates that all old tuples have been purged from the sender machine. At this time, the sender machine can send an *AllOldPurged* message back to the distributed manager. This is shown as the Step 4 in Figure 21.3.

PTLB Phase 3: Stopping Duplicates

In the last phase of the PTLB protocol, each split operator modifies its mapping table again so that now the tuples in to-be-moved partitions are mapped to the receiver machine only. This stops the duplicates of the tuples and completes the PTLB process. This phase is accomplished by Step 5 as depicted in Figure 21.4. The DM sends a *DeletePartitions* message to the sender machine and all machines with active split operators. Each active split operator will then remove the entries that map the to-be-moved partition IDs to the sender machine. This allows the split operator to forward new tuples belonging to these partitions to the receiver machine only. The split operator then puts an *DeletePartitions* meta-data tuple to all the output queues connecting to the sender machine. This is shown as the Step 5.1 in Figure 21.4. When a join operator has received the *DeletePartitions* tuple from all its input queues, it can delete the to-be-moved partitions from its states. The join operator forwards the *DeletePartitions* tuple to its parent. When the root join operator on the sender has received one *DeletePartitions* tuple from each of its input queues, it sends a *PartitionDeleted* message back to the DM. After this step, the PTLB process is over.

PTLB Algorithms

Algorithms 9 and 10 sketch the high-level interactions between the distribution manager and the processors on each machine during the runtime PT load balancing process. Algorithm 9 describes the basic operations of the distribution manager. The algorithm follows the actions in the timeline

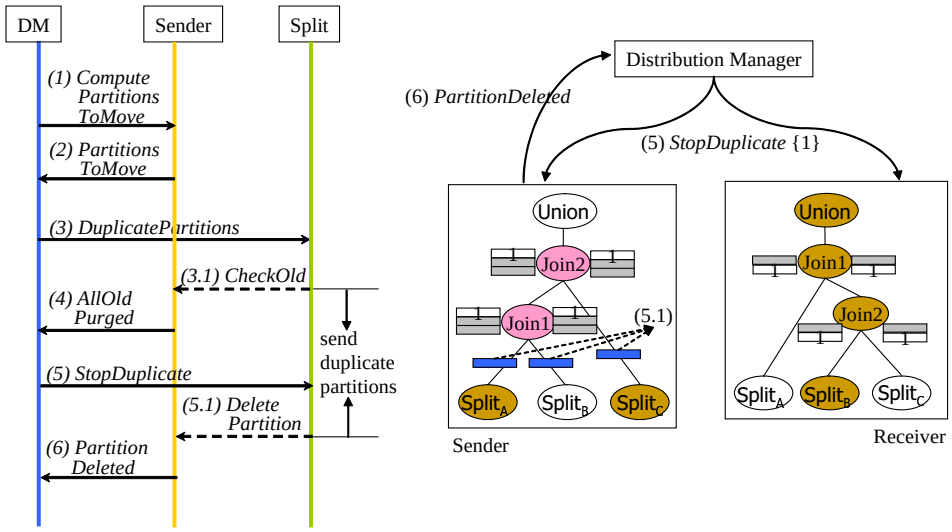


Figure 21.4: PT Load Balance: Delete Partitions.

diagram (shown on the left of Figure 21.4) by sending protocol messages and waiting for the corresponding responses. Similarly, Algorithm 10 describes the steps performed on a participating processor during the PTLB process. Here, the *send* and *wait* are primitive operators designed to send or wait for messages across machines.

In summary, the PTLB is a general strategy that does not need to care about the detailed properties about the plan itself, such as the types of the operators and the shapes of the plans on different machines. This simplifies the process of load balancing, especially when the plan shapes are different between the sender and the receiver. It also has the advantage of not having to stop the query execution in the to-be-moved partitions at any point of time. It thus does not have to deal with on-the-fly tuples. However, this simplicity comes with a price of both CPU and memory overhead, which

Algorithm 9 PT-State-Rebalance: DistributionManager(sender, receiver, amount)

*/*It controls load balancing process by sending control messages to participating machines and waiting for corresponding responses.*/*

1: **send** *ComputePartitionsToMove*(amount) msg to *sender*;
2: **wait** until get *PartitionsToMove* msg;
3: **send** *DuplicatePartitions* to sender & machines with active split operator(s);
4: **wait** until get *AllOldPurged* msg from the sender machine;
5: **send** *StopDuplicates* msg to sender & machines with active split operator(s);
6: **wait** until get *PartitionDeleted* msg from sender machine;

will prevail as long as the balancing process is ongoing. As discussed earlier in Section 21.1, the whole process takes at least 2W timeframe to finish. This is undesirable for continuous queries with large windows, which are in fact the ones that most likely need to be executed in a distributed system in the first place. It also incurs the extra overhead of having to store the same set of tuples for these to-be-moved partitions on both the sender machine and the receiver machine. To overcome these shortcomings, I design the second runtime load balancing protocol, the moving state protocol, which is described in the next section.

Algorithm 10 PT-State-Rebalance:Processor()

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

```
1: while (keepGoing) do
2:   wait for control message of PTLB protocol;
3:   switch(message)
4:   ComputePartitionsToMove: /*compute partitions to be moved*/
5:     compute partitions to move;
6:     send PartitionsToMove msg to DM;
7:   DuplicatePartitions: /*sends new tuples to both machines*/
8:     split operators sends CheckOld tuple to operators on sender;
9:     split operators start sending new tuples in to-be-moved partitions
       to both machines;
10:    root join operator waits for all old tuples to be purged and AllOld-
       Purged tuples from all its children except split operators;
11:    root join operator sends AllOldPurged msg to DM;
12:  DeletePartitions: /*delete to-be-moved partitions*/
13:    split operators send DeletePartition tuple to operators on sender;
14:    split operators stop sending tuples in to-be-moved partitions to the
       sender;
15:    join operators on sender remove to-be-moved partitions when re-
       ceive DeletePartition tuples from all its children except split operators;
16:    sender machine sends PartitionDeleted message back to DM;
17:    keepGoing = false;
18: end while
```

Chapter 22

Moving Partition Load Balancing

22.1 Distributed Moving State Load Balancing

The basic idea of the *Moving State Load Balancing* (MSLB), which is inspired by the moving state plan migration strategy described in Chapter 12, is to safely move to-be-moved partitions (instead of whole states in plan migration) on the sender machine directly into the states on the receiver machine without losing or duplicating data. The main steps of MSLB and its corresponding steps in the MS plan migration strategy are shown in Figure 22.1.

As we can see from Figure 22.1, MSLB and MS migration have very similar steps in the flow charts. However, there are two major differences between the two processes.

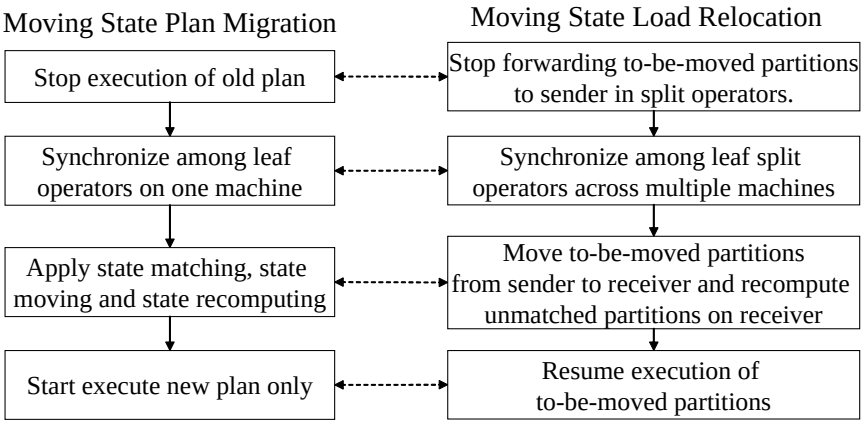


Figure 22.1: Moving State Load Balancing Strategy

First, the synchronization step, which is critical for both to ensure correctness of query results during each process, are different in terms of synchronization scales. For MS migration, the synchronization process synchronizes the execution of leaf operators to the old box. However, for MSLB, the synchronization is among split operators where the data is first being partitioned and distributed to both sender and receiver machines. Since split operators in a query plan can be activated on multiple machines, the synchronization process is also applied to multiple machines. This complicates the synchronization process, because now we need to synchronize among multiple machines which normally execute query plans at their own paces. In the designed MSLB strategy, this multi-machine synchronization is coordinated by the distribution manager. Each machine sends and receive messages to and from the distribution manager so that the executions on multiple machines are able to be synchronized by using the common knowledge received from the distribution manager. Once the multiple ma-

chines with active split operators are brought to a synchronized point, the rest of the state matching, state moving and state recomputing steps are very similar to the MS plan migration strategy.

Secondly, as in the PTLB strategy, the MSLB only moves to-be-moved partitions from the sender to the receiver. Other partitions should be processed as normal. So the split operators need to be able to distinguish between these two types of partitions and apply different actions to different partition types.

The next section describes the protocol designed to apply the ideas of MSLB to distributed load balancing.

22.2 Distributed MSLB Protocols

The key of the distributed moving state load balancing (MSLB) strategy is that now we need to carefully synchronize among multiple participating machines, including the distribution manager (DM), the sender and the receiver and all other machines with active split operators in the query plan in order to achieve the load balancing without resulting in any loss, duplicate or incorrect query results. Hence I have developed a communication protocol to achieve the MS load rebalancing. Each step consists of one or more message exchanges between distribution manager (DM) and one of the query processors.

Furthermore, I divide these steps into 4 phases of the MSLB process, including computing partitions, synchronizing execution, relocating partitions and reactivating partitions. The four phases of the protocols are

depicted in Figures 22.2, 22.3, 22.4 and 22.5 respectively.

MSLB Phase 1: Computing Partitions

The first phase of MSLB is the same as the first phase of the PTLB, which is to calculate the set of partitions IDs that need to be moved from the sender machine to the receiver machine. Steps 1 and 2 in the MSLB protocol correspond to communication between the distribution manager and the sender machine, as shown in Figure 22.2, which are the same as in the PTLB protocol depicted in Figure 21.2 of Section 21.2.

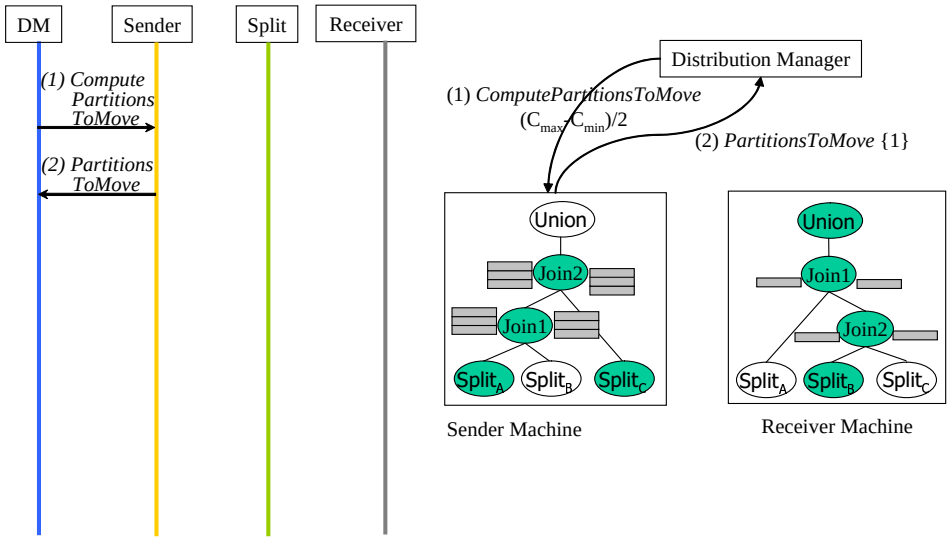


Figure 22.2: MS Load Balance: Compute Partitions to Move.

MSLB Phase 2: Synchronizing Execution

During phase 2 of the MSLB, messages are exchanged between the DM and processors to deactivate to-be-moved partitions before they are really

moved between machines. Furthermore, this phase also has the task of synchronizing the executions among all the active split operators to bring the system to a point when it is safe to move partitions between a pair of matching states on the sender and the receiver respectively. After this phase, the partitions can be send to the receiver without losing any useful information, as proved in the local MS plan migration in Chapter 12.

The phase 2 of the MSLB consists of steps 3, 4, 5 and 6, which are depicted in Figure 22.3.

In Step 3, the DM sends a *DeactivatePartitions* message, along with the to-be-moved partition IDs calculated in Steps 1 and 2, to the sender machine and all machines with active split operators. In this example, both machines have active split operators and both will receive such message from the DM.

On machines with active split operators, after receiving such *deactivatePartitions* message, an active split operator will take the following three actions in that order: 1) First, it removes the to-be-moved partition IDs from its partition mapping table, so that newly arriving tuples belonging to these partitions will no longer be forwarded to the sender machine. 2) Because after the first action, any new arriving tuples belonging to these partitions won't be forwarded to any machine, the split operator needs to create buffers to temporarily hold new tuples belonging to these partitions. 3) Lastly, the split operator calculates the minimal timestamp of tuples that belong to the to-be-moved partitions in its input queues. It then sends a message containing this *MinTimestamp* back to the distribution manager. This is illustrated as Step 4 in Figure 22.3.

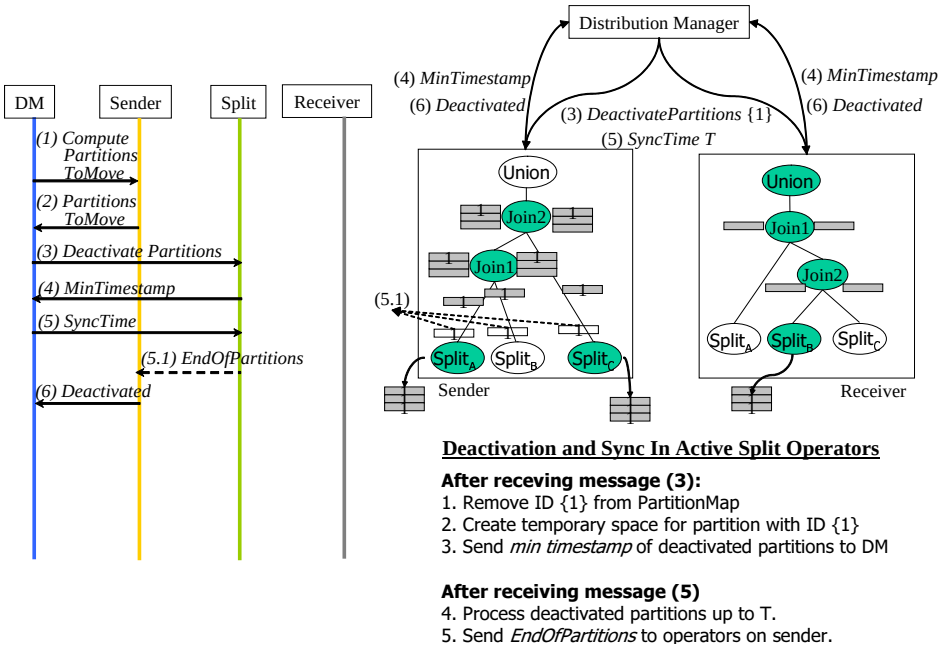


Figure 22.3: MS Load Balance: Deactivate and Synchronize To-be-moved Partitions.

The distribution manager waits for the *MinTimestamp* from all the split operators and calculates the max timestamp of all these min timestamps received. This is the global synchronization timestamp among all machines with active split operators. The generation of this synchronization timestamp is essentially the same as generating a synchronization timestamp for all leaf operators in the centralized moving state plan migration, which is described in Chapter 12. The distribution manager then sends this *SyncTime* to all machines with active split operators. This is illustrated as Step 5 in Figure 22.3.

When receiving the *SyncTime* message, each split operator then takes

two actions. 1) First, it keeps on processing tuples that belong to the to-be-moved partitions up to the received global *SyncTime*. Note that at this point, the tuples belong to the to-be-moved partitions can reside in both input queues and the temporary buffers created in Step 3 of MSLB. The split operator needs to process tuples from both locations if any tuple has a timestamp smaller than the global sync time. 2) The split operator then inserts an *EndOfPartition* tuple into each output queue that connects to the sender machine. This is shown as the Step 5.1 in Figure 22.3. This tuple is propagated through the operators on the sender machine: When a join operator has received the same number of the *EndOfPartition* as its input queues, it forwards this tuple to its parent operator. Therefore when this tuple reaches the top root operator on the sender machine, this indicates that from now on no more tuples belonging to the to-be-moved partitions will enter the sender machine. The sender machine then sends a *Deactivated* message back to the DM as Step 6.

The phase 2 of the MSLB not only deactivates to-be-moved partitions and synchronizes the execution of split operators, it also allows the operators on the sender machine to finish processing all on-the-fly tuples in these input queues that belong to these to-be-moved partitions (by sending and propagating the *EndOfPartitions* in Step 5.1). This clean-up stage is necessary, because if the partitions were to be moved right away without the clean-up, the on-the-fly tuples won't be able to join with these already moved partitions on the sender machine. We thus may miss some of the query results due to load balancing process.

MSLB Phase 3: Relocating Partitions

This phase does the actual partition movement by sending to-be-moved partitions from the sender to the receiver. This is achieved in Steps 7, 8 and 9, as depicted in Figure 22.4. The DM first waits for the *Deactivated* message from all the involved machines. After that, as Step 7, the DM sends a *SendPartitions* message to the sender machine. Upon receiving such a message, as Step 8 the sender machine packs all the partition groups with the selected IDs and sends them to the receiver machine using a *ReceivePartitions* message. After receiving and recomputing all partitions on the receiver machine, in Step 9, the receiver sends a *Received* message back to the distribution manager.

After receiving the *ReceivePartitions* message from the sender, along with all the partition groups, the receiver machine then conducts the following process:

- First, the receiver machine extracts each individual partition from the received partition groups.
- It then applies the *state partition matching* step, as described in Section 22.1, to match each single partition's schema with the existing states on the sender machine. A partition's schema, as well as a state's schema, is defined as the string concatenation of all the column IDs the tuple in this partition has. If a match is found, the single partition is then inserted to the state that has the same schema. At this point, this single partition should have a partition ID different from any existing partitions in that state. If a single partition cannot be matched

with any state, this single partition is then discarded by the receiver. Using the example in Figure 18.3, the moved partition group contains four single partitions P_{A2}, P_{B2}, P_{C2} and P_{AB2}. The first three single partitions will be inserted into states P_A, P_B and P_C on machine M2 respectively, while the single partition P_{AB2} is discarded since it does not match any states on machine M2.

- After the partition matching step, all the states that do not have a matching partition inserted will require a partition recomputation to regain the partitions that have the moved partition IDs. This can be done by recursively recomputing these single partitions in a bottom up fashion. Again using the example in Figure 18.3, the state P_{BC} does not have any matching partition. So the partition P_{BC2} that should have been moved from the sender machine would now be recomputed by joining the moved single partitions P_{B2} and P_{C2}. Note that we only need to join partitions with the same ID as the to-be-generated partition.

After the partitions have been moved and recomputed, in Step 9, the receiver machine sends a *Received* message back to the DM. This partition moving procedure is general, that is, it would also work when local plan optimization had not been invoked in the system, meaning the shape of query plans had stayed unchanged. In the latter case, all partitions transfered between two machines will be matching partitions on the receiver machine. Therefore no partition recomputation is necessary.

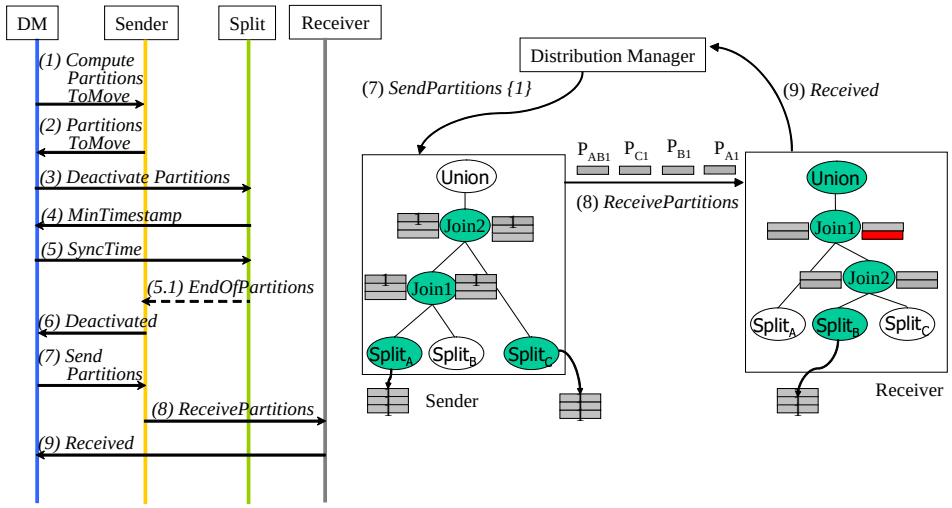


Figure 22.4: MSLB: Move and Recompute Partitions.

MSLB Phase 4: Reactivating Partitions

In the last phase of the MSLB, the split operators reactivate the processing of the tuples belonging to the to-be-moved partitions and send them to the receiver only. This phase has Step 10 as depicted in Figure 22.5, which is the last step of the MSLB protocol.

In Step 10, the DM sends a *ReactivatePartitions* message to all machines with active split operators. This is shown in Figure 22.5. Upon receiving such a message, the split operator will start forwarding new tuples belonging to the moved-partitions to the receiver machine. The tuples in the temporary buffers will also be forwarded to the receiver machine all at once, after which the temporary buffers are removed from the machine. The process of MS load balancing is then finished.

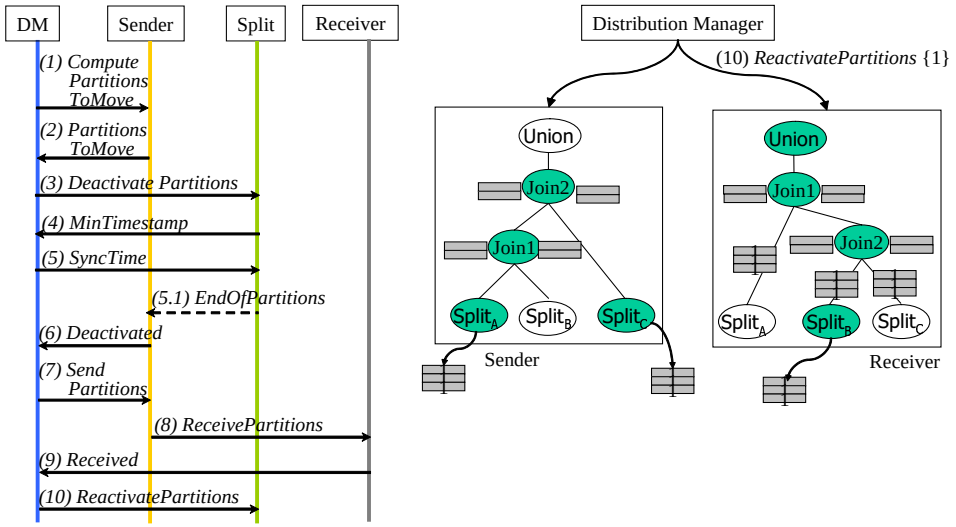


Figure 22.5: MS Load Balance: Reactivate Partitions.

MSLB Algorithms

Algorithms 11 and 12 sketch the high-level communication between the distribution manager and the query processors on each machine during the distributed MSLB process.

Algorithm 11 describes the basic operations of the distribution manager. The algorithm follows the actions in the timeline diagram (depicted on the left of Figure 22.5) by sending protocol messages and waiting for the corresponding responses.

Similarly, Algorithm 12 describes the steps performed on a participating processor during the state relocation process. The algorithm waits for control messages in the MSLB protocol. It performs corresponding actions based on the messages it has received.

In summary, the MS load rebalance strategy selects partitions to move

Algorithm 11 MS-State-Rebalance:Manager(sender, receiver, amt)

*/*It controls load balancing process by sending control messages to local machines and waiting for corresponding responses.*/*

1: **send** *ComputePartitionsToMove*(amt) msg to *sender*;
2: **wait** until get *PartitionsToMove* msg from sender;
3: **send** *DeactivatePartitions* to sender & machines with split operator(s);
4: **wait** until get *MinTimestamp* msgs from all split operators;
5: **send** *SyncTime* to machines with split operator(s);
6: **wait** until get all *Deactivated* msg from sender;
7: **send** *SendPartitions* msg to *sender*;
8: **wait** until get *Received* msg from receiver;
9: **send** *ReactivatePartitions* msg to machines with split operator(s);

and then directly moves these partitions from the sender machine to the receiver. Different from the PTLB strategy, it needs to have the knowledge of the detailed information about the query plan. However, it directly moves partitions from the sender and the receiver without delay. Therefore it can release the burden of the sender right away, which is expected to be the over-loaded machine of the two. It also does not incur the extra overhead of having to send new tuples to both the sender and the receiver, as in the PTLB strategy.

Algorithm 12 MS-State-Rebalance:Processor()

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

```
1: while (keepGoing) do
2:   wait for control messages of the MSLB protocol;
3:   switch(received message)
4:   ComputePartitionsToMove:
5:     compute partitions to move;
6:     send PartitionsToMove msg to Distribution Manager;
7:   DeactivatePartitions:
8:     split operators deactivate partition inputs;
9:     split operators send MinTimestamp msg to Distribution Manager;
10:  SyncTime:
11:    split operators execute tuples up to the SyncTime;
12:    split operators send EndOfPartitions to operators on sender;
13:    operators on sender propagates EndOfPartitions to root operator;
14:    sender machine sends Deactivated msg to Distribution Manager;
15:  SendPartitions: /*send out partitions*/
16:    wait for on-the-fly tuples finished being processed;
17:    send partitions via ReceivePartitions msg to receiver;
18:  ReceivePartitions: /*receive, insert and recompute partitions*/
19:    extract single partitions from partition groups received;
20:    insert matching single partitions to corresponding states;
21:    recompute single partitions in unmatched states;
22:    send Received msg to Distribution Manager;
23:  ReactivatePartitions: /*resume & redirect inputs for moved partitions */
24:    Split operators send tuples hold in temporary buffers to receiver;
25:    Split operators start sending tuples in to-be-moved partitions to re-
       ceiver only;
26:    keepGoing = false;
27: end while
```

Chapter 23

Experimental Evaluation

This part of the experimental evaluation focuses on two studies. First, I show the benefits of adding local plan optimization in the distributed continuous query processing along with the load balancing adaptation (Section 23.2). Second, I compare the performances of the two proposed load balancing strategies (Section 23.3).

23.1 Experimental Setup

I have implemented the dynamic query optimization and the two proposed load balancing strategies in a distributed continuous query processing system called D-CAPE [LZJ+05]. The D-CAPE system consists of a distribution manager, a stream generator and arbitrary number of query engines. Each machine runs a query engine. The distribution manager collects statistics from each query engine and initiates global load balancing among machines. The stream generator generates tuples with arrival pat-

terns modeled as the widely adopted Poisson process. That is, the mean inter-arrival delay between two consecutive tuples is exponentially distributed in order to model the Poisson arrival pattern. In each experiment, the stream generator continuously generates streams for 30,000ms. System parameters such as stream input rates and global time windows are varied to reflect the changes in workload and data characteristics, as will be further discussed in each experiment.

All experiments are run on a 10-node cluster. Each node has dual 2.4Hz Xeon CPUs with 2G main memory. I use the query in Figure 18.2 as the experiment query. The join operators have instances installed on all machines. Split and union operators are added to the plan accordingly. Each split or union operator is only active on one machine, while the join operators are active on all machines. I devote one machine each to run the distribution manager, the stream generator and the end application that receives query results. The remaining nodes can each be utilized to execute one D-CAPE processor, which is the central query engine to execute a query plan.

23.2 Benefits of Local Query Optimization

The main motivation for this work is to study the practicality and the potential benefit gainable by applying the local plan optimization in distributed continuous query processing. As we have discussed in Chapter 18, this leads to the necessity of designing new load balancing techniques to support heterogeneous query shapes due to local query optimization. There-

fore, our first goal for experimental evaluation is to show that local query optimization does boost the performance of partitioned CQ processing. To show the added benefits of local optimization, I compare the query performances in the following four settings:

- *No-Adapt*: In this setting, the same query plan is executed from the beginning to the end. Neither local optimization nor load balancing is applied during query execution.
- *LM-only*: Only Local Machine query optimization (LM) is applied as the form of adaptation during query execution.
- *PTLB-only* or *MSLB only*: Only either PTLB or MSLB is applied as the adaptation method to shift the workload across the available machines. But no local query optimization is applied during query execution.
- *LM-PTLB* or *LM-MSLB*: Both local query optimization and global load balancing are applied as adaptation techniques during query execution.

In this set of experiments, each of the three stream inputs (streams A, B and C) is partitioned into 100 partitions. The initial input rates for the three streams are each set to be 100 tuples per second. The initial plan joins streams A and B first, and the intermediate results then join with stream C. At the 30th second, the input rates of B and C are both changed to 5 tuples per second. This motivates the switch of the two join operators to

get a more efficient query plan (to reduce the number of intermediate results), which can be achieved only by local plan optimization. The partition functions in the split operators are initially set so that one machine in the system gets 50% of the total workload, while the rest of the workload is divided evenly among all machines in the system. This results in a scenario where load balancing is necessary in order to obtain a better utilizations of machine resources and hence result in better query performance. Load balancing is only between the heaviest loaded and the lightest loaded machines each time it is applied.

Here I focus on evaluating the benefits of local query optimization. To highlight my focus, I show the benefits of applying optimization with each one of the two load balancing strategies separately. The comparisons of the two load balancing strategies are left to Section 23.3 instead.

The following figures in this section depict the query performances, with each chart comparing the four settings mentioned above. I compare the query performances in various aspects, including the accumulated throughput, total tuples in system, total tuples in states and average query output rate.

I first show the results of applying LM with PTLB in Figures 23.1 and 23.2. These results compare the performances of the four settings described above in terms of query throughput and total tuples in system, respectively, when PTLB is applied as the load balancing strategy. Here the term "total tuples" accounts for all tuples across all machines, not just tuples on one machine. This shows the system performance as a whole.

The performance comparisons in term of throughput (accumulated) is shown in Figure 23.1. It is clear that the execution with neither forms of adaptation performs the worse. When applying the *PTLB-only*, the performance improves about 100% because the workloads are more balanced on all machines as a result of runtime load balancing. The execution with only local plan optimization (*LM-only*) but no load balancing also generates about twice the number of tuples generated by *No-Adapt* in the same amount of time. This shows that local plan optimization, as a runtime adaptation technique, can be as powerful as the widely used load balancing. Lastly, the execution with both forms of adaptions (*LM-MSLB*) has the best performance, producing about 330% more tuples than the *No-Adapt*. This illustrates that both adaptation techniques can significantly improve continuous query performance in distributed systems. Furthermore, combining the two techniques can lead to a better performance than applying either adaptation alone.

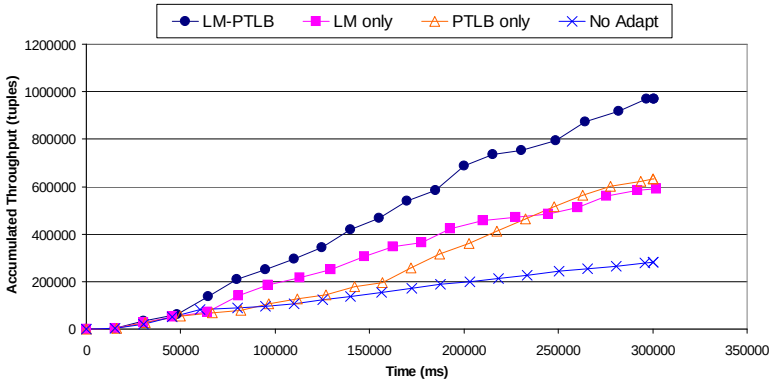


Figure 23.1: Throughput Comparisons (PTLB).

The total number of tuples in the system is a good indicator for how well the query performs. A build-up of tuples in the system indicates that the query engine is not able to keep up with the current workload. In the ideal case, given stable data statistics, a good query plan should lead to a stable number of tuples. In fact, in this case the number of tuples in the system should be very close to the number of tuples in states. This indicates close to zero accumulation in input queues. This is when the system is in stable state. On the other hand, the number of tuples in state is determined by several parameters, such as window sizes, stream input rates and selectivities. An optimized query plan, which can be achieved by plan optimization, can be expected to result in the least number of state tuples because this is when the number of intermediate results is minimal.

Figure 23.2 depicts the comparisons in terms of the total system tuples among the four settings. The two settings with local plan optimization, including *LM-only* and *LM-PTLB*, have much lower system tuple build-up than the other two settings without plan optimization (*PTLB-only* and *No-Adapt*). This is because both settings can apply query optimization early on to optimize the query plan as soon as the changes in stream input rates are first detected. Thus we observe much more in-time reduction in the number of tuples for these two settings before too many tuples are being build-up in the system. The reduction of system tuples happens for *PTLB-only* as well but it falls behind the above two settings because it needs to deal with a much higher tuple build-up. The *No-Adapt* has about the same highest build-up as *PTLB-only*. The number of tuples slowly drops as a result of slower stream input rates. But this drop may happen too slowly

and thus too late for a system with limited amount of memory. Both the *PTLB-only* and *NO-Adapt* have higher likelihood of causing system overflow than the other two settings with plan optimization applied. This set of results shows that load balancing itself may sometimes have very limited impact on lowering the total memory costs of the system. Local plan optimization can be much more critical than load balancing when it comes to releasing the burden of memory in the system as a whole.

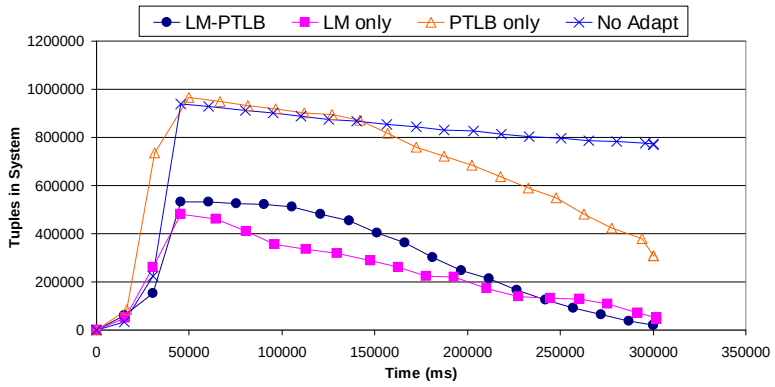


Figure 23.2: Total Tuples Comparisons (PTLB).

The comparisons in terms of number of tuples in states, as shown in Figure 23.3, tells a similar story as in Figure 23.2, although the cause is slightly different. Here both settings with plan optimization applied (*LM-only* and *LM-PTLB*) have smaller numbers of state tuples than the two without plan optimization. This is because after the query is being optimized, the number of intermediate results is smaller and therefore the number of tuples stored in the intermediate states is also smaller. The number of state tuples for *No-Adapt* and *PTLB-only* get smaller when the arrival rates of input

streams slow down. The lines do not suddenly drop because it takes time to purge all the state tuples from all states. Eventually, if the data statistics remain stable, the number of state tuples approaches a steady amount as well. We can see that the *LM-only* and the *LM-PTLB* in the end still have slightly less state tuples than the *No-Adapt* and *PTLB-only* because the plans are better for the former two settings and generate a smaller amount of intermediate tuples.

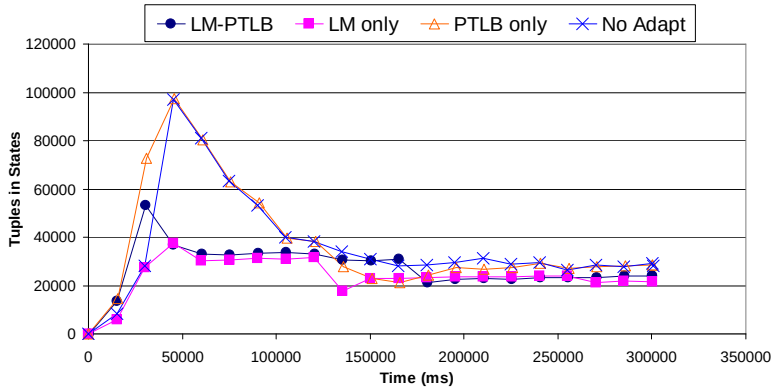


Figure 23.3: State Tuples Comparisons (PTLB).

Figure 23.4 compares the four settings in term of average output rate. The *No-Adapt* has the lowest output rate among the four, while both *LM-Only* and *PTLB-Only* both double the output rate. The *LM-PTLB* has the highest output rate, which is about 3.3 times faster than the *No-Adapt* towards the end of the experiment. For all four settings, the output rates are becoming flat, indicating that the rates are approaching stable when the data statistics remain stable.

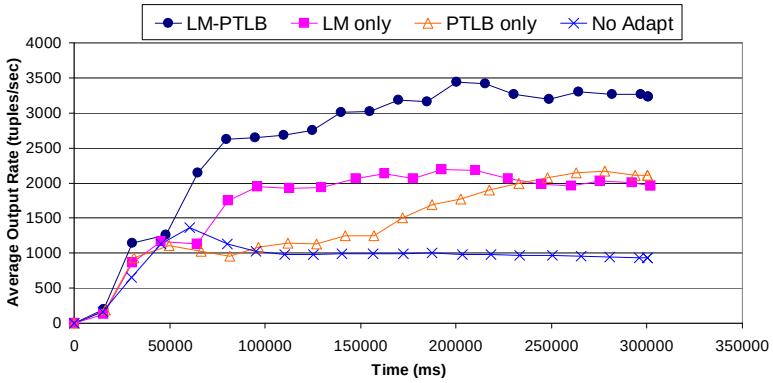


Figure 23.4: Output Rate Comparisons (PTLB).

When MSLB is applied as the load balancing strategy, we observe very similar patterns in all performance charts as compared to the corresponding performance charts depicted in Figures 23.1, 23.2, 23.3 and 23.4, respectively. In Figure 23.5, again the *No-Adapt* execution performs the worse, while the *MSLB-Only* performs about 80% better due to runtime load balancing. The *LM-Only* generates twice the number of tuples generated by *No-Adapt* in the same amount of time, and it even performs about 20% better than *MSLB-Only*. The *LM-MSLB* further improves the query performance and is about 2.7 times more productive than the *No-Adapt*. Figure 23.6 shows that the two settings with local query optimization, including *LM-only* and *LM-PTLB*, have much lower system tuple build-up and much faster memory relief in terms of total tuples than the other two settings without query optimization (*PTLB-only* and *No-Adapt*). Finally, Figure 23.8 illustrates that both *LM-Only* and *MSLB-Only* are able to produce a 100% faster output rate than the one with *No-Adapt*, with the combined *LM-MSLB* producing the fastest output rate among all four settings.

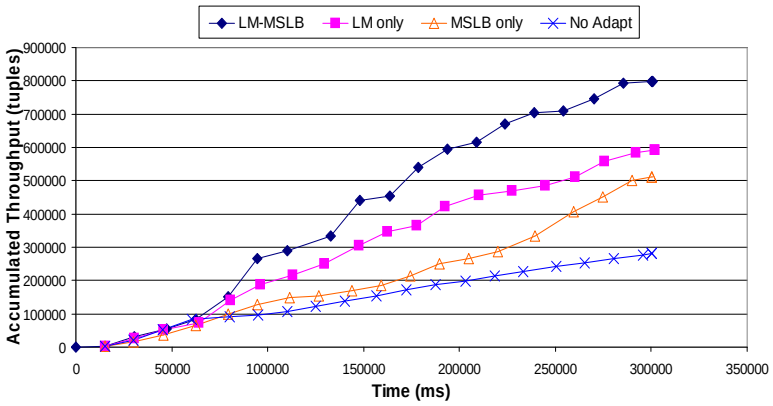


Figure 23.5: Throughput Comparisons (MSLB).

In summary, our experiments have shown that applying query optimization is essential in a distributed continuous query system. Furthermore, I have made the following three observations: 1) local query optimization can be as effective as load balancing in terms of improving partitioned continuous query performance in distributed systems. 2) query optimization has the effect of decreasing total system resource consumptions while load balancing only balances the workload but does not decrease it. 3) Combining both adaptation techniques can significantly improve query performances more than applying one of the two adaptations alone.

23.3 Comparing PTLB and MSLB

In this evaluation, I compare the runtime performances of the two proposed load balancing techniques, namely PTLB and MSLB. The overhead of the two load balancing strategies is largely determined by the number of

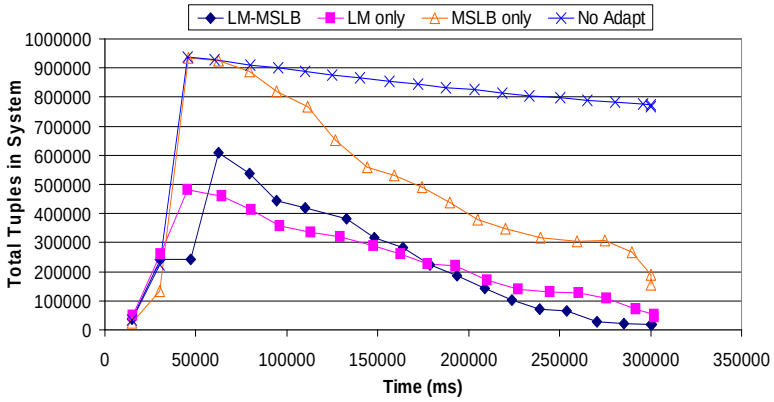


Figure 23.6: Total Tuples Comparisons (MSLB).

tuples in the corresponding states, which in turn is controlled by parameters including window sizes and stream arrival rates. For PTLB, the number of tuples in states determines how many old tuples need to be purged before the process is over. For MSLB, the number of tuples in states determines the cost of moving state partitions to another machine and the cost of recomputing unmatched partitions if needed. Since the state sizes are controlled by parameters including the window sizes and stream arrival rates, these parameters control the overhead of load balancing strategies.

As described in Section 21.1, PTLB is a general strategy that does not need to care about the detailed information about the plan itself. But this simplication comes at the price of overhead: the total process theoretically takes at least 2W to finish. [1] MSLB safely moves loads between machines by comparing the detailed shapes of the query plans. It can be more complicated than PTLB but potentially may take less time to finish.

[1]This is for a query plan tree with the height higher than 1.

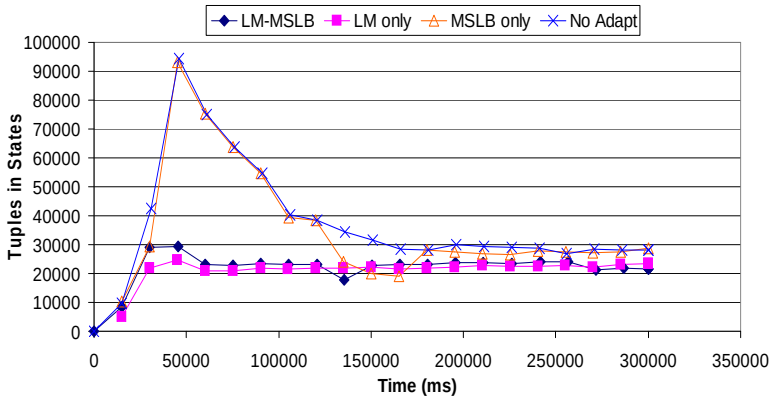


Figure 23.7: State Tuples Comparisons (MSLB).

I vary window sizes and stream rates in order to compare the two strategies in a range of parameter settings, from low, medium to high. The stream rates are set to be one of the three values: 30, 40 or 50 tuples/second, while the window sizes are set to be one of the four settings: 15, 30, 45, 60 second. The total number of combinations of stream rates and window sizes is thus 3 x 4 = 12. Therefore I have 12 different experimental settings. During our experiments, each experiment runs for 30,000ms. I run each setting at least 5 times, and get the average of the total throughput as the throughput of that setting. All the other environment setup is the same as in the previous section.

For each setting, I also run the experiment with no adaptation to serve as the base performance. The average throughput of this base run is the *base throughput*. I then run the experiment by applying either the PTLB or the MSLB to adapt the query plan. The average throughputs of the PTLB run and the MSLB run are then divided by the base average throughput

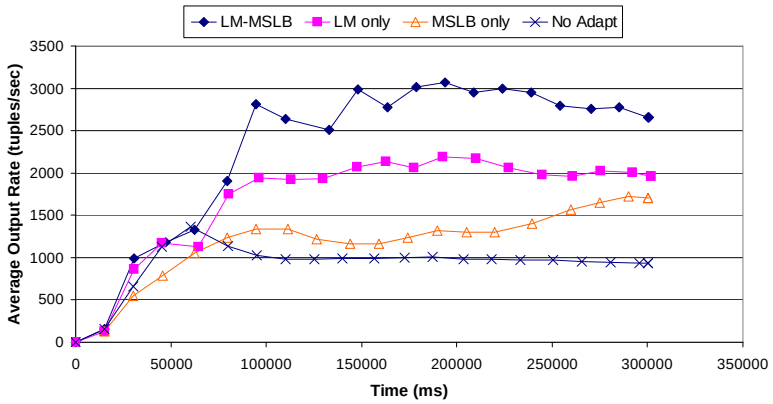


Figure 23.8: Output Rate Comparisons (MSLB).

to get the scaled *throughput ratio*. The throughput ratio for the run with no adaptation is 1 because it is divided by itself. The larger the throughput ratio is, the better the query performs as compared to the run without adaptation.

Figures 23.9, 23.10 and 23.11 depict the results of the 12 settings with different combinations of window sizes and stream rates. Each figure compares the throughput ratio of the base case, the PTLB and the MSLB.

Figure 23.9 shows the results of the 4 settings in which the stream rates are set to be 30 tuples/sec. We can see that as the size of the window grows, the difference of average throughput ratios between the base case and either the PTLB run or the MSLB run are getting larger.

The difference between the PTLB and the MSLB also changes from insignificant, when window size is small, to about 25% difference, with the MSLB gaining the edge. This is because, as the window size grows, the total time for the PTLB to finish also becomes larger (it's estimated as 2W

in Section 21.1). This means the over-loaded machine will continue to be overloaded because it needs to purge out all the old tuples. This slow relief can have a negative impact on the overall system performance. In comparison, the MSLB is able to release the overloaded machine (the sender) right away by moving tuples from the machine to another machine (the receiver). Even if some states are unmatched and need to be recomputed, this work will be done at the receiver side. The later is expected to be the underloaded machine. Therefore the impact of such recomputation to the overall query performance would be rather light.

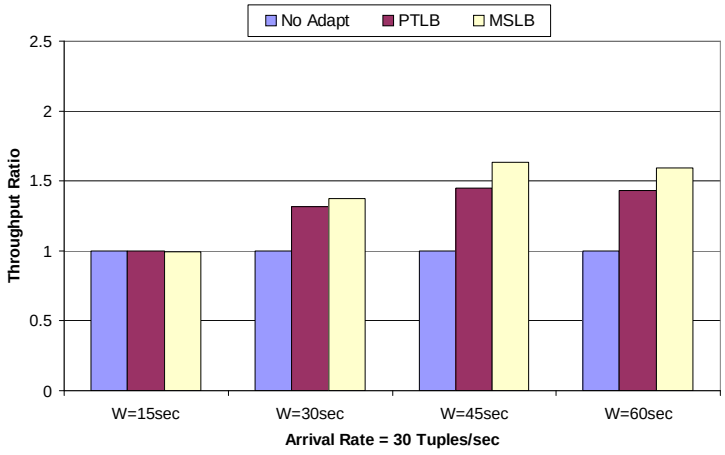


Figure 23.9: Throughput comparisons (λ = 30).

We can observe similar but more dramatic trends in Figure 23.10, where the stream rates are all set to be 40 tuples/sec. Since the stream rate is higher than in the previous set of results, the lead of the PTLB and MSLB versus the base case is much larger even when the window size is small. This performance lead still continues to grow as the window size becomes

larger. Here we can still observe that the MSLB has better performance than the PTLB when the window size becomes larger.

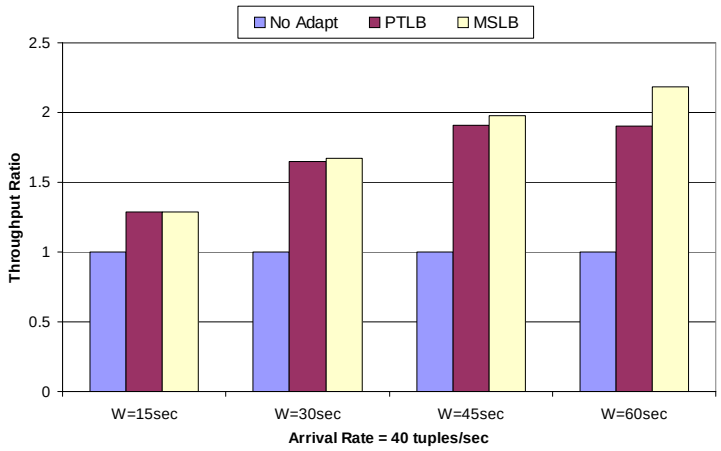


Figure 23.10: Throughput comparisons (λ = 40).

In Figure 23.11, when the stream rate is set to the relatively high (namely 50 tuples/sec), the trend is a bit different than in the previous two comparisons. First, when the window size is small, the difference between the PTLB or the MSLB and the base case is very large. On average, the PTLB produces about 90% more tuples than the base case, and the MSLB produces about 100% more tuples than the base case. However, as the window size grows larger, this difference is not further enlarged. Instead, the gap between the base and the PTLB is getting narrower. This is because as both stream rates and window sizes are set to high values, the PTLB starts to take a long time and to consume large amounts of system resources in order to purge all old tuples on the already overloaded sender machine. Therefore the PTLB strategy becomes less and less efficient. On the other hand, the

MSLB is becoming more efficient in comparison to the PTLB, demonstrating that MSLB is a better choice when the parameters have high values.

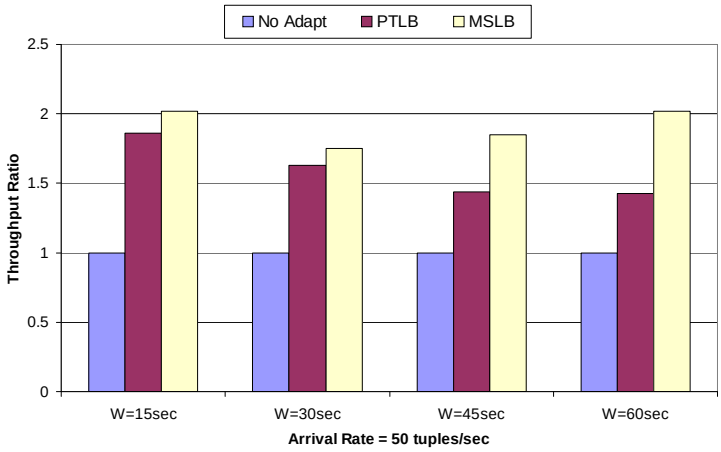


Figure 23.11: Throughput comparisons ($\lambda = 50$).

Figures 23.12, 23.13 and 23.14 compare the average total time taken by the two load balancing strategies in the 12 experimental settings. As I have estimated using cost models, the PTLB always takes approximately 2W time to finish, while the MSLB usually takes much shorter time to complete the whole process.

So far our experimental results have shown that the MSLB strategy is winning. However, given certain combinations, the PTLB can perform better than the MSLB as well. This is when the cost for state moving and recomputation is high (large state sizes) while the cost for processing new tuples is relatively low (low stream rates). Such situation will happen when the stream statistics changes shortly before the load balancing process. For example, if the stream rates start at very high, this results in very large

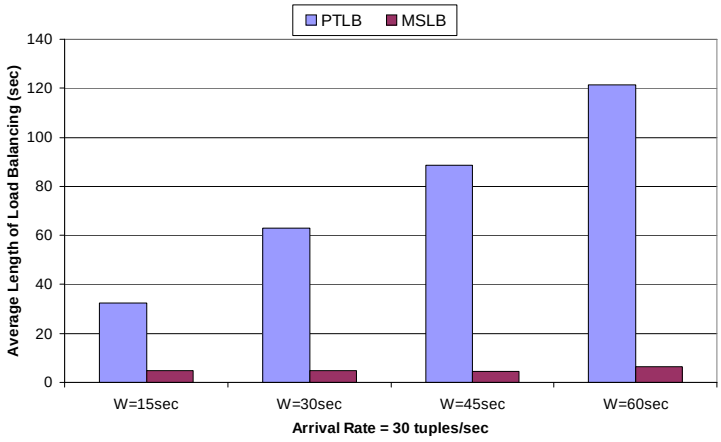


Figure 23.12: Average Lengths of Load Balance (λ = 30).

state sizes. However, at the time the load balance is triggered, the stream rate may have become very low. Thus the cost of purging old tuples is relatively low (lower cost on processing fewer new tuples). However, since the number of tuples accumulated in the states are high, the cost of moving the state and the cost of recomputation can potentially be very high. In this case, the PTLB can be more efficient than the MSLB.

I set up an experiment to reflect this situation. The same query is used as before. For the three input streams, A, B and C, the input rates all start to be 100 tuples/sec. At 30th second, the input rates for B and C slow down to 5 tuples/sec. This also triggers a local query optimization on the machine with the highest workload. The load balancing process is then invoked. Figure 23.15 shows the experimental results. As we can see, the PTLB starts to have better performance after the load balancing process is triggered. As mentioned earlier, such stream changes benefit PTLB because it lowers the

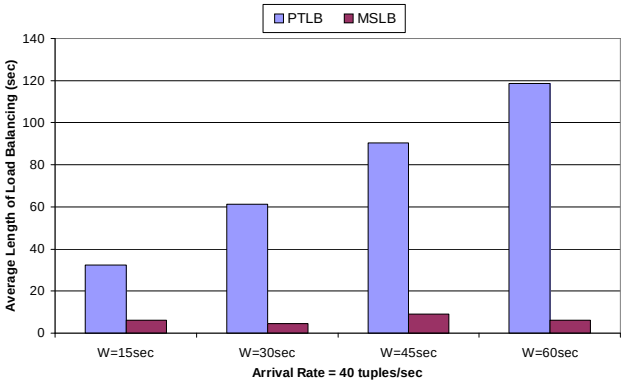


Figure 23.13: Average Lengths of Load Balance (λ = 40).

cost of purging old tuples. However, since the state size has already grown very large at this point, the cost of moving the partitions and recomputing the unmatched partitions can be high. So in this case PTLB is winning.

In summary, our experiments have demonstrated that MSLB has better performance than PTLB because the former utilizes the underloaded machine more while the latter continues to use the already overloaded machine to purge old tuples. However, under certain circumstances, the cost of state purging can be smaller than the cost of state moving and state recomputing. This may occur when the data statistics change towards the direction that decreases the cost of PTLB. In this case applying PTLB can be more efficient than applying MSLB.

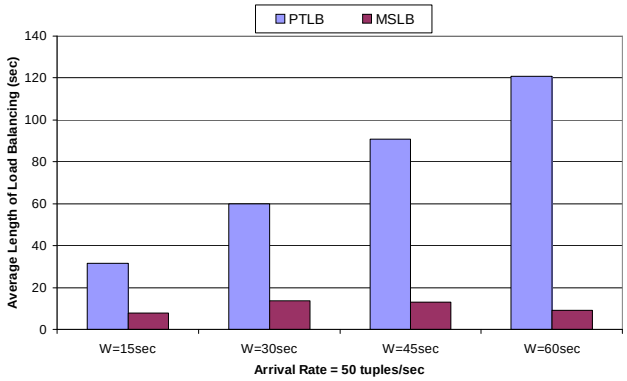


Figure 23.14: Average Lengths of Load Balance ($\lambda = 50$).

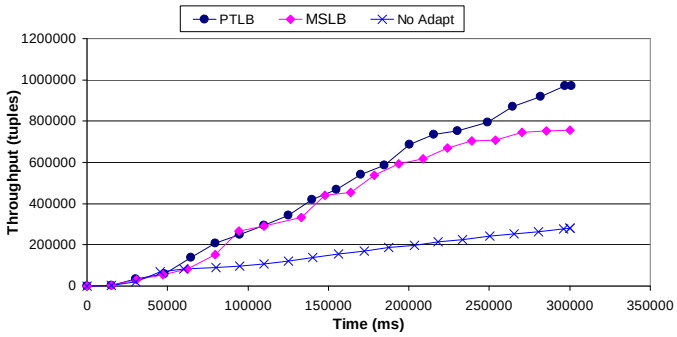


Figure 23.15: PTLB-better-than-MSLB Case.

Chapter 24

Related Work

Existing distributed continuous query systems [AAB+05, CBB+03, DH04] use an operator as the basic unit of load balancing. This assumes that each operator is small enough to fit on one machine. Partitioned parallelism is a general query plan distribution strategy [Has95, Gra90]. The Flux system [MJSM03], the first to apply partition-level load redistribution to continuous queries, has demonstrated promising performance. However, Flux has been focused on group-by and assumes that all query instances installed on machines have the same query shapes. My research instead propose load balancing strategies to deal with the heterogeneity of plan shapes with stateful join operators among different machines.

Continuous query optimization has been studied in the literature in recent years [BBD+02a, VN02a, CDN02, IHW02]. [VN02a] proposes a rate-based algorithm to optimize continuous multiple joins to achieve a high output rate. [BMM+04] proposes heuristics-based join ordering algorithms for mjoin that consider dependent join selectivities. [MSHR02] introduces

the Eddy approach of adaptively executing a query by routing tuples among operators. Eddy's always-adapting solution makes it suitable for a highly dynamic environment. These solutions all focus on optimizing continuous queries based on statistics collected at runtime.

My earlier work on dynamic plan migration [ZRH04], as presented in Part II of this dissertation, is the first to deal with the problem of safely transferring the currently running plan to the new plan generated by the optimizer. The migration strategies described earlier inspire my designs of the two load balancing strategies proposed in this part of the dissertation, as both solve the problem of switching among different plan shapes. However, the two problems are significantly different: The former changes the currently running query plans with stateful operators on a local machine, while the latter moves operator partitions among machines with different query plans. The latter requires carefully synchronized coordinations among participating machines.

Part IV

Conclusions and Future Work

Chapter 25

Conclusions of This Dissertation

Continuous queries process real-time streaming data and output results in streams for a wide range of applications. Due to the fluctuating stream characteristics, a streaming database system needs to dynamically adapt query executions. This dissertation proposes novel solutions to continuous query adaptations in three core areas, namely dynamic query optimization, dynamic plan migration and partitioned query adaptation.

The first part of this dissertation proposes two polynomial-time optimization strategies, namely *mjoin-init* and *bjtree-init*, that generate continuous multi-join plans meeting resource constraints of both CPU and memory. The proposed strategies consider mjoin, bjtree, and the tree structures in-between as solution candidates. They search the entire query plan space in polynomial time when a typical exhaustive search would take at least

exponential time. I have designed four new optimization algorithms, two for each proposed optimization strategy. Within each strategy, the first algorithm utilizes the positive correlation to decrease both memory and CPU costs, while the second utilizes the negative correlation to further tune the trade-off between the two resources. Besides the two efficient optimization strategies and the four optimization algorithms mentioned above, I also design an exhaustive search strategy using bottom-up dynamic programming, which searches the whole multi-join search space to find a qualified query plan. This exhaustive search strategy guarantees that a qualified plan can be found if there exist one. All proposed strategies in Part I are implemented in the DCAPE [RDS+04, LZJ+05] continuous query system and have been evaluated and compared through a comprehensive experimental study. The experimental results show that both proposed optimization strategies are as reliable in finding qualified query plans as the exhaustive search strategy, while taking much less time and space than the exhaustive strategy. The qualified query plans generated by the mjoin-init strategy tend to have lower memory but higher cpu costs as compared to the qualified query plans generated by bjtree-init strategy. Therefore, an runtime optimizer may choose which optimization strategy to apply depend on which resource factor (memory or cpu) is the more ample one in the current system or which one is expected to become more critical in the near future.

In the second part of the dissertation, I have designed two dynamic query plan migration strategies, namely the moving state strategy and the parallel track strategy, for migrating continuous query plans at runtime.

The first strategy exploits reusability of existing stream states and the second employs parallel query execution to seamlessly migrate between continuous join plans without affecting the results of the query. I first present the basic ideas of the two migration strategies focusing on joins only while assuming a particular system execution model. I then generalize and significantly extend the existing migration strategies along several dimensions, including to cover all common types of operators (and not just joins), all execution models and timestamp representations common in the current stream literature. Various execution models are identified and categorized to illustrate how different execution model can affect the runtime plan migration strategy. Each identified execution model has its unique properties on tuple execution order and operator scheduling. I describe how to apply these migration strategies to query plans that contain Select, Project and Join (SPJ) operators, and Group-by and Aggregate operators. The proposed migration strategies are implemented in the DCAPE system. Experimental evaluations have been conducted to compare their performances. The experimental results demonstrate performance improvements in the order of magnitude by dynamically applying the migration strategies in the middle of query processing in a variety of system settings.

In the third part of this dissertation, I point out that existing load balancing solutions have made the simplifying assumption that query plan instances on all machines are static, i.e., no query optimization is conducted at runtime. This is clearly unrealistic for dynamic stream systems. In this work, I point out that adding plan optimization to distributed continuous query processing is beneficial but doing so also creates new problems in dy-

namic load balancing. The new problem is the heterogeneity of query plan shapes among machines as a result of applying local query optimization, which has yet to be dealt with by current state-of-the-art load balancing strategies. I therefore propose two new load balancing strategies, namely the PTLB and the MSLB strategies, along with their corresponding protocols, that can balance the workload while seamlessly handling the complexity caused by local plan changes in the system. The PTLB strategy is a general load balancing strategy that requires no knowledge of the underlying query plan optimization. The MSLB strategy, on the other hand, rebalances the workload by comparing the detailed shapes of the query plans among different machines. Both proposed load balancing strategies are implemented in the DCAPE system. The experiments are conducted in the DCAPE system using a real cluster. The results show that the combination of query optimization and load balancing exhibits significantly superior performances than applying each adaptation technique alone. Applying query optimization in partitioned query processing have shown dramatic performance improvement by more than 300%. Between the two load balancing strategies, the MSLB is shown to be more efficient than the PTLB in many situations, while the PTLB can win under certain conditions.

Chapter 26

Ideas for Future Work

This chapter discusses several topics, with each containing several problems for possible future work that are important for runtime continuous query adaptation. For each topic and its contained problems, I discuss relatively detailed thoughts on the possible solutions to these problems. In particular, these topics for future work include:

- *Choosing Optimization Timing*: Choosing *when* to apply query optimization is a critical problem in runtime continuous query processing. Since the data characteristics in streams can change over time, if we optimize too soon, we may optimize too frequently and the benefits of optimization may be overshadowed by its overhead. On the other hand, if we optimize too late, we may miss optimization opportunities for improving the query performance. Therefore, this is an important yet tricky problem. I discuss it further in Section 26.1.

- *Plan Migration Scope*: Once the optimizer decides to optimize a query

plan, which part of the query plan should be optimized and migrated. Should we optimize the whole query plan or only part of the query plan? Should we migrate a query plan as a whole or step-by-step? These are also unsolved yet crucial problems to runtime query optimization and migration. I will discuss my thoughts and possible solutions to these problems in Section 26.2.

- *Distributed Query Optimization and Allocation*: In a distributed system, we need to decide how to optimize a continuous query and how to allocate the query plans across machines. I will give my thoughts on these future work in Section 26.3.

26.1 Future Work on Choosing Optimization Timing

In this section, I discuss my thoughts on the problem of choosing when to optimize and migrate a continuous query (CQ) at run time. Since the query migration is basically the last step in a dynamic query optimization process, the problem of choosing the timing for migration is indeed the problem of choosing the optimization timing at runtime. I here present simple timing strategies that could be adopted in runtime optimizer. I then describe possible improvements.

The most challenging aspect of the dynamic optimization for continuous queries is the unpredictability of the stream characteristics. This means that the data statistics, including stream arrival rates, arrival patterns and value distributions, may keep on changing in an unpredictable way during a CQ execution. In general, it is hard or even impossible to know what the

data characteristics will be like even in the very next moment. The optimizer only has the knowledge of what has happened so far, but does not have the knowledge of what is going to happen, unless the optimizer has gained the knowledge that there is a repeating pattern which I will discuss later in this section.

26.1.1 Data-Driven Optimization

Data-driven optimization for continuous queries here is defined as optimization triggered by detecting changes in data statistics at runtime. In other words, the optimization is directly triggered by the changes in the data itself. The data statistics considered include data arrival rates and predicate selectivities.

A typical way to choose a good timing for data-driven optimization is to select an *optimization interval* and a *statistics threshold*. If one optimization interval has passed and the changes in data statistics are beyond the defined statistics threshold, the optimizer is invoked to optimize the query plan. A more aggressive version would be that even if the optimization interval is not passed, the optimizer may still be activated just because the statistics have changed beyond the threshold. Prototype systems such as POP [MRS+04] define upper and lower bounds (thresholds) of cardinalities expected in the inputs to or outputs from an operator. If such thresholds are violated, the runtime optimizer is then triggered.

The data-driven optimization is easy to be explained and accepted logically. However, it has several problems that make it hard to be adopted for a continuous query optimizer.

First, finding the proper optimization interval is a difficult task. If the interval is defined to be too large, the optimizer may wait for too long to take actions and thus may miss some critical moments of optimization. On the other hand, if the threshold is defined too small, the optimization may happen too frequently. This can lead to *thrashing*. When a query optimizer enters the thrashing state, the gain of the optimization may be overshadowed by the overhead of the frequent optimization. Therefore, the optimization interval itself may need to be tuned dynamically.

Second, it is difficult, if not impossible, to define the threshold which indicates that the changes of statistics are so significant so that runtime optimization becomes necessary. In some cases, changes in data statistics do not necessarily imply that the current query has become sub-optimal.

Third, as illustrated in Part I of this dissertation, the optimization goal for the continuous query optimizer is to find a qualified query plan that satisfies both CPU and memory constraints. However, this concept of a qualified plan is hard to embed into the data-driven optimization.

Therefore, data-driven optimization can be highly complicated, if not impossible, to apply to runtime optimization for continuous queries.

26.1.2 Memory-Driven Optimization

Among all the information that an optimizer is observing in order to make a decision on timing, what it really needs is a simple yet clear indication that the current query is not good enough to keep up with the incoming workload. In this section, I illustrate that this simple indication may be achieved by simply observing memory consumptions.

Memory usage is a good choice for the purpose of triggering runtime optimization for the following reasons. First, a qualified plan is bound by both memory and CPU. So memory usage itself is a critical indicator to show if a query plan is qualified or not. Second, a query execution that is lacking of CPU will also result in newly arriving tuples being accumulated in input queues, and thus eventually it will also cause memory build-up. So any major increase in memory usage can be a signal that either or both of the system resources are not sufficient for running the current query plan. Hence a more efficient plan needs to be generated by the runtime optimizer. Otherwise, the system may be running out of memory. Last but not the least, memory usage is easy to measure at runtime, and therefore its increasing or decreasing trend is easy to discover. When the memory usage has increased for a period of time, we know that the current query plan is most likely no longer a qualified plan and a change needs to be made by the optimizer.

The *memory-driven optimization* is controlled by two memory related parameters: the total memory usage and the number of tuples accumulated in input queues. A runtime optimization is necessary when:

- The total memory usage is approaching the total memory available in the system for this processing task. This would avoid the system from memory overflow.

- A runtime optimization is also necessary if the number of tuples in input queues has the trend of increasing steadily for a period of time. This indicates that a relatively permanent change has occurred in the

data statistics and the current query plan is no longer good enough to keep up with the current workload.

A sudden increase in the amount of tuples in input queues is an indication that a sudden change has occurred. In this case, it may be more useful to apply load shedding or data spilling than to invoke query optimization. This is because the impact of query optimization can be delayed for a certain amount of time and thus would not be worthwhile if the change is only sudden for a very short period of time.

In general, we can view the data-driven optimization as trying to converge to an optimal plan, while the memory-driven optimization instead as trying to converge to a steady state.

However, by using the memory-driven optimization, the effects of sub-optimality can be delayed. For example, if two joins are being executed in the wrong order (the higher one has much larger selectivity than the lower one), and if they are located high up in the query tree, tuples will accumulate mainly in the input queue of the lower join. That is, it is not an input queue to the query plan but is rather an intermediate queue inside the query plan. It may take a very long time to have the effects of accumulating tuples to propagate down through the query tree and finally to reach the input queues of the query plan. So by the time tuples are shown to be accumulated in the input queues, the cause of this accumulation may already persist for a while and may even not be present any more at this time.

26.1.3 Refined Memory-Driven Optimization

To solve the problem of delayed impact, we could adopt a *refined memory-driven optimization*. Instead of monitoring the number of tuples in input queues, we can also monitor the number of tuples in intermediate queues. If a build-up in the input queues or intermediate queues is detected and if the trend holds for a certain amount of time, we can invoke the runtime optimizer to optimize the query plan. This enables a quicker detection of a possibly troubled execution.

Note that when using memory-driven optimization, we still need to collect data statistics like the ones used in data-driven optimization. However, instead of using these data statistics to control optimization timing, the optimizer, once invoked by memory-related indicators, would use these statistics to generate a better query plan.

26.1.4 Query Logging

It is possible that the data changes follow certain pattern. For example, a typical day of stock trading may experience certain patterns over the course of the day. The pattern may be in the volume of data streams or in the value distributions in data. If both parameters follow similar patterns, the optimizer may store a library of candidate query plans, each labeled by its data statistics involved in the query. This solution is referred to as *Query Logging*, an concept that exists in most commercial static DBMSs. At run time, the optimizer can pick a query plan from the library that best matches with the current data statistics. This query logging strategy can be highly

efficient if the number of combinations of data statistics is not large.

The above presented strategies for choosing optimization timing can be useful for different types of stream changes, such as random change, periodic changes, or sudden bursty changes. An interesting future work is to compare the performance of these strategies given difference stream change patterns.

26.2 Future Work on Choosing Migration Scope

The problem of choosing the proper migration scope is closely related to how the query is being optimized by the runtime optimizer. Here I describe two distinct yet close related concepts, namely the *optimization box* and the *migration box*. The optimization box contains the query plan or sub-plan that needs to be optimized, while the migration box contains the plan or sub-plan that needs to be migrated. I first give my thoughts on how to determine the size of the migration box given different runtime optimization methods. I view this problem of choosing the scope (size) of the migration box to have two aspects:

- The first aspect is to determine where to place the migration box in the query plan and what is the size of each migration box. This is in fact the problem of choosing which part of the query plan (network) needs to be migrated, based on how the query is being optimized. The sub-query to be migrated can be large or small depending on the overhead of the migration and optimization process. Sometimes more than one part of the query needs to be optimized and therefore

we may have more than one migration box. So this aspect determines how many migration boxes and the size, i.e, the scope, of each migration box in the migration process.

- The second aspect is to determine the *migration steps*. The problem is that once a part of the query plan is chosen to be migrated, should we put this sub-query into one migration box and migrate the sub-query together, or should we migrate step-by-step, i.e, which each step covers the migration of switching two consecutive operators such as joins. So this presents an option of migrating using the box chosen in the first aspect, or to further divide the box into smaller boxes and migrate the smaller boxes one at a time.

26.2.1 Determining The Size of Migration Box

The size of the migration box can be determined by the optimization algorithm used by the runtime optimizer and can be equal to or smaller than the optimization box. For a relatively simple query plan that has a small number of operators, the optimizer can afford to use dynamic programming to exhaustively search through the problem space to find the best query plan given current data statistics. In this case, the optimization box would contain the whole query plan, and the migration box is the same as the optimization box.

However, if the query plan is a complex query plan or query network that contains a large number of operators, such as more than 7 or 8 joins, heuristic-based algorithms may be necessary to find a good plan based on

the current statistics. In this case, the optimization box still covers a query plan since it needs to optimize the plan. However, to avoid any high optimization overhead, the optimizer may only optimize some parts of the query plan that it sees the most necessary and would most likely improve the performance of the query plan. For example, the optimizer may focus on reordering parts of the query plan that contain multiple join operators. Therefore, the migration boxes are placed over those parts of the query network that contain the multiple joins whose orders have been changed by the optimizer.

Figures 26.1 (a) and (b) depict the two examples when the optimization box and the migration box are the same and when they are different. The query plan on the left (Figure 26.1(a)) contains two join operators, so the optimization box and the migration box are the same. However, the query plan depicted in Figure 26.1(b) has several join operators, and the optimizer may choose to switch two pairs of joins, resulting in two smaller migration boxes.

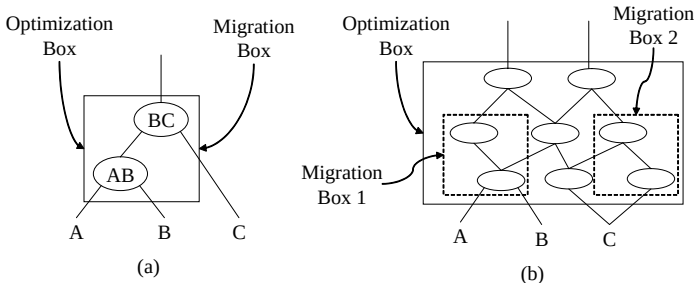


Figure 26.1: Optimization Box and Migration Box

The prerequisite of the runtime optimization is to keep the benefits of

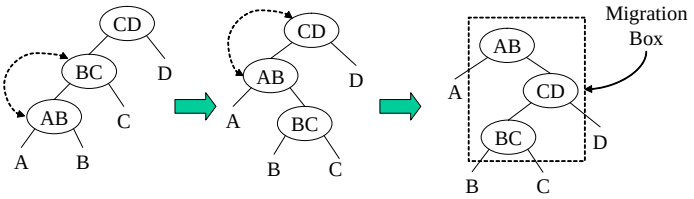
optimization to be higher than its cost. Therefore, the number of migration boxes and the size of migration boxes are largely determined by the cost (overhead) of the whole optimization process, including both the optimization cost and the migration cost. Before each optimization step, the optimizer would get an allocated quota, which limits the upper bound of the total amount of overhead this round of optimization should incur. This quota can be achieved by analyzing the most recent changes in the data statistics and estimating when the next change would possibly occur.

When an optimizer rebuilds a whole new query plan using dynamic programming, it does not take the cost of migration into account. Therefore this type of optimization is only suitable for simple queries, where the cost of both the optimization and the migration are likely to be low. On the other hand, if the optimizer uses a step-by-step optimization method, for example, switching joins one pair at a time as described above, it is easier to estimate the overall cost of optimization and migration after each step. Thus for large query plans and query networks, step-by-step optimization gives the optimizer better control on the cost of the migration, and also makes it easier to place the proper migration boxes.

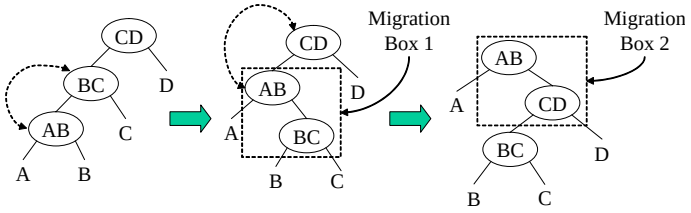
26.2.2 Choosing Migration Step

Another important problem when choosing the size of a migration box is to decide the *migration step*. Assuming the step-by-step optimization is being applied, two options exist to place the migration box: First, one single large migration box could be placed around the sub-query that the optimizer has changed. Second, a small migration box can be placed around the opera-

tors the optimizer has just changed in the last step and migrate that part of the query right away. In the next optimization step, we place the next migration box until the optimization process is done.



(a) One big migration box after optimization



(b) Several small migration boxes along optimization steps

Figure 26.2: Size of Migration Box and Migration Steps

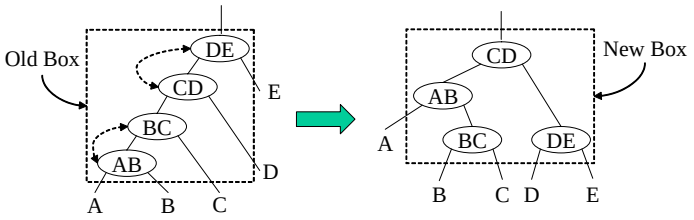
Figures 26.2 (a) and (b) depict these two choices of migration steps respectively. The optimization box contains three binary joins, and the optimizer first switches joins AB and BC, and then it switches joins AB and CD. After the two optimization steps, CD becomes the bottom join operator and AB the top join operator in the query tree. In Figure 26.2 (a), the migrator decides to delay the migration process until both optimization steps are done. So a migration box that includes all three join operators is placed over the query plan. On the other hand, a smaller migration box could be placed to include the two joins that are just switched by the optimizer and migrate the query plan right away. Another migration box is placed to contain the two joins that are switched in the next step of the optimization. So

in total, two migration boxes, each contains two joins, are used during this optimization process.

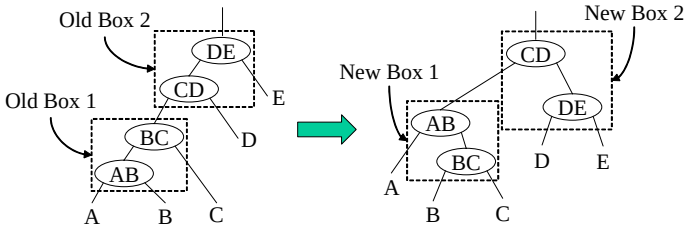
Now the question is which one of two choices of migration steps (thus migration box placement) incur less overhead, or they are the same in terms of overhead? Are the answers the same for the moving state migration strategy and for the parallel track migration strategy?

The replacement of one migration box by several smaller migration boxes can be broken into two different cases. One case is that a migration box can be divided into several *non-overlapping migration boxes*. Another case is that one migration box is replaced by several *overlapping migration boxes*. Which replacement can be applied depends on the actual changes made to the sub-query by an optimizer. *Non-overlapping migration boxes* are migration boxes that do not have common operators among them. Figure 26.3 illustrates an example where one migration box can be replaced by several non-overlapping migration boxes. One migration box may also be replaced by several *overlapping migration boxes*, as shown in Figure 26.4. Are the migration steps in these two examples causing any differences in the migration overhead?

These are all interesting future tasks as extensions to Part II on dynamic plan migration of this dissertation.

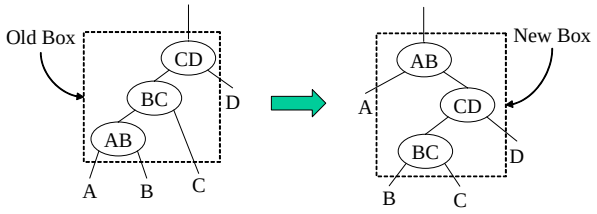


(a) One big migration box at the end of optimization

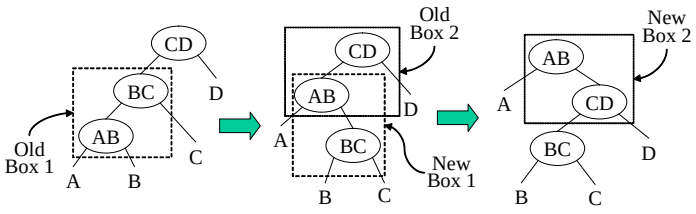


(b) Two small migration boxes at the end of optimization

Figure 26.3: Non-overlapping Migration Boxes



(a) One big migration box at the end of optimization



(b) Two small migration boxes along optimization steps

Figure 26.4: Overlapping Migration Boxes

26.3 Future Work on Distributed Optimization and Allocation

When executing queries in a distributed environment, we face several problems that are not encountered in centralized systems. These are all valuable

future work that can make contributions to distributed continuous query processing.

- How to optimize the query in a distributed system?
- How to distribute the query operators across multiple machines?
- How should the above two aspects cooperate with each other?

Considering both of the two aspects at the same time when choosing a query plan can dramatically increase the complexity of the already complicated query optimization process. A classic method is to use a two-phase approach proposed in [Hon92, Has95], that is, to separate the query optimization and distribution into two steps. A query is first optimized into a query tree, which is then being distributed to multiple machines based on the cost-related annotations associated with the query plan that were generated during the query optimization stage.

For distributed continuous query processing, this same solution can also be applied. The optimization to generate a qualified continuous multi-join query plan can still be applied first with some modifications, which are discussed in the next section. The difference between task scheduling for a traditional query and the task scheduling for a distributed query is that the latter does not have blocking operators. Therefore they can apply pipelined parallelism and partitioned parallelism without any temporal restrictions caused by a blocking execution. This requires modification to the existing solutions for scheduling static queries. However, the general ideas should still be applicable.

In the next two sections, I discuss my thoughts on possible approaches for optimizing and distributing multi-join continuous query plans in a homogeneous distributed system. The homogeneous system here implies that all machines have the same processing power, meaning the available memory and CPU on each machine is the same. To simplify the problem, I also assume that the number of processors can be arbitrary but known to the distributing component. In Section 26.3.3 I will give my thoughts on query optimization and distribution in a heterogeneous distributed system.

26.3.1 Distributed Query Optimization

The assumption of a homogeneous distributed system indicates that the amount of memory and CPU on each machine is the same. Because the number of machines are also know, therefore at the optimization step, two aspects of the system are known to the optimizer:

- The total amount of memory and CPU in the distributed system.
- The ratio between memory and CPU in the system as well as on each machine.

The first aspect also holds true for both a homogeneous system and a heterogeneous system. However, the second aspect is only true for a homogeneous distributed system. This information is important, because it implies that the ratio of memory and CPU is approximately the same for the whole system as well as for each machine.

Utilizing this information, the query optimizer can optimize the query based on the total amount of memory and CPU in the system. Since the

optimizer can make different choices based on the availability of both resources, the query plan chosen for a system that has sufficient memory but limited CPU would be different from the query plan chosen for a system with sufficient CPU but limited memory. Based on the property of similar memory-CPU ratio for the whole system and for each machine, the optimizer only needs to generate qualified query plans based on the total memory and CPU, and the generated query plan should in principle be suitable in proportion to each machine as well. Therefore the optimizer can work the same as in a centralized system, and any optimization problems and solutions should be applied here as well.

In my earlier work as described in Part I of this dissertation, I have studied the problem of choosing a qualified multi-join plan assuming that the joins are all equi-join and each stream only has one column involved in the predicate. This is the most simplified case, which is depicted in Figure 26.5. For such join predicates, we can apply a hash-based join for each join operation in the query plan, be it an mjoin operator, a btree or a mjoin tree. The query plan for the mjoin and btree are shown in Figure 26.5. Each operator state is marked by the hash key used for that hash table.

We can see that in this case, the memory cost of an mjoin operator is sure to be smaller than the memory cost of a semantically equivalent btree, because the intermediate results are not stored in the mjoin. On the other hand, the mjoin may need to recompute these intermediate results and may thus result in higher CPU costs.

We now consider two other types of common join predicates. The first type is an equi-join but with at least one stream that has more than one col-

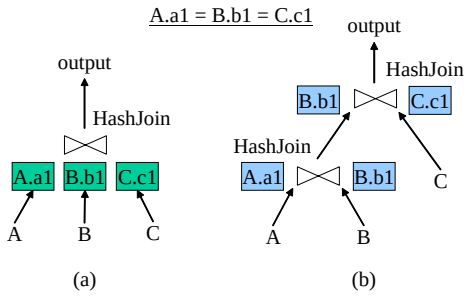


Figure 26.5: Multi-way Join Plan with Same-Column Equi-Join Predicates.

umn involved in the join predicates. An example three-way join with this type of join predicates is depicted in Figure 26.6. In the mjoin on the left, since input B has two columns, b1 and b2, involved in the join predicates, in order to utilize a hash-based join, we need to store two states for input B, with each having a different hash key. An alternative mjoin operator is depicted in the middle. This mjoin keeps hash tables for inputs A and C, but instead keeps a FIFO queue in the state for input B. Any tuple that probes the state of B would join with the tuples in that state using NLJ. However, a tuple B is able to join with tuples in state A and C using hash-based join. So this mjoin operator applies a mix of hash-based join and NLJ. The third query plan on the right is a binary tree. Here both join operators in the query plan can still apply hash-based joins. Here the intermediate state at the left side of the upper join uses a different hash key, in this case B.b2, from the hash key used in the right state of the lower join, which is B.b1.

For such join predicates, the mjoin operator that uses a pure hash-based join may have to store duplicate copies of a state, thus increasing its memory cost. If the number of intermediate results of joining tuples from A and

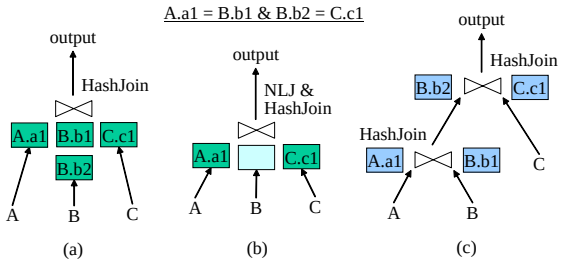


Figure 26.6: Multi-way Join Plan with Different-Column Equi-Join Predicates.

B is small, it is possible that the btree may have both less memory and less CPU cost than the mjoin operator on the far left. Using the mixed mjoin operator in the middle can be a better choice. However, for a distributed query plan, the drawback of using a NLJ is that it is difficult to apply partitioned parallelism, while it is much easier to apply this parallelism to a hash-based join because we can partition both the states and the input data into non-overlapping partitions. So we may favor hash-based join over the NLJ when generating a query plan intended for later distribution.

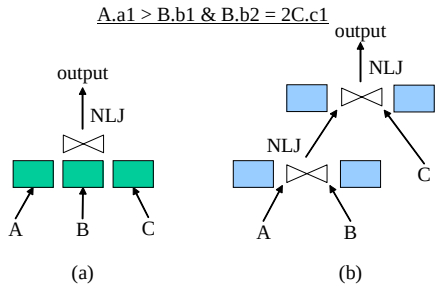


Figure 26.7: Multi-way Join Plan with Complex Join Predicates.

The join predicates can also be more complex, such as the ones shown in Figure 26.7. In this case, we may only apply NLJ to all the join operators

in the query plan.

After the query optimization step, the output is an annotated query plan, as depicted in Figure 26.8, that contains a profile inside each operator. The profile indicates the types of join used in this operator, as well as the estimated memory and CPU cost required by this operator. The links between two operators are also marked by the estimated network communication cost if the two operators were to be placed on two different machines.

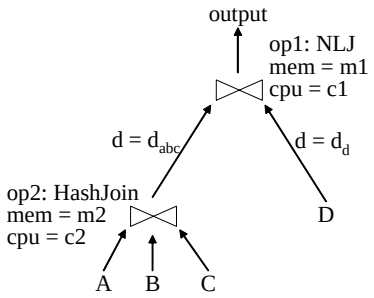


Figure 26.8: Annotated Query Plan.

26.3.2 Distributed Query Allocation

The annotated query plan generated from the optimization stage is then allocated to multiple processors by a distributor. Two possible approaches can be applied here.

In the first approach, we aim to put the query plan on as few machines as possible. We can start by allocating the whole query plan on one machine. If the machine is overloaded, we select part of the query plan and distribute it to another machine. When putting two connecting operators

on different machines, we need to add CPU cost to both the parent operator and the children operators corresponding to the CPU cost spent on receiving and sending tuples. This is shown in Figure 26.9. The CPU costs of both the join operators are increased when the lower join is distributed to a different machine. Note that an operator can be further partitioned using partitioned parallelism. Then we create a copy on each machine that it is distributed to. For example, as shown in Figure 26.9, two copies of the lower join operator are put on two different machines, with each copy only processing part of the data, and the states of the join operator are also divided into two parts based on the partition function. For such a case, the memory cost and CPU cost of the operator should both be updated: the memory is decreased by half and the CPU is also decreased by half, but the communication costs is increased.

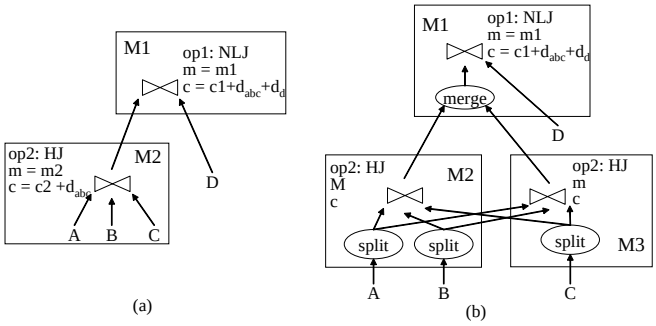


Figure 26.9: Query Distribution and Cost Updates.

The advantage of such a query allocation approach is that the communication costs between operators may be kept minimal. However, the drawback is that some machines are fully loaded, while others may be idle. This

can result in frequent runtime load rebalancing when one machine is detected to be overloaded due to changes in the data streams.

Another possible approach is to first divide the total cost of the query plan by the number of machines in the system, and then try to divide the query plan into smaller pieces with each piece have the same amount of costs. An obvious advantage of this approach is that all machines are utilized in the system. However, if the workload is not large, all machines may be under-utilized. To solve this problem, we may use a load factor when dividing the query plan. Instead of dividing the total cost of the query plan by the total number of machines in the system, we may decrease the number of machines to ensure that each machine used for executing the query is loaded more than the load factor.

The two allocation approaches described above have different performance goals and may as well have different performances. It would be interesting to compare these two and see which one has better performances under what conditions.

We may need to re-optimize the query if a valid distribution plan cannot be formed. This is usually caused by an operator being too large to fit on one machine and thus pipelined parallelism not being usable, or the total memory and CPU costs exceeding the system available resources. The former can be fixed by re-optimization, while the latter needs more resources or needs to push part of the data to disk in order to execute the query. The former can be done by a feedback loop to inform the optimizer which part of the plan is too large to fit on one machine. Actually, at the initial optimization stage, we may add a check step to make sure that the part that

cannot utilize partitioned parallelism should not be exceeding the memory and CPU cost on a single machine.

26.3.3 More Possible Future Work

Several other critical issues still remain unaddressed, which would be interesting future work as well.

How to efficiently apply partitioned parallelism to NLJ operators is a challenge for continuous queries. NLJ requires that one tuple has to probe the complete state in order to generate complete results. Putting states of a NLJ operator on different machines may dramatically increase the communication costs. It would be interesting to investigate when and how to apply such partitioned parallelism.

Another problem is the query optimization and query distribution in a heterogeneous distributed system, where the ratio of memory and CPU can vary on different machines. A possible approach is equip the optimizer with detailed information regarding the distribution of the memory and CPU in the system. The optimizer may then divide the query plan into clusters, with each cluster having different memory and CPU requirements.

Bibliography

[AAB+05] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Heong-Hyon Hwang, Walfgan Lindner, Anurag S. Maskey, Alexander Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stan Zdonik. The design of the borealis stream processing engine. In *Proceedings of CIDR*, 2005.

[ABB+03] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, I. Nishizawa, J. Rosenstein, and J. Widom. STREAM: The stanford stream data manager. In *SIGMOD Demonstration*, June 2003.

[ABW03] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The cql continuous query language: Semantic foundations and query execution. In *Stanford University Computer Science Department Technical Report*, 2003.

[ACC+03] D. Abbadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A new model and architecture for data stream management. *VLDB Journal*, pages 120–139, 2003.

[AH00] Ron Avnur and Joseph M. Hellerstein. Eddies: continuously adaptive query processing. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 261–272. ACM Press, 2000.

[AN04] Ahmed Ayad and Jeffrey F. Naughton. Static optimization of conjunctive queries with sliding windows over infinite streams. In *ACM SIGMOD*, pages 419–430, June 2004.

[Ant96] G. Antoshenkov. Dynamic optimization of index scan re-

stricted by booleans. In *Proceedings of the IEEE Conference on Data Engineering*, pages 430–440, 1996.

[BBD+02a] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proceedings of PODS*, pages 1–16, 2002.

[BBD+02b] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–16. ACM Press, 2002.

[BDM04] Brian Babcock, Mayur Datar, and Rajeev Motwani. Load shedding for aggregation queries over data streams. In *Proceeding of ICDE*, pages 350–361, 2004.

[BKS01] S. Borzsonyi, D. Kossmann, and K. Stocker. The skyline operator. In *ICDE*, pages 421–430, 2001.

[BMM+04] Shivnath Babu, Rajeev Motwani, Kamech Munagala, Itaru Nishizawa, and Jennifer Widom. Adaptive ordering of pipelined stream filters. In *Proceeding of ACM SIGMOD Conference*, pages 407–418, 2004.

[BMW05] Shivnath Babu, Kamesh Munagala, and Jennifer Widom. Adaptive caching for continuous queries. In *Proceeding of ICDE*, March, 2005.

[CBB+03] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemal, Y. Xing, and S. Zdonik. Scalable distributed stream processing. In *Proceeding of CIDR Conference*, 2003.

[CCC+02] D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams: A new class of data management applications. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB '02)*, pages 215–226, 2002.

[CCD+03] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurphy, S. R. Madden, F. Reiss, and M. A. Shah. TelegraphCQ: Continuous dataflow processing. In *SIGMOD Demostration*, June 2003.

[CCea03] D. Carney, U. Cetintemel, and A. Rasin et al. Operator scheduling in a data stream manager. In *VLDB*, pages 838–849, 2003.

[CDN02] J. Chen, D. J. DeWitt, and J. F. Naughton. Design and evaluation of alternative selection placement strategies in optimizing continuous queries. In *Proceedings of International Conference on Data Engineering*, pages 345–356, 2002.

[CF02] S. Chandrasekaran and M. J. Franklin. Streaming queries over streaming data. In *Proceedings of VLDB Conference*, pages 203–214, 2002.

[CM95] Sophie Cluet and Guido Moerkotte. On the complexity of generating optimal left-deep processing trees with cross products. In *ICDT*, pages 54–67, January 1995.

[CS94] S. Chaudhuri and K. Shim. Including Group-By in Query Optimization. In *Proceedings of the Twentieth International Conference on Very Large Databases*, pages 354–366, Santiago, Chile, 1994.

[DH04] Amol Deshpande and Joseph M. Hellerstein. Lifting the burden of history from adaptive query processing. In *VLDB*, pages 948–959, 2004.

[DMRH04a] L. Ding, N. Mehta, E. A. Rundensteiner, and G. T. Heineman. Joining punctuated streams. In *EDBT Conference*, pages 587–604, March 2004.

[DMRH04b] Luping Ding, Nishant Mehta, Elke Rundensteiner, and George Heineman. Joining punctuated streams. In *EDBT*, pages 587–604, 2004.

[DTW00] David J. DeWitt, Feng Tian, and Yuan Wang. Niagaracq: a scalable continuous query system for internet databases. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 379–390. ACM Press, 2000.

[EHJ+96] O. Etzioni, S. Hanks, T. Jiang, R. M. Karp, O. Madari, and O. Waarts. Efficient information gathering on the internet. In *IEEE Symp. on Foundations of Computer Science*, pages 234–243, 1996.

[GBLP96] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-total. In *ICDE*, pages 152–159, 1996.

[GC94] G. Graefe and R. Cole. Optimization of dynamic query evaluation plans. In *Proceedings of ACM-SIGMOD Conference*, pages 150–160, 1994.

[GI96] Minos N. Garofalakis and Yannis E. Ioannidis. Multi-dimensional resource scheduling for parallel queries. In *ACM SIGMOD*, pages 365–376, 1996.

[GO03] L. Golab and M. Tamer Ozsu. Processing sliding window multi-joins in continuous queries over data streams. In *VLDB*, pages 500–511, September 2003.

[Gra90] G. Graefe. Encapsulation of parallelism in the volcano query processing system. In *Proceeding of ACM SIGMOD*, pages 102–111, 1990.

[HAE03] Moustafa A. Hammad, Walid G. Aref, and Ahmed K. Elmagarmid. Stream window join: Tracking moving objects in sensor-network dbs. In *SSDBM*, pages 75–84, 2003.

[Has95] W. Hasan. Optimizing response time of relational queries by exploiting parallel execution. In *Ph.D Thesis, Stanford University*, 1995.

[HFAE03] M. A. Hammad, M. J. Franklin, W. G. Aref, and A. K. Elmagarmid. Scheduling for shared window joins over data streams. In *Proceedings of VLDB Conference*, pages 297–308, 2003.

[HH99] P. J. Hass and J. M. Hellerstein. Ripple joins for online aggregation. In *Proceedings of ACM-SIGMOD Conference*, pages 287–298, 1999.

[HM94] W. Hasan and R. Motwani. Optimization algorithms for exploiting the parallelism-communication tradeoff in pipelined parallelism. In *Proceeding of VLDB*, pages 36–47, 1994.

[HMA+04] Moustafa A. Hammad, Mohamed F. Mokbel, Mohamed H. Ali, Walid G. Aref, and et. al. Nile: A query processing engine for data streams. In *ICDE*, page 851, 2004.

[Hon92] W. Hong. Parallel query processing using shared memory multiprocessors and disk arrays. In *Ph.D Thesis, University of California, Berkeley*, August, 1992.

[HS91] W. Hong and M. Stonebraker. Optimization of parallel query execution plans in xprs. In *Proceeding of the first international conference on Parallel and Distributed information systems*, pages 218–225, 1991.

[IHW02] Z. G. Ives, A. Y. Halevy, and D. S. Weld. An xml query engine for network-bound data. In *VLDB Journal*, pages 11(4): 380–402, 2002.

[IK84] T. Ibaraki and T. Kameda. On the optimal nesting order for computing n-relational joins. In *ACM Transaction on Database Systems*, pages 9(3):482–502, 1984.

[IK91] Y. E. Ioannidis and Y. C. Kang. Left-deep vs. bushy trees: An analysis of strategy spaces and its implications for query optimization. In *Proceeding of ACM SIGMOD*, pages 168–177, Denver, USA, May 1991.

[ILW+00] Zachary G. Ives, Alon Y. Levy, Daniel S. Weld, Daniela Florescu, and Marc Friedman. Adaptive query processing for internet applications. *IEEE Data Engineering Bulletin*, 23(2):19–26, 2000.

[INSS92] Y. Ioannidis, R. T. Ng, K. Shim, and T. Sellis. Parametric query optimization. In *Proceedings of 18th VLDB Conference*, pages 103–114, 1992.

[Ive02] Zachary G. Ives. Efficient Query Processing for Data Integration. Ph.D Dissertation, University of Washington, Auguest, 2002.

[KBZ86] R. Krishnamurthy, H. Boral, and C. Zaniolo. Optimization of non-recursive queries. In *Proceeding of VLDB*, pages 128–137, 1986.

[KD98] N. Kabra and D. J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *Proceedings of ACM-SIGMOD Conference*, pages 106–117, 1998.

[KNV03] J. Kang, J. F. Naughton, and S. D. Viglas. Evaluating window joins over unbounded streams. In *Proceedings of ICDE Conference*, pages 341–352, 2003.

[LMT+05] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. Semantics and evaluation techniques for window aggregates in data streams. In *ACM SIGMOD*, pages 311–322, 2005.

[LVZ93] R. Lanzelotte, P. Valduriez, and M. Zait. On the effectiveness of optimization search strategies for parallel execution spaces. In *Proceeding of VLDB*, pages 493–504, September, 1993.

[LZJ+05] Bin Liu, Yali Zhu, Mariana Jbantova, Bradley Momberger, and Elke Rundensteiner. A dynamically adaptive distributed system for processing complex continuous queries. In *VLDB Demonstration*, pages 1338–1341, 2005.

[LZR06] Bin Liu, Yali Zhu, and Elke A. Rundensteiner. Run-time operator state spilling for memory intensive long-running queries. In *ACM SIGMOD*, pages 347–358, 2006.

[MJSM03] M.A.Shah, J.M.Hellerstein, S.Chandrasekaran, and M.J.Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *Proceeding of ICDE*, pages 25–36, 2003.

[MRS+04] Volker Markl, Vijayshankar Raman, David E. Simmen, Guy M. Lohman, and Hamid Pirahesh. Robust query processing through progressive optimization. In *ACM SIGMOD*, pages 659–670, 2004.

[MS79] C. Monma and J. Sidney. Sequencing with series-parallel precedence constraints. In *Mathematics of Operations Research*, pages 4:215–224, 1979.

[MSHR02] S. Madden, M. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *Proceedings of ACM-SIGMOD*, pages 49–60, 2002.

[MWA+03] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, resource management, and approximation

in a data stream management system. In *Proceedings of the First Biennial Conference on Innovative Data Systems Research (CIDR 2003)*, pages 245–256, 2003.

[NWAea02] R. Notwani, J. Widom, A. Arasu, and et al. Query processing, appromixation, and resource management in a data stream management system. In *Proceedings of CIDR Conference*, pages 1–16, January 2002.

[NWMN99] K. W. Ng, Z. Wang, R. R. Muntz, and S. Nittel. Dynamic query re-optimization. In *Proceedings of International Conference on Scientific and Statistical Databases*, pages 264–273, July 1999.

[NWMS98] K. W. Ng, Z. Wang, R. R. Muntz, and E. C. Shek. On reconfiguring query execution plans in distributed object-relational dbms. In *Proceedings of International Conference on Parallel and Distributed Systems*, pages 59–66, 1998.

[OL90] Kiyoshi Ono and Guy M. Lehman. Measuring the complexity of join enumeration in query optimization. In *Proceeding of VLDB*, pages 314–325, 1990.

[PY01] Christos H. Papadimitriou and Mihalis Yannakakis. Multiobjective query optimization. In *PODS*, 2001.

[RDH03] V. Raman, A. Deshpande, and J. M. Hellerstein. Using state modules for adaptive query processing. In *ICDE*, 2003.

[RDS+04] Elke A. Rundensteiner, Luping Ding, Timothy Sutherland, Yali Zhu, Brad Pielech, and Nishant Mehta. Continuous Query Engine with Heterogeneous-Grained Adaptivity. In *VLDB Demo Session*, pages 1353–1356, 2004.

[SAC+79] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceeding of ACM SIGMOD*, pages 23–34, Boston, USA, May 1979.

[SAL+96] M. Stonebraker, P. Aoki, W. Litwin, A. Pfeffer, A. Sah, J. Sidell, C. Staelin, and A. Yu. Mariposa: a wide-area distributed database system. In *The VLDB Journal*, pages 5, 1(Jan.), 48–63, 1996.

[SI93] A. Swami and B. Iyer. A polynomial time algorithm for optimizing join queries. In *Proceeding of ICDE*, pages 345–354, Vienna, Austria, April 1993.

[SLMK01] M. Stillger, G. M. Lohman, V. Markl, and M. Kandil. Leo db2s learning optimizer. In *VLDB*, pages 19–28, 2001.

[SMK97] Machael Steinbrunn, Guido Moerkotte, and Alfons Kemper. Heuristic and randomized optimization for the join ordering problem. In *The VLDB Journal*, pages 6(3):191–208, 1997.

[SN95] Ambuj Shatdal and Jeffrey F. Naughton. Adaptive parallel aggregation algorithms. In *SIGMOD*, pages 104–114, 1995.

[Ste86] R. E. Steuer. Multiple criteria optimization. Wiley, New York, NY, 1986.

[SZDR05] Timothy M. Sutherland, Yali Zhu, Luping Ding, and Elke A. Rundensteiner. An Adaptive Multi-Objective Scheduling Selection Framework for Continuous Query Processing. In *IDEAS*, pages 445–454, 2005.

[TCZ+03] N. Tatbul, U. Cetintemel, S. B. Zdonik, M. Cherniack, and M. Stonebraker. Load shedding in a data stream manager. In *Proceedings of VLDB Conference*, pages 309–320, 2003.

[TMSF03a] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras. Exploiting punctuation semantics in continuous data streams. *IEEE Transactions on Knowledge and Data Engineering*, 15(3):555–568, May/June 2003.

[TMSF03b] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras. Exploiting punctuation semantics in continuous data streams. In *IEEE Transactions on Kowledge and Data Engineering*, pages 15(3):555–568, May/June 2003.

[UF99] T. Urhan and M. Franklin. XJoin: Getting fast answers from slow and bursty networks. Technical Report CS-TR-3994, University of Maryland, 1999.

[VM96] B. Vance and D. Maier. Rapid bushy join-order optimization with cartesian product. In *Proceeding of ACM SIGMOD*, Montreal, Canada, June, 1996.

[VN02a] S. D. Viglas and J. F. Naughton. Rate-based query optimization for streaming information sources. In *Proceedings of ACM-SIGMOD*, pages 37–48, 2002.

[VN02b] Stratis Viglas and Jeffrey F. Naughton. Rate-based query optimization for streaming information sources. In *ACM SIGMOD*, pages 37–48, 2002.

[VNB03] Stratis Viglas, Jeffrey F. Naughton, and Josef Burger. Maximizing the output rate of multi-way join queries over streaming information sources. In *Proceeding of VLDB*, pages 285–296, 2003.

[WA93a] Annita N. Wilschut and Peter M. G. Apers. Dataflow query execution in a parallel main-memory environment. *Distributed and Parallel Databases*, 1(1):103–128, 1993.

[WA93b] Annita N. Wilschut and Peter M. G. Apers. Dataflow query execution in a parallel main-memory environment. *Distributed and Parallel Databases*, 1(1):103–128, 1993.

[WW94] Carl A. Waldspurger and William E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Operating Systems Design and Implementation*, pages 1–11, 1994.

[YL94] Weipeng P. Yan and Paul Larson. Performing Group-By before Join. In *Proceedings of the 10th IEEE International Conference on Data Engineering*, pages 89–100, Houston, Texas, 1994.

[ZRH04] Yali Zhu, Elke A. Rundensteiner, and George T. Heineman. Dynamic plan migration for continuous queries over data streams. In *ACM SIGMOD*, pages 431–442, Paris, France, June 2004.