

**State-Slice: A New Stream Query
Optimization Paradigm for Multi-query and
Distributed Processing**

by

Song Wang

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

March 17, 2008

APPROVED:

Professor Elke A. Rundensteiner
Advisor

Professor George Heineman
Committee Member

Professor Murali Mani
Committee Member

Professor Ugur Çetintemel
Brown University
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. Modern stream applications such as sensor monitoring systems and publish/subscription services necessitate the handling of a large number of continuous queries specified over high volume data streams. Continuous queries utilize constraints, such as windows, to unblock stateful and otherwise blocking query operations. The window constraints impose new challenges for effective query processing. This dissertation proposes novel solutions to continuous query optimization based on state-slicing in three core areas, namely multiple continuous query sharing, ring-based multi-way join query distributed processing and distributed multi-query optimization.

The first part of the dissertation proposes efficient optimization strategies that utilize the novel state-slicing concept to achieve optimal memory and computation sharing for multiple stream join queries with different window constraints. Extensive analytical and experimental evaluations demonstrate that the proposed strategies can minimize the memory or CPU consumptions for multiple join queries.

The second part of this dissertation proposes a novel scheme for the distributed execution of generic multi-way joins with window constraints. The proposed scheme partitions the states into disjoint slices in the time domain, and then distributes the fine-grained states in the cluster, forming a virtual computation ring. New challenges to support this distributed state-slicing processing are answered by numerous new techniques, including synchronized processing on slices without locking in asynchronous cluster, maintenance and termination logic in ring-based query plan, interleaving of join runs, cost-based state allocation, and distributed time-slice adaptation. The extensive experimental evaluations show that the proposed strategies achieve significant performance improvements in terms of response time and memory usages for a wide range of configurations and workloads on a real system compared to the state-of-the-art distributed processing techniques.

Ring-based distributed stream query processing and multi-query sharing both are based on the same state-slice concept. The third part of this dissertation combines the first two parts of this dissertation work and proposes a novel distributed multi-query optimization technique.

Acknowledgments

This dissertation and the growth in my knowledge over the last few years owe a great deal to many professors, colleagues, and friends. First, I want to thank my advisor, Professor Elke A. Rundensteiner, for all her support during the process of my Ph.D. study. She inspired my interests in database research and gave me direction on every aspects of research. It has been my luck to have her as my advisor. Her technical and editorial advice was essential to the completion of this dissertation. I express my sincere thanks for her support, advice, patience, and encouragement throughout my graduate studies. Her persistence in tackling problems, confidence, and great teaching will always be an inspiration.

My thank goes to the members of my Ph.D. committee, Prof. Murali Mani, Prof. George Heineman and Prof. Ugur Çetintemel, who provided valuable feedback and suggestions to my comprehensive-exam, my dissertation proposal talk and dissertation drafts. All these helped to improve the presentation and contents of this dissertation. I thank Prof. Kathi Fisler for her time and efforts discussing with me in my research qualifying exam.

I would like to thank the whole CAPE and D-CAPE team members,

previous and current, in DSRG for their hard work on building the system that we can share together as a team. I would also like to thank the Rainbow and Raindrop team members for their help in my early Ph.D. study. The friendship of Yali Zhu, Ling Wang, Bin Liu, Andreas Koeller, Li Chen, Hong Su, Xin Zhang, Songting Chen, Maged El-Sayed, Luping Ding, Rimma V. Nehme, Jinghui Jian and all the other previous and current DSRG members is much appreciated. They have contributed to many interesting and good-spirited discussions related to this research.

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, the school and NEC Labs America during my study and research in WPI. My thank also goes to NSF for providing funding for the computing cluster used in my dissertation.

Finally, I would like to thank my wife Mingyu for her understanding and love during the past few years. Her support and encouragement was in the end what made this dissertation possible. My parents receive my deepest gratitude and love for their dedication and the many years of support during my studies. Special thanks also go to my little daughter Allison, her sweet smile comes along with me forever.

Contents

1	Introduction	1
1.1	Research Motivation	1
1.1.1	Continuous Query Optimization in General	1
1.1.2	New Challenges in Continuous Query Processing	4
1.1.3	Motivation for Operator Granularity Control	6
1.1.4	Relation with State-of-the-Art Stream Query Processing Techniques	8
1.2	Research Focus of This Dissertation	11
1.2.1	Multiple Continuous Query Optimization	13
1.2.2	Distributed Multi-way Stream Join Query Optimization	18
1.2.3	Distributed Multiple Query Processing	24
1.2.4	Overview of the CAPE/D-CAPE System	26
1.3	Dissertation Road Map	28
I	State-Slice Multi-query Optimization of Stream Queries	29
2	Introduction	30
2.1	Research Motivation	30
2.2	Proposed Strategies	33
2.3	Road Map	36
3	Background	37
3.1	Stateful Operators in Continuous Queries	37
3.2	Window Constraints and Sliding Window Join	39
3.3	Assumptions and Simplifications	42
4	Review of Existing Strategies for Sharing Continuous Queries	45
4.1	Naive Sharing with Selection Pull-up	47
4.2	Stream Partition with Selection Push-down	49

5	State-Slice Sharing Paradigm	52
5.1	State-Sliced One-Way Window Join	53
5.2	State-Sliced Binary Window Join	59
5.3	Discussion and Analysis	65
6	Case Study: State-Slice Sharing for Join Tree	70
6.1	Continuous Queries and Terms Used in Cost Model	71
6.2	Strategies of Sharing Queries	74
6.2.1	Selection PullUp Sharing	74
6.2.2	State-Slice Sharing	76
6.3	Cost Model for State Memory Consumption	77
6.3.1	Isolated Execution without Sharing	77
6.3.2	Selection PullUp Sharing	79
6.3.3	State-Slice Sharing	80
6.3.4	Comparison and Analysis	81
6.4	Cost Model for CPU Consumption	84
6.4.1	Isolated Execution without Sharing	85
6.4.2	Selection PullUp Sharing	86
6.4.3	State-Slice Sharing	87
6.4.4	Comparison and Analysis	88
7	State-slice: Building the Chain	92
7.1	Memory-Optimal State-Slicing and its Cost Analysis	93
7.2	CPU-Optimal State-Slicing	95
7.3	Online Migration of the State-Slicing Chain	98
7.4	Push Selections into Chain	101
7.4.1	Mem-Opt Chain with Selection Push-down	101
7.4.2	CPU-Opt Chain with Selection Push-down	103
8	Experimental Evaluation	105
8.1	Experimental System Overview	105
8.2	State-Slice vs. Other Sharing Strategies	106
8.3	State-slice: Mem-Opt vs. CPU-Opt	110
9	Related Work	113
II	Distributed Multi-way Stream Join Processing	116
10	Introduction	117
10.1	Research Motivation	117

10.2	Proposed Strategies	120
10.3	Our Contributions:	124
10.4	Road Map	125
11	Background	126
11.1	Semantics of Multi-way Window Join	126
11.2	Distributed Continuous Query Processing in DCAPE	130
12	PSP: State-Slicing for Multi-way Joins	133
12.1	New Challenges in State-slicing for Multi-way Joins	133
12.2	State-Slice Ring with Life Control	137
12.2.1	Coordinated State Maintenance.	140
12.2.2	Intermediate Result Propagation and Processing.	143
12.2.3	Life Span Control in the Ring.	145
12.3	Execution Algorithm and Time Line	148
12.4	PSP with Interleaved Processing	151
12.5	PSP with Dynamic Head and Tail	153
12.6	Interleaving Processing with Dynamic Ring Structure	155
13	PSP: Cost Analysis and Tuning	157
13.1	Cost Model	157
13.2	Cost-based Tuning	160
13.2.1	Maximize Output Rate.	160
13.2.2	Minimize Average Response Latency.	161
13.3	Initial State Slicing	162
13.4	Workload Balancing	163
14	PSP: Adaptive Load Diffusion	164
14.1	Workload Smoothing	165
14.2	State Relocation	166
15	Discussion	169
15.1	State Replication Based Distribution	169
15.2	Stream Tuple Processing Order	173
15.2.1	Execution Models	174
15.2.2	State Sliced Join Processing with Semi-Ordered Execution	176

16 Experimental Evaluation	180
16.1 Experiment Settings	180
16.2 Experiment 1: Sensitivity Analysis for PSP	182
16.3 Experiment 2: PSP with Interleaved Processing	185
16.4 Experiment 3: PSP vs. ATR and CTR	187
16.5 Experiment 4: Runtime Adaptation of PSP	190
17 Related Work	192
III Distributed Multiple Multi-way Join Query Optimization	194
18 Introduction	195
18.1 Research Motivation	195
18.2 Proposed Strategies	196
18.3 Road Map	199
19 Selection Pushdown for Multi-way Join	200
19.1 Selection Pull Up and Window based State Slicing	201
19.2 Selection Push Down	202
20 Routing the Joined Results	207
20.1 Routing Bitmaps for the Logical Window Slices	208
20.2 Bitmaps for Evaluation of the Selections	210
21 Logical Query Plan Deployment in the Cluster	211
21.1 Extended Cost Model	212
21.2 Minimize Average Response Latency	213
21.3 Workload Balancing	213
IV Conclusions and Future Work	215
22 Conclusions of This Dissertation	216
23 Future Work	221
23.1 State Slicing Aware Continuous Query Optimization	221
23.2 Computation Sharing for Complex Event Query Processing	222
23.3 Approximate Continuous Query Processing	223

List of Figures

1.1	Continuous Query Plan Network.	4
1.2	Overall Research Focus.	12
1.3	D-CAPE System Architecture.	27
2.1	A Brute Force State Slicing with Incomplete Result	34
3.1	Sliding Window Join Operators and Their States	38
3.2	Execution of Sliding-window join.	41
4.1	Query Plans for Q_1 and Q_2	46
4.2	Selection Pull-up.	48
4.3	Selection Push-down.	50
5.1	Sliced One-Way Window Join.	54
5.2	Execution of $A[W^{start}, W^{end}] \times^s B$	54
5.3	Chain of 1-way Sliced Window Joins.	55
5.4	Chain of Binary Sliced Window Joins.	60
5.5	Execution of Binary Sliced Window Join.	61
5.6	State-Slice Sharing for Q_1 and Q_2	64
5.7	The Processing of the Union Operator.	67
5.8	Memory Consumption Comparison	67
5.9	CPU Cost Comparison: State-Slice vs. Selection PullUp . . .	68
5.10	CPU Cost Comparison: State-Slice vs. Selection PushDown. . .	68
6.1	Query Plans for Q_1 and Q_2	71
6.2	Selection PullUp Sharing Query Plan.	75
6.3	State-Slice Sharing for Q_1 and Q_2	76
6.4	Memory Consumption Comparison: State-Slice Sharing vs. Selection PullUp Sharing.	83

6.5	Memory Consumption Comparison: State-Slice Sharing vs. Isolated Execution.	83
6.6	CPU Cost Comparison: State-Slice Sharing vs. Selection PullUp Sharing.	90
6.7	CPU Cost Comparison: State-Slice Sharing vs. Isolated Execution.	91
7.1	Mem-Opt State-Slice Sharing.	93
7.2	Merging Two Sliced Joins by Introducing Router Operator.	96
7.3	Directed Graph of State-Slice Sharing.	97
7.4	Online Splitting of the Sliced Join J_i	98
7.5	Online Merging of the Sliced Join J_i and J_{i+1}	100
7.6	Selection Push-down for Mem-Opt State-Slice Sharing.	102
7.7	Merging Sliced Joins with Selections.	104
8.1	Memory Comparison with Various Parameters	108
8.2	Service Rate Comparison with Various Parameters	109
8.3	Service Rate Comparison of Mem-Opt. Chain vs. CPU-Opt. Chain	111
11.1	Binary Join Trees	128
11.2	Multi-way Join Operator for Query $A \bowtie B \bowtie C \bowtie D \bowtie E$	129
11.3	Pipelined parallelism and Partitioned Parallelism	130
11.4	Example of Data Partitioned Plan Distribution	132
12.1	Ring-based Query Plan with Multi-way State-slice Joins.	136
12.2	Snapshot of Runtime State Deployment in the Ring-based Query Plan. The Current Sliding Window is Composed of the Colorful/Gray Slots for Each Stream.	139
12.3	Execution Steps of Sliced Join op_i	149
12.4	Execution Time Line of the PSP	150
12.5	Purge Steps with StateStart and StateEnd in $node_i, 1 \leq i \leq n$	152
12.6	PSP with Dynamic Head and Tail.	154
12.7	PSP-D with Multiple Heads and Tails.	156
14.1	Aggressive State Relocation.	167
15.1	CPU Consumption Comparison	172
16.1	Cost Breakdown for an Example 3-Way Join Query.	183
16.2	Performance Analysis of the PSP scheme	184

16.3 Performance Analysis of the PSP-Int scheme	186
16.4 Performance Analysis of the ATR scheme	188
16.5 Performance Analysis of the CTR scheme	189
16.6 Experimental Results of Adaptation	190
16.7 Experiments Results of Adaptation	191

List of Tables

4.1	System Settings Used in Chapter 4.	47
5.1	Execution of the Chain: J_1, J_2	56
6.1	Terms Used in Cost Model	73
6.2	Value of Terms Used in Cost Model	78
8.1	System Settings Used in Chapter 8.2.	107
8.2	Window Distributions Used for 12 Queries.	110
13.1	Terms Used in Cost Model	158

Chapter 1

Introduction

1.1 Research Motivation

1.1.1 Continuous Query Optimization in General

Over the past decades, database systems have emerged as the core technology for managing data [RG00]. After many years of development, relational database technology has matured and contributed significantly to the rapid growth of various industries. Relational database management systems (DBMS) are a proven technology for managing business data [RG00]. Commercial relational database products, such as Oracle, DB2, Microsoft SQL Server, Sybase, PostgreSQL, MySQL and etc., embody years of research and development in areas as diverse as modeling, storage, retrieval, update, indexing, transaction processing, and concurrency control, to just name a few. Work continues to add capabilities to a DBMS to address new kinds of data in the past decade, such as multimedia [MS96], object-

oriented [CA93], spatial-temporal data types [PdBG94, JCE⁺94], XML and semistructured data [BPSM97], and most recently stream data types [BW01].

Recently, the development of the web and network techniques also has necessitated the widely used stream data processing. Recent years have witnessed a rapidly increasing attention on streaming database systems [MWA⁺03, BMW02, ACC⁺03, AH00, DTW00, VN02, ILW⁺00, AAB⁺05a]. Different from traditional database systems with statically stored data and one-time queries, in a streaming database, data are streaming in as time goes by. User queries are generally long-running or even continuous, and the results of the queries are also in the format of streams. This type of query is generally referred to as a *continuous query*. Many applications require the processing of continuous queries on streaming data, including sensor networks, online financial tickers and medical monitoring systems. Hence a database system that specialized in processing streaming data and continuous queries is likely to be beneficial for a large range of applications.

Efficiency is the key point in all these above data processing systems because of the tremendous data size. Modern relational databases usually host several TB data. Query optimization is a core component of any database system. Query optimization has been intensively studied for decades for the now mature relational databases [RG00, Cha98, Ioa96]. The well known optimization techniques include query plan rewriting, answering queries using materialized views, sharing computation for multi-queries and others [Cha98]. These optimizations are generally cost-based and algebraic-based [Cha98]. Query optimization aims to produce an efficient execution plan.

Continuous queries significantly differ from traditional static queries in several aspects. (1) Data availability. For traditional relational queries, data is stored *a priori* on disk; while the stream data arrives at the system on-the-fly. This means that various data access methods with well-studied indices existing for relational databases may not be valid for stream data. (2) Query execution mode. Users of relational databases submit one-time queries against the data in the tables. A valid relational query should not run indefinitely on finite data. On the contrary, users register a set of continuous queries on the incoming streams. These queries will be running in the query engine until the user explicitly deactivates them. (3) Result generation. Generation of query results for relational queries are driven by the execution steps in a pull-based fashion, which is usually defined by the *iterator* interfaces of the operators. However, when stream data arrives at runtime, the query processing will be driven by the data and output the update of results in a push-based fashion.

Figure 1.1 shows a simplified architecture of a continuous query engine. As illustrated, the stream data arrives on the fly, while the result is also streaming out of the system at runtime. A streaming continuous query system usually hosts multiple registered continuous queries having the same input streams. Since all these queries are continuous queries, they have to be executed simultaneously instead of sequentially. The corresponding query operators of the registered query form a query network, which is a *directed acyclic graph* of operators. All the operators in the query network will be connected with queues to buffer intermediate results temporarily. The final result will be sent out as streaming result.

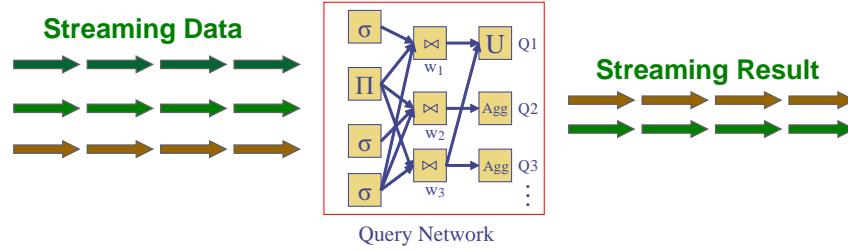


Figure 1.1: Continuous Query Plan Network.

The stream data model is different from the well-studied relational data model in the sense of on-the-fly arrival. The corresponding query models bring new semantics, such as timestamps and window constraints. All these differences require us to revisit the traditional database processing techniques in the streaming database scenario, since the former 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 continuous queries.

1.1.2 New Challenges in Continuous Query Processing

The optimization of continuous query processing [MWA⁺03, MSHR02, CCC⁺02] differs from traditional query optimization in several aspects. Below we list several aspects of the differences and illustrate the new challenges for continuous query processing.

First, the quality of a continuous query plan is typically judged by its runtime performance measurements, including output rate [VN02] and response time. In a static database, the quality of a query plan is often judged by its total estimated execution cost measured in terms of CPU process-

ing and disk I/O costs [SMK97]. Since ideally the estimated execution cost might be a “good” indication of actual query execution time in a real system, the query optimizer usually picks the query plan that has the minimal estimated cost. However for a stream query, the query execution time is not essential because of the long running nature. New performance indicators are defined then for continuous query processing. As observed in [AN04], a continuous query plan produces the optimal throughput without shedding as long as the system can process all incoming stream data within the stream arrival rates. When the continuous query engine faces high-volume input streams, it is thus critical to devise methods to catch up with the stream speed.

Second, continuous queries are usually main-memory-resident to satisfy 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 CPU-intensive and memory-intensive. When the system is overloaded, we have to either spill in-memory data to disk [LZR06, UF00, 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 demand accurate real-time results, the query optimizer instead should aim to generate query plans with minimal memory and CPU costs.

Third, continuous query must be aware of constraints such as the window constraints, which is new semantics of stream queries. Window constraints for stream data can be time-based [BW01], tuple-based [BW01] or

punctuation-based [TMSF03]. The window constraints are used to unblock the stateful operators from infinite waiting time before generation of any result. The window constraints determine the memory consumption of the stateful operators, and along with it the CPU resources needed to process the tuples in the states. Usually the CPU cost is proportional, quadratic or even higher degree to the window constraints, since the complexity of the operators are usually not linear.

Lastly, the statistics of the streams are usually unknown before a query starts. In fact they may continue to change during the query execution. Thus a query plan that is currently optimal can become sub-optimal at a later time. Therefore, runtime optimization is critical and inevitable. Thus the initial generated query plan must be flexible for adaptive continuous query processing. More importantly, effective runtime query optimization methods must be developed.

1.1.3 Motivation for Operator Granularity Control

Query optimization is one of the most critical techniques for improving query performance in any database system. Among these techniques the optimization of join queries, especially for the multi-way joins with arbitrary join graphs, is essential since join operations tend to dominate the CPU and memory usage in a database system [Gra93, KRB85]. For stream query optimization, the real-time query response requirement and in-memory processing of stream operators exacerbate the situation.

In stream processing, the CPU and memory usage is directly related to the window constraints. The join and group-by operators are *stateful* opera-

tors. A stateful operator must store all tuples that have been processed thus far from input streams so to be able to join or group them with future incoming tuples from the other input streams. For a long-running query as in the case of continuous queries, the number of tuples stored inside a stateful operator can potentially be quite large for large window constraints. Such operators are serious obstacles for stream query optimizations.

- First, a huge operator limits the granularity of query optimization, since an operator is the basic unit for query rewriting. Window constraints for the stateful operators add semantics beyond the relational query model. Since the operator is the basic unit for the query optimizer to work on, a huge stateful operator limits the scope and effectiveness of the query optimizer in terms of rule based query rewriting. Localized intra micro operator optimization may be possible but cannot achieve plan level optimality.
- Second, a huge stateful operator brings new issues for operator scheduling strategies, since an operator is the minimal unit for an execution thread scheduled in a continuous query engine. In the case when huge stateful operators exist in the query plan and consume most of the CPU time at runtime, special care must be taken to avoid starvation of other operators.
- Third, a huge stateful operator is not suitable for distributed query processing, since an operator is the basic unit for parallelism. Distributed stream query processing is a natural direction when the input stream arrival rates and stream query processing requirements

go beyond the ability of a single processor. However, huge stateful operators may be too large to fit into any single processor and thus they must be split.

In summary, the size of the operators determines the granularity of query processing in almost all aspects. Operators of fine granularity provide potentially more opportunities for the runtime query optimizer and query execution engine.

In this dissertation, I propose a novel solution of slicing the states in the time domain called *state slicing*, designed to split a huge stateful operators into a group of smaller stateful operators at the optimizer's will. Our proposed method is generic in the sense that the key idea of state slicing does not rely on the query semantics such as type of predicates, attribute domain and attribute distribution. Our solution is versatile and generic for arbitrary join predicates, with minimal extra cost. Based on the state slicing concept, we show solutions of two important problems, namely, computation sharing among stream queries with overlapped window constraints and distributed query processing of generic stateful join queries.

1.1.4 Relation with State-of-the-Art Stream Query Processing Techniques

Many aspects of stream query processing techniques have been proposed and studied recently. Listed below are the most commonly used techniques in current continuous query systems [MWA⁺03, BBD⁺02, ACC⁺03, AH00, DTW00, VN02, ILW⁺00, AAB⁺05a].

- *Adaptive continuous query processing.* Since several important parameters (such as characteristics of the incoming streams, system workload and registered queries) may change during the usually long execution of a continuous query, runtime query optimization is necessary for stream query processing. Existing adaptations including: (1) dynamic tuple routing through operators such as Eddies [AH00, TD03] and content based routing [BBDW05], (2) dynamic operator scheduling such as Chain [DBBM03] and [CCea03, L. 00, PSR03], (3) intra-operator adaptation such as XJoin [UF00] and PJoin [DMRH04], (4) run-time operator scheduling [WW94, MWA⁺03, CCea03, SZDR05], (5) query plan re-optimization and migration [ZRH04], and (6) query run-time re-distribution [LZJ⁺05]. These techniques in general work as follows: (1) collect runtime stream and system statistics, (2) runtime optimize to minimize or maximize certain performance measurements, and (3) execute the evolved query plan with any necessary compensation. State-slicing based stream query optimization support runtime adaptation and can be combined with other adaptive query processing techniques.
- *Distributed stream query processing.* A share-nothing cluster has been used widely for distributed query processing in the contexts of both relational and continuous query processing. 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

such as memory and CPU resources 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 further delay the query results. Hence a streaming system needs to scale well in regards to its potentially very large workload, which cannot be achieved by a centralized system with a single machine. For distributed stream query processing, several generic challenges have been tackled: (1) operator deployment in a distributed network environment, such as [Ac04], (2) distributed plan migration, such as [ZR07], (3) fault tolerance architecture, such as [HXcZ07], (4) robust query plan deployment, such as [XHcZ06]. State slicing based operator splitting provides a novel pipeline construction approach for distributed stream query processing. State slicing is a network-aware query optimization method, which considers both the query plan optimization and operator distribution of stream join queries in one cost model. The proposed state slicing technique extends the above techniques in the sense that it provides new opportunities for dynamic operator deployment.

- *Load-shedding and Quality of Service (QoS)*. In the case of bursty input streams that exceed the current system resource limitations, some research has proposed to apply load shedding [TcZ⁺03, BDM04, TcZ07] in order to decrease the workload that the system needs to handle. The basic idea is to drop workload that has the minimal possible im-

impact on the quality of the output. As a side effect, this introduces approximation of the results that the continuous query processing engine produces, which by itself is another important research area in the streaming database field. Practically, end users would need to provide parameters to describe their requirement of the result in terms of response time, accuracy or other preferences (all these are QoS parameters). Load-shedding, especially *semantic load-shedding*, is usually considered with such QoS specifications [TcZ⁺03, TcZ07]. However for a streaming database application that needs to get exact results, approximation techniques such as load shedding are not applicable. The proposed state slicing techniques aim to provide accurate answers assuming the availability of a set of computing resources. However if the total workload goes beyond the computational resources of the cluster, load-shedding or other approximate processing approaches have to be involved to avoid system crashes.

1.2 Research Focus of This Dissertation

The overall goal of this dissertation is to build a continuous query processing system that can effectively scale up to tens of registered queries and large window constraints (≥ 30 minutes) with high-volume streams (≥ 300 tuples/second). As depicted in Figure 1.2, the main techniques discussed in this dissertation are multiple query computation sharing and distributed multi-way join query processing. The core concept of our proposed solutions is the state slicing method, which enables the split of multi-way

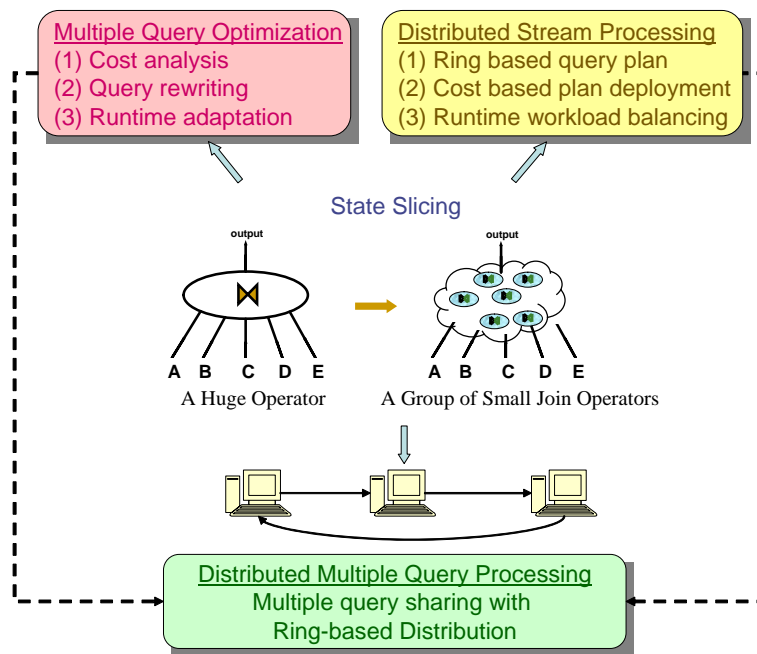


Figure 1.2: Overall Research Focus.

join operators at the optimizer's will. As discussed in Section 1.1.3, macro stream operators may be inefficient for dynamic stream processing. Intuitively if the stream query optimizer can split macro-operators into smaller micro-operators according to an optimization goal, the resulting query plan tends to be more suitable for fine grained query processing. Additionally, operator splitting may bring more optimization opportunities to the stream query. The state slicing method can transform a huge multi-way join operator into a group of small join operators inter-connected, which each imposes its own challenges with much smaller and thus more manageable CPU and memory requirements.

The *Multiple Query Optimization* shown in Figure 1.2 is an application of

the state slicing method to solve the multiple query optimization problem. State slicing is applied to achieve fine grained sharing of state memory and join computation among multiple stream queries that join the same streams with arbitrary window constraints. As a result, after cost-based query rewriting with state slicing, multiple stream queries can achieve maximum computation sharing with minimal extra cost.

The *Distributed Stream Processing* provides a novel paradigm for pipelined join processing with partitioning of the states into manageable slices to be distributed across multiple processing nodes in a shared-nothing cluster environment. This technique generates a ring shape state sliced join query plan based on a cost model. We then deploy the query plan in a cluster, with one state sliced join assigned to a processing node.

Both the above two solutions support runtime query plan optimization in terms of further slicing of the sliced joins or merging connected sliced joins. For the distributed state slicing processing, it also includes state relocation with additional or reduced processing nodes at runtime.

Lastly the *Distributed Multiple Query Processing* combines the multiple query optimization and distributed query processing techniques to provide an integrated solution for scalable stream query processing of a set of queries.

1.2.1 Multiple Continuous Query Optimization

In the first part of this dissertation, we focus on the problem of sharing of window join operators across multiple continuous queries. The window constraints may vary according to the semantics of each query. The

sharing solutions employed in existing streaming systems, such as NiagaraCQ [CDN02], CACQ [MSHR02] and PSoup [CF02], focus on exploiting common sub-expressions in queries, that is, they closely follow the traditional multi-query optimization strategies from relational technology [Sel88, RSSB00]. Their shared processing of joins ignores window constraints. That is, their approaches will treat joins with distinct window sizes as different joins and not share them.

New Challenges in Multiple Stream Query Optimization

The problem of sharing the work between multiple queries is not new. For traditional relational databases, multiple-query optimization [Sel88] seeks to exhaustively find an optimal shared query plan. Recent work, such as [RSSB00, MRSR01], provides heuristics for reducing the search space for the optimally shared query plan for a set of SQL queries. These works differ from this dissertation work in that we focus on the computation sharing for window-based continuous queries. The traditional SQL queries do not have window semantics.

Continuous query based applications involving hundreds of, or even thousands of, concurrent queries over high volume data streams are emerging in a large variety of scientific and engineering domains. Examples of such applications include environmental monitoring systems [AAB⁺05b] that allow multiple continuous queries over sensor data streams, with each query issued for independent monitoring purposes. Another example is the publish-subscribe service [BBDW05, Pc05] that hosts a large number of subscriptions monitoring published information from data sources. The

number of input data streams is usually much smaller than the number of continuous queries issued on them. Thus commonly many continuous queries are similar in flavor against the same input streams.

Processing each such compute-intensive query separately is inefficient and certainly not scalable to the huge number of queries encountered in these applications. One promising approach in the database literature to support large numbers of queries is *computation sharing*. Efficient sharing of computations among multiple continuous queries is equally paramount. Many previous works [CCC⁺02, MSHR02, CDN02, HFAE03] have highlighted the importance of computation sharing in continuous queries. The previous work in the early stage, e.g. [CCC⁺02], has focused primarily on sharing of filters with overlapping predicates, which are stateless and have simple semantics.

However in practice, stateful operators such as joins and aggregations tend to dominate the usage of critical resources such as memory and CPU in a DSMS. These stateful operators tend to be bounded using window constraints on the otherwise infinite input streams. Efficient sharing of these stateful operators with possibly different window constraints thus becomes critical, offering the promise of major reductions in resource consumption.

Compared to traditional multi-query optimization, one new challenge in the sharing of stateful operators comes from the preference of in-memory processing of stream queries. Frequent access to hard disk will be too slow when arrival rates are high. Any sharing blind to the window constraints might keep tuples unnecessarily long in the system. A carefully designed sharing paradigm beyond traditional sharing of common sub-expressions

is thus needed.

Recent works [AW04, ZKOS05] have focused on sharing computations of stateful aggregations. The work in [AW04], addressing operator-level sharing of multiple aggregations, has considered the effect of different sliding windows constraints. The work in [ZKOS05] discusses shared computation among aggregations with fine-grained *phantoms*, which is the smallest unit for sharing the aggregations. However, efficient sharing of window-based join operators has thus far been ignored in the literature until our work [WRGB06].

The Proposed Approach

In order to efficiently share computations of window-based join operators, we propose a new paradigm for sharing join queries with different window constraints and filters. The two key ideas of the approach are *state-slicing* and *pipelining*.

The window states of the shared join operator are sliced into fine-grained pieces based on the window constraints of individual queries. Multiple sliced window join operators, with each joining a distinct pair of sliced window states, can be formed into a chain. Selections now can be pushed down between the appropriate sliced window joins to avoid unnecessary computation and memory usage shown above.

Based on the state-slice sharing paradigm, two algorithms are proposed for the chain buildup, one that minimizes the memory consumption and the other that minimizes the CPU usage. The algorithms are guaranteed to always find the optimal chain with respect to their targeted resource of

either minimizing memory or CPU costs, for a given query workload and statistic estimations. Chains in the “middle” can also be built considering tradeoffs between the system memory consumption and CPU usage. The experimental results show that our strategy achieves respected optimization goals for memory or CPU costs over a diverse range of workload settings among alternate solutions in the literature.

Dissertation Contributions to Multiple Stream Query Sharing

- We categorize the existing sharing strategies in the literature, highlighting their memory and CPU consumptions.
- We introduce the concept of a chain of pipelining sliced window join operators, and prove its equivalence to the regular window-based join.
- The memory and CPU costs of the chain of sliced window join operators are evaluated and analytically compared with the existing solutions.
- Based on the insights gained from this analysis, we propose two algorithms to build the chain that minimizes the CPU or the memory cost of the shared query plan, respectively. We prove the optimality of both algorithms.
- We provide methods for the online adaptation of the shared slice join plan. Such optimization can be done dynamically at run time. According to run time statistics, adjacent state sliced join operators can

be merged by combining the corresponding states and adding necessary routing operators. The online splitting of operator is also supported by further splitting the states.

- The proposed techniques are implemented in an actual DSMS (*CAPE*). Results of performance comparison of our proposed techniques with state-of-the-art sharing strategies are reported. Our solution has been shown to be more efficient than other sharing strategies for various workloads of stream queries.

1.2.2 Distributed Multi-way Stream Join Query Optimization

New Challenges in Distributed Multi-way Join Processing

Stream applications such as scientific sensor network infrastructures require filtering, aggregation and correlation of high-volume stream data. The data streams can include text data, multimedia data and other complex objects such as network packets and sensor data. Multi-way window-based Join operations (MJ) are commonly used to explore the correlation among multiple such stream tuples in scientific and engineering domains [AAB⁺05b, RRWM07, JAA⁺06, KDY⁺06]. For example, environmental monitoring systems use sensor networks that analyze data streams with possibly complex pattern matching methodologies [AAB⁺05b, RRWM07]. Network monitoring systems use deep packet inspection queries to evaluate network traffic flows with content-based analysis methods [KDY⁺06]. The multi-way joins in such applications tend to have complex join conditions on high volume input stream data.

Stream applications can be time critical, which causes additional challenges for the processing of stream joins among multiple high-speed streams. Stateful operators such as MJs tend to dominate the critical resources such as memory and CPU in a DSMS. When facing high-volume input streams, the in-memory processing may at times be beyond the capacity of a single machine [GYW07]. The resource pressure includes not only CPU processing power, but also memory used for the stateful MJ operations, given that processing tends to be main memory resident to ensure timely response. To scale such memory- and CPU-intensive applications without violating result accuracy nor real-time response requirements, resorting to a shared-nothing cluster has been recognized as one of the most practical solutions [ABcea05].

The basic distribution techniques used in the relational database systems can be classified as *pipelined parallelism* and *partitioned parallelism* [Kun00]. By streaming the output of one operator into the next operator, the two operators can work in series, termed pipelined parallelism. By partitioning the input data among multiple processors, an operator can often be instantiated as many independent instances each working on a part of the data, termed partitioned parallelism.

Distributed continuous query processing has been considered in recent years, such as distributed Eddies [TD03], Borealis [Ac04, ABcea05], System S [JAA⁺06] and D-CAPE [LZ]⁺05]. Correspondingly two distribution techniques are usually supported: operator distribution and data distribution. Using operator distribution, disjoint sub-plans of the query plan are executed on different machines with the intermediate results being routed be-

tween the machines. Data distribution instead installs instances of the same operator into multiple machines, each then processing a different partitions of the input data on its respective machine. Both methods are orthogonal and can in fact be combined.

However direct application of these distribution methods is not always guaranteed to be effective for distributing MJs with arbitrary join conditions. 1) For pipelined parallelism, the macro MJ operator must fit into one single machine — which is not always feasible when large window constraints and high volume input streams are encountered. Though we could translate an MJ operator into a join tree composed of a sequence of smaller binary join operators, such method would lose the flexibility of join orderings shown to be extremely useful for MJ processing in dynamic environments [VNB03]. Moreover such join tree distribution will scale to at most $k - 1$ machines for a k -way MJ operator, while the number of machines available may be much larger than k . 2) On the other hand, partitioned parallelism only supports equi-joins, since it requires some hash function for disjoint partitioning of tuples. For non-equi-joins, value-based data partitioning cannot be applied without potentially huge data duplication, as shown in [GYW07]. Data duplication may abuse memory and cause increased data shipping and processing costs. Moreover, data partitioning [SHea03] assumes that every partition is small enough to be processed by one single machine. This assumption may not always be valid or could rapidly be violated at run-time, especially when processing skewed data.

In the second part of this dissertation, we focus on distributed process-

ing of generic MJs with arbitrary join predicates, especially of MJs with large window constraints. Generic stream joins occur in many practical situations, from simple range (or band) join queries to complicated scientific queries with equation-based predicates [AAB⁺05b, RRWM07]. Such join operators tend to be complex and CPU intensive. Our goal is to minimize the query response time to meet the real-time response requirement of the stream applications.

Applying a data-replication based distribution approach [GYW07] for window-based MJ operators with generic join predicates can be inefficient, because: 1) the state memory used for the MJ operators dominates the memory consumption, and thus data replication would further exacerbate the memory shortage; 2) An extra cost for state management with data replication arises, including cost for duplication elimination. Such cost can be rather significant for large window constraints and high volume data streams.

The Proposed Approach

A novel MJ operator distribution scheme called Pipelined State Partitioning (PSP) is proposed in this dissertation. The PSP scheme is a new form of pipelined parallelism. Our solution is based on the state-slicing concept introduced for query sharing in Chapter 5. We propose a novel solution to split a macro MJ operator into a series of smaller state-sliced MJ operators. Different from value-based partitioning, the PSP scheme is join predicate agnostic and thus general. It slices the states into disjoint slices in the time domain, and then distributes these fine-grained state slices among process-

ing nodes in the cluster. Different from traditional plan-based pipelined parallelism, whose degree of parallelism is bounded by the longest sequence of operators in the query plan, PSP instead can split the MJ to any number of state-sliced MJ operators at the optimizer's will to achieve maximum parallelism.

Beyond this basic PSP scheme, we design two extensions. One, PSP-I (with I for Interleaving) introduces a delayed purging technique for the states to enable interleaved processing of multiple stream tuples with asynchronous processor coordination. Such interleaved processing is used to avoid idle processors which exist in the synchronized basic PSP scheme. Two, beyond interleaved processing, PSP-D (with D for Dynamic) further incorporates a dynamic state ring structure to avoid repeated maintenance cost of sliced states, which comes from the standard tuple insertion and state purging routines.

A cost model is developed to achieve the optimal state slicing and allocation, in terms of query response latency. The tradeoff between employing more processing nodes and having more transmission hops is considered. Runtime adaptive state relocation are also employed for achieving load balancing and re-optimization in a fluctuating environment by smoothing the sliced state size and adding/removing processing nodes dynamically.

Compared to existing work on distributed generic MJ processing in [GYW07], the PSP scheme has the following benefits: 1) no state duplication and thus no repeated computations during PSP distribution; 2) applicable for any window constraints; 3) arbitrary number of sliced operators at the optimizer's will to achieve optimality with given statistics; and 4) controllable

adaptive state partitioning and allocation in the time domain.

To illustrate the benefits of our PSP scheme, we have implemented the proposed PSP scheme within the *D-CAPE* DSMS. Since the operator distribution has been supported in *D-CAPE*, we reuse this part of *D-CAPE* to distribute the generated state sliced joins among multiple processing nodes. A series of experimental studies are conducted to illustrate the performance of the PSP scheme (in term of response time and state memory usage) under various workloads. Comparisons with other distributed generic MJ processing approaches in [GYW07] are also discussed. The experimental results show that our strategy provides significant performance improvements under diverse workload settings.

Dissertation Contributions to Distributed Stream Query Processing

- We introduce the novel ring architecture of sliced window join operators, and prove its equivalence to the regular window-based join.
- We extend the based PSP model with two key features: interleaved tuple processing and dynamic ring structure to improve the response time.
- The memory and CPU costs of PSP ring are analytically evaluated based on a cost model.
- Based on insights gained from this analysis, a cost-based optimizer is proposed that achieves optimal state slicing in terms of maximum output rate and minimal query response latency. The optimality is proved.

- The runtime state migration algorithms in terms of slice allocation and relocation is described.
- The proposed techniques are implemented in the *D-CAPE* DSMS. Performance of the PSP scheme under various workloads, in term of response time and state memory usage is reported. The effect of runtime adaptation is also illustrated in the experimental study.
- Results of performance comparison of our proposed PSP scheme with state-of-the-art distributed generic MJ processing approached in [GYW07] are also conducted.

1.2.3 Distributed Multiple Query Processing

Challenges in Distributed Multiple Query Sharing

In the first two parts of the dissertation, we discussed the state slicing based binary stream join query sharing and distributed multi-way join query processing. In the third part of dissertation work, we will integrate these two solutions to tackle the problem of multiple query optimization in a distributed system. The common state slicing concept behind these two parts makes the seamless integration possible. Based on the approaches proposed in the first two parts of dissertation work, we need to solve following issues.

- Extend the selection pushdown algorithm to multi-way state sliced join ring.

- Fast routing strategy is needed to send the joined results to the corresponding users. Each state sliced join operator can generate joined result for different queries. A fast routing method is necessary to dispatch the joined result to serve multiple queries.
- Deploy the state sliced join ring with selections in a cluster of processing nodes with consideration of workload balancing.

The Proposed Approach

We propose a two phase query plan generation to share the computation of multi-way stream joins in a cluster. In the first phase, the selections are pushed into the ring and the state sliced joins based on the selection predicates are formed. In the second phase, the ring of query plan is deployed in the processing nodes with consideration of balanced workload in each node. To achieve balanced workload, the state sliced joins generated in the first phase may be further sliced. Also one processing node may host multiple state sliced joins together with the selections between them. A cost based deployment is used to achieve the balanced workload.

To achieve fast routing of the joined result, we propose a bitmap based routing strategy. Since the number of distinct sub-joins between sliced states may be huge for multi-way join sharing, we use one routing operator to dispatch all the joined results instead of using one routing operator for the joined results from each sub-joins. Based on the bitmap in the joined result, it can be routed to the corresponding query user.

1.2.4 Overview of the CAPE/D-CAPE System

The techniques in this dissertation have been implemented in a prototype continuous query system named *CAPE/D-CAPE* [RDS⁺04, LZJ⁺05] developed at WPI as a team effort to serve as the testbed for our research of continuous query processing. D-CAPE stands for **D**istributed **C**ontinuous **A**daptive **P**rocessing **E**ngine). The D-CAPE system is a prototype streaming database system designed 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]. D-CAPE adopts a novel architecture that enables adaptive services at all levels of query processing, including reactive operator execution [RDS⁺04], adaptive operator scheduling [SZDR05], runtime query plan re-optimization [ZRH04] and across-machine plan redistribution [ZR07].

The D-CAPE system architecture is depicted in Figure 1.3. The system can be degraded to run on a single machine as well as across multiple machines. Each machine (processor) can run an instance of the CAPE query engine. If the system is run on multiple machines, a distributed manager overlooks these multiple CAPE query engines and makes system-wide adaptation decisions according to runtime statistics collected by the QoS Inspector. The key adaptive components in D-CAPE 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

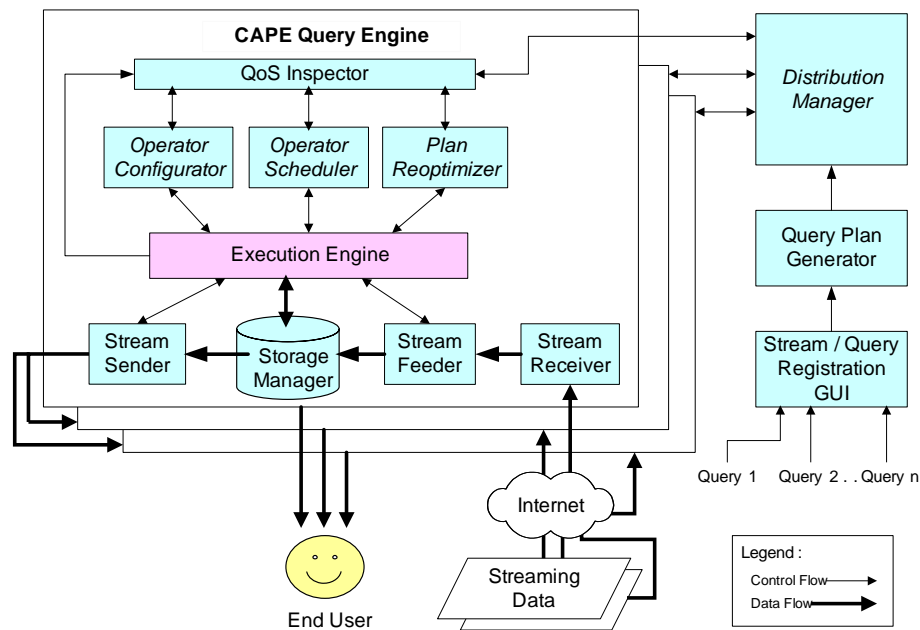


Figure 1.3: D-CAPE System Architecture.

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 components in the D-CAPE architecture that are directly related to this dissertation are Plan Generator, Plan Reoptimizer and Distribution Manager, which are in charge of the static query plan generation for multiple query optimization and ring based distribution, adaptive re-optimization of the state sliced query plans, and state relocation in the distributed system, respectively.

1.3 Dissertation Road Map

The rest of this dissertation is organized as follows: The 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 motivation, problem introduction, background, solution description, experimental evaluation and discussions of related work. Finally Part IV concludes this dissertation and discusses possible future work.

Part I

State-Slice Multi-query Optimization of Stream Queries

Chapter 2

Introduction

2.1 Research Motivation

Modern stream applications such as sensor monitoring systems and publish/subscription services necessitate the handling of large numbers of continuous queries specified over high volume data streams. Examples of such applications include environmental monitoring systems [AAB⁺05b] that allow multiple continuous queries over sensor data streams, with each query issued for independent monitoring purposes. Another example is the publish-subscribe services [BBDW05, Pc05] that host a large number of subscriptions monitoring published information from data sources. Such systems often process a variety of continuous queries that are similar in flavor on the same input streams.

Efficient sharing of computations among multiple continuous queries, especially for the memory- and CPU-intensive window-based operations, is critical. Many papers [CCC⁺02, MSHR02, CDN02, HFAE03] have high-

lighted the importance of computation sharing in continuous queries. In practice, stateful operators such as joins and aggregations tend to dominate the usage of critical resources such as memory and CPU in a DSMS. These stateful operators tend to be bounded using window constraints on the otherwise infinite input streams. A novel challenge in this scenario is to allow resource sharing among similar queries, even if they employ windows of different lengths.

The intuitive sharing method for joins [HFAE03] with different window sizes employs the join having the largest window among all given joins, and a routing operator which dispatches the joined result to each output. Such method suffers from significant shortcomings as shown using the motivation example below. The reason is two folds, (1) the per-tuple cost of routing results among multiple queries can be significant; and (2) the selection pull-up (see [CDN02] for detailed discussions of selection pull-up and push-down) for matching query plans may waste large amounts of memory and CPU resources.

Motivation Example: Consider the following two continuous queries in a sensor network expressed using CQL [AAB⁺05b], an SQL-like language with window extension.

```
Q1: SELECT A.* FROM Temperature A, Humidity B
      WHERE A.LocationId=B.LocationId
      WINDOW 1 min
```

```
Q2: SELECT A.* FROM Temperature A, Humidity B
      WHERE A.LocationId=B.LocationId AND
```

```
A.Value>Threshold  
WINDOW 60 min
```

Q_1 and Q_2 join the data streams coming from temperature and humidity sensors by their respective locations. The WINDOW clause indicates the size of the sliding windows of each query. The join operators in Q_1 and Q_2 are identical except for the filter condition and window constraints. The naive shared query plan will join the two streams first with the larger window constraint (60 min). The routing operator then splits the joined results and dispatches them to Q_1 and Q_2 respectively according to the tuples' timestamps and the filter. The routing step of the joined tuples may take a significant chunk of CPU time if the fanout of the routing operator is much greater than one. If the join selectivity is high, the situation may further escalate since such cost is a per-tuple cost on every joined result tuple. Further, the state of the shared join operator requires a huge amount of memory to hold the tuples in the larger window without any early filtering of the input tuples. Suppose the selectivity of the filter in Q_2 is 1%, a simple calculation reveals that the naive shared plan requires a state size that is 60 times larger than the state used by Q_1 , or 100 times larger than the state used by Q_2 each by themselves. In the case of high volume data stream inputs, such wasteful memory consumption is unaffordable and renders inefficient computation sharing.

The problem of multiple continuous query optimization with window constraints contains two sub-problems:

- First, we need rewriting algorithms that efficiently split continuous

join query plans into equivalent plans with identical join signatures, including predicates and window constraints.

- Second, we need cost based optimization algorithms to determine how to optimize the overall shared query plan, considering other operators such as selection.

2.2 Proposed Strategies

In order to efficiently share computations of window-based join operators, I propose a new paradigm for sharing join queries with different window constraints and filters. The two key ideas of my approach are: *state-slicing* and *pipelining*.

We slice the window states of the shared join operator into fine-grained pieces based on the window constraints of individual queries. Multiple sliced window join operators, with each joining a distinct pair of sliced window states, can be formed. Selections now can be pushed down below any of the sliced window joins to avoid unnecessary computation and memory usage shown above.

State slicing is not trivial. Let's consider a brute force state slicing as follows. Figure 2.1 shows a possible state slicing solution for stateful join operators with window constraints. The original join operator now can be split to connected J_1 and J_2 with J_2 as the down stream operator. J_2 will accept the up stream tuples from J_1 . Assume that the queues between J_1 and J_2 are empty, then at any time, the snapshot of the combined state content of J_1 and J_2 is equivalent to that of the original join operator. Also the

sliced states are disjoint for J_1 and J_2 . Such state slicing is very straightforward and seems achieving our goals for operator splitting. However such naive state slicing can not produce the same result as the original join operator. Apparently, the possible results coming from crossing probings of both J_1 and J_2 are lost. Eventually an incomplete joined result is generated.

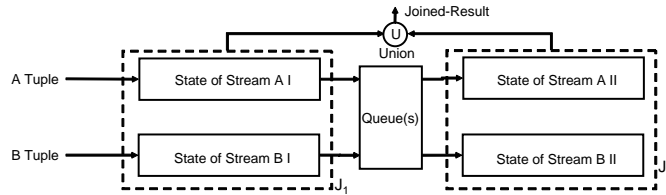


Figure 2.1: A Brute Force State Slicing with Incomplete Result

It seems that N^2 joins appear to be needed to provide a complete answer if each of the window states were to be sliced into N pieces and each join works on one combination of sliced states. The number of distinct join operators needed would then be too large for a DSMS to hold for a large N . We overcome this hurdle by elegantly pipelining the slices. This enables us to build a chain of only N sliced window joins to compute the complete join result. This also enables us to selectively share a subsequence of such a chain of sliced window join operators among queries with different window constraints.

Based on the state-slice sharing paradigm, two algorithms are proposed for the chain buildup, one that minimizes the memory consumption and the other that minimizes the CPU usage. The algorithms are guaranteed to always find the optimal chain with respect to either memory or CPU cost, for a given query workload.

Those two algorithms are based on the cost model developed for the state slicing paradigm. The cost model can be used to estimate the CPU and memory usage of the shared query plans. After comparing with alternatives, the optimal state slicing query plan can be achieved.

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

- First, I review the existing sharing strategies in the literature with consideration of the new window constraints. By comparing their memory and CPU consumptions, their drawbacks are illustrated and motivate my research.
- Second, I propose a novel paradigm for splitting large window join operators with window constraints. By introduce the state slicing concept, the CPU and memory consumptions can be split accordingly into small pieces. I also prove its equivalence to the semantics of regular stream window join operator. To the best of my knowledge, this work is the first in multiple continuous query optimization to 1) consider both predicates and window constraints, 2) utilize chain of pipelining sliced window join operators to rewrite join query plan.
- Third, I develop a set of cost models to analytically compare the memory and CPU costs of the chain of sliced window join operators with other existing solutions.
- Fourth, based on the insights gained from the cost base analysis, I propose two algorithms to build the chain that minimizes the CPU or

the memory cost of the shared query plan, respectively. I prove the optimality of both algorithms.

- Methods for the online adaptation of the shared slice join plan are provided and discussed. Such optimization can be done dynamically at running time.
- The proposed techniques are implemented in an actual DSMS (*CAPE*). A thorough experimental evaluation is conducted. Results of performance comparison of our proposed techniques with state-of-the-art sharing strategies are reported. I compare the CPU and memory consumptions of different sharing strategies with various workload queries. The experimental results show that the proposed solutions are the best among them.

2.3 Road Map

The rest of the part I is organized as follows. Chapter 3 presents the background and preliminaries used in this paper. Chapter 4 shows the motivation example with detailed analytical performance comparisons of alternative sharing strategies of window-based joins. Chapter 5 describes the proposed chain of sliced window join operators. Chapter 6 provides a detailed case study on stream query join trees applying the state slicing concept. Chapter 7 presents the algorithms to build the chain. Chapter 8 presents the experimental results. Chapter 9 contains related work.

Chapter 3

Background

3.1 Stateful Operators in Continuous Queries

Continuous queries generally require real time responses. Query results need to be sent to the downstream user in a pipelined manner. This requires that all operators in the query plans need to be operated in a unblocked fashion: the operator needs to be able to generate 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 and relate to future processing. *Operator state* is some data structure inside stateful operators, such as joins and group-bys, that stores tuples received so far for future processing. An operator may output partial results based on the already received tuples. To make blocking operators, such as joins or group-bys, become non-blocking, we can store tuples received so far in this *state* data structure. For a long-running query as in the case of continuous queries, the number of tuples stored

inside a stateful operator can potentially be very large and unbounded. 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, MWA⁺03, HFAE03] and punctuation-based constraints [DMRH04, TMSF03]. 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.

Stateful join operator is one of the most important stateful operators in continuous query processing, and is the focus of the research in this part of the dissertation. As commonly used by continuous query plans in most streaming database systems [KNV03, CCC⁺02, MWA⁺03], in this dissertation we adopt the symmetric window-based join algorithm [WA93, HH99] for join processing.

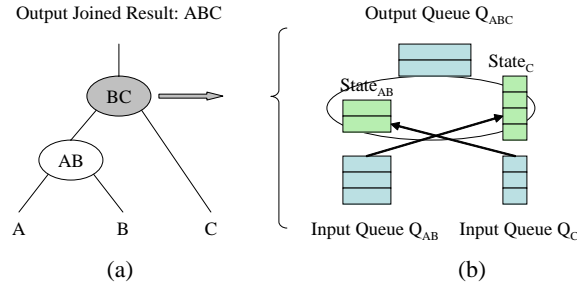


Figure 3.1: Sliding Window Join Operators and Their States

A sample query plan for the query $A \bowtie B \bowtie C$ is depicted in Figure 3.1(a). The join operator $B \bowtie C$ in Figure 3.1(b) has two *states* S_{AB} and S_C , one associated with each input queue. Each *state* stores the tuples that

fall within the current window frame 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 one window or further away from tuple AB; 2) probe – 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 . We call this 3-step process as *purge-probe-insert* algorithm.

3.2 Window Constraints and Sliding Window Join

An operator state stores tuples received so far for future processing. 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 time-frame 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 $C(c1, c2)$ where $a1$ and $a2$ denote attributes of stream A , $b1$ and $b2$ denote attributes of stream B and etc., a user may submit the following query with window constraints:

```
SELECT    Count(*)
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 [ABW06]. 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 COUNT. In this example, all operators are evaluated 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.

Without any constraints, the states of a stateful operator can grow infinitely, and the system can eventually grow out of memory. To solve this problem, streaming databases usually adopt sliding window constraints to limit the size of states. A sliding window-based constraint [KNV03, CCC⁺02, MWA⁺03] can be used to purge unwanted tuples stored in the state. Usually two kinds of window constraints are posed over an operator: time-based [KNV03] and count-based [MWA⁺03]. See [GÖ03b] for a survey on window-based join operations in the literature. The size of a window constraint is specified using either a time interval (time-based) or a count on the number of tuples (count-based). In this part, we present our sharing paradigm using time-based windows. However, our proposed techniques can be applied to count-based window constraints in the same way by using different purging condition. The proposed solution is applicable to generic join operators with arbitrary join conditions.

Formally, the sliding window join of streams A and B , with window

sizes W_1 and W_2 respectively having the join condition θ can be denoted as $A[W_1] \bowtie_{\theta} B[W_2]$. The semantics [SW04] for such sliding window joins are that the output of the join consists of all pairs of tuples $a \in A, b \in B$, such that join condition $\theta(a, b)$ holds (we omit θ in the future and instead concentrate on the sliding window only) and at certain time t , both $a \in A[W_1]$ and $b \in B[W_2]$. That is, either $0 < T_b - T_a < W_1$ or $0 < T_a - T_b < W_2$. T_a and T_b denote the timestamps of tuples a and b respectively in this paper. The timestamp assigned to the joined tuple is $\max(T_a, T_b)$. The execution steps for a newly arriving tuple of A are shown in Fig. 3.2. In this part we only consider cross-purge, while self-purge is also applicable. Symmetric steps are followed for a B tuple.

-
1. *Cross-Purge*: Discard expired tuples in window $B[W_2]$
 2. *Probe*: Emit $a \bowtie B[W_2]$
 3. *Insert*: Add a to window $A[W_1]$
-

Figure 3.2: Execution of Sliding-window join.

The most commonly used window constraint is the *global window constraint* in which a single stream has an unique window constraint with respect to any other stream. For example in the CQL query shown in this chapter, stream A has a 30 minute window. This window is a global window on stream A and is not bound to stream B in the join condition. Theoretically it is possible that the window constraints are defined on join pairs and may be inconsistent for the same stream appearing in different join pairs. These type of window constraints are referred to as *local window constraints*. In this dissertation we assume the window constraints in the

stream queries are valid in semantics, no matter whether global or local.

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, MWA⁺03]. When two tuples are joined together, the timestamp for the joined tuple is an array that concatenates the timestamps from both joining tuples. Both timestamps are kept because either of them might be used by other join operators in the query plan if local window constraints are used. Such a tuple with a combined timestamp is referred to as a *combined tuple*. Usually, the largest timestamp is enough for the purpose of purging when the global window constraints are used.

3.3 Assumptions and Simplifications

In this part of dissertation, the following assumptions and simplifications are used.

- Each join operator processes the input stream tuples in the order of their timestamps. Applying time-based window constraints requires that each tuple has a timestamp. Thus we assume that all the stream tuples have unique timestamps. Tuples are usually assumed to be ordered by their timestamps [KNV03, CF02, MWA⁺03]. We follow this assumption in this dissertation. Thus we assume that the stream

tuples are ordered. In practice, out-of-order stream processing is necessary since the asynchronous nature of stream collection processes. A stream data cleaning step can be adopted before continuous query processing to tackle such timestamp discrepancies.

- Each join operator processes an input tuple to completion before processing the next one. That is, the join operator is single-threaded. However multiple join operators can run concurrently each in its own thread. Under this assumption, each operator will process stream tuples in a sequential manner. Thus at any time, there is at most only one thread purging the states of each join operator. This assumption is commonly used [CCC⁺02] in the sense that at most one thread is used for each operator to avoid multi-threading issue in the operator.
- To simplify further discussions in this part, we omit the join conditions from each join expression and instead use the equi-join notation. Since the stream join algorithm used throughout this part is the symmetric nested loop join, this assumption can be dropped straightforwardly. However our solutions do not require specific join algorithms. Other join algorithms, like symmetric hash joins, also can be employed for processing equi-joins.
- To simplify further descriptions in this part, we only show our proposed algorithm with cross-purge strategy. However our solution does not limit us to any specific window-based purge strategies and can work together with self-purge or combined purge strategies.

- In this part we only consider global window constraints. Local window constraints can be handled by using pairwise window constraints in the purging strategies. This will not change the principle of our proposed solutions. Our solution is orthogonal to the purging strategies. Further discussion of purging strategies is beyond the scope of this dissertation.

Chapter 4

Review of Existing Strategies for Sharing Continuous Queries

Using the example queries Q_1 and Q_2 from Chapter 2 with window constraints, we review the existing strategies in the literature for sharing continuous queries. Figure 4.1 shows the query plans for Q_1 and Q_2 without computation sharing. The states in each join operator hold the tuples in the window. We use σ_A to represent the selection operator on stream A . For easy reference, the queries Q_1 and Q_2 are listed again below.

```
Q1: SELECT A.* FROM Temperature A, Humidity B
      WHERE A.LocationId=B.LocationId
      WINDOW 1 min
```

```
Q2: SELECT A.* FROM Temperature A, Humidity B
```

WHERE A.LocationId=B.LocationId AND

A.Value>Threshold

WINDOW 60 min

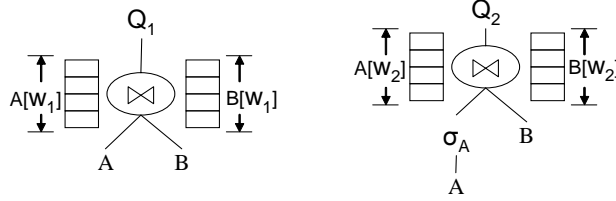


Figure 4.1: Query Plans for Q_1 and Q_2 .

For the following cost analysis, we use the notations of the system settings in Table 4.1. We define the selectivity of σ_A as: $\frac{\text{number_of_outputs}}{\text{number_of_inputs}}$. The *number_of_inputs* and *number_of_outputs* are defined for the input and the output streams of σ_A as the total stream tuple counts from beginning of the execution time of the query. We define the join selectivity S_{\bowtie} as: $\frac{\text{number_of_outputs}}{\text{number_of_outputs_from_Cartesian_Product}}$. For stream join with sliding windows, the join selectivity equals to the probability of satisfying the join conditions when one probing of a pair of stream tuples happens.

We focus on state memory when calculating the memory usage. To estimate the CPU cost, we consider the cost for value comparison of two tuples and the timestamp comparison. We assume that comparisons are equally expensive and dominate the CPU cost. We thus use the count of comparisons per time unit as the metric for estimated CPU costs. In this part, we calculate the CPU cost assuming the nested-loop join algorithm. Calculation using the hash-based join algorithm can be done similarly using an adjusted cost model [KNV03].

Symbol	Explanation
λ_A	Arrival Rate of Stream A (Tuples/Sec.)
λ_B	Arrival Rate of Stream B (Tuples/Sec.)
W_1	Window Size for Q_1 (Sec.)
W_2	Window Size for Q_2 (Sec.)
M_t	Tuple Size (KB)
S_σ	Selectivity of σ_A
S_{\bowtie}	Join Selectivity

Table 4.1: System Settings Used in Chapter 4.

Without loss of generality, we assume $0 < W_1 < W_2$. For simplicity, in the following computation, we set $\lambda_A = \lambda_B$, denoted as λ .

4.1 Naive Sharing with Selection Pull-up

The PullUp or Filtered PullUp approaches proposed in [CDN02] for sharing continuous query plans containing joins and selections can be applied to the sharing of joins with different window sizes. That is, we need to introduce a router operator to dispatch the joined results to the respective query outputs. The intuition behind such sharing lies in that the answer of the join for Q_1 (with the smaller window) is contained in the join for Q_2 (with the larger window). The shared query plan for Q_1 and Q_2 is shown in Figure 4.2.

By performing the sliding window join first with the larger window size among the queries Q_1 and Q_2 , computation sharing is achieved. The router then checks the timestamps of each joined tuple with the window

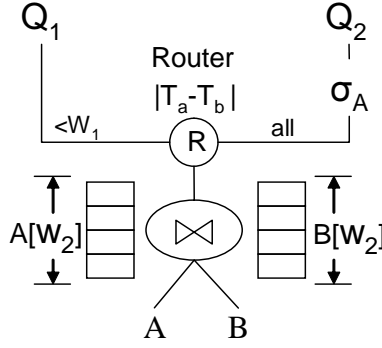


Figure 4.2: Selection Pull-up.

constraints of registered CQs and dispatches them correspondingly. The compare operation happens in the probing step of the join operator, the checking step of the router and the filtering step of the selection. We can calculate the state memory consumption C_m (m stands for memory) and the CPU cost C_p (p stands for processor) as:

$$\begin{cases} C_m = 2\lambda W_2 M_t \\ C_p = 2\lambda^2 W_2 + 2\lambda + 2\lambda^2 W_2 S_{\bowtie} + 2\lambda^2 W_2 S_{\bowtie} \end{cases} \quad (4.1)$$

In this part, the CPU cost C_p is defined as the count of primary operation numbers in one unit time. The primary operations include join probing, purging tuple routing and filtering. For simple illustration, we assume each primary operation includes a major cost for one comparison and thus all primary operations cost the same. Thus the costs of different primary operations are not weighted in this part.

During each time unit, λ number of tuples arrive from both stream A and B . The first item of C_p denotes the join probing costs; the second the

cross-purge cost; the third the routing cost; and the fourth the selection cost. The routing cost is the same as the selection cost since each of them perform one comparison per result tuple.

As pointed out in [MSHR02], the selection pull-up approach suffers from unnecessary join probing costs. With strong differences of the windows the situation deteriorates, especially when the selection is used in continuous queries with large windows. In such cases, the states may hold tuples unnecessarily long and thus waste huge amounts of memory.

Another shortcoming for the selection pull-up sharing strategy is the routing cost of each joined result. The routing cost is proportional to the join selectivity S_{\bowtie} . This cost is also related to the fanout of the router operator, which corresponds to the number of queries the router serves. To address this overhead, similarly as in [CDN02], a router having a large fanout could be implemented as a range join between the joined tuple stream and a static profile table, with each entry holding a window size. Then the routing cost is proportional to the fanout of the router, which may still be much larger than one.

4.2 Stream Partition with Selection Push-down

To avoid unnecessary join computations in the shared query plan using selection pull-up, we employ the selection push-down approach proposed in [CDN02]. Selection push-down can be achieved using multiple join operators, each processing part of the input data streams. We then need a split operator to partition the input stream A by the condition in the σ_A

operator. Thus the two sub-streams of A sent into the different join operators are disjoint. We also need an order-preserving (on tuple timestamps) union operator [ACC⁺03] to merge the joined results coming from the multiple joins. Such sharing paradigm applied to Q_1 and Q_2 will result in the shared query plan as shown in Figure 4.3.

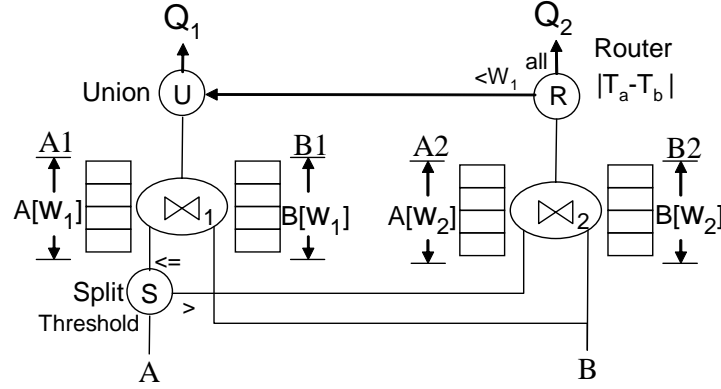


Figure 4.3: Selection Push-down.

The compare operation happens during the splitting of the streams, the merging of the tuples in the union operator, the routing step of the router and the probing of the joins. We can calculate the state memory consumption C_m and the CPU costs C_p for the selection push-down paradigm as:

$$\left\{ \begin{array}{l} C_m = (2 - S_\sigma)\lambda W_1 M_t + (1 + S_\sigma)\lambda W_2 M_t \\ C_p = \lambda + 2(1 - S_\sigma)\lambda^2 W_1 + 2S_\sigma\lambda^2 W_2 + \\ \quad 3\lambda + 2S_\sigma\lambda^2 W_2 S_{\bowtie} + 2\lambda^2 W_1 S_{\bowtie} \end{array} \right. \quad (4.2)$$

The first item of C_m refers to the state memory in operator \bowtie_1 ; the second to the state memory in operator \bowtie_2 . The first item of C_p corresponds to the

splitting cost; the second to the join probing cost of \bowtie_1 ; the third to the join probing cost of \bowtie_2 ; the fourth to the cross-purge cost; the fifth to the routing cost; the sixth to the union cost. Since the outputs of \bowtie_1 and \bowtie_2 are sorted, the union cost corresponds to a one-time merge sort on timestamps.

Different from the sharing of identical file scans for multiple join operators in [CDN02], the state memory B_1 cannot be saved since B_2 may not contain B_1 at all times. The reason is that the sliding windows of B_1 and B_2 may not move forward simultaneously, unless the DSMS employs a synchronized operator scheduling strategy.

Stream sharing with selection push-down tends to require more joins (mn , where m and n are the number of sub-streams of A and B respectively) than the naive sharing. With the asynchronous nature of these joins as discussed above, extra memory is consumed for the state memory. Such memory waste might be significant.

Obviously, the CPU cost C_p of a shared query plan generated by the selection push-down sharing is much smaller than the CPU cost of using the naive sharing with selection pull-up. However this sharing strategy still suffers from similar routing costs as the selection pull-up approach. Such cost can be significant, as already discussed for the selection pull-up case.

Chapter 5

State-Slice Sharing Paradigm

In this section, the new sharing paradigm is discussed for sharing sliding window joins with different window constraints. As discussed in Chapter 4, existing sharing paradigms suffer from one or more of the following cost factors: (1) expensive routing step; (2) state memory waste among asynchronous parallel joins; and (3) unnecessary join probings without selection push-down. Our proposed state-slice sharing successfully avoids all three types of costs.

We first introduce the proposed concept of state-slice using a one-way sliding window join. Then we extend this concept to the binary state-sliced join operator. Lastly we show the state-slice sharing for the running example queries in Chapter 4 and compare its performance with other alternatives listed in Chapter 4 analytically.

5.1 State-Sliced One-Way Window Join

A one-way sliding window join [KNV03] of streams A and B is denoted as $A[W] \times B$ (or $B \times A[W]$), where stream A has a sliding window of size W . The output of the join consists of all pairs of tuples $a \in A, b \in B$, such that $T_b - T_a < W$, and tuple pair (a, b) satisfies the join condition. It has been shown in [KNV03] that:

$$A[W_1] \times B[W_2] = (A[W_1] \times B) \text{ Union } (A \times B[W_2])$$

Definition 1 (Sliced One-way Sliding Window Join) *A sliced one-way window join on streams A and B is denoted as $A[W^{start}, W^{end}] \overset{s}{\times} B$ (or $B \overset{s}{\times} A[W^{start}, W^{end}]$), where stream A has a sliding window of range: $W^{end} - W^{start}$. The start and end window are W^{start} and W^{end} respectively. The output of the join consists of all pairs of tuples $a \in A, b \in B$, such that $W^{start} \leq T_b - T_a < W^{end}$, and (a, b) satisfies the join condition.*

We can consider the sliced one-way sliding window join as a generalized form of the regular one-way window join. That is $A[W] \times B = A[0, W] \overset{s}{\times} B$. Figure 5.1 shows an example of a sliced one-way window join. This join has one output queue for the joined results, two output queues (optional) for purged A tuples and propagated B tuples, respectively. These purged tuples will be used by another down-stream sliced window join as input streams.

The execution steps to be followed for the sliced window join $A[W^{start}, W^{end}] \overset{s}{\times} B$ are shown in Figure 5.2.

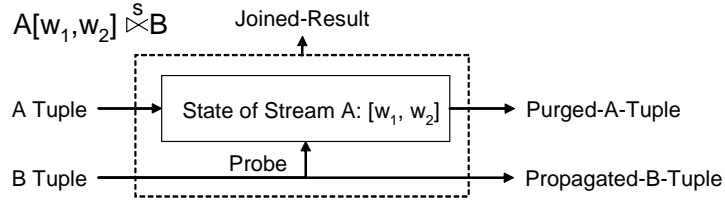


Figure 5.1: Sliced One-Way Window Join.

When a new tuple a arrives on A

1. *Insert*: Add a into sliding window $A[W^{start}, W^{end}]$

When a new tuple b arrives on B

1. *Cross-Purge*: Update $A[W^{start}, W^{end}]$ to purge expired A tuples, i.e., if $a' \in A[W^{start}, W^{end}]$ and $(T_b - T_{a'}) > W^{end}$, move a' into *Purged-A-Tuple* queue (if exists) or discard (if not exists)
 2. *Probe*: Emit result pairs (a, b) according to Def. 1 for b and $a \in A[W^{start}, W^{end}]$ to *Joined-Result* queue
 3. *Propagate*: Add b into *Propagated-B-Tuple* queue (if exists) or discard (if not exists)
-

Figure 5.2: Execution of $A[W^{start}, W^{end}] \stackrel{s}{\bowtie} B$.

The semantics of the state-sliced window join require the checking of both the upper and lower bounds of the time-stamps in every tuple probing step. In Figure 5.2, the newly arriving tuple b will first purge the state of stream A with W^{end} , before probing is attempted. Then the probing can be conducted without checking of the upper bound of the window constraint W^{end} . The checking of the lower bound of the window W^{start} can also be omitted in the probing. The reason is that when the stream tuples are inserted into the corresponding state, their timestamps are stipulated to be larger than the lower bound of the window by the purging step of the up-

stream sliced join operator.

Definition 2 (Chain of Sliced One-way Sliding Window Join) *A chain of sliced one-way window joins is a sequence of pipelined N sliced one-way window joins, denoted as $A[0, W_1] \overset{s}{\bowtie} B, A[W_1, W_2] \overset{s}{\bowtie} B, \dots, A[W_{N-1}, W_N] \overset{s}{\bowtie} B$. The start window of the first join in a chain is 0. For any adjacent two joins, J_i and J_{i+1} , the start window of J_{i+1} equals the end window of prior J_i ($0 \leq i < N$) in the chain. J_i and J_{i+1} are connected by both the Purged-A-Tuple output queue of J_i as the input A stream of J_{i+1} , and the Propagated-B-Tuple output queue of J_i as the input B stream of J_{i+1} .*

Figure 5.3 shows a chain of state-sliced window joins having two one-way joins J_1 and J_2 . We assume the input stream tuples to J_2 , no matter from stream A or from stream B , are processed strictly in the order of their global time-stamps. Thus we use one logical queue between J_1 and J_2 . This does not prevent us from using physical queues for individual input streams.

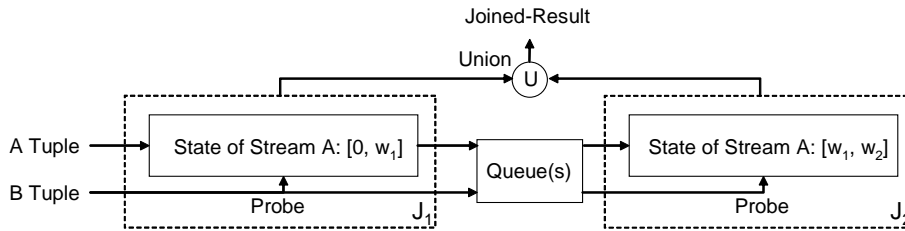


Figure 5.3: Chain of 1-way Sliced Window Joins.

Table 5.1 depicts an example execution of this chain. For this example, let us assume that one single tuple (an a or a b) will only arrive at the start of each second, $w_1 = 2sec$, $w_2 = 4sec$ and every a tuple will match every

b tuple (Cartesian Product semantics). During every second, an operator will be selected to run. Each running of the operator will process one input tuple. The content of the states in J_1 and J_2 , and the content in the queue between J_1 and J_2 after each running of the operator are shown in Table 5.1.

T	Arr.	OP	$A :: [0, 2]$	Queue	$A :: [2, 4]$	Output
1	a_1	J_1	$[a_1]$	$[\]$	$[\]$	
2	a_2	J_1	$[a_2, a_1]$	$[\]$	$[\]$	
3	a_3	J_1	$[a_3, a_2, a_1]$	$[\]$	$[\]$	
4	b_1	J_1	$[a_3, a_2]$	$[b_1, a_1]$	$[\]$	$(a_2, b_1), (a_3, b_1)$
5	b_2	J_1	$[a_3]$	$[b_2, a_2, b_1, a_1]$	$[\]$	(a_3, b_2)
6		J_2	$[a_3]$	$[b_2, a_2, b_1]$	$[a_1]$	
7		J_2	$[a_3]$	$[b_2, a_2]$	$[a_1]$	(a_1, b_1)
8	a_4	J_1	$[a_4, a_3]$	$[b_2, a_2]$	$[a_1]$	
9		J_2	$[a_4]$	$[a_3, b_2]$	$[a_2, a_1]$	
10		J_2	$[a_4]$	$[a_3]$	$[a_2, a_1]$	$(a_1, b_2), (a_2, b_2)$

Table 5.1: Execution of the Chain: J_1, J_2 .

Execution in Table 5.1 follows the steps in Figure 5.2. For example at the 4th second, first a_1 will be purged out of J_1 and inserted into the queue by the arriving b_1 , since $T_{b_1} - T_{a_1} \geq 2sec$. Then b_1 will purge the state of J_1 and output the joined result. Lastly, b_1 is inserted into the queue.

Note that the union of the join results of $J_1: A[0, w1] \overset{s}{\times} B$ and $J_2: A[w1, w2] \overset{s}{\times} B$ is equivalent to the results of a regular sliding window join: $A[w2] \times B$. The order among the joined results is restored by the merge union operator.

To prove that the chain of sliced joins provides the complete join answer, we first introduce the following lemma.

Lemma 1 For any sliced one-way sliding window join $A[W_{i-1}, W_i] \stackrel{s}{\bowtie} B$ in a chain, at the time that one b tuple finishes the cross-purge step, but has not yet began the probe step, we have: (1) $\forall a \in A :: [W_{i-1}, W_i] \Rightarrow W_{i-1} \leq T_b - T_a < W_i$; and (2) $\forall a$ tuple in the input steam A , $W_{i-1} \leq T_b - T_a < W_i \Rightarrow a \in A :: [W_{i-1}, W_i]$. Here $A :: [W_{i-1}, W_i]$ denotes the full state of stream A .

Proof: (1). In the cross-purge step (Figure 5.2), the arriving b will purge any tuple a with $T_b - T_a \geq W_i$. Thus $\forall a_i \in A :: [W_{i-1}, W_i], T_b - T_{a_i} < W_i$. For the first sliced window join in the chain, $W_{i-1} = 0$. We have $0 \leq T_b - T_a$. For other joins J_i in the chain, at any moment there must exist a tuple $a_p \in A :: [W_{i-1}, W_i]$ that has the maximum timestamp among all the a tuples in $A :: [W_{i-1}, W_i]$. Tuple a_p must have been purged by b' of stream B from the state of the up-stream join operator in the chain. If $b' = b$, then we have $T_b - T_{a_p} \geq W_{i-1}$, since W_{i-1} is the upper window bound of the up-stream join operator. If $b' \neq b$, then $T_{b'} - T_{a_p} > W_{i-1}$, since $T_b > T_{b'}$. We still have $T_b - T_{a_p} > W_{i-1}$. Since $T_{a_p} \geq T_{a_k}$, for $\forall a_k \in A :: [W_{i-1}, W_i]$, we have $W_{i-1} \leq T_b - T_{a_k}$, for $\forall a_k \in A :: [W_{i-1}, W_i]$.

(2). We use a proof by contradiction. If $a \notin A :: [W_{i-1}, W_i]$, then first we assume $a \in A :: [W_{j-1}, W_j], j < i$. Given $W_{i-1} \leq T_b - T_a$, we know $W_j \leq T_b - T_a$. Then a cannot be inside the state $A :: [W_{j-1}, W_j]$ since a would have been purged by b when it is processed by the join operator $A[W_{j-1}, W_j] \stackrel{s}{\bowtie} B$. We got a contradiction. Similarly a cannot be inside any state $A :: [W_{k-1}, W_k], k > i$. ■

Theorem 1 The union of the join results of all the sliced one-way window joins in a chain $A[0, W_1] \stackrel{s}{\bowtie} B, \dots, A[W_{N-1}, W_N] \stackrel{s}{\bowtie} B$ is equivalent to the results of a

regular one-way sliding window join $A[W] \times B$, where $W = W_N$.

Proof: “ \Leftarrow ”. Lemma 1(1) shows that the sliced joins in a chain will not generate a result tuple (a, b) with $T_a - T_b > W$. That is, $\forall (a, b) \in \bigcup_{1 \leq i \leq N} A[W_{i-1}, W_i] \overset{s}{\times} B \Rightarrow (a, b) \in A[W] \times B$.

“ \Rightarrow ”. We need to show: $\forall (a, b) \in A[W] \times B \Rightarrow \exists i, s.t. (a, b) \in A[W_{i-1}, W_i] \overset{s}{\times} B$. Without loss of generality, $\forall (a, b) \in A[W] \times B$, there exists unique i , such that $W_{i-1} \leq T_b - T_a < W_i$, since $W_0 \leq T_b - T_a < W_N$. We want to show that $(a, b) \in A[W_{i-1}, W_i] \overset{s}{\times} B$. The execution steps in Figure 5.2 guarantee that the tuple b will be processed by $A[W_{i-1}, W_i] \overset{s}{\times} B$ at a certain time. Lemma 1(2) shows that tuple a would be inside the state of $A[W_{i-1}, W_i]$ at that same time. Then $(a, b) \in A[W_{i-1}, W_i] \overset{s}{\times} B$. Since i is unique, there is no duplicated probing between tuples a and b . ■

From Lemma 1, we see that the state of the regular one-way sliding window join $A[W] \times B$ is distributed among different sliced one-way joins in a chain. These sliced states are disjoint with each other in the chain, since the tuples in the state are purged from the state of the previous join. This property is independent from operator scheduling, be it synchronous or even asynchronous.

Lemma 2 *At any time, the sliced states in one-way sliding window join chain are disjoint with each other, no matter synchronized or unsynchronized operator scheduling is used.*

Proof: Consider two arbitrary distinct states in the chain, $A :: [W_{i-1}, W_i]$ and $A :: [W_{j-1}, W_j]$ ($i \leq j - 1$). Let b_i be the last B tuple being processed by $A[W_{i-1}, W_i] \overset{s}{\times} B$ and b_j be the last B tuple being processed by

$$A[W_{j-1}, W_j] \stackrel{s}{\times} B.$$

(1). When synchronized scheduling is used, two situations exist:

(1a). $b_i = b_j$ if $i = j - 1$. From Lemma 1, we have $\forall a_i \in A :: [W_{i-1}, W_i] \Rightarrow W_{i-1} \leq T_{b_i} - Ta_i < W_i$, and $\forall a_j \in A :: [W_{j-1}, W_j] \Rightarrow W_{j-1} \leq T_{b_j} - Ta_j < W_j$. Since $i = j - 1$ and $b_i = b_j$, we have: $T_{a_i} > T_{a_j}$. That is: $a_i \neq a_j$.

(1b). $b_i \neq b_j$ if $i < j - 1$. Then $T_{b_i} > T_{b_j}$ since the chain is a pipeline. Now we have: $W_{i-1} \leq T_{b_i} - Ta_i < W_i < W_{j-1} \leq T_{b_j} - Ta_j < W_j$, i.e., $T_{b_i} - Ta_i < T_{b_j} - Ta_j$. Since $T_{b_i} > T_{b_j}$, then $T_{a_i} > T_{a_j}$. That is: $a_i \neq a_j$.

(2). When unsynchronized scheduling is used, two situations exist:

(2a). In case $i = j - 1$, $b_i = b_j$ if the queue between the two joins are empty. Then it is proved from (1a). If the queue is not empty, $b_i \neq b_j$. It is covered by (1b).

(2b). In case $i < j - 1$, $b_i \neq b_j$. This case is proved by (1b). ■

From Lemma 1, we see that the state of the regular one-way sliding window join $A[W] \times B$ is distributed among different sliced one-way joins in a chain. These sliced states are disjoint with each other in the chain from Lemma 2.

5.2 State-Sliced Binary Window Join

Similar to Definition 1, we can define the state sliced binary sliding window join. The definition of the chain of sliced binary joins is similar to Definition 2 and is thus omitted. Figure 5.4 shows an example of a chain of

state-sliced binary window joins.

Definition 3 (Sliced Binary Sliding Window Join) A sliced binary window join of streams A and B is denoted as $A[W_A^{start}, W_A^{end}] \bowtie^s B[W_B^{start}, W_B^{end}]$, where stream A has a sliding window of range: $W_A^{end} - W_A^{start}$ and stream B has a window of range $W_B^{end} - W_B^{start}$. The join result consists of all pairs of tuples $a \in A, b \in B$, such that either $W_A^{start} \leq T_b - T_a < W_A^{end}$ or $W_B^{start} \leq T_a - T_b < W_B^{end}$, and (a, b) satisfies the join condition.

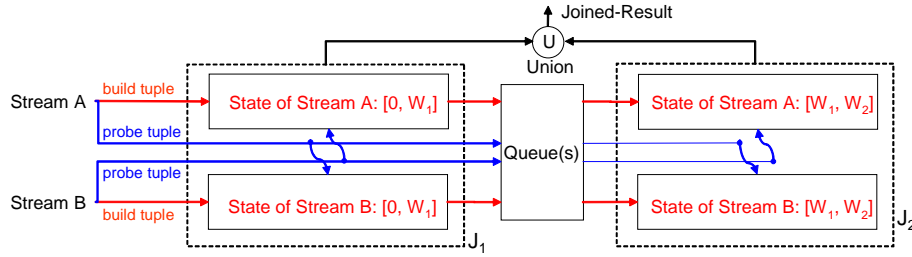


Figure 5.4: Chain of Binary Sliced Window Joins.

The execution steps for sliced binary window joins can be viewed as a combination of two one-way sliced window joins. Each input tuple from stream A or B will be captured as two reference copies. This is done before the tuple is processed by the first binary sliced window join. The copies are made by the first binary sliced join in the chain. One reference is annotated as the *probe* tuple (denoted as a^p) and the other as the *build* tuple (denoted as a^b).

The execution steps to be followed for the processing of a stream A tuple by $A[W^{start}, W^{end}] \bowtie^s B[W^{start}, W^{end}]$ are shown in Figure 5.5. The execution procedure for the tuples arriving from stream B can be similarly defined.

When a new tuple a^p arrives

1. *Cross-Purge*: Update $B[W^{start}, W^{end}]$ to purge expired B tuples, i.e., if $b^b \in B[W^{start}, W^{end}]$ and $(T_{a^p} - T_{b^b}) > W^{end}$, move b^b into the queue (if exists) towards next join operator or discard (if not exists)
2. *Probe*: Emit a^p join with $b^b \in B[W^{start}, W^{end}]$ to *Joined-Result* queue
3. *Propagate*: Add a^p into the queue (if exists) towards next join operator or discard (if not exists)

When a new tuple a^b arrives

1. *Insert*: Add a^b into the sliding window $A[W^{start}, W^{end}]$
-

Figure 5.5: Execution of Binary Sliced Window Join.

Intuitively the probe tuples of stream B and build tuples of stream A are used to generate join tuples equivalent to a one-way join: $A[W^{start}, W^{end}] \overset{s}{\bowtie} B$. The probe tuples of stream A and build tuples of stream B are used to generate join tuples equivalent to the other one-way join: $A \overset{s}{\bowtie} B[W^{start}, W^{end}]$.

Note that using two copies of a tuple will not require doubled system resources since: (1) the combined workload (in Figure 5.5) to process a pair of build and probe tuples equals the processing of one tuple in a regular join operator, since one tuple takes care of purging/probing and the other filling up the states; (2) the state of the binary sliced window join will only hold the build tuple; and (3) assuming a simplified queue (M/M/1), doubled arrival rate (from the two copies) and doubled service rate (from above (1)) still would not change the average queue size, if the system is stable. In our implementation, we use a copy-of-reference instead of a copy-of-object, aiming to reduce the potential extra queue memory during bursts of arrivals. In this dissertation work, we only count state memory as mem-

ory usage since in a stable system, the state memory is the major memory usage for stateful join operators with window constraints. Discussion of scheduling strategies and their effects on queues is beyond the scope of this work.

Theorem 2 *The union of the join results of the sliced binary window joins in a chain $A[0, W_1] \overset{\$}{\bowtie} B[0, W_1], \dots, A[W_{N-1}, W_N] \overset{\$}{\bowtie} B[W_{N-1}, W_N]$ is equivalent to the results of a regular sliding window join $A[W] \bowtie B[W]$, where $W = W_N$.*

Using Theorem 1, we can prove Theorem 2. Since we can treat a binary sliced window join as two parallel one-way sliced window joins, the proof is fairly straightforward.

Theorem 3 *At any time, the sliced states in the sliding window join chain are disjoint with each other, no matter if synchronized or unsynchronized operator scheduling is used.*

Since a binary state slice join chain can be viewed as the combination of two one-way state slice join chains, Theorem 3 is true for binary state slice join chain from Lemma 2.

Lemma 3 *A select operator, which has predicate on stream attributes except the timestamps, can be pushed down a state sliced join operator.*

Proof: Without loss of generality, let the select operator σ_A with predicates on the attributes of stream A. We need to show: $\sigma_A(A[W_{i-1}, W_i] \overset{\$}{\bowtie} B[W_{i-1}, W_i]) = \sigma_A(A[W_{i-1}, W_i]) \overset{\$}{\bowtie} B[W_{i-1}, W_i]$.

" \Rightarrow ". $\forall (a, b) \in \sigma_A(A[W_{i-1}, W_i] \bowtie B[W_{i-1}, W_i]) \rightarrow \sigma_A(a) = true$. From Definition 3, $(a, b) \in \sigma_A(A[W_{i-1}, W_i] \bowtie B[W_{i-1}, W_i])$.

" \Leftarrow ". $\forall (a, b) \in \sigma_A(A[W_{i-1}, W_i] \bowtie B[W_{i-1}, W_i]) \rightarrow \sigma_A(a) = true$. From Definition 3, $(a, b) \in \sigma_A(A[W_{i-1}, W_i] \bowtie B[W_{i-1}, W_i])$.

■

Theorem 4 *The select operator, which has predicate on stream attributes except the timestamps, can be pushed down into the chain without changing of the query semantics. That is, when the selection σ is pushed into the chain between sliced join $J_i : A[W_{i-1}, W_i] \bowtie B[W_{i-1}, W_i]$ and $J_{i+1} : A[W_i, W_{i+1}] \bowtie B[W_i, W_{i+1}]$, the union of the join results of the sliced binary window joins in a chain $\sigma(A[0, W_1] \bowtie B[0, W_1]), \dots, \sigma(A[W_{i-1}, W_i] \bowtie B[W_{i-1}, W_i]), \sigma, A[W_i, W_{i+1}] \bowtie B[W_i, W_{i+1}], \dots, A[W_{N-1}, W_N] \bowtie B[W_{N-1}, W_N]$ is equivalent to the results of a regular sliding window join $\sigma(A[W] \bowtie B[W])$, where $W = W_N$.*

Proof: From Theorem 2, we have:

$$\sigma(A[W] \bowtie B[W]) = \bigcup_{1 < j \leq N} \sigma(A[W_{j-1}, W_j] \bowtie B[W_{j-1}, W_j]).$$

Assume the select operator σ is pushed down into the chain between sliced join operator J_i and J_{i+1} . From Lemma 3, we have:

$$\sigma(A[W] \bowtie B[W]) =$$

$$\bigcup_{1 < j \leq i} \sigma(A[W_{j-1}, W_j] \bowtie B[W_{j-1}, W_j]) \bigcup_{i < j \leq N} \sigma(A[W_{j-1}, W_j]) \bowtie B[W_{j-1}, W_j]$$

Since the state sliced operators are connected in a pipeline, the σ operator between J_i and J_{i+1} will suppress down-stream select operators. All the down-stream select operators thus can be safely removed. ■

We now show how the proposed state-slice sharing can be applied to the running example in Chapter 4 to share the computation between the two queries. The shared plan is depicted in Figure 5.6. This shared query plan includes a chain of two sliced sliding window join operators \Join_1^s and \Join_2^s . The purged tuples from the states of \Join_1^s are sent to \Join_2^s as input tuples. The selection operator σ_A filters the input stream A tuples for \Join_2^s . The selection operator σ'_A filters the joined results of \Join_1^s for Q_2 . The predicates in σ_A and σ'_A are both $A.value > Threshold$.

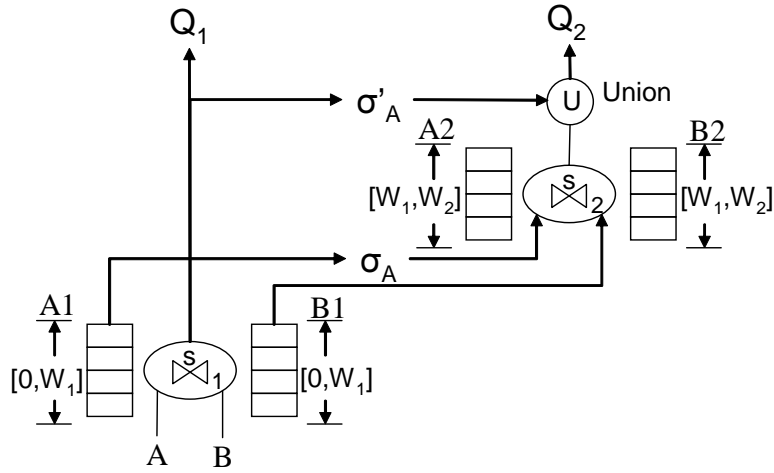


Figure 5.6: State-Slice Sharing for Q_1 and Q_2 .

5.3 Discussion and Analysis

Compared to the alternative sharing approaches discussed in Chapter 4, the state-slice sharing paradigm offers the following benefits:

- Selection can be pushed down into the middle of the join chain. Thus unnecessary probing in the join operators are avoided.
- The routing cost is saved. Instead a pre-determined route is embedded inside the query plan.
- States of the sliced window joins in a chain are disjoint with each other, independent from if synchronized or unsynchronized operator scheduling is used. Thus no state memory is wasted.

Using the same settings as in Chapter 4, we now calculate the state memory consumption C_m and the CPU cost C_p for the state-slice sharing paradigm as follows:

$$\left\{ \begin{array}{l} C_m = 2\lambda W_1 M_t + (1 + S_\sigma)\lambda(W_2 - W_1)M_t \\ C_p = 2\lambda^2 W_1 + \lambda + 2\lambda^2 S_\sigma(W_2 - W_1) + \\ \quad 4\lambda + 2\lambda + 2\lambda^2 S_{\times} W_1 \end{array} \right. \quad (5.1)$$

The first item of C_m corresponds to the state memory in \mathfrak{N}_1^s ; the second to the state memory in \mathfrak{N}_2^s . The first item of C_p is the join probing cost of \mathfrak{N}_1^s ; the second the filter cost of σ_A ; the third the join probing cost of \mathfrak{N}_2^s ; the fourth the cross-purge cost; while the fifth the union cost; the sixth the filter cost of σ'_A .

The union cost in C_p is proportional to the input rates of streams A and B . The reason is that the probe tuple of the last sliced join \bowtie_2^s acts as punctuation [TMSF03] for the union operator. For example, the probe tuple a_1^b is sent to the union operator after it finishes probing the state of stream B in \bowtie_2^s , indicating that no more joined tuples with timestamps smaller than a_1^b will be generated in the future. Such punctuations are used by the union operator for the merge sorting of joined tuples from multiple join operators [TMSF03].

The detail of using the punctuations in the union operator is shown in Figure 5.7. Assume there are n sliced join operators connected to the union operator, then the union operator will have an individual temporary storage buffer for each sliced join operator. Each buffer is filled with buckets of joined results coming from the probing of tuples a_1, b_1, a_2, \dots . The timestamps of the tuples in the buckets $T_{a_1}^1, T_{a_1}^2, \dots, T_{a_1}^n$ are exactly the same, which equal to the timestamp of the probe tuple a_1 . When the probe tuple a_1 arrives at the union operator as a punctuation, this means that all the buckets of a_1 are ready for output. Joined tuples in the buckets for a_1 are then sent out in sequence. Here unlike regular merge sorting, no comparisons are needed at all. The output of the union operator is guaranteed to be ordered, since the input tuples are processed in order. The CPU cost of such sorting then is only related to the number of the punctuations, instead of the number of result tuples.

Comparing the memory and CPU costs for the different sharing solutions, namely naive sharing with selection pull-up (Equation 4.1), stream partition with selection push-down (Equation 4.2) and state-slice chain (Equa-

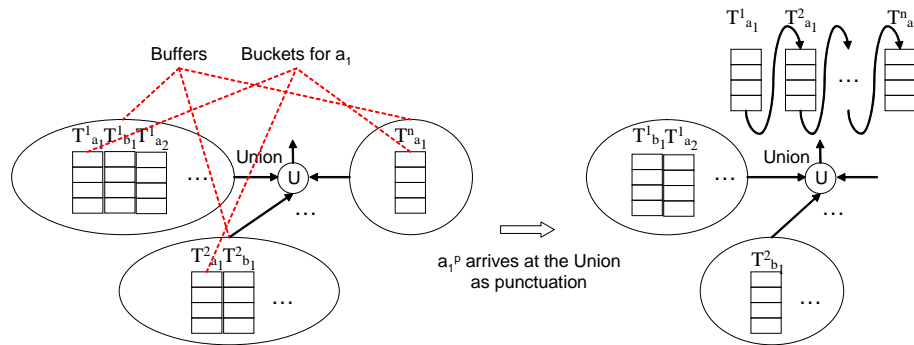


Figure 5.7: The Processing of the Union Operator.

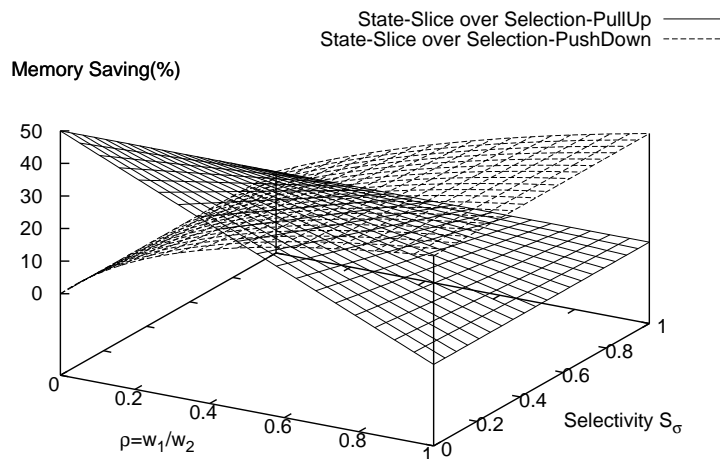


Figure 5.8: Memory Consumption Comparison

tion 5.1), the savings of using the state slicing sharing are:

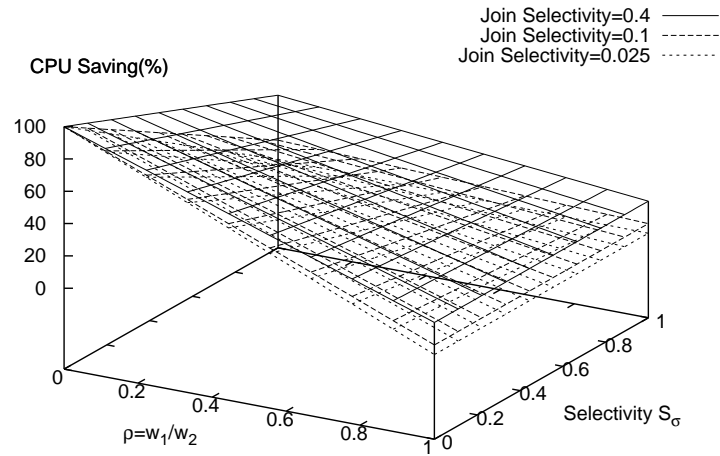


Figure 5.9: CPU Cost Comparison: State-Slice vs. Selection PullUp

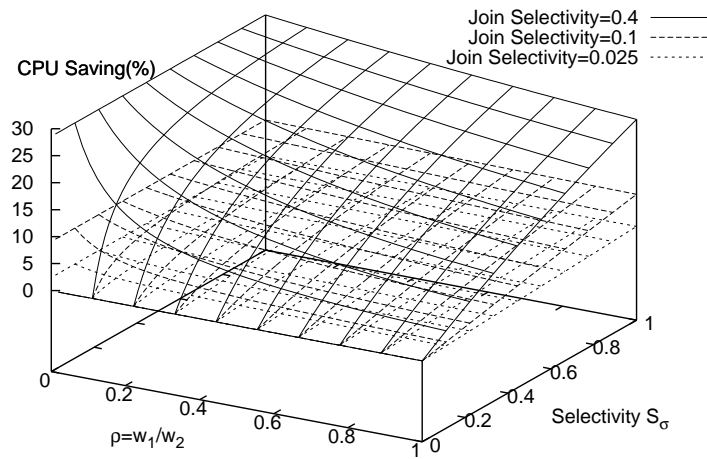


Figure 5.10: CPU Cost Comparison: State-Slice vs. Selection PushDown.

$$\left\{ \begin{array}{l} \frac{C_m^{(1)} - C_m^{(3)}}{C_m^{(1)}} = \frac{(1-\rho)(1-S_\sigma)}{2} \\ \frac{C_m^{(2)} - C_m^{(3)}}{C_m^{(2)}} = \frac{\rho}{1+2\rho+(1-\rho)S_\sigma} \\ \frac{C_p^{(1)} - C_p^{(3)}}{C_p^{(1)}} = \frac{(1-\rho)(1-S_\sigma) + (2-\rho)S_{\bowtie}}{1+2S_{\bowtie}} \\ \frac{C_p^{(2)} - C_p^{(3)}}{C_p^{(2)}} = \frac{S_\sigma S_{\bowtie}}{\rho(1-S_\sigma) + S_\sigma + S_\sigma S_{\bowtie} + \rho S_{\bowtie}} \end{array} \right. \quad (5.2)$$

with $C_m^{(i)}$ denoting C_m , $C_p^{(i)}$ denoting C_p in Equation i ($i = 1, 2, 3$); and window ratio $\rho = \frac{W_1}{W_2}$, $0 < \rho < 1$.

The memory and CPU savings under various settings calculated from Equation 5.2 are depicted in Figures 5.8, 5.9 and 5.10. Compared to the sharing alternatives in Chapter 4, state-slice sharing achieves significant savings. As a base case, when there is no selection in the query plans (i.e., $S_\sigma = 1$), state-slice sharing will consume the same amount of memory as the selection PullUp while the CPU saving is proportional to the join selectivity S_{\bowtie} . When selection exists, state-slice sharing can save about 20%-30% memory, 10%-40% CPU over the alternatives on average. For the extreme settings, the memory savings can reach about 50% and the CPU savings about 100% (Figure 5.8, 5.9). The actual savings are sensitive to these parameters. Moreover, from Equation 5.2 we can see that all the savings are positive. This means that the state-sliced sharing paradigm achieves the lowest memory and CPU costs under all these settings. Note that we omit λ in Equation 5.2 for CPU cost comparison, since its effect is small when the number of queries is only 2. The CPU savings will increase with increasing λ , especially when the number of queries is large.

Chapter 6

Case Study: State-Slice Sharing for Join Tree

This chapter gives a detailed cost analysis on two queries with join trees and selections. This case study intends to show the state slicing sharing for queries with multiple selections and multiple window constraints. With detailed cost calculation, the benefits of the state sliced sharing is illustrated for this case. We will first define the variables used in the cost model. Then we will compare the state memory consumptions of the given queries Q_1 and Q_2 in the following three cases: no sharing, naive sharing with selections pull up and the proposed state-slice sharing. Next we will analyze the CPU cost of the given queries in the same three cases. In the following discussion, the sharing with selections push down is not included for comparisons. The sharing with selection push down will generate exponential amount of sub-stream partitions in case of multiple selections. The mem-

ory usage will be much larger than the other alternatives. Thus selection push down is not considered for sharing of join trees with multiple select operations.

6.1 Continuous Queries and Terms Used in Cost Model

In this chapter, the cost model of state memory usage and CPU cost are both developed for the following two queries: Q_1 and Q_2 . Each of these queries includes selections on the input data streams (A, B, C) and a join tree of $A \bowtie B \bowtie C$. All the join conditions are equi-join. In the cost model, the join ordering of A, B, C is picked. Other cost models can be developed similarly for other join orderings. Figure 6.1 shows the query plans for Q_1 and Q_2 .

$$Q_1 : \sigma_A^1(A) \bowtie \sigma_B^1(B) \bowtie \sigma_C^1(C)$$

$$Q_2 : \sigma_A^2(A) \bowtie \sigma_B^2(B) \bowtie \sigma_C^2(C)$$

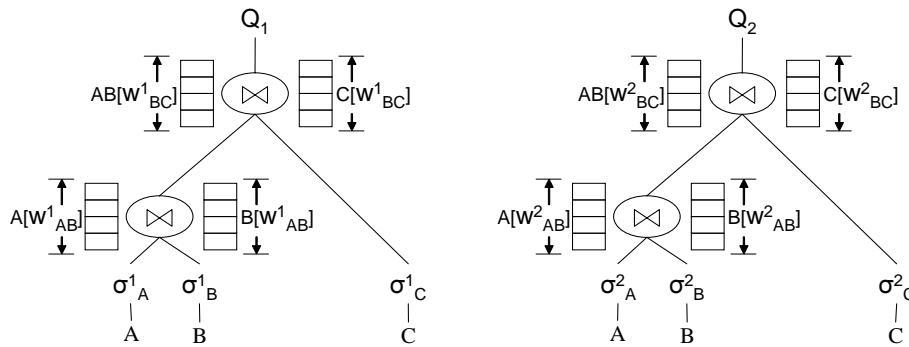


Figure 6.1: Query Plans for Q_1 and Q_2 .

In Figure 6.1, only state memory is shown since the state memory is determined by the semantics of the query and the arrival rates of the streams. The queue memory is not considered in this chapter. To estimate the CPU cost, we consider the cost for value comparison of two tuples and the timestamp comparison. We assume that comparisons are equally expensive and dominate the CPU cost. We thus use the count of comparisons per time unit as the metric for estimated CPU costs.

A list of terms and their meanings used in our model are listed in Table 6.1. Some of the symbols have the same definitions as in Chapter 5. For easy reference, we give descriptions of the full list of terms below. We define the selectivity of σ_A as:

$$\frac{\textit{number_of_outputs}}{\textit{number_of_inputs}}$$

We define the join selectivity S_{\bowtie} as:

$$\frac{\textit{number_of_outputs}}{\textit{number_of_outputs_from_Cartesian_Product}}$$

In order to estimate the state memory and CPU cost spent on the shared plan capturing Q_1 and Q_2 , we first develop a general model that can be applied to each of the queries. Unless necessary, the super scripts in the terms are omitted in the general cost model.

In the cost model, we have the following assumptions:

- All the tuples from stream A , B or C are of the same size. M_t is used to represent the size of all the tuples.

Table 6.1: Terms Used in Cost Model

Term	Meaning
λ_A	Arrival Rate of Stream A (Tuples/Min.)
λ_B	Arrival Rate of Stream B (Tuples/Min.)
λ_C	Arrival Rate of Stream C (Tuples/Min.)
W_{AB}^1	Window Size of $A \bowtie B$ for Q_1 (Min.)
W_{AB}^2	Window Size of $A \bowtie B$ for Q_2 (Min.)
W_{BC}^1	Window Size of $(A \bowtie B) \bowtie C$ for Q_1 (Min.)
W_{BC}^2	Window Size of $(A \bowtie B) \bowtie C$ for Q_2 (Min.)
M_t	Tuple Size (KB)
S_A^1	Selectivity of σ_A^1 for Q_1
S_B^1	Selectivity of σ_B^1 for Q_1
S_C^1	Selectivity of σ_C^1 for Q_1
S_A^2	Selectivity of σ_A^2 for Q_1
S_B^2	Selectivity of σ_B^2 for Q_1
S_C^2	Selectivity of σ_C^2 for Q_1
$S_{A \bowtie B}$	Join Selectivity of $A \bowtie B$
$S_{B \bowtie C}$	Join Selectivity of $B \bowtie C$
$ State $	Memory used by tuples in the state (KB)

- The join result is the combination of the matched input tuples. For example, the tuple size of join result $A \bowtie B$ is $2M_t$.
- The selection predicates are on different columns with the join columns. The join selectivities used in different queries can be assumed to be the same as each other. Also, the predicates are independent with each other and the selectivity of combined predicates can be calculated by product of individual selectivities.
- A uniform distribution of the values in the join columns is assumed.

A similar cost model can be developed with all these assumptions dropped. However with increased complexity, no essential benefit is achieved. All the assumptions do not change the nature of the cost model.

6.2 Strategies of Sharing Queries

In this chapter, two sharing strategies are compared in the cost model. The selection PullUp sharing and the State-Slice sharing.

6.2.1 Selection PullUp Sharing

The PullUp or Filtered PullUp approaches proposed in [CDN02] for sharing continuous query plans containing joins and selections can be applied to the sharing of joins with different window sizes. That is, we need to introduce a router operator to dispatch the joined results to the respective query outputs. The intuition behind such sharing lies in that the answer of the join for query with the smaller window is contained in the join for

query with the larger window. The shared query plan for Q_1 and Q_2 is shown in Figure 6.2.

Without loss of generality, we let $0 < W_{AB}^1 < W_{AB}^2$ and $0 < W_{BC}^2 < W_{BC}^1$. This ordering is picked for the purpose of showing the comparisons with arbitrary order among windows. For simplicity, in the following computation, we set $\lambda_A = \lambda_B = \lambda_C$, denoted as λ . The analysis can be extended similarly for unbalanced input stream rates.

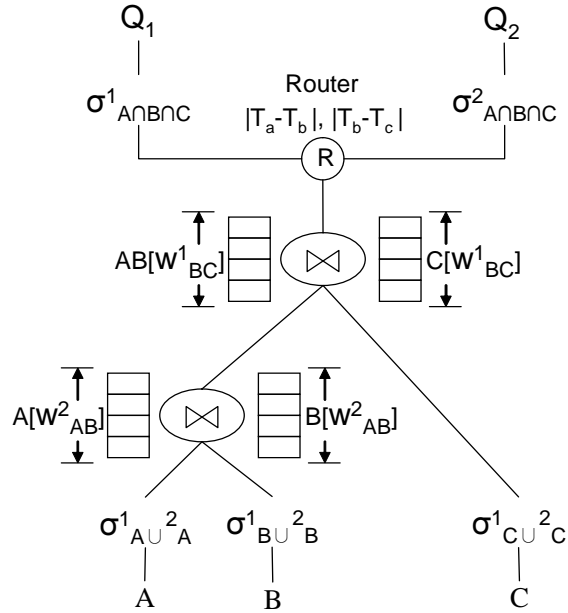


Figure 6.2: Selection PullUp Sharing Query Plan.

By performing the sliding window join first with the larger window size among the queries Q_1 and Q_2 , computation sharing is achieved. The router then checks the timestamps of each joined tuple with the window constraints of registered CQs and dispatches them correspondingly. Finally the selections filter the joined results according to the predicates for each of

the queries.

6.2.2 State-Slice Sharing

We now show how the proposed state-slice sharing can be applied to the running example of Q_1 and Q_2 to share the computation between the two queries. The shared plan is depicted in Figure 6.3.

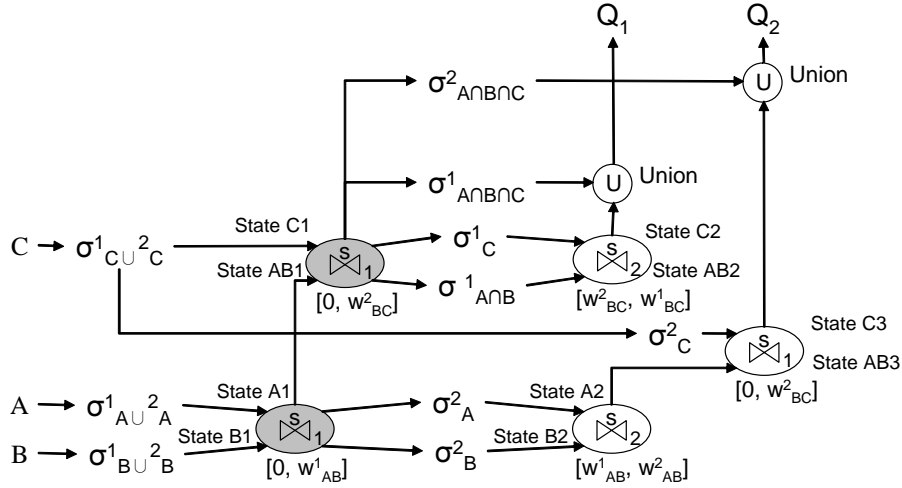


Figure 6.3: State-Slice Sharing for Q_1 and Q_2 .

Given $0 < W_{AB}^1 < W_{AB}^2$ and $0 < W_{BC}^2 < W_{BC}^1$, the shared query plan includes three chain of sliced sliding window join operators. The first chain computes $A \bowtie B$ and generates two disjoint join results. One of the join results are feed to the second chain to join with stream C . Similarly, the other join results are feed to the third chain¹. The gray state sliced join operators are shared by Q_1 and Q_2 . All the predicates are pushed down as low as possible.

¹This chain has one state slice join only, since $0 < W_{BC}^2 < W_{BC}^1$.

6.3 Cost Model for State Memory Consumption

Main memory is used for the states of the join operators (*state memory*) and queues between operators (*queue memory*). State memory is determined by the window constraints in the continuous queries. The size of the state memory is independent from the runtime environment. That is, no matter how fast the machine is, the state memory is unchanged. In this chapter, we focus on the state memory only.

State memory consumption C_m (m stands for memory) is calculated for the isolated execution without sharing, selection PullUp sharing and the proposed state-slice sharing in the following sections.

6.3.1 Isolated Execution without Sharing

As a baseline, we first calculate the memory consumption for Q_1 and Q_2 without sharing. In Figure 6.1, the query plan of Q_1 is exactly the same as Q_2 's, ignoring the parameters. We first develop the cost model for generic query tree and calculate the memory usage for Q_1 and Q_2 individually.

For a generic query tree, let $|A|$, $|B|$, $|C|$ and $|AB|$ stand for the size of the states, we have:

$$|A| = \lambda_A S_A W_{AB} M_t$$

$$|B| = \lambda_B S_B W_{AB} M_t$$

$$|C| = \lambda_C S_C W_{BC} M_t$$

$$|AB| = [\lambda_A S_A (\lambda_B S_B W_{AB}) S_{A \bowtie B} + \lambda_B S_B (\lambda_A S_A W_{AB}) S_{A \bowtie B}] W_{BC} 2M_t$$

So let $\lambda_A = \lambda_B = \lambda_C$ be denoted by λ :

$$\begin{aligned}
 C_m &= |A| + |B| + |C| + |AB| \\
 &= [\lambda(S_A + S_B)W_{AB} + \lambda S_C W_{BC} + 4\lambda^2 S_A S_B S_{A \bowtie B} W_{AB} W_{BC}] M_t
 \end{aligned} \tag{6.1}$$

Assume the values of the parameters in Table 6.2 for Q_1 and Q_2 are plugged into, we have:

$$Q_1 : C_m = 75.8MB$$

$$Q_2 : C_m = 2253.375MB = 2.25GB$$

Table 6.2: Value of Terms Used in Cost Model

Term	Value
λ	1K (Tuples/Min.)
W_{AB}^1	1 (Min.)
W_{AB}^2	60 (Min.)
W_{BC}^1	30 (Min.)
W_{BC}^2	15 (Min.)
M_t	0.1 (KB)
S_A^1	0.25
S_B^1	0.25
S_C^1	0.25
S_A^2	0.25
S_B^2	0.25
S_C^2	0.25
$S_{A \bowtie B}$	0.1
$S_{B \bowtie C}$	0.1

6.3.2 Selection PullUp Sharing

Figure 6.2 shows the selection PullUp sharing query plan. Here we assume that the predicates of Q_1 and Q_2 are 80% overlapped. That is, the selectivity of $\sigma_A^1 \cup_A^2$ (denoted as $S_{A_{1 \cup 2}}$, \cup means “or” here) is:

$$S_{A_{1 \cup 2}} = S_A^1 + S_A^2 - 0.8S_A^1 = 0.3$$

Similarly rule is followed for B and C .

Let $|A|$, $|B|$, $|C|$ and $|AB|$ stand for the size of the states, we have:

$$\begin{aligned} |A| &= \lambda_A S_{A_{1 \cup 2}} W_{AB}^2 M_t \\ |B| &= \lambda_B S_{B_{1 \cup 2}} W_{AB}^2 M_t \\ |C| &= \lambda_C S_{C_{1 \cup 2}} W_{BC}^1 M_t \\ |AB| &= [\lambda_A S_{A_{1 \cup 2}} (\lambda_B S_{B_{1 \cup 2}} W_{AB}^2) S_{A \bowtie B} + \lambda_B S_{B_{1 \cup 2}} (\lambda_A S_{A_{1 \cup 2}} W_{AB}^2) S_{A \bowtie B}] W_{BC}^1 2M_t \end{aligned}$$

So let $\lambda_A = \lambda_B = \lambda_C$ be denoted by λ , then:

$$\begin{aligned} C_m &= |A| + |B| + |C| + |AB| \\ &= [2\lambda S_{A_{1 \cup 2}} W_{AB}^2 + \lambda S_{A_{1 \cup 2}} W_{BC}^1 + 4\lambda^2 (S_{A_{1 \cup 2}})^2 S_{A \bowtie B} W_{AB}^2 W_{BC}^1] M_t \end{aligned} \tag{6.2}$$

Note that $W_{AB}^2 \neq W_{AB} W_{AB}$. We will use $(W_{AB}^2)^2$ to represent $W_{AB}^2 W_{AB}^2$.

Assume the parameters in Table 6.2 for Q_1 and Q_2 are given. Then we have:

$$C_m = 6484.5MB \approx 6.5GB$$

6.3.3 State-Slice Sharing

Figure 6.3 shows the state-slice sharing paradigm for the example queries. Five state sliced joins are used in our example. The states are $A1$, $A2$, $B1$, $B2$, $C1$, $C2$, $C3$, $AB1$, $AB2$, and $AB3$. Same as the previous section, we have:

$$S_{A1 \cup 2} = S_A^1 + S_A^2 - 0.8S_A^1 = 0.3$$

Similarly, we can calculate the state memory usage as follows:

$$\begin{aligned} |A1| &= \lambda_A S_{A1 \cup 2} W_{AB}^1 M_t \\ |A2| &= \lambda_A S_A^2 (W_{AB}^2 - W_{AB}^1) M_t \\ |B1| &= \lambda_B S_{B1 \cup 2} W_{AB}^1 M_t \\ |B2| &= \lambda_B S_B^2 (W_{AB}^2 - W_{AB}^1) M_t \\ |C1| &= \lambda_C S_{C1 \cup 2} W_{BC}^2 M_t \\ |C2| &= \lambda_C S_C^1 (W_{BC}^1 - W_{BC}^2) M_t \\ |C3| &= \lambda_C S_C^2 W_{BC}^2 M_t \\ |AB1| &= [\lambda_A S_{A1 \cup 2} \frac{|B1|}{M_t} S_{A \bowtie B} + \lambda_B S_{B1 \cup 2} \frac{|A1|}{M_t} S_{A \bowtie B}] W_{BC}^2 2M_t \\ |AB2| &= [\lambda_A S_A^1 (\lambda_B S_B^1 W_{AB}^1) S_{A \bowtie B} + \lambda_B S_B^1 (\lambda_A S_A^1 W_{AB}^1) S_{A \bowtie B}] (W_{BC}^1 - W_{BC}^2) 2M_t \\ |AB3| &= [\lambda_A S_A^2 \frac{|B2|}{M_t} S_{A \bowtie B} + \lambda_B S_B^2 \frac{|A2|}{M_t} S_{A \bowtie B}] W_{BC}^2 2M_t \end{aligned}$$

Thus:

$$C_m = |A1| + |A2| + |B1| + |B2| + |C1| + |C2| + |C3| + |AB1| + |AB2| + |AB3| \quad (6.3)$$

Assume the parameters in Table 6.2 for Q_1 and Q_2 are given. We have:

$$C_m = 0.03+1.475+0.03+1.475+0.45+0.375+0.375+54+37.5+2212.5 = 2308.21MB$$

6.3.4 Comparison and Analysis

From the above calculations, we now can summarize the results as follows:

- Isolated Execution: $C_m = 2325.8MB$.
- Selection PullUp Sharing: $C_m = 6484.5MB$.
- State-slice Sharing: $C_m = 2308.21MB$.

We can see that the selection PullUp sharing consumes the largest memory. Obviously, selection PullUp will largely increase the state memory requirement. In this example, the selection PullUp sharing will consume about three times of the memory as the other two strategies.

From comparison, the state-slice sharing and isolated execution consumes almost the same amount of memory. We can see that the state $AB3$ in Figure 6.3 and the state AB for Q_2 in Figure 6.2 dominate the memory consumptions respectively. These two states are almost of the same size. Intuitively, since $W_{AB}^1 \ll W_{AB}^2$, little sharing is achieved. Otherwise, huge difference is possible in general, as further shown in this section. In this section, several important parameters are defined and the performance comparisons under different system settings are discussed.

We noticed that the window constraints are important parameters in the cost model. For easy illustration, we define following two parameters:

$$\begin{aligned} m &= \frac{W_{AB}^1}{W_{AB}^2} \\ n &= \frac{W_{BC}^2}{W_{BC}^1} \end{aligned}$$

Since we assume $W_{AB}^1 \leq W_{AB}^2$ and $W_{BC}^2 \leq W_{BC}^1$, we have:

$$0 < m \leq 1, 0 < n \leq 1$$

Thus in this example, we can rewrite the windows as:

$$W_{AB}^1 = m * 60, W_{AB}^2 = 60, W_{BC}^1 = n * 30, W_{BC}^2 = 60$$

Let $C_{m'}^1$, C_m^2 and C_m^3 denote the memory consumption for isolated execution, selection PullUp sharing and state slice sharing respectively. Assume the values for the other parameters from Table 6.2. Then we compare the Equation 6.1, 6.2 and 6.3 as follows:

$$\begin{aligned} \frac{C_m^3}{C_m^1} &= \frac{1500(m+n) - 840mn}{1500(m+n)} \\ \frac{C_m^3}{C_m^2} &= \frac{1500(m+n) - 840mn}{2160} \end{aligned} \tag{6.4}$$

The memory consumptions under various settings calculated using Equation 6.4 are depicted in Figures 6.4 and 6.5. Compared to sharing alternatives, state-slice sharing achieves significant savings of memory. State-slice

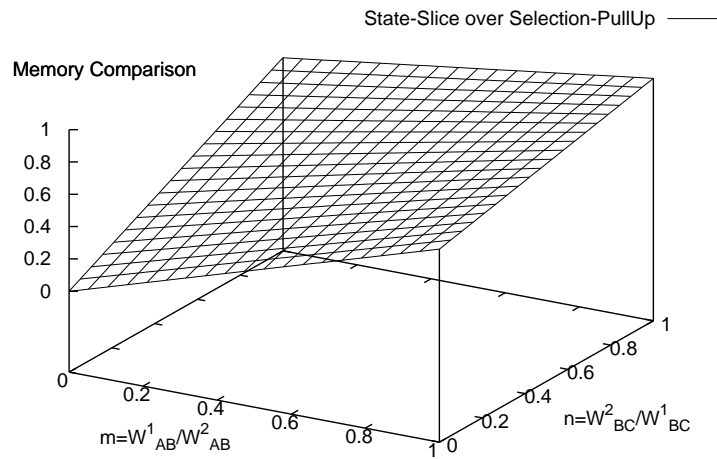


Figure 6.4: Memory Consumption Comparison: State-Slice Sharing vs. Selection PullUp Sharing.

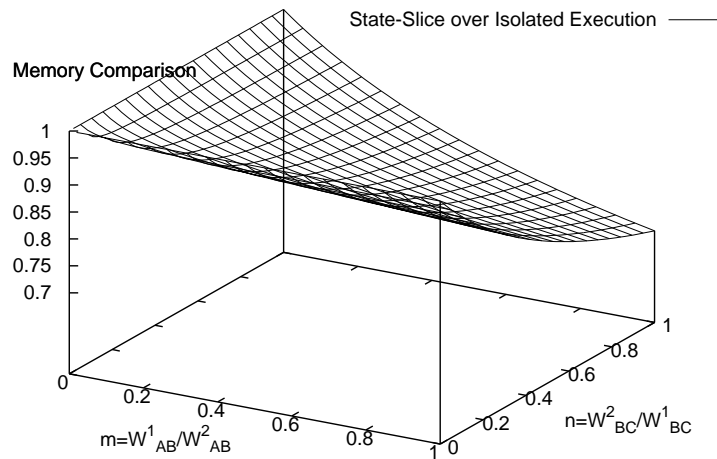


Figure 6.5: Memory Consumption Comparison: State-Slice Sharing vs. Isolated Execution.

sharing can save about 20%-30% memory over the alternatives on average. The actual savings are determined by these parameters. Moreover, from Equation 6.4 we can see that the state-sliced sharing paradigm achieves the lowest memory consumption under all these settings.

6.4 Cost Model for CPU Consumption

To estimate the CPU cost, we consider the cost for the value comparisons of two tuples and the timestamp comparisons. We assume that comparisons are equally expensive and dominate the CPU cost. We thus use the count of the number of comparisons per time unit as the metric for estimated CPU costs. In this chapter, we calculate the CPU cost using the nested-loop join algorithm. Calculation using the hash-based join algorithm can be done similarly using an adjusted cost model.

In Figures 6.2 and 6.3 there are several selection operators that have conjunctive predicates. Here I assume that only one comparison is needed to evaluate such conjunctive predicate. The reason is that the conjunctive predicates appear only at the last several steps in the query plan and each component of the predicates have been evaluated somewhere below in the query plan. That is, to avoid evaluating the same predicates against, each tuple can have a code indicating the previous evaluation history. Thus later evaluation of the conjunctive predicates only needs one comparison of the associated code. For disjunctive predicates, the number of comparisons is equal to the number of the primary predicates.

CPU cost C_p (p stands for processor) is calculated for the isolated execu-

tion without sharing, selection PullUp sharing and the proposed state-slice sharing in the following sections.

6.4.1 Isolated Execution without Sharing

As a baseline, we first calculate the CPU costs for Q_1 and Q_2 without sharing. In Figure 6.1, the query plan shape of Q_1 is exactly the same as Q_2 's, ignoring the parameters. We first develop the cost model for the generic query tree and calculate the CPU costs for Q_1 and Q_2 individually.

For a generic query tree, let C_{filter} , C_{purge} and C_{probe} denote the filtering, purge and join probing costs respectively. We have: (where λ_{AB} denotes the arrival rate at $B \bowtie C$)

$$\begin{aligned}\lambda_{AB} &= 2\lambda_A S_A (\lambda_B S_B W_{AB}) S_{A \bowtie B} \\ C_{filter} &= \lambda_A + \lambda_B + \lambda_C \\ C_{purge} &= \lambda_A S_A + \lambda_B S_B + \lambda_C S_C + \lambda_{AB} \\ C_{probe} &= \frac{\lambda_{AB}}{S_{A \bowtie B}} + 2\lambda_{AB} \lambda_C S_C W_{BC}\end{aligned}$$

We have:

$$C_p = C_{filter} + C_{purge} + C_{probe} \quad (6.5)$$

Assume the following parameters in Table 6.2 for Q_1 and Q_2 are given.

We have:

$$Q_1 : C_p = 187.64 * 10^6$$

$$Q_2 : C_p = 5633.25 * 10^6$$

6.4.2 Selection PullUp Sharing

Let C_{filter} , C_{purge} , C_{probe} and C_{route} denote the filtering, purge, join probing and tuple routing cost respectively, we have: (where λ_{AB} denotes the arrival input rate at $B \bowtie C$. λ_{ABC} denotes the output rate at $B \bowtie C$)

$$\lambda_{AB} = 2\lambda_A S_{A_{1U_2}} (\lambda_B S_{B_{1U_2}} W_{AB}^2) S_{A \bowtie B}$$

$$\lambda_{ABC} = 2\lambda_{AB} (\lambda_C S_{C_{1U_2}} W_{BC}^1) S_{B \bowtie C}$$

$$C_{filter} = \lambda_A + \lambda_B + \lambda_C + 2\lambda_{ABC}$$

$$C_{purge} = \lambda_A S_{A_{1U_2}} + \lambda_B S_{B_{1U_2}} + \lambda_C S_{C_{1U_2}} + \lambda_{AB}$$

$$C_{probe} = \frac{\lambda_{AB}}{S_{A \bowtie B}} + \frac{\lambda_{ABC}}{S_{B \bowtie C}}$$

$$C_{route} = 2\lambda_{ABC}$$

We have:

$$C_p = C_{filter} + C_{purge} + C_{probe} + C_{route} \quad (6.6)$$

Assume the parameters in Table 6.2 for Q_1 and Q_2 is given. We have:

$$C_p = 27227.88 * 10^6$$

6.4.3 State-Slice Sharing

Figure 6.3 shows the state-slice sharing paradigm for the example queries.

Similar to previous section, we have:

$$S_{A_1 \cup 2} = S_A^1 + S_A^2 - 0.8S_A^1 = 0.3$$

Let:

λ_{AB_1} denotes the output rate at $A_1 \bowtie B_1$

λ_{AB_2} denotes the output rate at $A_2 \bowtie B_2$

λ_{ABC_1} denotes the output rate at $AB_1 \bowtie C_1$

λ_{ABC_2} denotes the output rate at $AB_2 \bowtie C_2$

λ_{ABC_3} denotes the output rate at $AB_3 \bowtie C_3$;

Let C_{filter} , C_{purge} , C_{probe} and C_{union} denote the filtering cost, purge cost,

join probing cost and union cost respectively, we have:

$$\begin{aligned}
\lambda_{AB_1} &= 2\lambda_A S_{A_1 \cup 2} (\lambda_B S_{B_1 \cup 2} W_{AB}^1) S_{A \bowtie B} \\
\lambda_{AB_2} &= 2\lambda_A S_A (\lambda_B S_B (W_{AB}^2 - W_{AB}^1)) S_{A \bowtie B} \\
\lambda_{ABC_1} &= 2\lambda_{AB_1} (\lambda_C S_{C_1 \cup 2} W_{BC}^2) S_{B \bowtie C} \\
\lambda_{ABC_2} &= 2 \frac{\lambda_{AB_1} S_A S_B}{S_{A_1 \cup 2} S_{B_1 \cup 2}} (\lambda_C S_C (W_{BC}^1 - W_{BC}^2)) S_{B \bowtie C} \\
\lambda_{ABC_3} &= 2\lambda_{AB_2} (\lambda_C S_C W_{BC}^2) S_{B \bowtie C} \\
C_{filter} &= 2\lambda_A + 2\lambda_B + 3\lambda_C + 2\lambda_{ABC_1} + \lambda_{AB_1} \\
C_{purge} &= 2\lambda_A S_{A_1 \cup 2} + 2\lambda_B S_{B_1 \cup 2} + 3\lambda_C S_{C_1 \cup 2} + 2\lambda_{AB_1} + \lambda_{AB_2} \\
C_{probe} &= \frac{\lambda_{AB_1} + \lambda_{AB_2}}{S_{A \bowtie B}} + \frac{\lambda_{ABC_1} + \lambda_{ABC_2} + \lambda_{ABC_3}}{S_{B \bowtie C}} \\
C_{union} &= 2\lambda_{ABC_1} + \lambda_{ABC_2} + \lambda_{ABC_3}
\end{aligned}$$

We have:

$$C_p = C_{filter} + C_{purge} + C_{probe} + C_{union} \quad (6.7)$$

Assume the parameters in Table 6.2 for Q_1 and Q_2 is given. We have:

$$C_p = 6422.66 * 10^6$$

6.4.4 Comparison and Analysis

From the prior calculations, we now have the following results:

- Isolated Execution: $C_p = 5820.89 * 10^6$.

- Selection PullUp Sharing: $C_p = 27227.88 * 10^6$.
- State-slice Sharing: $C_p = 6422.66 * 10^6$.

We can see that the selection PullUp sharing has the largest CPU cost. Obviously, selection PullUp will largely increase the CPU requirements. In this example, the selection PullUp sharing will consume about 5 times more of the CPU power than the other two strategies.

The state-slice sharing strategy uses little more CPU resources than the isolated execution. We can see that the probe cost at $AB_3 \bowtie C_3$ in Figure 6.3 dominates the CPU consumptions ($5531.25 * 10^6$), which is not the sharing part. Intuitively, since $W_{AB}^1 \ll W_{AB}^2$, little sharing is achieved. However, the union cost of the state-slice join ($595 * 10^6$) now is a more significant factor, in spite of being linear to the total output.

Further in this section, several important parameters are defined and the performance comparisons under different system settings are discussed.

Similarly to the memory analysis, we noticed that the window constraints are important parameters in the cost model. We define the following two parameters:

$$m = \frac{W_{AB}^1}{W_{AB}^2}$$

$$n = \frac{W_{BC}^2}{W_{BC}^1}$$

Assume $W_{AB}^1 \leq W_{AB}^2$ and $W_{BC}^2 \leq W_{BC}^1$, we have:

$$0 < m \leq 1, 0 < n \leq 1$$

Thus in this example, the windows can be rewritten as:

$$W_{AB}^1 = m * 60, W_{AB}^2 = 60, W_{BC}^1 = n * 30, W_{BC}^2 = 60$$

Let C_p^1 , C_p^2 and C_p^3 denote the CPU costs for isolated execution, selection PullUp sharing and state slice sharing respectively. Assume the values plugged in as indicated in Table 6.2. Then we compare the Equations 6.5, 6.6 and 6.7 as follows:

$$\frac{C_p^3}{C_p^1} = \frac{228.61mn + 206.33m + 18.75n + 0.1375}{187.5(m + n) + 0.125m + 0.125} \quad (6.8)$$

$$\frac{C_p^3}{C_p^2} = \frac{228.61mn + 206.33m + 18.75n + 0.1375}{453.798}$$

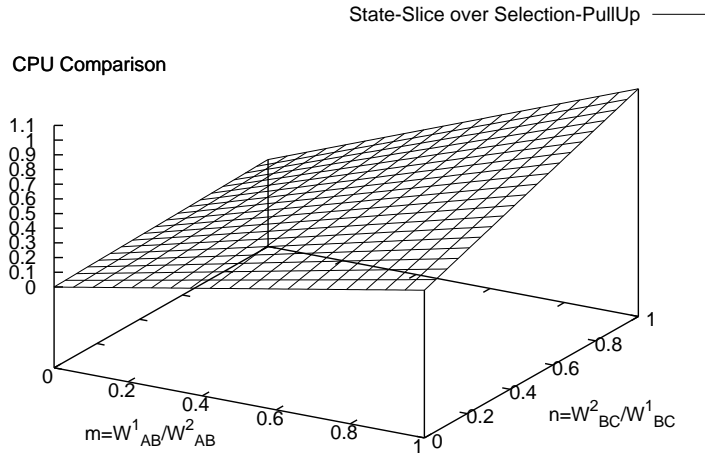


Figure 6.6: CPU Cost Comparison: State-Slice Sharing vs. Selection PullUp Sharing.

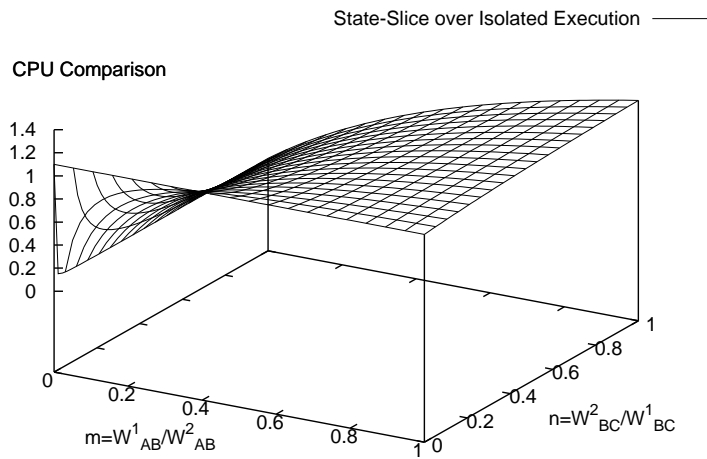


Figure 6.7: CPU Cost Comparison: State-Slice Sharing vs. Isolated Execution.

The CPU cost comparisons under various settings calculated from Equation 6.8 are depicted in Figures 6.6 and 6.7. Compared to the sharing alternatives, state-slice sharing achieves significant savings of CPU resources for most of the situations. The actual savings are determined by these parameters.

Chapter 7

State-slice: Building the Chain

In this chapter, we discuss how to build an optimal shared query plan with a chain of sliced window joins. Consider a DSMS with N registered continuous queries, where each query performs a sliding window join $A[w_i] \bowtie B[w_i]$ ($1 \leq i \leq N$) over data streams A and B . The shared query plan is a DAG with multiple roots, one for each of the queries.

Given a set of continuous queries, the queries are first sorted by their window lengths in ascending order. We propose two algorithms for building the state-slicing chain (Chapters 7.1 and 7.2). The choice between them depends on the availability of CPU versus memory resources in the system. The chain can also first be built using one of the two algorithms and then later migrated towards the other by merging or splitting the slices at runtime (Chapter 7.3).

7.1 Memory-Optimal State-Slicing and its Cost Analysis

Without loss of generality, we assume that $w_i < w_{i+1}$ ($1 \leq i < N$). Let's consider a chain of the N sliced joins: J_1, J_2, \dots, J_N , with J_i as $A[w_{i-1}, w_i] \bowtie B[w_{i-1}, w_i]$ ($1 \leq i \leq N, w_0 = 0$). A union operator U_i is added to collect joined results from J_1, \dots, J_i for query Q_i ($1 < i \leq N$), as shown in Figure 7.1. We call this chain the *memory-optimal state-slice sharing* (Mem-Opt).

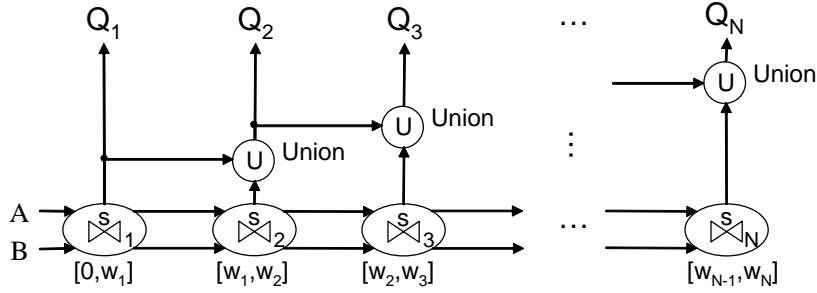


Figure 7.1: Mem-Opt State-Slice Sharing.

The correctness of Mem-Opt state-slice sharing is proven in Theorem 5 by using Theorem 2. We have the following equivalence for i ($1 \leq i \leq N$):

$$Q_i : A[w_i] \bowtie B[w_i] = \bigcup_{1 \leq j \leq i} A[W_{j-1}, W_j] \bowtie B[W_{j-1}, W_j]$$

Theorem 5 *The total state memory used by a Mem-Opt chain of sliced joins J_1, J_2, \dots, J_N , with J_i as $A[w_{i-1}, w_i] \bowtie B[w_{i-1}, w_i]$ ($1 \leq i \leq N, w_0 = 0$) is equal to the state memory used by the regular single sliding window join: $A[w_N] \bowtie B[w_N]$.*

Proof: From Lemma 1, the maximum timestamp difference of tuples (e.g.,

A tuples) in the state of J_i is $(w_i - w_{i-1})$, when continuous tuples from the other stream (e.g., B tuples) are processed. Assume the arrival rate of streams A and B is denoted by λ_A and λ_B respectively. Then we have:

$$\begin{aligned} & \sum_{1 \leq i \leq N} Mem_{J_i} \\ &= (\lambda_A + \lambda_B)[(w_1 - w_0) + (w_2 - w_1) + \dots + (w_N - w_{N-1})] \\ &= (\lambda_A + \lambda_B)w_N \end{aligned}$$

■

$(\lambda_A + \lambda_B)w_N$ is the minimal amount of state memory that is required to generate the full joined result for Q_N . Thus the Mem-Opt chain consumes the minimal state memory.

Let's again use the count of comparisons per time unit as the metric for estimated CPU costs. Comparing the execution (Figure 5.5) of a sliced window join with the execution (Figure 3.2) of a regular window join, we notice that the probing cost of the chain of sliced joins: J_1, J_2, \dots, J_N is equivalent to the probing cost of the regular window join: $A[w_N] \bowtie B[w_N]$.

Comparing the alternative sharing paradigms in Chapter 4, we notice that the Mem-Opt chain may not always win since it requires CPU costs for: (1) $(N - 1)$ more times of purging for each tuple in the streams A and B ; (2) extra system overhead for running more operators; and (3) CPU cost for $(N - 1)$ union operators. In the case that the selectivity of the join S_{\bowtie} is rather small, the routing cost in the selection pull-up sharing may be less than the extra cost of the Mem-Opt chain. In short, the Mem-Opt chain may

not be the CPU-optimal solution for all settings.

7.2 CPU-Optimal State-Slicing

We hence now discuss how to find the CPU-Optimal state-slice sharing (*CPU-Opt*) which will yield minimal CPU costs. We notice that the Mem-Opt state-slice sharing may result in a large number of sliced joins with very small window ranges each. In such cases, the extra per tuple purge cost and the system overhead for holding more operators may not be ignored.

In Figure 7.2(b), the state-sliced joins from J_i to J_j are merged into a larger sliced join with the window range being the summation of the window ranges of J_i and J_j . A routing operator then is added to split the joined results to the associated queries. Such merging of concatenated sliced joins can be done iteratively until all the sliced joins are merged together. In the extreme case, the totally merged join results in a shared query plan, which is equal to that formed by using the selection pull-up sharing method shown in Chapter 4. The CPU costs may decrease after this merge.

Both the shared query plans in Figure 7.2 have the same join probing costs and union costs. Using the symbols defined in Chapter 4 and C_{sys} denoting the system overhead factor, we can calculate the difference of partial CPU cost $C_p^{(a)}$ in Figure 7.2(a) and $C_p^{(b)}$ in Figure 7.2(b) as:

$$C_p^{(a)} - C_p^{(b)} = (\lambda_A + \lambda_B)(j - i) - 2\lambda_A\lambda_B(w_j - w_{i-1})\sigma_{\bowtie}(j - i) + C_{sys}(j - i + 1)(\lambda_A + \lambda_B)$$

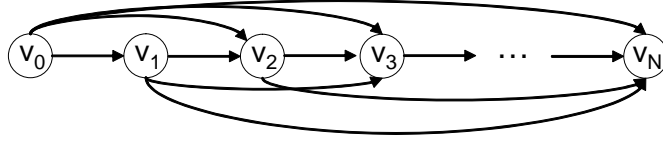


Figure 7.3: Directed Graph of State-Slice Sharing.

Similar to the above calculation of $C_p^{(a)}$ and $C_p^{(b)}$, we can calculate the CPU cost of the merged sliced window joins represented by every edge. We denote the CPU cost $c_{i,j}$ of the sliced join as the length of the edge $l_{i,j}$. We have the following lemma.

Lemma 4 *The calculations of CPU costs $l_{i,j}$ and $l_{m,n}$ are independent if $0 \leq i < j \leq m < n \leq N$.*

The proof of Lemma 4 is straightforward since when $l_{i,j}$ and $l_{m,n}$ do not overlap, the CPU costs $c_{i,j}$ and $c_{m,n}$ are unrelated to each other.

Based on Lemma 4, we can apply the *principle of optimality* [Ata99] here and transform the optimal state-slice problem to the problem of finding the shortest path from v_0 to v_N in an acyclic directed graph. Using the well-known *Dijkstra's algorithm* [Dij59], we can find the CPU-Opt query plan in $O(N^2)$, with N being the number of the distinct window constraints in the system. Even when we incorporate the calculation of the CPU cost of the $\frac{N(N+1)}{2}$ edges, the total time for getting the CPU optimal state-sliced sharing is still $O(N^2)$.

In case the queries do not have selections, the CPU-Opt chain will consume the same amount of memory as the Mem-Opt chain. With selections, the CPU-Opt chain may consume more memory. See Chapter 7.4 for more

discussion of pushing selections into the chain.

7.3 Online Migration of the State-Slicing Chain

Online migration of the shared query plan is important for efficient processing of stream queries. The state-slicing chain may need maintenance when: (1) queries enter or leave the system, (2) queries update predicates or window constraints, and (3) fluctuations in the runtime stream statistic may arise.

The chain migration can be achieved by two primitive operations: merging and splitting of the sliced join. For example when query Q_i ($i < N$) leaves the system, the corresponding sliced join $A[w_{i-1}, w_i] \bowtie B[w_{i-1}, w_i]$ could be merged with the next sliced join in the chain. Or on the contrary when a new query arrives, certain sliced join may need to be split.

The execution steps to be followed for the online splitting of the sliced join J_i are shown in Figure 7.4.

-
1. Stopping the system execution for J_i .
 2. Updating the end window of J_i to w'_i , where $w_{i-1} < w'_i < w_i$.
 3. inserting a new sliced join J'_i with window $[w'_i, w_i]$ to the right of J_i in the query plan.
 4. Connecting the output queues of J_i to the corresponding input queues of J'_i .
 5. Resuming the execution.
-

Figure 7.4: Online Splitting of the Sliced Join J_i .

Intuitively since the queue between J_i and J'_i is empty right after the insertion, then after resuming the execution, the execution of J_i will purge

tuples, due to its new smaller window, into the queue between J_i and J'_i and eventually fill up the states of J'_i .

Lemma 5 *The online splitting steps shown in Figure 7.4 will generate correct joined results without missing or duplicate tuples.*

Proof: (1). *No missing result.* Without loss of generality, we only consider the case of arrival of a new b tuple. After resume the execution, in the cross-purge step (Figure 5.2) of the sliced join J_i , the arriving b will purge any tuple a with $T_b - T_a \geq w'_i$. Thus $\forall a_i \in A :: [w_{i-1}, w'_i], T_b - T_{a_i} < w'_i$. Thus any joined result (a, b) with $T_b - T_a < w'_i$ will be generated in the first sliced join J_i .

The state tuple a_i with $w'_i \leq T_b - T_{a_i} < w_{i+1}$ will be purged from J_i and inserted into the states of down-stream sliced join J'_i . When the b tuple is processed by J'_i , the a_i tuple will stay in the states since $w'_i \leq T_b - T_{a_i} < w_{i+1}$. Thus any joined result (a, b) with $w'_i \leq T_b - T_a < w_{i+1}$ will be generated in the new inserted sliced join J'_i .

(2). *No duplicate result.* According to the probing step (Figure 5.2) of the sliced join J_i , and J'_i , each joined result will only be generated once. The reason is that the states of J_i , and J'_i are disjoint at any time. ■

Online merging of two adjacent sliced joins J_i and J_{i+1} requires the queues between these two joins to be empty. This can be achieved by scheduling the execution of J_{i+1} but stopping the scheduling of J_i . Thus J_i will stop put any new tuples into the queues and J_{i+1} will continuously consume the tuples from the queue. Eventually the queues between them

will be empty.

Once the queue between J_i and J_{i+1} is empty, we can follow the steps of online merging shown in Figure 7.5.

-
1. Attaching the states J_{i+1} to the corresponding states of J_i .
 2. Updating the end window of J_i to w_{i+1} , which is the end window of J_{i+1} .
 3. Removing J_{i+1} from the chain and connect the corresponding queues of J_i and J_{i+2} . Here J_{i+2} is the down-stream sliced join of J_{i+1} .
 4. Resuming the execution.
-

Figure 7.5: Online Merging of the Sliced Join J_i and J_{i+1} .

Intuitively since the queue between J_i and J_{i+1} is empty right before the merging, the states of J_{i+1} can be attached to the corresponding states of J_i without loss of any state tuples in between of J_i and J_{i+1} .

Lemma 6 *The online merging steps shown in Figure 7.5 will generate correct joined results without missing or duplicate tuples.*

Proof: (1). *No missing result.* Without loss of generality, we only consider the case that the last tuple processed by J_i before merging is a tuple from stream B. Let us assume this tuple as b . After the queues between J_i and J_{i+1} are empty, b must also be processed by J_{i+1} and it is also the last tuple being processed by J_{i+1} before merging. At this time, $\forall a_i \in A :: [w_{i-1}, w_i]$, $w_{i-1} \leq T_b - T_{a_i} < w_i$ and $\forall a_i \in A :: [w_i, w_{i+1}]$, $w_i \leq T_b - T_{a_i} < w_{i+1}$. That is, the states of J_i and J_{i+1} are synchronized at this time in the sense that they are purged by the same tuple b . Thus after attaching the states of J_{i+1} to the corresponding states of J_i , no state tuple is lost in between of J_i and J_{i+1} .

When the execution is resumed, complete joined result will be produced.

(2). *No duplicate result.* According to the online merging steps shown in Figure 7.5, no state tuples will be duplicated in the steps and thus no duplication will be generated after the execution is resumed. ■

The overhead for chain migration corresponds to a constant system cost for operator insertion/deletion. The system suspending time during join splitting is neglectable, while during join merging it is bound by the execution time needed to empty the queues in-between of the sliced joins. Extra processing costs may also arise for attaching the states of corresponding sliced join operators.

7.4 Push Selections into Chain

When the N continuous queries each have selections on the input streams, we aim to push the selections down into the chain of sliced joins. For clarity of discussion, we focus on the selection push-down for predicates on one input stream. Predicates on multiple streams can be pushed down similarly. Here we will denote the selection predicate on the input stream A of query Q_i as σ_i and the condition of σ_i as $cond_i$.

7.4.1 Mem-Opt Chain with Selection Push-down

According to Theorem 4, the selections can be pushed down into the chain of sliced joins as shown in Figure 7.6. The predicate of the selection σ'_i corresponds to the disjunction of the selection predicates from σ_i to σ_N . That is:

unnecessary state tuples exist after push-down of selections. The reason is that the predicates have the most tight conditions.

■

Intuitively each join probing performed by \bowtie_i in Figure 7.6 is a joined result at least for one of the queries: Q_i, Q_{i+1}, \dots, Q_N . The state tuples in the Mem-Opt state-slice sharing are all required to produce the complete set of joined results.

7.4.2 CPU-Opt Chain with Selection Push-down

The merging of adjacent sliced joins with selection push-down can be achieved following the scheme shown in Figure 7.7. Merging sliced joins having selection between them will cost extra state memory usage due to selection pull-up. The tuples, which have been filtered out by the selection before, will now stay unnecessarily long in the state memory. Also, the consequent join probing cost would thus increase accordingly. Repeated merging of the sliced joins will result in the selection pull-up sharing approach discussed in Chapter 4.

Similarly to the CPU optimization in Chapter 7.2, the Dijkstra's algorithm can be used to find the CPU-Opt sharing plan with minimized CPU costs in $O(N^2)$ time. Such CPU-Opt sharing plan may not be Mem-Opt.

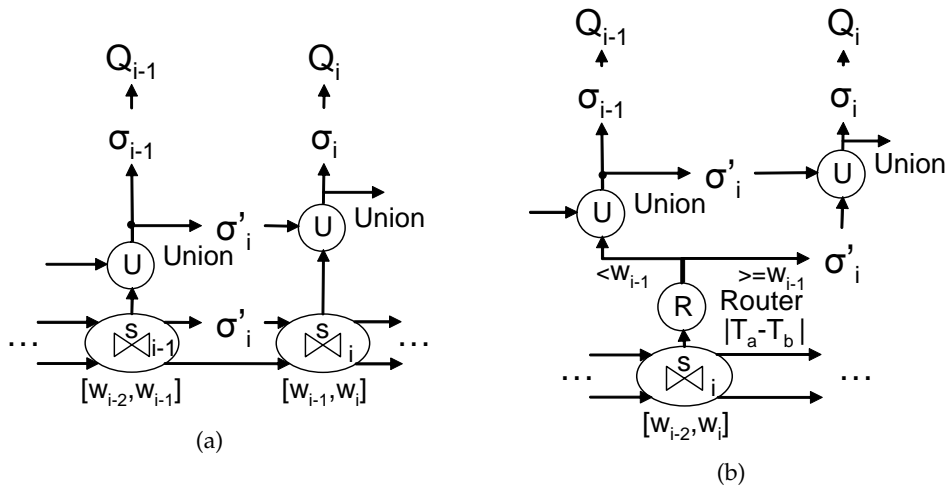


Figure 7.7: Merging Sliced Joins with Selections.

Chapter 8

Experimental Evaluation

We have implemented the proposed state-slice sharing paradigm in a DSMS system (*CAPE*) [RDS⁺04]. Experiments have been conducted to thoroughly test the ability of the sharing paradigm under various system resource settings. We compare the CPU and memory usages for the same set of continuous queries using different sharing approaches.

8.1 Experimental System Overview

The *CAPE* is implemented in Java. All experiments are conducted on a machine running windows XP with a 2.8GHz processor and 1GB main memory. The DSMS includes a synthetic data stream generator, a query processor and several result receivers. The query processor employs round-robin scheduling for executing the operators. The query processor has a monitoring thread that collects the runtime statistics of each operator. In all the experiments, the stream generator will run for 90 seconds. All the experi-

ments start with empty states for all operators.

We measure the runtime memory usage in terms of the number of tuples staying in the states of the joins. We measure the CPU cost of the query plans in terms of the average service rate ($\frac{Total\ Throughput}{Running\ Time}$).

The tuples in the data streams are generated according to the Poisson arrival pattern, which is usually used to model events that occur with a known average rate and independently of the time since the last event. The stream input rate is changed by setting the mean inter-arrival time between two tuples. To control the join selectivity on the chain of sliced window joins, we simulate the evaluation of the join predicates using a probabilistic model. The selectivity is changed in the experiments. The queries used in the experiments are similar to the example queries Q_1 and Q_2 in Chapter 4 with different window constraints.

8.2 State-Slice vs. Other Sharing Strategies

Equation 5.2 analytically compares the performance of state-slice sharing with other sharing alternatives. The experiments in this section aim to verify these benefits empirically.

We use three queries and the Mem-Opt chain buildup in these experiments. The queries are: $Q_1 (A[W_1] \bowtie B[W_1])$, $Q_2 (\sigma(A[W_2]) \bowtie B[W_2])$ and $Q_3 (\sigma(A[W_3]) \bowtie B[W_3])$. Apparently these three queries can share partial computations among each other. Using the Mem-Opt state-slice sharing, the shared query plan has a chain of three sliced joins with window constraints as $[0, W_1]$, $[W_1, W_2]$ and $[W_2, W_3]$. The joined results are

unioned and sent to each data receiver respectively. We compare the state-slice sharing with the naive sharing with selection pull-up and the stream partition with selection push-down (see Chapter 4). Using the naive sharing approach with selection pull-up, the shared plan will have one regular sliding window join: $A[W_3] \bowtie B[W_3]$. Using the sub-stream partition with selection push-down, the shared plan will have two regular joins: $A[W_1] \bowtie B[W_1]$ and $A[W_3] \bowtie B[W_3]$. The input stream A is partitioned by σ and sent to these two joins.

We vary the parameters as shown in Table 8.1. All the settings are moderate instead of extreme cases such as selectivities being close to 0 or 1. Experiments with all the combination of these settings are conducted. The input rates of the streams vary from 20 tuples/sec. to 80 tuples/sec in all the experiments.

Window Distribution(Sec.)	Mostly-Small: 5, 10, 30	Uniform: 10, 20, 30	Mostly-Large: 20, 25, 30
S_σ	Low(0.2)	Middle(0.5)	High(0.8)
S_\bowtie	Low(0.025)	Middle(0.1)	High(0.4)

Table 8.1: System Settings Used in Chapter 8.2.

The results showing memory consumption comparisons are depicted in Figure 8.1. Figures 8.1(a), 8.1(b) and 8.1(c) show that the memory usage is sensitive to the window distributions. Figures 8.1(d), 8.1(e) and 8.1(f) illustrate the effect of S_σ on the memory usage. Comparing Figures 8.1(b) and 8.1(e), we can see that S_\bowtie does not affect the memory usage since the number of joined tuples is unrelated to the state memory of the join. Over-

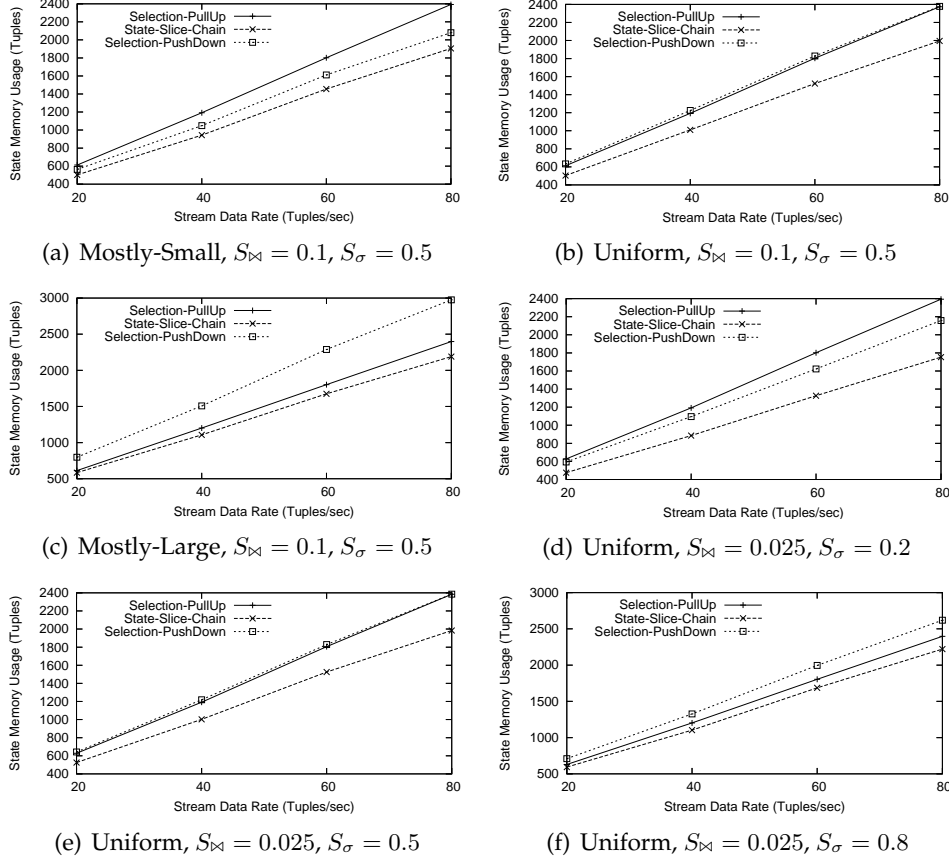


Figure 8.1: Memory Comparison with Various Parameters

all, the state-slice sharing always achieves the minimal memory consumption, with the memory savings ranging from 20% to 30%, depending on the overlap ratio of the corresponding windows.

Figure 8.2 shows the comparison of the service rate under various settings. Figures 8.2(a), 8.2(b) and 8.2(c) show the change of service rate under different window distributions. Figures 8.2(d), 8.2(e) and 8.2(f) illustrate the effect of S_{\times} on the service rate. Overall, the state-slice sharing always

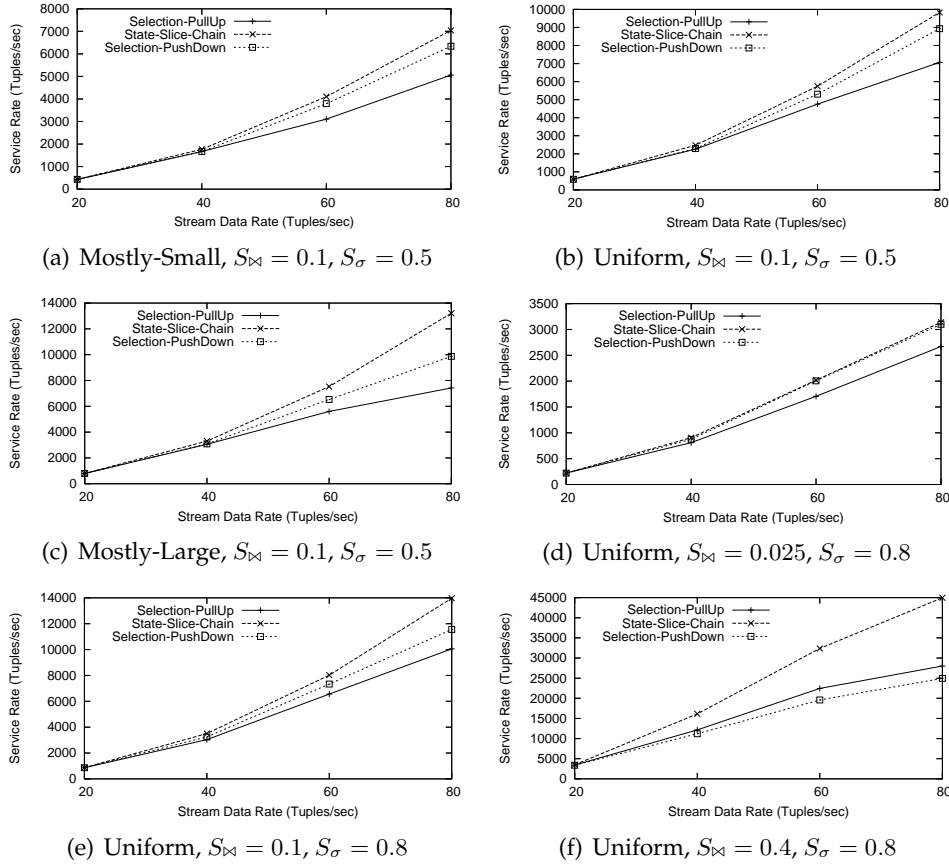


Figure 8.2: Service Rate Comparison with Various Parameters

achieves the maximum service rate.

From Figure 8.2 we can see that with increasing data input rate, more performance improvements can be expected from the state-slice sharing. One reason is that the number of joined tuples is proportional to $\lambda_A * \lambda_B$. Thus the routing cost increases quadratically. On the contrary, the extra purging cost in the state-slice sharing is proportional to $\lambda_A + \lambda_B$. Thus the purging cost only increases linearly. Then the state-slice sharing is more

scalable in the data input rates. Under the scenario of large join selectivities and high-volume input streams, the performance improvement of using state-slice sharing can reach 40%, as shown in Figure 8.2(f).

8.3 State-slice: Mem-Opt vs. CPU-Opt

In this second set of experiments, we focus on the performance comparison between the Mem-Opt and the CPU-Opt chains under different system settings. We use similar queries as in Chapter 8.2 with the selections removed. We also use the service rate to measure the CPU consumptions. The CPU-Opt chain is built from the Mem-Opt chain by merging some of the slice joins according to the algorithm discussed in Chapter 7.2. To control the selectivities, we use a probabilistic probing algorithm that will match the predicates according to the settings of the selectivities. The experiments are conducted using different numbers of queries (12, 24, 36) and various window distributions. The window distributions for the 12 queries are shown in Table 8.2. The window distributions for other number of queries are set accordingly. We set the join selectivity to be 0.025. The input rates of the streams vary from 20 tuples/sec to 80 tuples/sec in all experiments. The service rate comparisons are shown in Figure 8.3.

Uniform(Sec.)	2.5, 5, 7.5, 10, 12.5, 15, 17.5, 20, 22.5, 25, 27.5, 30
Mostly-Small(Sec.)	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 30
Small-Large(Sec.)	1, 2, 3, 4, 5, 6, 25, 26, 27, 28, 29, 30

Table 8.2: Window Distributions Used for 12 Queries.

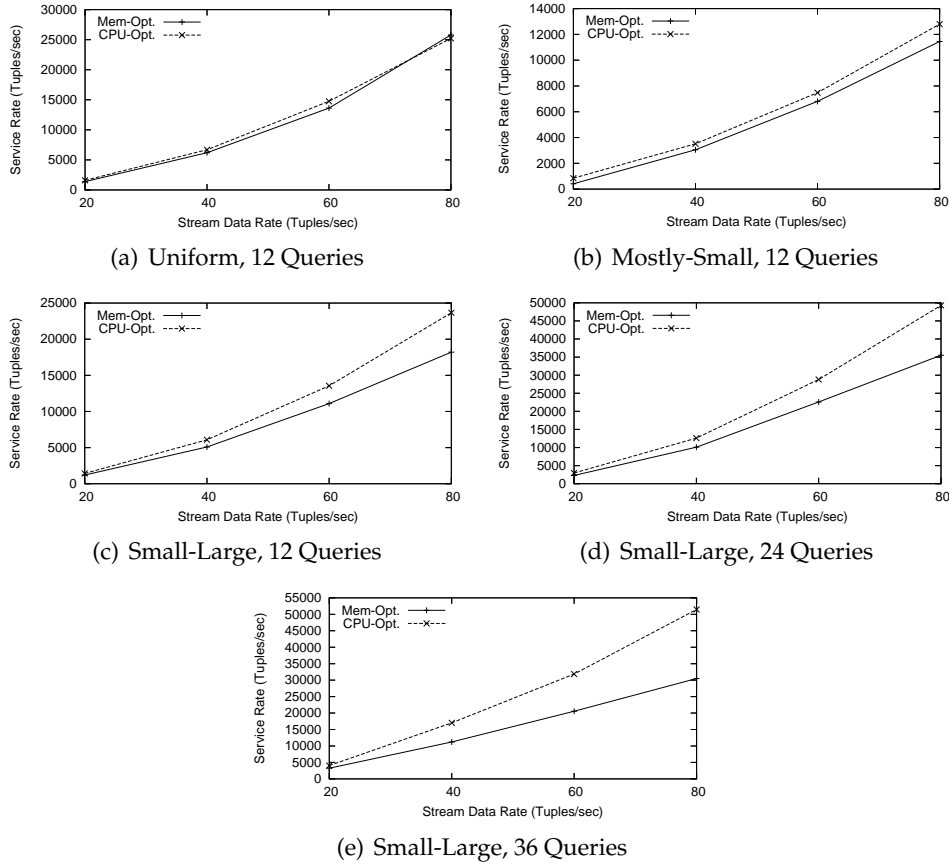


Figure 8.3: Service Rate Comparison of Mem-Opt. Chain vs. CPU-Opt. Chain

In Figure 8.3(a), the CPU-Opt chain is actually the same as the Mem-Opt chain. However, for skewed window distributions, the CPU-Opt chain has fewer operators than the Mem-Opt chain. In Figure 8.3(b), all the small windows are merged together in the CPU-Opt chain. In Figure 8.3(c), the CPU-Opt chain actually will have only 2 sliced joins, after merging all the small windows and all the large windows by the optimization algorithm. The benefit of CPU-Opt over Mem-Opt chain increases as the number of

queries increases, as shown in Figures 8.3(d) and Figure 8.3(e).

Chapter 9

Related Work

There has been considerable work recently on data stream processing. [GÖ03b] is a survey of stream and continuous query processing. We discuss only the body of work related to sharing of multiple queries in stream processing.

The problem of sharing the work between multiple queries is not new. For traditional relational databases, multiple-query optimization [Sel88] seeks to exhaustively find an optimal shared query plan. Recent work, such as [RSSB00, MRSR01], provides heuristics for reducing the search space for the optimally shared query plan for a set of SQL queries. These works differ from our work since we focus on the computation sharing for window-based continuous queries. The traditional SQL queries do not have window semantics.

Sharing the computation of multiple continuous queries has been considered recently. Many papers [CCC⁺02, MSHR02, CDN02, HFAE03, KFHJ04] in the literature have highlighted the importance of computation sharing in continuous queries. The sharing solutions employed in existing systems,

such as NiagaraCQ [CDN02], CACQ [MSHR02] and PSoup [CF02], focus on exploiting common subexpressions in queries. Their shared processing of joins simply ignores window constraints which are critical for most continuous queries, given that memory becomes unbounded when no window constraints are employed.

Recent papers in [AW04, ZKOS05, KWF06] have focused on sharing computations for stateful aggregations. The work in [AW04], addressing operator-level sharing of multiple aggregations, has considered the effect of different windows constraints on a single stream. The basic idea is to explore the overlapping relations of the states for aggregations in multiple stream queries. By split the aggregation states into several small partitions, the calculation of the aggregations over partitions then can be shared among multiple queries. The original aggregations can be achieved by combining aggregation values over small window partitions.

The work in [ZKOS05] discusses shared computations among aggregations with fine-grained *phantoms*, which is the smallest unit for sharing the aggregations. The work in [KWF06] discusses runtime aggregation sharing with different periodic windows and arbitrary predicates. However, efficient sharing of window-based join operators has thus far been ignored in the literature.

In [HFAE03] the authors propose various strategies for intra-operator scheduling for shared sliding window joins with different window sizes. Using a cost analysis, the strategies are compared in terms of average response time and query throughput. Our focus instead is on how we can minimize the memory and CPU cost for shared sliding window joins. The

intra-operator scheduling strategies proposed in [HF_{AE}03] can naturally be applied for inter-operator scheduling of our sliced joins.

Load-shedding [TZ⁺03] and spilling data to disk [UF00, LZR06] are alternate solutions for tackling continuous query processing with insufficient memory resources. Approximated query processing [SW04] is another general direction for handling memory overflow. Different from these, we minimize the actual resources required by multiple queries for accurate processing. These works are orthogonal to our work and can be applied together with our state-slice sharing.

Ideas from some previously proposed techniques are implemented in our sharing paradigm. The lineage of the tuples proposed in [MSHR02] can be used to avoid repeated evaluation of the same selections on a tuple in a chain of sliced joins. The precision sharing in the TULIP [KFHJ04] can be used in our paradigm for selections on multiple input streams. The grouping of similar queries with same window constraints in [CDN02] can be used for discovering shared join expressions among multiple continuous queries. These ideas are complementary to our state-slice concept, and can be applied to our sharing paradigm.

Part II

Distributed Multi-way Stream

Join Processing

Chapter 10

Introduction

10.1 Research Motivation

Modern stream applications are usually time critical in scientific and engineering domains [AAB⁺05b, RRWM07, JAA⁺06, KDY⁺06], which is a challenge goal when the processing of stream joins among multiple high-speed streams are involved. The multi-way joins in such applications usually have complex join conditions on high volume input stream data [AAB⁺05b, RRWM07]. Given the memory- and CPU-intensive nature of stream queries, distributed query processing on cluster must be employed for tackling this challenge [GYW07].

Distributed continuous query processing has been considered in recent years, such as distributed Eddies [TD03], Borealis [Ac04, ABcea05], System S [JAA⁺06] and D-CAPE [LZJ⁺05]. Two distribution techniques are usually supported: operator distribution and data distribution. Using operator distribution, disjoint sub-plans of the query plan are executed on

different machines with the intermediate results being routed between the machines. Data distribution instead installs instances of the same operator into multiple machines, each then processing a different partitions of the input data on its respective machine. Both methods are orthogonal and can in fact be combined.

Hash-based data partitioning has been proved effective for distributing equi-joins [Kun00], both for relational and stream queries. Beyond equi-join stream queries, which can be distributed with hash-based partitioning, generic joins with arbitrarily join conditions are used widely in non-trivial stream applications such as image matching and biometric recognizing. Hash-based partitioning invokes potentially huge duplications when distributing generic joins. A more efficient scheme for the distributed execution of generic multi-way joins with window constraints is thus critical.

Moreover, for operator distribution, the macro Multi-way window-based Join operations (MJ) operator must fit into one single machine — which is not always feasible when large window constraints and high volume input streams are encountered. Though we could translate an MJ operator into a join tree composed of a sequence of smaller binary join operators, such method would lose the flexibility of join orderings shown to be extremely useful for MJ processing in dynamic environments [VNB03]. Also, such join tree distribution will scale to at most $k - 1$ machines for a k -way MJ operator, while the number of machines available may be much larger than k .

In this part, we focus on distributed processing of generic MJs with arbitrary join predicates, especially for MJs with large window constraints.

Generic stream joins occur in many practical situations, from simple range (or band) join queries to complicated scientific queries with equation-based predicates. Such join operators tend to be complex and CPU intensive, as further motivated below. Our goal is to minimize the memory consumptions and the query response time to meet the time requirement of the stream applications.

Motivation Example:

In a fire spread monitoring system [RRWM07], sensor(s) deployed at diverse physical locations provide real-time environmental measurements of the space, including temperature, humidity, images, or video. The system will recognize the fire pattern and predict fire spreading in order to for instance provide safe escape routes. The information collected from sensor(s) per location correspond to a stream source with tuples having multiple columns, each column for a measurement. To predicate the trend of fire spread, a window-based MJ operator is used to employ phenomenon matching functions among data streams from multiple locations, to determine if it is an isolated incident or a wide-spread fire affected area. The join predicates are based on mathematical models from the fire protection domain, and possibly matching against comparative simulation snippets of classified fire patterns. The join predicates are far more complex than the simple equi-join predicates, and can be expensive to evaluate. Such computation may include Discrete Fourier Transform, pattern recognition and etc. The sliding windows can be large, such as from minutes to tens of minutes, since for some fire patterns the spreading can be very slow at beginning but change rapidly. Without knowledge of which fire patterns

may arise *a priori*, the largest window size must be set to cover all possible patterns with various fire pattern characteristics. Also, clearly these time-consuming join evaluations must be finished in real-time to alert personnel around the fire.

10.2 Proposed Strategies

A novel MJ operator distribution scheme called Pipelined State Partitioning (PSP) is proposed in this part of the dissertation. The PSP scheme is a new form of pipelined parallelism. Our solution is based on the state-slicing [WRGB06] concept introduced in Chapter 5 for query sharing. We propose a novel solution to separate a macro MJ operator into a series of smaller state-sliced MJ operators. The sliced MJ operators are connected in a virtual ring architecture. Different from value-based partitioning, the PSP scheme is join predicate agnostic and thus general. It slices the states into disjoint slices in the time domain, and then distributes these fine-grained state slices among processing nodes in the cluster. Different from traditional plan-based pipelined parallelism, whose length of pipeline is bounded by the longest sequence of operators in the query plan, PSP instead can split the MJ to any number of state-sliced MJ operators at the optimizer's will to achieve maximum parallelism.

We design two critical extensions of the basic PSP scheme. PSP-I (with I for Interleaving) and PSP-D (with D for Dynamic).

PSP-I introduces a delayed purging technique for the states to enable interleaved processing of multiple stream tuples. Without interleaving, only

one input stream tuple and the intermediate results generated in the probing steps triggered by this tuple can be processed in the ring of nodes. That is, all input stream tuples must be processed in a sequence manner in order to avoid any state tuple being purged too early. This will happen when a stream tuple with larger timestamp purges the state tuple but another stream tuple with smaller timestamp still should probe this state tuple. The reason for this mess-up is the possible out-of-order process of the multiple stream tuples and corresponding intermediate results at each node. This limitation is removed by separating the purging steps into two sub-tasks: propagation of purged tuple and deletion from the state. The state deletion is postponed until the state tuple is out of the sliced windows of all the currently being processed tuples. Such interleaved processing is used to avoid idle processors which exist in the synchronized basic PSP scheme.

Beyond interleaved processing, PSP-D (with D for Dynamic) further incorporates a dynamic state ring structure to avoid repeated maintenance costs of sliced states along the ring of nodes. In the basic PSP scheme, any state tuples must be purged multiple times and propagated step-by-step along the ring. To avoid this extra cost caused by a tuple's repeated insertion into and purging from of a state-slice in the basic PSP scheme, we extend the PSP scheme by introducing dynamic head and tail abstraction. Instead of moving all the state tuples through all the nodes, we move the start and end location of the windows along the ring. Thus this portion of costs for state maintenance, which can be significant for fast incoming streams, can be saved.

The key principles of the basic PSP scheme and its varieties are listed as

follows.

- *Ring-based query plan.* Instead of a chain architecture used in Chapter 7 for state sliced binary join processing, a ring-based query plan with loop back of the intermediate results is proposed for handling multi-way join operators. Recall that a binary join is treated as the combination of two one-way joins in Chapter 6, which implicitly enforces the binding of state slicing approach with the join orderings. The ring-based query plan instead enables the state slicing approach to work with any optimizer choice of join orderings. Thus ring-based query plan makes the state slicing and join ordering orthogonal, which largely simplifies the optimization process.
- *Synchronized processing on slices without locking.* The pipelined processing of state sliced joins requires synchronized state maintenance to avoid incomplete or duplicated join results. In a homogeneous but asynchronous cluster, synchronized processing usually requires special support of locking mechanism. Instead in our proposed PSP schemes, the specially designed execution strategies stipulate the synchronized processing on slices without using locks. Thus no extra locking support is required from the cluster and makes our PSP schemes applicable to any homogeneous computation environments.
- *Cost-based state allocation, and distributed time-slice adaptation.* Since the fluctuate nature of the streaming data, runtime adaptive state allocation and relocation of the sliced states is essential for fine-grained load

balancing. Our proposed PSP schemes support online state adaptation with low extra cost.

We develop a cost model for the basic PSP scheme and use it to tune the parameters for different performance objectives. Our cost model provides the necessary analytical equations to model the relationships between the following key parameters of the PSP model: (1) stream data characteristics, including stream arrival rates; (2) query parameters, including window sizes; (3) join selectivities, which is related to both query and stream data; (4) PSP ring parameters, such as the number of nodes in the ring; (5) performance measurements, such as the system throughput and average response latency of joined results. A cost-based optimizer is also developed to achieve the optimal state slicing and allocation.

Runtime adaptive state relocation are also employed for achieving load balancing and re-optimization in a fluctuating environment. Adaptive workload diffusion is critical for realistic long running query processing, when stream arrival rates, join selectivities and load of processing nodes change at runtime. In the PSP schemes, adaptive workload diffusion is achieved by state relocation among the nodes by setting the corresponding window ranges. We tackle two major load re-balancing scenarios: *workload smoothing* among same amount of nodes and *state relocation* with more/less nodes.

Compared to existing work on distributed generic MJ processing in [GYW07], the PSP scheme has the following benefits: 1) there is no state duplication and thus no repeated computations during PSP distribution; 2) the PSP scheme is applicable for large window constraints; 3) the PSP scheme can

slice the MJ operator into a ring with optimal number of sliced joins, which is orthogonal to other optimizations such as join ordering optimization; and 4) controllable adaptive state partitioning and allocation in the time domain.

The proposed PSP schemes have been implemented within the *D-CAPE* [SLJR05], which is the distributed version of the *CAPE* DSMS. We use multi-way stream joins comparing the similarity of the synthetic image streams in the experiments. The experiments have been conducted to thoroughly test the ability of the proposed solution under various system resource settings. The experimental results show that our strategy provides significant performance improvements over the state replication based solutions in [GYW07] under a diverse workload settings.

10.3 Our Contributions:

- We introduce the novel ring architecture of sliced window join operators, and prove its equivalence to the regular window-based join.
- We extend the based PSP model with two key features: interleaved tuple processing and dynamic ring structure to improve the system performance.
- The memory and CPU costs of PSP-D ring are analytically evaluated based on a cost model.
- Based on insights gained from this analysis, a cost-based optimizer is proposed that achieves optimal state slicing in terms of maximum

output rate or minimal query response latency, respectively.

- The runtime state migration in terms of slice allocation and relocation is discussed. Based on the cost model of the PSP scheme, algorithms for state migration are developed for workload smoothing among same amount of nodes and state relocation with more/less nodes.
- The proposed techniques are implemented in the *D-CAPE* DSMS. Results of performance comparison of our proposed techniques with state-of-the-art state replication based strategies in [GYW07] are reported, confirming the superiority of our PSP schemes.

10.4 Road Map

The rest of this part is organized as follows. Chapter 11 presents the preliminaries used in this part, briefly reviewing the state-slice concept in Chapter 5. Chapter 12 defines the problem tackled and introduces the PSP distribution scheme. Chapter 13 present the cost based analysis. Chapter 14 discusses the cost based runtime adaptive optimization. Chapter 15 compares the PSP with other generic join distribution schemes. Chapter 16 reports the experimental results while Chapter 17 contains related work.

Chapter 11

Background

11.1 Semantics of Multi-way Window Join

In this part, we consider a multi-way join operator (MJ) on input streams with unbounded sequences of tuples. Each stream input tuple has an associated timestamp identifying its arrival time at the system. Similar to [BMW05], we assume that the timestamps of stream tuples are globally ordered. *Sliding windows* [BBMW02] define the scope of the otherwise infinite streams for stateful operators.

A multi-way join operator on data streams S_1, S_2, \dots, S_n with window constraints W_1, W_2, \dots, W_n respectively is denoted as $J^n : S_1[W_1] \bowtie S_2[W_2] \bowtie \dots \bowtie S_n[W_n]$ with join conditions $\theta(S_1, S_2, \dots, S_n)$. In this part of dissertation, the input stream tuples are assumed to be processed in the order of their timestamps. We extend the semantics of the window constraints defined previously in Chapter 3.2 for multi-way joins. That is, the output of the MJ consists of all joined tuples (s_1, s_2, \dots, s_n) , such that $T - T_{s_i} < W_i$

$(\forall i \in [1, n])$ and $\theta(s_1, s_2, \dots, s_n)$ hold. Here T_{s_i} denotes the timestamp of tuple s_i and T denotes $\max(T_{s_i}), i \in [1, n]$. The timestamp assigned to the joined tuple is T .

In [VNB03, BMM⁺04], the efficient execution algorithms for multi-way stream joins, in particular the non-blocking multi-way symmetric join algorithms with flexible join orderings, are introduced. Compared to the traditional evaluation of multi-way joins using fixed binary join trees [Kun00], such adaptive execution results in less blocking and a distinct optimized join ordering for each input stream. Also different from Eddies [AH00], join orderings are selected per stream instead of per tuple, in order to avoid per tuple routing cost. Our proposed approach inherits this flexibility of customized join orderings per stream to assure high performance. Clearly, selection of efficient join orderings is orthogonal to our focus, and any algorithms in [VNB03, BMM⁺04] could be used for this purpose. We briefly review the two execution methods of multi-way joins below.

There are two common methods for executing multi-way continuous joins, namely binary join trees [VN02] as shown in Figure 11.1 and multi-way join operators [GO03a, VNB03, BMM⁺04, HAE03] as shown in Figure 11.2.

A binary join tree, as shown in Figure 11.1 in two of many possible different shapes, is a query plan composed of binary join operators. It is a direct extension of the typical query plans used in static query processing [SAC⁺79, IK84, KBZ86]. Figure 11.1 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 one on the right is a *bushy tree*, in which both inputs of a join operator can be intermediate results produced by some join operators below it. Each binary join operator applies a symmetric join algorithm [WA93, HH99], such as symmetric hash join or symmetric nested-loop join. To implement the window constraints, each binary join operator keeps two states that stores tuples that the operator has received so far and in the current window. Some states, such as state S_A in Figure 11.1, keep the stream input tuples. Other states, such as S_{AB} and S_{ABC} , keep intermediate join results.

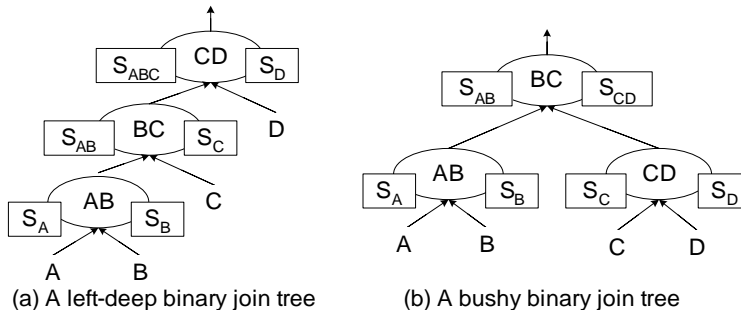


Figure 11.1: Binary Join Trees

Different from a binary join tree, a single multi-way join operator that takes in all joining stream inputs and outputs the joined results can be used. Figure 11.2(a) shows the basic data structure of a multi-way join operator in a continuous query that implements a five-way join $A \bowtie B \bowtie C \bowtie D \bowtie E$. The operator takes in five input streams and outputs joined tuple of the form $ABCDE$. Five states are kept in the operator, each associated with one of the input streams. Suppose now the multi-way join operator takes one tuple a from input stream 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, which is selected to minimize the size of intermediate results and thus the join cost. The processing of new tuples from other input streams follows the same procedure, except that they may join with remaining states in a different order. Figure 11.2(b) shows possible join orders for tuples from input stream A and input stream B .

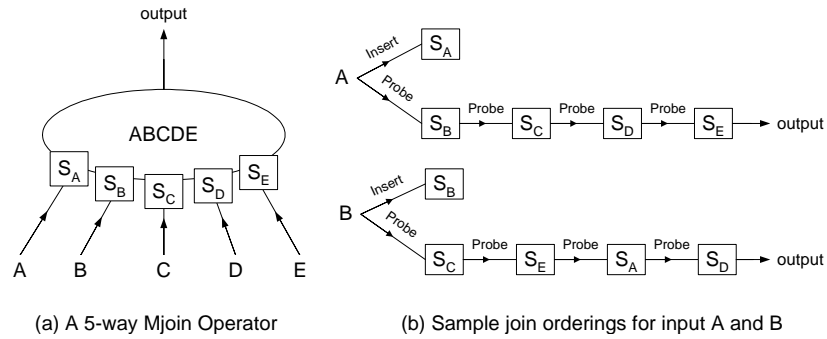


Figure 11.2: Multi-way Join Operator for Query $A \bowtie B \bowtie C \bowtie D \bowtie E$

As we can see a binary join tree keeps all intermediate results in operator states, thus saves CPU cost on recomputing these intermediate results but requires high memory costs. On the contrary, a multi-way join operator does not keep any intermediate results, thus saves memory but requires extra CPU for re-computation. Because of not maintaining any history of partially computed join results, multi-way join operator is more flexible in terms of enabling different join orderings for each input stream and also in terms of being able to quickly switch between different join orderings even for the same input stream.

11.2 Distributed Continuous Query Processing in DCAPE

The basic distribution techniques can be classified as *pipelined parallelism* and *partitioned parallelism* [Kun00]. By streaming the output of one operator into the next operator through network connections, the two operators which located on different processing nodes can work in series, termed pipelined parallelism¹. By partitioning the input data among multiple processors, an operator can be instantiated as many independent copies of the operator (called operator instances), each working on a subset of the data, termed partitioned parallelism². Figure 11.3 illustrates the pipelined parallelism and partitioned parallelism in two processing nodes using an example of three-way join query. The DCAPE system support both of these modes of parallelism.

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.

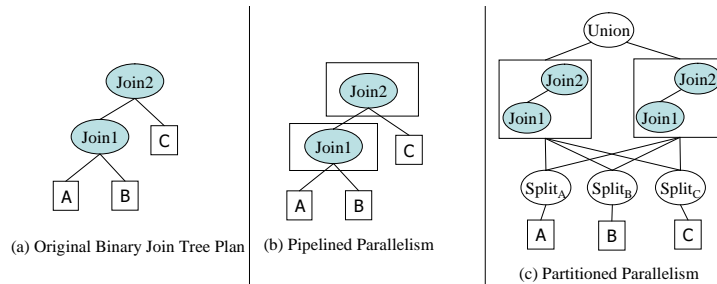


Figure 11.3: Pipelined parallelism and Partitioned Parallelism

¹Blocking operator, such as sorting, may not be allowed for pipelined processing.

²This may not work for all operators (e.g. calculation of the standard derivation) or special treatment of combining the result is needed (e.g. getting the max value).

For example, the continuous query plan with two joins in Figure 11.4(a) can be assigned to two machines as in Figure 11.4(b). Each machine runs instances of both join operators. To partition the data, we add three *split operators* (one for each input stream) and one *union operator* (for collecting together all outputs) 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.

The number of split operators is the same as the number of inputs to the query plan. The *union operator* combines the outputs from all involved (in this example, two) machines to produce the final outputs. This can be viewed as making a copy of the query plan (with added split and union operators) and putting one copy of the plan on each machine. The darker operator indicates that the operator is active on that machine. As we can see, the two joins are executed on both machines. One copy of the split is activate in the system since the input streams may be connected to different nodes, as shown in Figure 11.4. Also one union is needed to be active, being the single collecting operator.

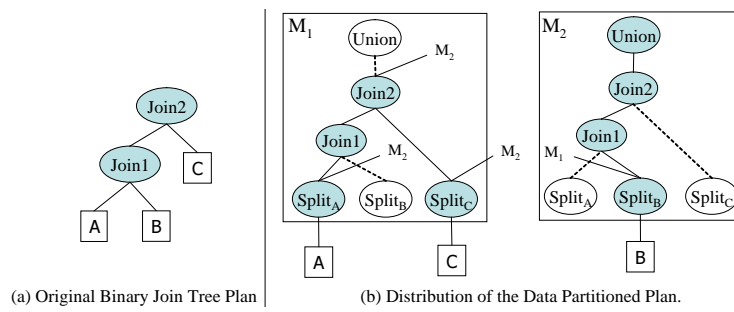


Figure 11.4: Example of Data Partitioned Plan Distribution

Chapter 12

PSP: State-Slicing for Multi-way Joins

12.1 New Challenges in State-slicing for Multi-way Joins

Applying the state-slicing concept to MJ operators faces new challenges beyond the binary state-slicing discussed in Chapter 5 and also published in [WRGB06]. In this chapter, we present our proposed solution to apply state slicing concept to multi-way join operators.

If we were to first convert an MJ operator into a binary join tree, then thereafter we would reuse the binary state-slice method in a naive way to process MJs. However a binary join tree implies a fixed join ordering for all input stream tuples, which may be sub-optimal compared to having individual join orderings for each input streams in a holistic MJ operator [BMM⁺04]. More over since it is a fixed tree, most certainly it is rather

rigid for runtime join re-ordering when the stream characteristics change. Extra effort is needed for migration of a binary join tree at runtime [ZRH04].

Alternatively, directly applying the binary state-slicing method to MJ operators faces several problems, as explained below with an example. Assume the MJ to be processed is a four-way join $A \bowtie B \bowtie C \bowtie D$ and n state-sliced join operators, J_1 to J_n , are connected in a chain structure. The state for each stream in the MJ is partitioned into n parts, denoted as (A_1, \dots, A_n) , (B_1, \dots, B_n) and so on. Thus sliced join J_i ($1 \leq i \leq n$) will hold the sliced states A_i , B_i , C_i and D_i . For one incoming tuple a from stream A , all the sub-join tasks $a \bowtie B_i \bowtie C_j \bowtie D_k$, ($1 \leq i, j, k \leq n$) must be conducted to generate the complete joined results. Without loss of generalization, consider one sub-join task with $k \leq j \leq i$, then first $a \bowtie D_k$ must be conducted at J_k , since the sliced state D_k is held only at J_k , which is ahead of J_i and J_j in the chain. Similarly, $(a \bowtie D_k) \bowtie C_j$ then is conducted at J_j and so on. That is, the join ordering $(A \rightarrow D \rightarrow C \rightarrow B)$ is imposed automatically by the monotonous increasing order of the i, j, k in each sub-join task, but not by the plan optimizer. All the sub-join tasks are required to produce the complete joined results in the state slicing approach. That is, sub-join tasks with all the possible orders of i, j, k , which imposes all join orderings, are processed in every nodes. From the optimizer's point of view, most of these join orderings certainly may not be optimal and there is no freedom for the optimizer to pick a better join ordering other than the imposed one. Thus not even a single best join ordering can be picked as in a binary join tree. This strategy is the worst one in all the design choices in terms of options of optimal ordering for max filtering of intermediate joined results.

The benefit of holistic MJ processing is totally lost and the join performance may be significantly decreased.

More concretely, assume two state sliced joins are employed and the states of A, B, C are each partitioned into two parts as A_1, A_2, B_1, B_2 and C_1, C_2 . Then $a \bowtie B_1 \bowtie C_1$ and $a \bowtie B_2 \bowtie C_2$ are performed at J_1 and J_2 respectively. However to ensure the correctness, $a \bowtie C_1$ must be also performed in J_1 and then join with B_2 in J_2 . Moreover, the intermediate result of $a \bowtie B_1$ needs to be send to J_2 to join with C_2 . That is,

$$\begin{aligned} a \bowtie B \bowtie C = & (a \bowtie B_1) \bowtie C_1 + (a \bowtie B_2) \bowtie C_2 \\ & + (a \bowtie B_1) \bowtie C_2 + (a \bowtie C_1) \bowtie B_2 \end{aligned}$$

We can see that in J_1 , the a tuple needs to probe both B and C states. Thus in fact all possible orderings ($A \rightarrow B \rightarrow C$ and $A \rightarrow C \rightarrow B$ in this example) are used and no freedom exists to pick different join orderings. This may decrease the performance when the selectivity of $A \bowtie C$ is much larger than that of $A \bowtie B$.

As a consequence of having to use all possible sub-optimal join orderings, large system cost may exist for processing all the intermediate results. There are exponential kinds of intermediate results for an n -way join, since every subset of the n input streams can be mapped to a kind of intermediate result.

In a summary, both of the approaches of extending binary state slicing concept to MJ operators may interfere the optimizer's choice of the optimal join orderings, no matter if a binary join tree or a brute force extension is

used. Thus a new state slicing approach is required to avoid the interference of join ordering optimization.

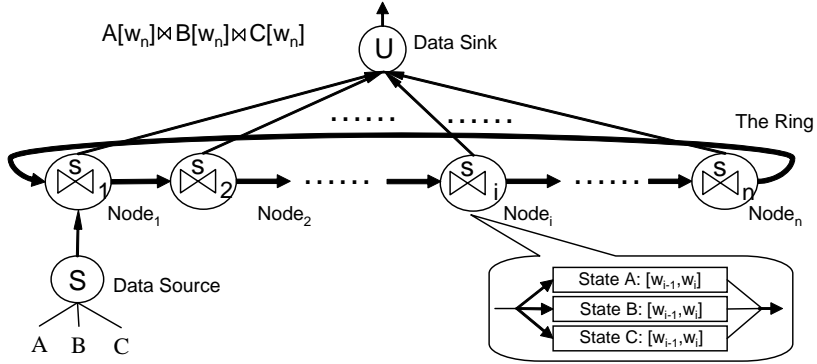


Figure 12.1: Ring-based Query Plan with Multi-way State-slice Joins.

To inherit the merit of holistic MJ processing with optimal join orderings, in this part we propose a *ring-based query plan* execution framework for multi-way state sliced join processing. Figure 12.1 shows the logical ring-based query plan with an example 3-way state sliced joins. The ring-based query plan redirects the output of the last sliced join (J_n) back to the input of the first sliced join (J_1). In this way, the selection of the join orderings is now made independent from the locations of the corresponding sliced states for a sub-join task, since the ring structure can bring tuples to any corresponding sliced states to be joined with next, according to the selected join orderings. The ring-based query plan and its control logic will be discussed in detail in this chapter.

In this part of dissertation, we allocate one state sliced join operator to each processing node. Thus we will use the term *Node i* interchangeably with the term *state-sliced join J_i*.

Naturally the PSP scheme provides a novel scheme for distributed processing of expensive join operators. The cost-based deployment of the PSP scheme in a cluster is discussed in Chapter 13.

PSP is designed to distribute the potentially huge state of the MJ operator to all the processing nodes and consequently render balanced CPU load diffusion. The adaptive workload balance is achieved by dynamic setting the window ranges of the sliced joins at runtime. The runtime ring plan adaptation will be discussed in Chapter 14.

[GYW07] proposed two *state replication* based distributions for generic MJ operators. Detailed comparisons between our *state partitioning* based PSP and state replication based approaches will be discussed in Chapter 15.

12.2 State-Slice Ring with Life Control

The logic ring model of PSP in Figure 12.1 corresponds to a series of multi-way state-sliced join operators connected in a ring structure. Besides the raw stream inputs and the final output of complete join results, each state-sliced join also has special input and output for pipelined propagation of intermediate join results.

Similar to binary state sliced join operator, we first define multi-way state-slice join as follows.

Definition 4 (Multi-way State-Slice Join) *A multi-way state-slice join operator J on data streams S_1, S_2, \dots, S_n is denoted as $S_1[W_1^s, W_1^e] \overset{m}{\bowtie} S_2[W_2^s, W_2^e] \overset{m}{\bowtie} \dots \overset{m}{\bowtie} S_n[W_n^s, W_n^e]$, where the superscripts s and e denote the start and end of the window constraints. The state for input stream S_i in J , denoted as $S_i[W_i^s, W_i^e]$,*

holds tuples within a window of $[W_i^s, W_i^e]$ with respect to the current timestamp. The joined results of J for arrival tuple s_i consist of all tuples in the form of $(s_1, s_2, \dots, s_i, \dots, s_n)$, such that $W_j^s \leq T_{s_i} - T_{s_j} < W_j^e$, where $j \in [1, n]$, $j \neq i$ and s_j is a state tuple from stream S_j .

A pipelined state-slice join ring is composed of multiple state-slice join operators on the same data streams. The states of the connected joins have abutting window ranges for each input stream (except for head and tail joins as explained next), that can be concatenated together conceptually into the full stream window. The slice join containing the $W^s = 0$ in the ring is called the *head* of the ring and the one containing the largest end window the *tail* of the ring. Similar to its binary counterpart, each state of the multi-way state-slice join is defined by a window range with upper and lower timestamp bounds. Since the window ranges of sliced joins in the ring are non-overlapping, all states of the join operators are partitioned disjointly among all the state sliced join operators.

Figure 12.2 shows a snapshot of the state partitioning and physical deployment for an example 3-way join on streams A , B and C in a 5-node cluster. Each stream has unique window size and the current sliding windows are illustrated with colorful/gray slots. The state of each stream is partitioned into five disjoint pieces which are deployed to Nodes 1 to 5 respectively. In order to achieve balanced state sizes and consequently balanced memory consumptions at runtime, the sliced state deployment can be flexible in terms of using different window sizes and different ring connections for each stream. For example, the Node 2 can hold sliced states

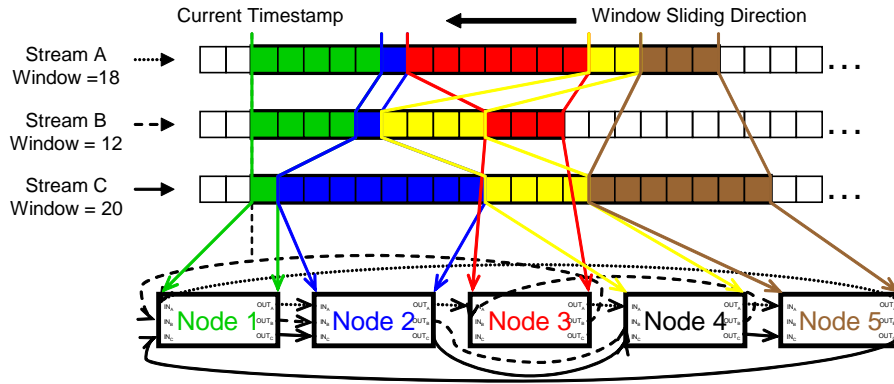


Figure 12.2: Snapshot of Runtime State Deployment in the Ring-based Query Plan. The Current Sliding Window is Composed of the Colorful/Gray Slots for Each Stream.

from streams *A* and *B* with 1 unit window size, while from stream *C* with 8 units window size. Also the ring connection of the sliced states for stream *A* is a loop as $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 1\dots$, while for stream *B* is another loop as $1 \rightarrow 2 \rightarrow 4 \rightarrow 3 \rightarrow 1\dots$. We can see that the ring connection for each stream state may not be identical and the ring length can also be different for each stream.

The PSP execution model includes three closely coupled components that stipulate that it produces complete yet no redundant join results, according to the semantics of the multi-way sliding window join. They are: (1) coordinated state maintenance among the sliced join operators in the ring to ensure state consistency; (2) propagation and processing of the intermediate results for generation of correct and complete join results; (3) execution control to avoid infinite looping of tuples in the ring.

The join processing among multiple nodes need coordination to ensure data consistency. However synchronization in a large cluster is expensive,

especially when the synchronization protocol needs to be invoked for every single input stream tuple in our case. In our PSP model, we instead design an implicit synchronization scheme based on the FIFO network transmission model. The execution process of each node then is synchronized by the tuples in the input network connections of this node. This achieves that each node can run synchronously with not extra cost since no synchronization mechanism from the cluster is needed.

We first discuss the coordination among multiple nodes for processing a single input stream tuple in Chapter 12.2.1. Then we will show how the head node “knows” the end of processing cycle for current input tuple in Chapter 12.2.3. Based on these two techniques, the execution algorithm of sequential processing of the input tuples in the ring is discussed in Chapter 12.3. The PSP with interleaved processing and dynamic ring structure are discussed later in Chapter 12.4 and Chapter 12.5 respectively.

12.2.1 Coordinated State Maintenance.

Recall that in Chapter 5 two representatives for each input tuple, called *build tuple* and *probe tuple*, are used, each with distinct assigned responsibilities. Build tuples will be inserted into the states of the join operators and stay there until being purged. The probe tuples instead will be propagated throughout the ring structure for probing corresponding states of other streams to identically matching the join predicates and then perform the actual result construction. Note that all the states are composed of the build tuples from corresponding streams. In the ring structure, the build tuples will “move” from the head node towards the tail node steadily, as

we will further explain this in Chapter 12.3, until finally it can be surly purged after it has been probed at the tail of the join pipeline. The following example routine shows how each node processes state insertion and purging.

Example: Suppose at time t , a tuple a_t arrives from stream A at the entry state-sliced join operator J_1 in the ring. Then two copies a_t^b (build) and a_t^p (probe) are made. Tuple a_t^b is inserted into the current state $(a_t^b, a_j^b, \dots, a_{i+1}^b, a_i^b, a_{i-1}^b)$, ordered in decreasing order of their timestamps. Suppose a_i^b and a_{i-1}^b are the only tuples which are now fall outside of the current window range due to the arrival of a_t^p at the sliced state. Then the state will be $(a_t^b, a_j^b, \dots, a_{i+1}^b)$ after purging triggered by a_t^p and the output queue then is augmented by pushing $(a_t^p, a_i^b, a_{i-1}^b)$ into the queue. Later when a_{i-1}^b and a_i^b are processed by the next join operator J_2 , they will be inserted into the state of J_2 . Thereafter when a_t^p is processed by J_2 , it will be used to purge and probe the corresponding state in J_2 .

From the above example, we have following lemma.

Lemma 7 For any node holding J_i with a state sliced window $S :: [W_i^s, W_i^e]$ on certain input stream S , at the time that a probe tuple s^p with timestamp T_{s^p} finishes the purge step, but has not yet began the probe step, we have: (1) $\forall s'^b \in S :: [W_i^s, W_i^e] \Rightarrow W_i^s \leq T_{s^p} - T_{s'^b} < W_i^e$; and (2) $\forall s'^b$ tuple in the input steam S that $W_i^s \leq T_{s^p} - T_{s'^b} < W_i^e \Rightarrow s'^b \in S :: [W_i^s, W_i^e]$. Here $S :: [W_i^s, W_i^e]$ denotes the sliced state of stream S at J_i .

Proof: (1). In the purge step, the processing of s^p will purge any tuple s'^b with $T_{s^p} - T_{s'^b} \geq W_i^e$. Thus $\forall s'^b \in S :: [W_i^s, W_i^e], T_{s^p} - T_{s'^b} < W_i^e$. For the first sliced window join in the ring, $W_i^s = 0$. We have $0 \leq T_{s^p} - T_{s'^b}$.

For other joins J_i in the ring, at any moment let tuple s'^b denote the tuple in $S :: [W_i^s, W_i^e]$ that has the maximum timestamp. Tuple s'^b must have been purged by s^p (or another probe tuple with smaller timestamp) from the state of the up-stream join operator in the ring. Thus we have $W_i^s \leq T_{s^p} - T_{s'^b}$, for $\forall s'^b \in S :: [W_i^s, W_i^e]$.

(2). We use a proof by contradiction. (a) If $s'^b \notin S :: [W_i^s, W_i^e]$, then we assume $s'^b \in S :: [W_j^s, W_j^e], j < i$. Given $W_i^e \leq T_{s^p} - T_{s'^b}$, we know $W_j^e \leq T_{s^p} - T_{s'^b}$. Then s'^b cannot be inside the state $S :: [W_j^s, W_j^e]$ since s'^b would have been purged by s^p when it is processed by the up-stream join operator J_j . $A[W_{j-1}, W_j] \stackrel{s}{\times} B$. (b) Let us assume s'^b in the input queue of J_i . Since s'^b is purged by s^p and is inserted into the queue before s^p , s'^b cannot be in the input queue when s^p is being processed by J_i . (c) Let us assume s'^b in the output queue of J_i or down-stream joins. Since $T_{s^p} - T_{s'^b} < W_i^e$, no probe tuple will purge s'^b from J_i . In a summary, we got contradictions in all the possible cases. ■

Lemma 7 indicates the implicit synchronization of the sliced states in the ring of nodes by using the probe tuples, since the probing tuple is placed behind all purged tuples in the output queue. Lemma 7 is guaranteed due to the FIFO property of the network connections between processing nodes. The state maintenance at each node is coordinated by every probe tuple. Thus even though the state maintenance processes will not happen at the same time at all the nodes, the states are guaranteed to be consistent in terms of join probing process.

Coordinated state maintenance achieves implicit synchronization in the cluster. That is, the state synchronization is postponed as long as possible

until right before the join probing process commences. Also this coordination involves no extra network messages since the probing tuples have to be propagated for join probing purposes anyway. Along with the join progress, the probe tuples are propagated step-by-step along the ring.

12.2.2 Intermediate Result Propagation and Processing.

Intermediate results are propagated along the ring to probe the next state in the join ordering. Since the intermediate results are only used to probe other states, we treat them as the probe tuples. There is no state holding intermediate result in the ring.

For an M -way sliding join, the number of types of possible intermediate results is $O(2^M)$. One way to distinguish the intermediate results of different schemas is to use distinct network connection between each nodes in the ring for each type of intermediate result. However when M is large, the number network connections is too huge to afford. Instead, all intermediate results are transmitted in one network connection along the ring and the intermediate result schema is piggy-backed to identify the schemas. Also at runtime the schema of the intermediate result is used to determine the next state to join with.

An intermediate join result schema is denoted as (I_1, I_2, \dots, I_n) , where I_i can be a stream " S_i " or null " $-$ ". We assume that only one state exists for each type of intermediate result to probe next. That is, two join orderings that share the same prefix will join with the same sequence of states next. This assumption is hold when the given set of join orderings are optimal and each join ordering has distinct cost. Otherwise the two different join

orderings cannot be both optimal. Next we describe how the intermediate results are propagated and how the joined results are generated using an example below.

Given a set of optimal join orderings, the prefixes of join orderings define all possible intermediate join result schemas. For ease of illustration, below the intermediate result schema is also used to describe input stream tuples, e.g., the schema of tuples from stream A 's in a 4-way join is denoted as $(A, -, -, -)$.

For input stream tuples and also intermediate result tuples, the state that holds the tuples to be joined with the incoming tuples is called the active state. The active state denotes the state of the state-slice join operator to be probed next by the incoming stream (or intermediate) tuples.

Example: Consider a four-way join $A \bowtie B \bowtie C \bowtie D$ with the join ordering $C \rightarrow B \rightarrow A \rightarrow D$ for the tuples from stream C . The join result (a_i, b_j, c, d_k) , where i, j , and k (without loss of generality, assume $i < j < k$) denotes the serial number of nodes ($i, j, k \in [1, N]$) in the ring holding the corresponding state, is formed as follow. Tuple c is propagated to J_j first to probe the B state, and it generates intermediate result $(-, b_j, c, -)$. Then the intermediate result is propagated along the ring looping back to J_i to generate $(a_i, b_j, c, -)$, since $j > i$ and thus looping back is necessary. Then the newly generated intermediate result will be propagated further along to J_k to finish the join probing. No looping back is needed since $i < k$.

In the worst case, $(M-1)N$ hops of intermediate propagation are needed for an M -way join evaluation using a sliced join ring of length N . The average hops of intermediate propagation is then $(M-1)N/2$ since for each

probing step in the join orderings, $N/2$ hops is needed in average. This, potentially causing long response times, motivates the cost-based PSP optimization discussed in Chapter 13. However this does not necessarily imply that any node would have long idle times waiting for the propagation of the intermediate results, since pipelined execution is employed.

Clearly the selection of join orderings is independent of the state deployment in the ring. Further the PSP scheme allows each node to pick distinct join orderings to optimal join costs of different state slices.

12.2.3 Life Span Control in the Ring.

The purpose of life span control is: (1) dropping the input stream tuples and intermediate result tuples out of the ring at the right time to avoid generating incomplete or redundant join results; and (2) identifying the end of the processing cycle of a current input stream tuple at the head node.

In Figure 12.2, we observe that at any time, the build tuples of current sliding window are disjunctively sliced and deployed in a logical ring among the processing nodes. Thus each probing tuple, either input stream tuple or intermediate result tuple, is propagated along the ring exactly one and only one round. To achieve this, every sliced join operator assigns its unique node ID to the intermediate result tuples it generates. When the tuple reached the same operator again after one round propagation along the ring, the tuple is dropped from the system.

Since the processing of next stream tuple at the head node will cause state shifting along the ring, to ensure the correctness, the head node need to know the time when the processing of current input stream tuple is fin-

ished at all nodes. We design a special scheme to indicate the end of processing of current input stream tuple in the ring as follows.

A special *END* flag is used to mark the last intermediate result tuple with a certain intermediate result schema generated at the head node. A tuple with the *END* flag set is called the *END* tuple. First the *END* flag is set for the last intermediate result tuple that has been generated by the probing of the input stream tuple against the state in the head node. Future in the next probing step in the join orderings, the *END* flag is set for the last intermediate result tuple generated by the probing of the previous *END* tuple. The previous *END* tuple is dropped according to life span control. That is the “death” of the previous *END* tuple occurs together with the “birth” of the new *END* tuple. Thus at any time, there is one and only one *END* tuple in the ring for the current being processed stream tuple. Refer to Figure 12.3 for detailed steps.

Lemma 8 *The END tuple of certain schema Sch_i is the last intermediate result tuple processed at the head node having the schema Sch_i .*

Proof: Proof by induction.

(1)Base Case: Without loss of generality, assume that the state of the stream S is the first one in the join ordering for input tuple t . The state of S is sliced and distributed in the PSP ring as S_1, S_2, \dots, S_n . The first *END* tuple e_1 is the last tuple of the $t \bowtie S_1$. At any node i ($1 < i \leq n$) in the ring, tuple t is processed before e_1 , since propagation of t is taken care of before any probing with t will commence at any of the nodes. Thus $t \bowtie S_i$ (if any) will appear before e_1 in the input queue of the head node.

(2) Induction: Since e_j is the last tuple processed by the head node having a certain schema, thus the next END tuple e_{j+1} will be generated as the last tuple of probing with e_j against corresponding state. According to the FIFO processing sequence along the ring, e_j is the last one to be processed among the intermediate result tuples of the same schema as e_{j+1} at any nodes, including the head node. ■

According to Lemma 8, when the head node see an END tuple, it knows all the intermediate results with this schema have been processed by all the nodes. Thus we have the completion criteria.

Theorem 7 *For each input stream tuple, the processing cycle is ended by the processing of its $(m - 1)^{th}$ END tuple at the head node for an m -way join.*

Proof: From Lemma 8, the i^{th} END tuple is the last one that processed at the head node for any intermediate result having the same schema as the i^{th} END tuple. For an m -way join, there will be totally $m - 1$ probing steps in the join ordering and thus $m - 1$ END tuples. When the $(m - 1)^{th}$ END tuple is processed at the head node, all intermediate result tuples have been processed by all the nodes. ■

Comparing the brute force method of broadcasting the intermediate results to all the nodes at the same time, our pipelined propagation of the intermediate result guarantees the completion of each probing steps by using the END tuples without extra message between the processing nodes.

12.3 Execution Algorithm and Time Line

The sliced join execution algorithm is composed of four primitive routines: insert, cross-purge, propagation and probe, denoted as $insert(state)$, $purge(state)$, $prop(op)$ and $probe(state)$ respectively. In an m -way sliced join of streams S^1, S^2, \dots, S^m , the execution steps for a newly arriving tuple t in the sliced join number op_i are shown in Figure 12.3. We define the ID of the intermediate result generated by op_i as the number i and the sliced state of S^j in node i as S_i^j .

The head sliced join generates an *END* tuple to denote the finishing of the current round of the propagation. The execution period for the m -way join includes $m - 1$ rounds of propagation. After collecting $m - 1$ *END* tuples, the head sliced join initializes a new execution period for the next incoming stream tuple.

Figure 12.4 illustrates the execution time line in a four node cluster (each node holding one sliced join operator) for a 3-way join operator processing arrived a tuple from stream A . The accumulated input queue content is also shown for node M_2 and M_3 . Assume the optimal join ordering for A tuples is $A \rightarrow B \rightarrow C$. When tuple a arrives at node M_1 at time 0, first a build tuple a^b is made and $a^b.insert(S_1^A)$ is called. Then the probe tuple a^p is used to purge and probe state B , i.e. $a^p.purge(S_1^B)$, $(a^p.purge(S_1^B)).prop(M_2)$ and $a^p.probe(S_1^B)$ (done at time t_2). The intermediate result is send to M_2 to join with the states of C and eventually sent back to M_1 to join with S_1^C . M_2 will receive the probe tuple of a at time t_1 and follow the same execution steps as M_1 . The intermediate result I_1 will arrive at M_2 following tuple

If the tuple t is a build tuple from stream S^j ,

1-1. *Insert*: $t.insert(S_i^j)$.

If the tuple t is a probe tuple from stream S^j ,

2-1. *Purge*: $t.purge(S_i^l)$, $1 \leq l \leq m$. If op_i is the tail op, drop purged tuples; otherwise propagate purged tuples to op_{i+1} .

2-2. *Propagate*: If op_i is the tail op, drop t ; otherwise $t.prop(op_{i+1})$.

2-3. *Probe*: $I_i = t.probe(S_i^l)$, S_i^l is the state of the next stream in the given join ordering. I_i is the intermediate result with ID i . For head node, mark the last tuple in the intermediate result as the *END* tuple (If the probing has no output, a Null *END* tuple is generated).

2-4. *Propagate*: Send I_i with $I_i.prop(op_{i+1})$.

If the tuple t is an intermediate result tuple,

3-1. *Propagate*: If $ID \neq i$, $t.prop(op_{i+1})$, otherwise drop t .

3-2. *Probe*: $I_i = t.probe(S_i^l)$, S_i^l is the state of the next stream in the given join ordering. I_i is the intermediate result with ID i . For head node, if tuple t is marked as the *END* tuple and $ID = i$, mark the last output tuple as the new *END* tuple (If the probing has no output, a Null *END* tuple is generated).

3-3. *Propagate*: If I_i is final joined result, send out. Otherwise send I_i with $I_i.prop(op_{i+1})$.

Figure 12.3: Execution Steps of Sliced Join op_i

a and will be processed next by M_2 . Then same steps are followed by M_3 and M_4 . At time t_4 , one period of execution is finished. All the probe tuples and intermediate results are dropped after going through the ring. The next input stream tuple can be processed after t_4 .

The correctness of the sliced join algorithm relies on the FIFO nature of the queue connections between operators in the ring. We have the following theorem.

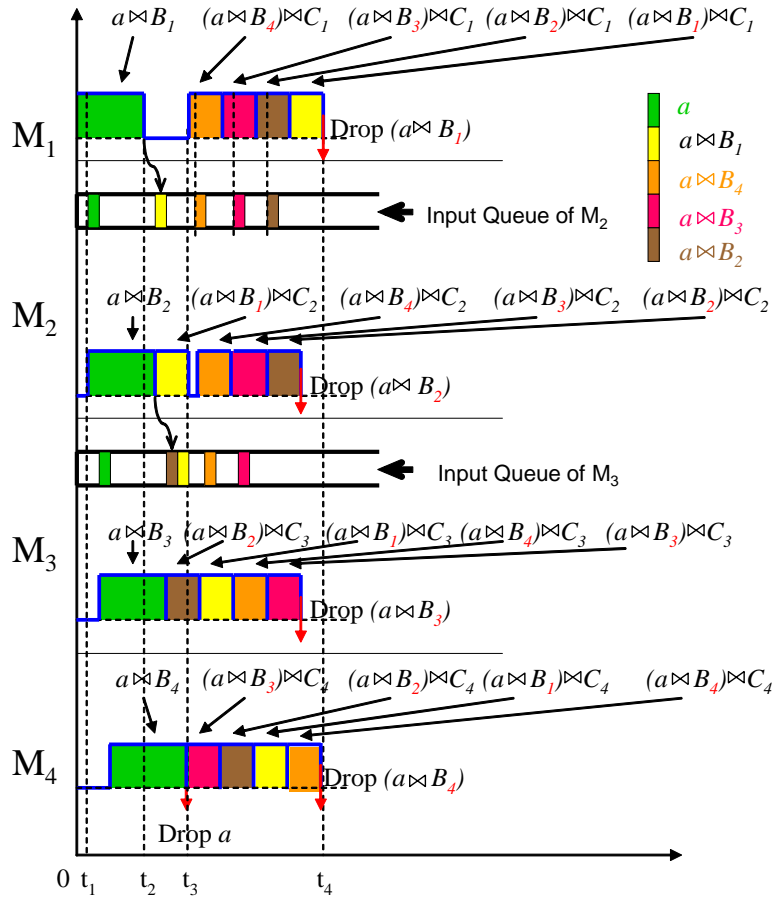


Figure 12.4: Execution Time Line of the PSP

Theorem 8 *The union of the join results of the sliced joins in the PSP ring is equivalent to the results of a regular multi-way sliding window join.*

Proof:

No missing results. From Lemma 7, the state slices is maintained consistently before any join probing. From Theorem 7, all state slices are probed before the end of processing cycle. The pipelined probings after a full round is guaranteed to cover all the corresponding states that need to be probed.

No extra results. Before any join probing, all the windows boundaries of the states are maintained consistently. All the probings are thus valid.

No redundant results. Lemma 7 guarantees that the window boundaries of the states are satisfied before probing. Thus the states are maintained disjointly in terms of probing. No redundant probing is conducted. ■

12.4 PSP with Interleaved Processing

The execution steps of the PSP scheme shown in Figure 12.3, does not allow interleaved stream tuple processing to assure consistent maintenance of the states in the basic PSP model. That is, only when the current stream tuple is “fully” processed before another stream tuple is admitted into the ring. When deploying the basic PSP scheme to the processing nodes of a cluster, a given processing node might be idle for some time waiting for other nodes to send them tuples to work on. Since the performance of the ring is determined by its slowest processing node, this may cause long idle periods. We thus extend PSP by means of a delayed purge strategy, called PSP-I, which enables the interleaved processing.

The processing of the next stream input tuple will cause insertion and purging of the states. To avoid state being purged too early by the new admitted tuple before being probed by the previous stream tuple (or the intermediate result tuples generated), every processing node maintains a list of active *StateStarts* and *StateEnds* pairs. Each pair marks the corresponding states for one of the currently being processed tuples in the system.

Instead of purging the states and removing purged tuples before probing, the StateEnds are used to mark the ends of states. The real state purging is postponed until no further probing requires the state anymore.

Although the purged state tuples are not physically deleted from the current node (they are just virtually marked as expired in some sense for a given tuple), they are propagated to the next processing node. During the join probing, only the part of the state within the appropriate StateStarts and StateEnds range relevant to the given tuple is used to join with the incoming tuple. The StateStart and StateEnd pair is expired and removed from the state when the corresponding END tuple is processed, because the later indicates that the tuple has successfully already visited all its join partners.

The purge step 2-1 in Figure 12.3 is now changed as shown in Figure 12.5.

2-1. *Purge:*

2-1-1. *Init:* Init a pair of StateStart and StateEnd.

2-1-2. *Mark:* Mark the states by setting the StateStart and StateEnd according to the input probe tuple.

2-1-3. *Propagate:* Propagate purged tuples to op_{i+1} (in case of not tail node) or drop them (in case of tail node).

2-1-4. *Delete:* After processing the END tuple, remove the corresponding StateStart and StateEnd and if this StateEnd has the smallest timestamp among all the StateEnd in the state, remove tuples older than this StateEnd.

Figure 12.5: Purge Steps with StateStart and StateEnd in $node_i$, $1 \leq i \leq n$

The interleaved processing of stream tuples induces duplicated states

among neighboring sliced operators due to the postponed deletion. In our implementation, we set a threshold to limit the number of concurrently processed stream tuples in the ring.

12.5 PSP with Dynamic Head and Tail

To avoid the extra cost caused by a tuple's repeated insertion into and purging from of a state-slice in the basic PSP scheme (namely, once for each of the n slices), we propose an extension of the PSP scheme in the form of a *dynamic head and tail abstraction*. Intuitively, we call the head and tail in the basic PSP model *static*, emphasizing that these two operators in the ring do not “move” during the evaluation of the sliding window join. On the other hand, the tuples in the states move from the head operator toward the tail operator along the ring and eventually leave the system after being purged in the tail operator. We now propose that instead of moving all the tuples through all the states to logically move the location of the logical head and the tail of the ring from operator to operator. Conceptually, this implies that the input stream tuples over time enter the system at a different operator, namely, at the current tail operator. We refer to this as PSP-D for *dynamic head and tail PSP scheme*. The PSP-D framework for a 3-way MJ is shown in Figure 12.6.

Instead of the window range, we now use window length to denote the window constraints of the states in each join operator. The window length of the state is the maximum possible difference of the timestamps of the tuples in the state, denoted as ΔW .

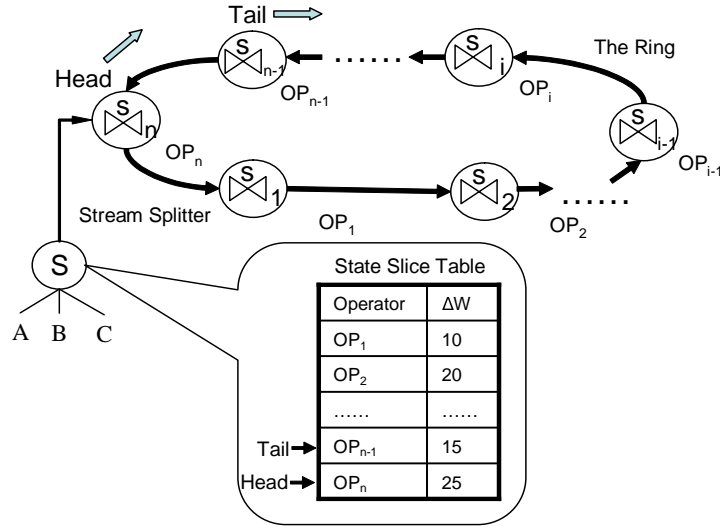


Figure 12.6: PSP with Dynamic Head and Tail.

In Figure 12.6, the current head and tail operator of the ring are OP_n and OP_{n-1} respectively. The stream splitter S maintains the state slice table. Each entry in the table describes the window length ΔW for an operator. The total window length of all the entries equals the sliding window constraints. The splitter also maintains the position and window slices of the all the operators in the ring. At runtime the stream splitter feeds the input tuples to the current head operator according to the state slice table. The insertion of tuples to the states happens only at the head and the purging of state tuples only happens at the tail. This mechanism now avoids the repeated tuple insertions and consequent purging. When the timestamp of the input tuple to be inserted is too large and is out of the window of the current head operator, it will be inserted in the current tail operator. That is, when the window of the current head operator is "full", the head of the

ring conceptually moves to the next operator in the reverse ring direction, i.e., to the tail operator. As time passes by, the state of the tail is eventually purged empty and then the tail also moves to the next operator along the reverse ring direction. During this transition, the logical head and tail may reside on the same operator. The stream tuples are inserted into the states of the head operator and stay there until being purged out of the system when this operator becomes the tail operator.

Intuitively the dynamic head and tail approach turns multiple state sliced sliding windows into one insertion only window at the head operator and one deletion only window at the tail operator with several fixed windows in the operators between the head and the tail operators. Thus only the states in the head and tail operators need to be maintained. This approach only affect the insertion and purging process since the head and tail will not change before processing next stream tuple. The execution steps of probing and propagation of intermediate results are exactly the same as that in the basic PSP model shown in Figure 12.3.

12.6 Interleaving Processing with Dynamic Ring Structure

During the execution, we may prefer to interleave the processing of multiple input tuples to avoid any operator waiting idly for the input. Since the execution period of one input tuple may be rather long for a join operator with a large number of input streams, the processing of the next input tuple should be started as soon as possible, instead of waiting until completion

of the previous processing period.

In the dynamic head and tail approach, the head and tail are determined by the current being processed stream tuple and may move when the next stream tuple is processed. Thus multiple heads and tails are necessary to allow the interleaved execution of stream tuples. Figure 12.7 shows the multiple heads and tails as the vertical dotted lines. The multiple heads and tails may distributed among multiple processing nodes. Also the state slice table need to be extended to keep track of all the heads and tails in the ring.

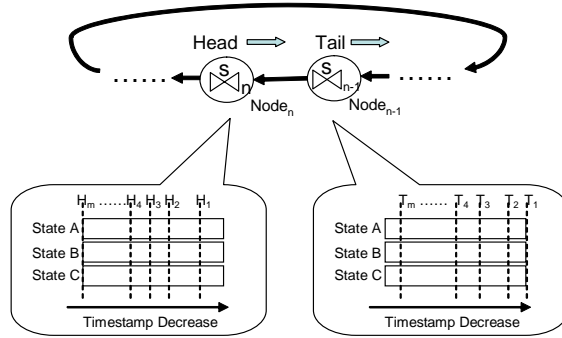


Figure 12.7: PSP-D with Multiple Heads and Tails.

The purging of the states in the tail node must be postponed until they are guaranteed not to be needed by any of the current being processed tuples and intermediate results. The delayed purge is the same as that in the PSP-I scheme using the corresponding head and tail as the StateStart and StateEnd marks.

Chapter 13

PSP: Cost Analysis and Tuning

In this chapter, we develop a cost model for PSP and use it to tune the parameters for different performance objectivities. Our cost model provides the necessary analytical equations to model the relationships between the following key parameters of the PSP model: (1) stream and query parameters, including stream arrival rates, window sizes and join selectivities; (2) PSP ring parameters, such as the number of nodes in the ring; (3) performance measurements, such as the average response latency (average time difference between sending out the joined result and reading in corresponding stream tuple) and output rate.

13.1 Cost Model

For an M -way join operation $S_1 \bowtie S_2 \bowtie \dots \bowtie S_M$, the parameters for the cost model are given in Table 13.1.

We assume that the network bandwidth is sufficient for our workload.

Table 13.1: Terms Used in Cost Model

Term	Meaning
λ_i	Arrival Rate of Stream i
W_i	Window Size of the Sliding Window on Stream i
$S_{\bowtie i}$	Join Selectivity for i^{th} probing step
T_j	Time spent to join a pair of tuples
T_p	Time spent to purge one tuple from a state
T_i	Time spent to insert one tuple into a state
T_s	Processing latency to send & receive one tuple
T_n	One hop network transmission latency
N	Number of processing nodes in the ring
M	Number of incoming streams
μ	Service rate of the PSP ring

The network latency then is proportional to the number of hops of transmission. We estimate the sending and receiving latency between processing nodes to be proportional to the number of tuples transmitted. The output rate is estimated with the assumption that all the processing nodes are 100% busy during the execution. That is, the input queues of the head operator are never empty. The output rate under such assumption is the maximum output rate possible.

We first calculate the processing workload L_C for the centralized join processing of one input tuple and the workload L_{PSP} for the processing of the same input tuple in the PSP scheme. The workload indicates the total time needed to process one input tuple. The workload can be calculated by summing up the CPU join time, the state maintenance time, and the network transmission time. We assume an in-memory nested loop join al-

gorithm is employed. We also assume that the optimal join ordering for the input tuple from stream S^1 is: $S^1 - > S^2 - > \dots - > S^M$ and the processing nodes and the network connections between them are homogeneous.

$$\begin{aligned}
 L_C &= T_i + T_p + T_j \sum_{2 \leq k \leq M} \left(\prod_{1 \leq i \leq k-1} \lambda_i W_i S_{\bowtie i} \right) \\
 L_{PSP} &= L_C + T_s N \left(1 + \sum_{2 \leq k \leq M-1} \left(\prod_{1 \leq i \leq k-1} \lambda_i W_i S_{\bowtie i} \right) \right)
 \end{aligned} \tag{13.1}$$

The third item for L_C is the total join probing cost. The second item for L_{PSP} is the total transmission cost for the input tuple and the intermediate results.

For succinctness of the analysis, we simplify the cost model by assuming $\lambda_i = \lambda_j$, $S_{\bowtie i} = S_{\bowtie j}$ and $W_i = W_j$. These assumptions can be relaxed without changing the principles of the cost analysis. Thus:

$$L_{PSP} \approx L_C \left(1 + \frac{T_s N}{\lambda W T_j} \right) \tag{13.2}$$

Every L_{PSP} seconds, PSP processes one input stream tuple. Thus the service rate μ (i.e. the number of tuples processed per second) is given as:

$$\mu = \frac{1}{L_{PSP}} \tag{13.3}$$

13.2 Cost-based Tuning

Based on the cost model, we perform PSP optimization for the following two important objectives: (1) given a fixed stream arrival rate, maximize the system output rate by tuning the length of the ring; and (2) given a fixed stream arrival rate, minimize the average response latency by tuning the length of the ring. In the following discussion, we assume that we have knowledge of the stream parameters and query parameters (i.e., all terms in Table 13.1 except μ). All these parameters are straightforward to measure in an actual implementation of PSP.

13.2.1 Maximize Output Rate.

In a homogeneous cluster, all processing nodes have identical CPU power. Assume the output rate for one single processing node with workload L_C is O_S . Then the output rate O_{PSP} in the PSP model with N processing nodes is:

$$O_{PSP} = \frac{O_S N}{1 + \frac{T_s N}{\lambda W T_j}} = \frac{O_S}{\frac{1}{N} + \frac{T_s}{\lambda W T_j}} \quad (13.4)$$

From Equation 13.4, as more processing nodes are deployed in the PSP ring, the output rate increases monotonically. That is, using more processing nodes will result in a higher output rate.

13.2.2 Minimize Average Response Latency.

To estimate the processing latency of the PSP model, we consider the average latency for join results from one input tuple. We estimate the latency assuming perfect load balancing among all processing nodes. That is there is no bottleneck processing node slowing down the flow along the ring. Such latency is the minimal latency achievable. The latency has mainly two parts: join probing and network latencies. These two latencies overlap in time during execution. For each processing node, the balanced workload is L_{PSP}/N on average. The final join results are generated after a total of M rounds of transmission of the intermediate results along the ring. Thus processing latency τ_i for node i , $1 \leq i \leq N$ is:

$$\tau_i = (i - 1)T_n + \max\left\{\frac{L_{PSP}}{N}, MNT_n\right\}$$

Thus the average processing latency τ is:

$$\tau = \frac{N - 1}{2}T_n + \max\left\{\frac{L_C}{N} + \frac{T_s L_C}{\lambda W T_j}, MNT_n\right\} \quad (13.5)$$

For clarity, we omit state insertion and deletion delay since each of them is one time cost for each input tuple. Such delay is independent of the number of join results generated.

From Equations 13.5, we see the response latency is sensitive to the number of the processing nodes N in the PSP ring. Intuitively, adding more processing nodes increases the CPU power. On the other hand, the longer

the length of the ring the higher the network transmission costs. Using standard calculus methodology, we can find exactly the value of N that minimizes the average response latency. We have following theorem:

Theorem 9 *The PSP ring has the minimal processing latency when $N = \min\{N_1, N_2\}$, where*

$$\frac{L_C}{N_1} + \frac{T_s L_C}{\lambda W T_j} = M N_1 T_n, \text{ and } \frac{N_2 - 1}{2} T_n = \frac{L_C}{N_2}$$

The processing latency for each node is decreasing with larger N , while the network latency is increasing. Both facts need to be considered for the optimal ring length with minimal processing latency.

13.3 Initial State Slicing

When the arrival rates and sliding window sizes are different for each input streams, naturally the optimal ring lengths would be different for individual streams. The problem of achieving global optimal lengths of rings is much harder than the simplified case discussed previously. In fact the search space is exponential since the optimal lengths of the PSP rings are correlated with each other. Thus searching for the optimal initial state slicing is expensive and may not be worthwhile, especially for stream processing in a highly dynamic environment. Instead we use the following heuristic to achieve a sub-optimal state slicing.

We first sort the streams by $\lambda_i W_i \overline{S_{\times i}}$ in ascending order. Here the $\overline{S_{\times i}}$ denotes the average join selectivity between stream i and other streams in the join graph. Then the optimal lengths of rings is calculated in the order

of the sorted list of streams. In the calculation, if the length of the ring for certain stream is unknown, then current length is assigned. For example when calculating the length for stream i and the length for stream j is not available (i.e., stream j is behind stream i in the sorted list), then the ring length for stream j is assigned the same as stream i . The intuition behind this heuristic is that the streams with large $\lambda_i W_i \overline{S_{\times i}}$ in the sorted list have larger impact on the total cost, such that should be processed later when more information about other streams is available.

13.4 Workload Balancing

In Figure 12.2 we indicate that the deployment of state sliced windows may not be even among all the nodes. Since PSP is a pipelined execution model, the performance of the PSP ring is determined by the busiest node in the ring. To avoid any bottleneck node, a workload re-balancing must thus be achieved for optimal performance.

From Equation 13.1, the dominant CPU cost for each node is the join probing cost, which is proportional to the total size of the sliced states in the node. To balance the workload of each node, we thus suggest as a heuristic to keep the number of state tuples balanced in every node. Since the state slicing boundaries between adjacent join nodes can be performed arbitrarily at the optimizer's will, the balanced state distribution can be achieved.

Chapter 14

PSP: Adaptive Load Diffusion

Adaptive workload diffusion is critical for realistic long running query processing, when stream arrival rates, join selectivities, and load of nodes change at runtime. In Chapter 13 the discussion is based on static statistics, however runtime statistics may change dramatically making runtime adaption critical. In PSP, adaptive workload diffusion is achieved by state relocation among the nodes by setting the corresponding window ranges. We tackle two major load re-balancing scenarios: *workload smoothing* among the same set of nodes and *state relocation* with more/less nodes. Both adaptations are rather straightforward and inexpensive to implement.

Two factors determine the performance of the state-slice ring, namely the length of the ring and the load balancing among the processing nodes in the ring. Both factors can be controlled through the state assignment among the nodes by the data sender.

We assume the runtime statistics are collected periodically by sampling the input streams. Runtime statistics collection is an orthogonal topic and

we assume given statistics in this dissertation. The runtime adaptation includes statistic collection, conducted periodically by sampling the input streams and system measurements that triggers the adaptation.

We will consider two major scenarios to be tackled by our adaptive optimization: namely, short term load burst with workload smoothing and long term load fluctuation with state relocation. The workload smoothing is suitable for the case when the system is not overloaded, while state relocation is conducted when system overloading is observed.

14.1 Workload Smoothing

The runtime stream arrival rates may always fluctuate, while the overall system is *not overloaded*. In a homogeneous cluster, we initially slice the time-based window ranges evenly among all the nodes, aiming for balanced workload. However such time-based state slicing may suffer from the short term load burst, since the state size, which determines the workload of each node, may vary significantly at runtime. The reason is that the fluctuating arrival rates will make the state size on each node unbalanced given fixed window ranges. System performance is slowed-down by the overloaded node in the ring.

Here we propose that instead of a time-based state slicing, a count-based state slicing can be employed to smooth the workload. Each sliced join, except the last one held by the tail node, has an upper bound on the state size and a count-based state purging is employed when the state size grows over the threshold. The upper bound is set periodically according

to the given statistics. When the average stream arrival rates and the total window sizes are given, the upper bound can be set to ensure even distribution of the tuples in the states. Since the upper bound is calculated using the average arrival rates, it can “smooth” the workload during short term load burst. The correct time-based semantics of the sliding window join continue to be ensured by the tail node since it still uses the time-based state purging.

With count-based workload smoothing, the state slice table in Figure 12.6 has one more column C , denoting the pre-determined count based upper bound for each processing node. Accordingly, the ΔW column is updated at runtime.

Such count-based workload smoothing is effective when the window constraint is large. For a small window to be close to the statistic sampling intervals, the statistics may be imprecise.

14.2 State Relocation

Adding/removing of nodes is needed when system is overloaded or the ring length is not optimal for response time. Two approaches for adaptive optimization are proposed: *passive adjustment* of the window range and *aggressive adjustment* by state relocation.

Passive State Adjustment. When long term load burst happens, passive adjustment aims to relocate the state by setting the window ranges. Consider an example of adding one node to a ring composed of 3 nodes. We assume the states are sliced equally among the processing nodes N_1 to

N_3 (N_4 finally). That is, the states in each processing node will be changed from $W/3$ to $W/4$, with W denoting the window constraint. The state slices in the original processing nodes N_1 to N_3 are step-by-step replaced and shrunk. Finally the new state allocation with one additional processing node is achieved. Similarly, node removal can be conducted.

The graceful state adjustment induces no extra migration cost. However a long adjustment latency may occur for large window size.

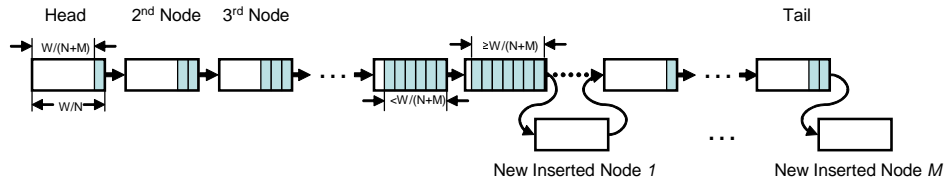


Figure 14.1: Aggressive State Relocation.

Aggressive State Relocation. To reduce the adaptation latency, aggressive state slice adjustment migrates some part of the states along the PSP-D ring. Such state relocation needs to suspend the execution and resume afterward.

To maintain the ring structure, the state slice movement happens only between adjacent processing nodes. Intuitively, a new processing node should be inserted into the ring at the appropriate position so that the shifted state slice can fill directly the new node. That is, let us assume that the ring has N nodes originally and that another M nodes need to be added into the ring, the i -th processing node from the head node need to move $\Delta S_i = \frac{M}{N}i - \lfloor \frac{M}{N}i \rfloor$ state tuples to the next nodes in the direction of the ring towards the tail node, as illustrated below.

$$\begin{aligned}
\Delta S_i &= \left(\frac{W}{N} - \frac{W}{N+M} \right) \frac{i}{\frac{W}{N+M}} - \left\lfloor \left(\frac{W}{N} - \frac{W}{N+M} \right) \frac{i}{\frac{W}{N+M}} \right\rfloor \\
&= \frac{M}{N} i - \left\lfloor \frac{M}{N} i \right\rfloor
\end{aligned} \tag{14.1}$$

The new processing node N_j , $1 \leq j \leq M$, needs to be inserted *after* the processing node N_k , such that k is the minimal number with $\frac{M}{N}k > j$. Figure 14.1 illustrates the addition of a new processing nodes. Similarly, the removal of processing node can be conducted.

The aggressive state relocation involves execution breaks and state migration during the adjustment. Frequent aggressive adjustment should be avoided.

Chapter 15

Discussion

15.1 State Replication Based Distribution

To the best of our knowledge, the only work on distributed processing of generic multi-way stream joins is [GYW07]. [GYW07] proposed two *state replication* based distributions for generic MJ operators: aligned tuple routing (ATR) and coordinated tuple routing (CTR). We briefly review these two approaches and then compare them with PSP below.

ATR picks one input stream as the master stream and partitions the master stream among the processing nodes. All the other slave input streams are distributed to the processing nodes with some overlaps of the states, to ensure the semantics of the window constraints. CTR is a multi-hop semantics preserving tuple routing where intermediate join results are transferred among nodes during each hop. A weighted minimum set covering is utilized to identify the best routing for each tuple to “find” all relevant states. Details of these two approaches can be referred from [GYW07].

Memory Cost.

The distribution strategies of both ATR and CTR are based on state (partial) duplication among the processing nodes. Compared to them, our proposed PSP approach does not have any duplicated states at any time.

In ATR the segment length T is an important parameter for the load diffusion. However, the ATR approach works under the condition that the window constraint $W \ll T$. When W is comparable with T , the memory waste and redundant computation can be significant, as illustrated below.

ATR duplicates the states of the slave input streams thus it may use extra memory and CPU to process them. For ease of illustration, we now assume that all input streams have the same arrival rate, and the master stream is not switched during the cost estimation. Besides the notations in Table 13.1, we now introduce T to denote the stream segment length.

For the $M - 1$ slave streams, each segment is set to be of size $T + 2W$ length, with W as the window size. Assume each segment is assigned to one processing node. Then the total memory consumption of the ATR approach is:

$$MEM_{ATR} = \lambda TN + \lambda(T + 2W)(M - 1)N$$

In other words, the duplicated (wasted) state memory is:

$$\frac{MEM_{ATR} - \lambda TNM}{\lambda TNM} = 2 \frac{W}{T} \frac{M - 1}{M}$$

From above equation, we can see that the memory waste is proportional to W/T . When $W \ll T$ is not the case, the memory waste can be significant.

In CTR the number of redundant states is determined by the minimum

set covering at runtime. CTR faces the following dilemma. The more redundant states, the smaller set covering of less nodes may exist. Then the incoming tuples will be stored in fewer states, which may make future set covering large. A more serious concern is that the states in CTR may converge to one (or a small subset of) node if sometimes only one copy of the input tuples is stored in a certain node, since future set covering will direct all later tuples to that node. Then no distribution is achieved. Unless an optimal insertion algorithm is employed (missing in [GYW07]), which predicts future workload diffusion, the CTR is uncontrollable and ad hoc.

The CTR scheme makes L copies ($L = 2$ in [GYW07]) of the input tuples and allocates them to multiple processing nodes. Thus L copies of each input tuple are stored in L different states of the processing nodes. Obviously the memory consumption is also L times of the input stream size.

Synchronization.

ATR results in a set of independent join operators that no synchronization is needed. However, CTR does need synchronization among nodes in different hops for maintenance of the states and processing of intermediate results. The synchronization is missed in [GYW07].

CPU Cost.

CPU cost comparison is summarized in Figure 15.1. Here we list only the main factors affecting CPU cost.

CTR employs a complex routing algorithm to determine the optimal routes for each segment of input streams. Such routing cost is per segment cost and may be significant with fine-grained segments. On the contrary, ATR and PSP do not require routing by employing one hop computation

Item	ATR	CTR	PSP
Routing Cost	Low	High	Low
Per Segment Metadata	No	Yes	No
Duplication Removal	No	Yes	No
Load Balancing Granularity	Large	Small	Small
State Management Cost	High	High	Low
Adaptation Cost	Unknown	Unknown	Low
Network Transmission	Low	Middle	High

Figure 15.1: CPU Consumption Comparison

and fixed routing respectively. The routing information and other metadata must be attached to each segment to ensure the correctness in CTR, while no such requirement exists for ATR and PSP-D. Further CTR needs extra work to avoid generating duplicated results while the other two will not generate duplication in the first place. The ATR and CTR approaches both duplicate states and thus the state management costs are much higher than for PSP. At runtime, each segment of the input stream is processed by only one processing node for ATR, several nodes for CTR and the optimal number of processing nodes for PSP. Thus the processing latency using PSP is expected to be the lowest.

The disadvantage of PSP is that the network transmission cost may be larger than that for ATR and CTR, since all input tuples and intermediate results need to be send along the ring. We limit the usage of the PSP scheme to a cluster with local high speed network only.

15.2 Stream Tuple Processing Order

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

The state slicing strategies described in previous chapters are based on the assumption that the *totally ordered execution model* is being used, where all the stream tuples are processed in a global order according to their timestamps. This model is the most strict execution model that guarantees that the timestamps are monotonously increasing in every operator. This simplifies the state purging process in the state slicing approach and avoids potential missing joined results that may otherwise arise if other less restricted models were being used.

In this section, we generalize the state slicing strategies proposed in previous chapters by relaxing the assumption about strict execution model. First we categorize the execution models and then discuss their corresponding tuple purging algorithms. These execution models include the *totally ordered model*, the *semi-ordered model* and the *unordered model*. We identify the necessity of applying at least *semi-ordered* processing model for correct state slicing. We then describe the changes that need to be made to the state slicing strategies when used in systems that employ the semi-ordered model.

15.2.1 Execution Models

For continuous query processing, tuples arrive at run time and need to be processed in certain orders. For a multi-way join operator (state sliced or not) that have multiple input streams, the operator determines that which tuple is processed next. Such execution orders can directly affect the purging and probing processes of the operator.

Totally Ordered Execution Model

This is the most strict execution model for continuous query processing. When using this model, tuples are being processed in *exactly* the order as their timestamp, independent of their stream source names. By applying the complete synchronized execution model, all probe tuples (or build tuples) in any queues in the state sliced query plan are ordered by their timestamp.

By using this model, a tuple t_1 that has a smaller timestamp than a tuple t_2 is guaranteed to be processed before t_2 , even if t_1 and t_2 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.

Semi-Ordered Execution Model

The *semi-ordered execution model* is a bit more relaxed than the previous model. This model only enforces that a operator processes tuples in each of its input queues in increasing order of their timestamps. Thus tuples from different input queues can be processed interactively. Such an execution

model is necessary for batch processing of the input stream tuples.

Different from the totally synchronized model, this model *only* enforces the tuple execution order to be the same as the tuple arrival order locally for each input queue. It does not enforce the tuple execution order across *all* input streams. Although this model is more relaxed in execution order, the tuples in each queue are still ordered by their timestamp because each operator processes tuples in its input queues in the right order.

Un-Ordered Execution Model

The *un-ordered execution model* does not pose any constraints on the tuple execution order. 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 joined result tuples are ordered neither by max nor by min timestamp of sub-tuples.

The execution model can determine the state purging and thus may affect the state sliced join processing. 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). To ensure the correctness of the state sliced join, as we will show below, the out-of-order execution creates a problem during the state sliced join process. Thus at least semi-ordered execution must be adopted.

15.2.2 State Sliced Join Processing with Semi-Ordered Execution

In previous chapters, all discussions are based on the total-ordered execution model. In this section, we first show that the un-ordered execution model cannot guarantee the completeness of the joined result. Then a lazy purge is proposed for state sliced join process in semi-ordered execution model.

Consider join $A[w] \bowtie B[w]$ of streams A and B , no purge based state slicing can be achieved with un-ordered execution model. The reason is that without any other information, any build tuple must stay in the first state sliced join operator to wait for possible future probing of out-of-order tuples.

Following is an example ($A[w] \bowtie B[w]$) of the purging and probing used with semi-ordered execution model. Assume tuples a_1, a_2 and tuples b_1, b_2 arrive at the system with timestamps $T_{a_1}, T_{a_2}, T_{b_1}, T_{b_2}$ respectively, where $T_{b_1} < T_{a_1} = T_{a_2} < T_{b_2}$. One possible semi-ordered execution sequence is: b_1, a_1, a_2 and b_2 . To generate the complete joined result, the processed tuple m need not only probe the state tuple n that satisfies $T_n > T_m - w$, but also $T_n < T_m + w$. Similar to the interleaved PSP scheme, the lazy purge is used to keep the state tuple until no other stream tuples will purge this tuple. To achieve this, each state maintain a mark for each crossing purge from other input streams. Only the part of the state that out of the sliced windows of all the other streams will be really removed from the current state.

For correctness, we have:

Lemma 15.1 (Complete Joined Result) *State sliced join processing with lazy*

purge will generate complete joined result in semi-ordered execution model.

Proof:

- No duplication. Proof by contradiction. Assume two joined tuples are exactly the same, in the form of t_1, t_1, \dots, t_n , where t_i is the tuple from stream I . Then these two tuples must be generated when processing different input probing tuples. Let them be t_m and t_n . Then it means tuple t_m is processed before t_n since t_n is already in the state. Similarly t_n is processed before t_m , which contradicts to the previous claim.
- No missing result. Assume tuple t_1, t_1, \dots, t_n is a valid joined result and input tuple t_i is the last one begin processed among t_1, \dots, t_n . Then from lazy purging, we know this joined result will be generated.

■

The timestamps of the output of state sliced join with semi-ordered execution model is not ordered by the max of the timestamps of the input tuples. However we have:

Lemma 15.2 (Output Timestamp Order Lemma) *Let t and t' be two tuples in the output queue of a state sliced window join operator. Both tuples have timestamps of size n , represented as $[TS_1, \dots, TS_n]$ and $[TS'_1, \dots, TS'_n]$ respectively. If tuple t appears earlier than tuple t' in the queue, then there must exist at least one i ($1 \leq i \leq n$), such that $TS_i < TS'_i$.*

Proof: Proof by induction on the size of timestamp array n .

Basic: $n = 2$. Let $[TS_1, TS_2]$ and $[TS'_1, TS'_2]$ be the timestamps of the intermediate result tuples t and t' respectively. When intermediate result t is generated before t' , there are only two possible cases. (1) If t and t' are both generated by the same probing tuple, then $TS_1 = TS'_1$ (or $TS_2 = TS'_2$). Thus $TS_2 < TS'_2$ (or $TS_1 < TS'_1$) since the state is ordered by the timestamps and the probing is in the same order. (2) If t and t' are not generated by same probing tuple, then the timestamps of the two corresponding probing tuples have: $TS_1 < TS'_1$ or $TS_2 < TS'_2$, according to the semi-ordered execution model.

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

Inductive Step: We 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 t_1 and t_2 with timestamp arrays as $[TS_1, \dots, TS_{i-1}, TS_{i+1}, \dots, TS_{k+1}]$ and $[TS_i]$, respectively. Similarly, t' can also be treated as the combination of two sub-tuples t_1' and t_2' with timestamp array as $[TS'_1, \dots, TS'_{i-1}, TS'_{i+1}, \dots, TS'_k]$ and $[TS'_i]$ respectively, where tuple with TS_i is the probing tuple. Using the same reasoning as in the base case, we have two cases possible: (1) $TS_i = TS'_i$, then from induction hypothesis, there must be one $TS_i < TS'_i$. Or (2) $TS_i < TS'_i$.

■

Above lemma is used to limit the memory of the union operator to sort the joined result. Any tuple that has the maximum timestamp of the timestamp array is smaller than the minimum timestamp of the timestamp array

of the incoming tuple can be safely removed from the union operator.

Chapter 16

Experimental Evaluation

In this chapter, we present an experimental study that showing the performance of the PSP model and comparing it with other state-of-the-art approaches.

16.1 Experiment Settings

Distributed Join Algorithms. We have implemented the proposed PSP in a real distributed DSMS system, the *D-CAPE* [SLJR05]. Experiments have been conducted to thoroughly test the ability of the proposed solution under various system resource settings.

D-CAPE is implemented in Java and the PSP model is implemented as a regular operator inside. The PSP ring-based query plan is formed first and then deployed in the cluster using the regular pipelined parallelism of the DSMS.

To compare the performance of PSP with other approaches, we also im-

plement the ATR and CTR proposed in [GYW07]. For ATR, a special stream data diffusion operator is implemented, who is in charge of centralized control of the segments from all input streams. The data diffusion operator in the ATR model has one important parameter, namely the segment length, for performance tuning. For CTR, the data diffusion operator has a routing table and can calculate the routing path for each input stream tuple. To avoid uncertainty of the minimum set cover algorithm, we add one parameter for CTR, enforcing the number of copies of the state tuples among all the processing nodes (this number is set to 2 in [GYW07]). We also enforce random deployment of the states, such that the minimum set cover algorithm will return a consistent number of coverings for each probing step. We also enforce the synchronization of the multi-hop execution in CTR. That is, no interleaved processing of multiple input segments is applied to avoid state overlap. To avoid bottleneck for the data diffusion operator, it is deployed separately in one node without other join operators.

To measure the system performance under different join selectivities, we using a probabilistic join probing that the join selectivities can be controlled.

Query Sets. The MJ operator is used by a clustering algorithm to identify similar images captured by different sensors for object movement detection. The similarity of images is defined based on their distances calculated from the RGB values for all image pixels. A symmetric nested loop join algorithm is used in the experiments. The tuples in the data streams are generated according to the Poisson arrival pattern. The stream input rate is changed by setting different mean inter-arrival times between two tuples.

Our experiments use three different join costs: *small*, *middle*, *large* corresponding to images with 5k, 10k and 20k pixels, respectively.

Evaluation Metrics. We use two measurements in this experimental study. We measure the runtime memory usage in terms of the number of tuples in the states of the joins. We also measure the output of the query plan in terms of the average response latency for the join results.

Experimental Platform. All experiments are conducted on a cluster that consists of 20 processing nodes and one master node. Each host has two AMD 2.6GHz Dual Core Opteron CPUs and 1GB memory. All the hosts are connected by gigabyte private networks. Each processing node runs an instance of our DSMS query processing engine executing one multi-way join operator. The master node acts as the synthetic stream data sender, which runs a stream generator and diffuses generated tuples to the processing node holding the head operator in the ring. The master node also collects the join results from each sliced join as the data sink. Each query processor has a monitor thread that collects the runtime statistics of each operator. All the experiments start with empty states in all operators.

16.2 Experiment 1: Sensitivity Analysis for PSP

In Chapter 13, we have presented an analytical cost model on the parameters of the PSP model. In the cost model, the most critical part is the optimal length of the ring for stream state slicing. In this experiment, we validate the cost model by varying the system parameters, including (1) the number of joins as: 3-way, 5-way, 7-way and 9-way; (2) the join cost as: *small*,

middle and large; (3) the join selectivities as: 0.05, 0.1 and 0.5; and (4) the number of processing nodes: 4-19.

The sliding window size is set to be 10k ms for all the streams. The input rate is set to 50 tuples/sec per stream. In all the experiments, the system will run for 600 seconds.

We first show the cost breakdown of the network cost and different join probing costs using a 3-way join as an example in Figure 16.1. The experiments are conducted using 4 nodes in the cluster and the join selectivity is set to be 0.1. Figure 16.1 shows the average cost, and the error bars of one standard derivation. All the tuple probings and transmissions are counted in, even no final joined results being generated. Since the three runs only differ with each other for the join cost and the network cost is the same, then only one network cost is shown. Figure 16.1 gives us a brief idea of the cost breakdown of the total response time.

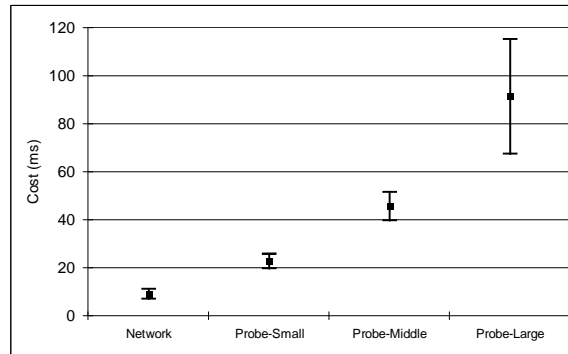
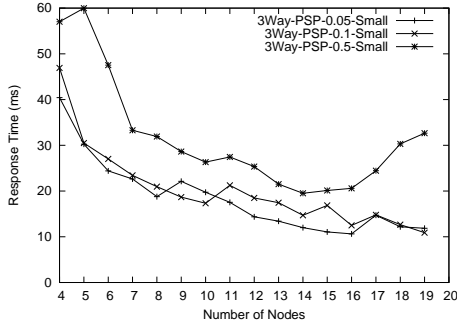
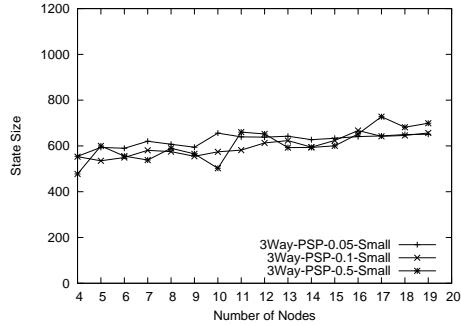


Figure 16.1: Cost Breakdown for an Example 3-Way Join Query.

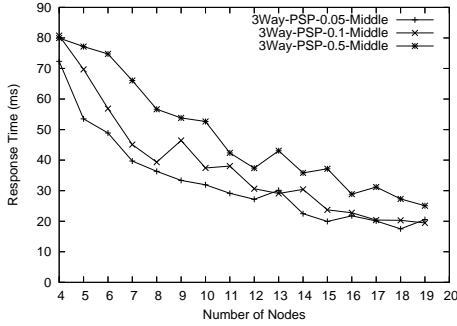
Figures 16.2(a) to 16.2(f) show several of the experimental results for the 3-way join with different join selectivities and number of nodes. The



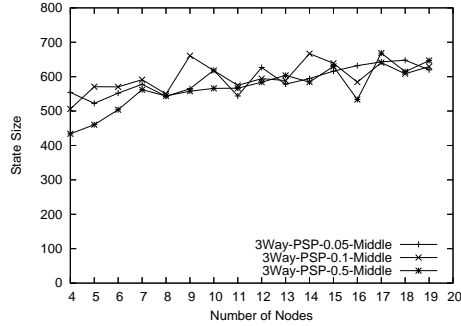
(a) PSP, Small Join, Response Time



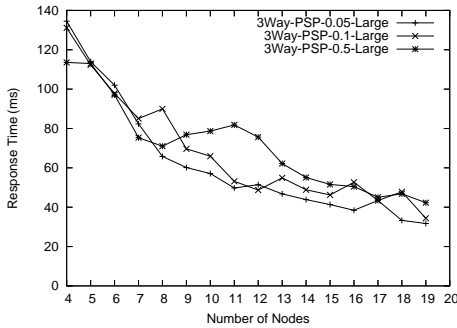
(b) PSP, Small Join, State Size



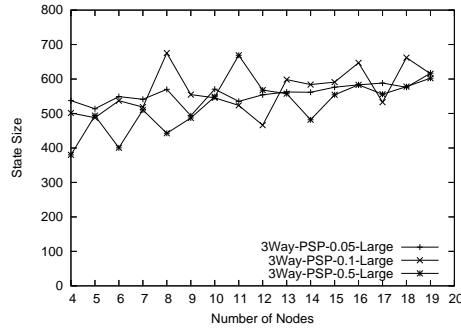
(c) PSP, Middle Join, Response Time



(d) PSP, Middle Join, State Size



(e) PSP, Large Join, Response Time



(f) PSP, Large Join, State Size

Figure 16.2: Performance Analysis of the PSP scheme

legend reads as: Number_of_way-Join_scheme-Join_selectivity-Join_cost.

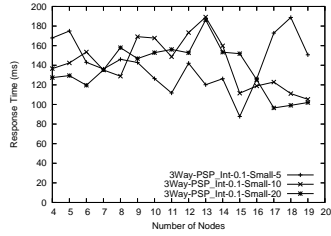
The PSP scheme does not have any duplicated states, thus the mem-

ory consumptions are pretty stable among all the experiments. The query response latency is sensitive to the join selectivities and the cluster size, which affect the number of intermediate join results and join probing respectively. We observe that the average response time increases for larger join selectivities, since the workload is increased accordingly. When the cluster size increases, the response time will not decrease all the time. Instead, it will increase when the size of cluster is too large. In Figure 16.2(a), the response time increases after having 14 nodes in the ring when join selectivity is 0.5. Also this number is sensitive to the join selectivities since different number of intermediate result will be generated and transmitted along the ring. For smaller join selectivities, we expect large optimal number of nodes in the ring. This is consistent with the cost based ring length optimization discussed in Chapter 13.

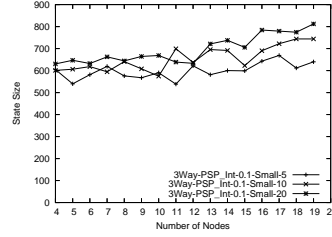
16.3 Experiment 2: PSP with Interleaved Processing

The PSP-Int scheme allows multiple input stream tuples to be processed concurrently in the PSP ring. The delayed purging is used to maintain correct sliced states in the corresponding operators.

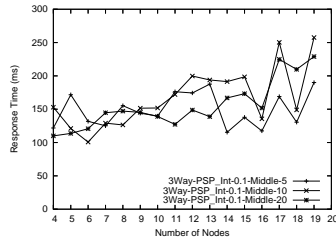
In this experiment, we compare the performance of PSP-Int with the PSP model under different workloads by varying the system parameters, including (1) number of ways of joins as: 3-way, 5-way, 7-way and 9-way; (2) join cost as: small, middle and large; (3) join selectivity as: 0.05, 0.1 and 0.5; (4) number of processing nodes: 4-19; (5) Number of concurrent processed tuples.



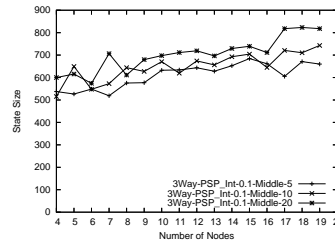
(a) PSP-Int, Small Join, Response Time



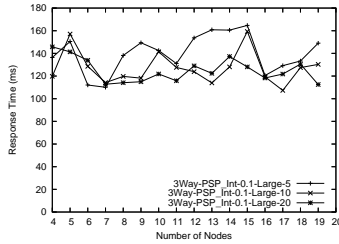
(b) PSP-Int, Small Join, State Size



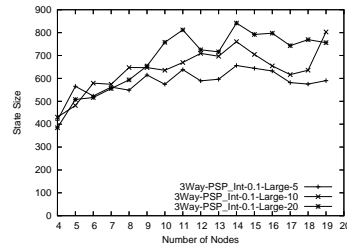
(c) PSP-Int, Middle Join, Response Time



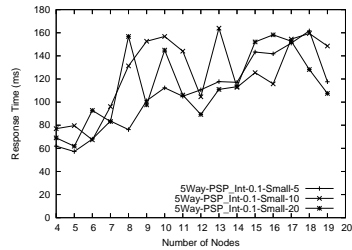
(d) PSP-Int, Middle Join, State Size



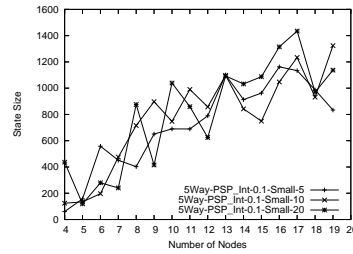
(e) PSP-Int, Large Join, Response Time



(f) PSP-Int, Large Join, State Size



(g) PSP-Int, Small Join, Response Time



(h) PSP-Int, Small Join, State Size

Figure 16.3: Performance Analysis of the PSP-Int scheme

The sliding window size is set to be 10k ms for all the streams. The input rate is set to 50 tuples/sec per stream. In all the experiments, the system will run for 600 seconds.

Figure 16.3(a) to 16.3(h) show several of the experiment results for 3-way and 5-way join with different join selectivities and number of nodes. The legend reads as: Number_of_way-Join_scheme-Join_selectivity-Join_cost-Concurrent_Tuple_Num.

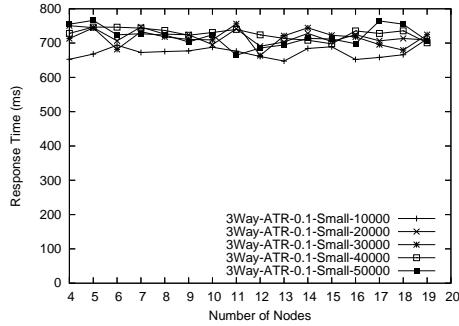
The PSP-Int scheme does have some duplicated states, thus the memory consumptions are more than the PSP corresponding among all the experiments. The query response latency is pretty stable since the PSP-Int allows more input tuples to be processed at the same time. Thus increasing the processing nodes will increase the currently processed tuples instead of reducing the response time of the joined results.

16.4 Experiment 3: PSP vs. ATR and CTR

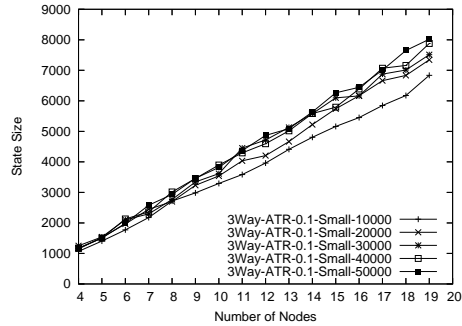
The next set of experiments compare the PSP scheme with the ATR and CTR solutions.

Figures 16.4(a) to 16.4(d) show several experimental results running the ATR approach. The legend reads as: Number_of_way-Join_scheme-Join_selectivity-Join_cost-Segment_length(ms).

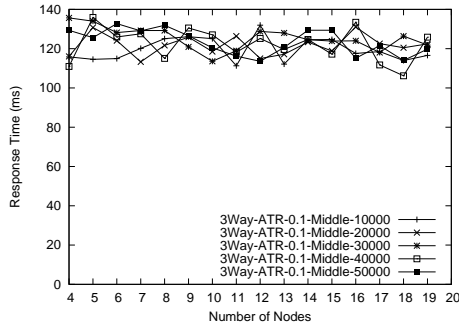
We vary the segment size for ATR from 10k ms, which is equal to the window size, to 50k ms. In ATR, the corresponding segments of the stream tuples are processed at each node. Thus all the workload to process single input stream tuple is done by single node. The result is that the average



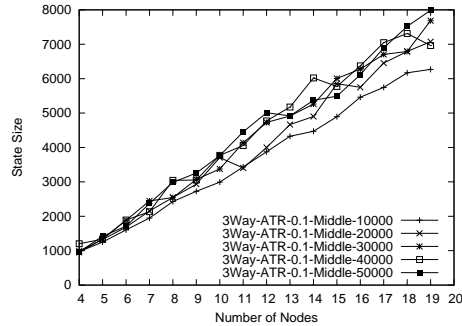
(a) ATR, Small Join, Response Time



(b) ATR, Small Join, State Size



(c) ATR, Middle Join, Response Time

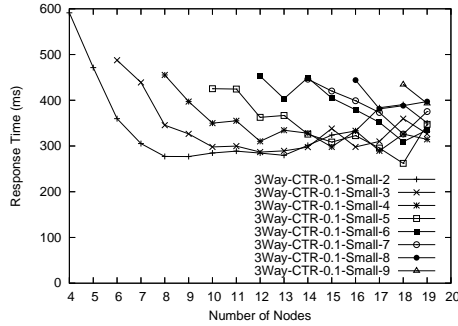


(d) ATR, Middle Join, State Size

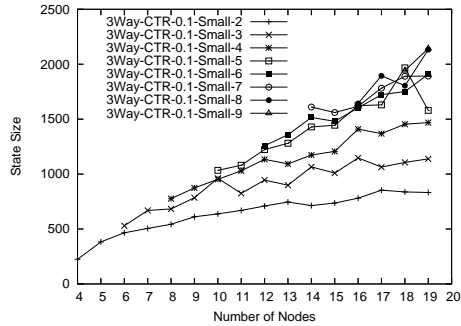
Figure 16.4: Performance Analysis of the ATR scheme

response time will not decrease by adding more nodes into the system. The state memory consumption for ATR is increasing steadily with the number of nodes in the system, since more duplicated segments are generated and stored in the states.

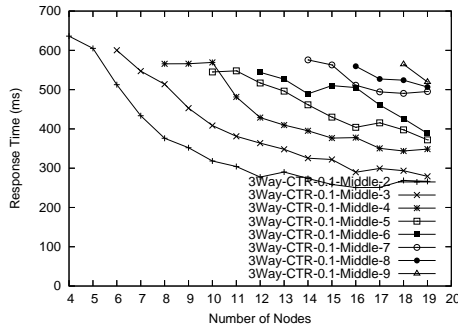
Figures 16.5(a) to 16.5(d) show several experimental results running the CTR approach. The legend reads as: Number_of_way-Join_scheme-Join_selectivity-Join_cost-Number_Copy. We enforce the number of copies in the CTR approach to stabilize the output of the minimal set cover algorithm. We limit the number of copies to at most 50% of the number of the



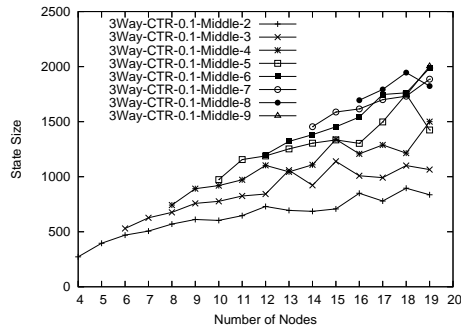
(a) CTR, Small Join, Response Time



(b) CTR, Small Join, State Size



(c) CTR, Middle Join, Response Time



(d) CTR, Middle Join, State Size

Figure 16.5: Performance Analysis of the CTR scheme

nodes running.

The response time is decreasing with the large number of nodes in the system. This result is consistent with the analysis in [GYW07] since the CTR is also a multi-hop join scheme. More nodes in the system will increase the CPU power and decrease the processing time. We also observe that for a fixed number of nodes, more copies of the states result in a larger response time. This can be explained by the minimal set cover algorithm. When more copies exist, less nodes will participate in processing this input stream tuple and the average response time increases accordingly. The state

memory usage remains rather stable when increasing the processing nodes. But it will increase when more copies is used.

Overall, the PSP model with optimal settings performs better than ATR and CTR schemes.

16.5 Experiment 4: Runtime Adaptation of PSP

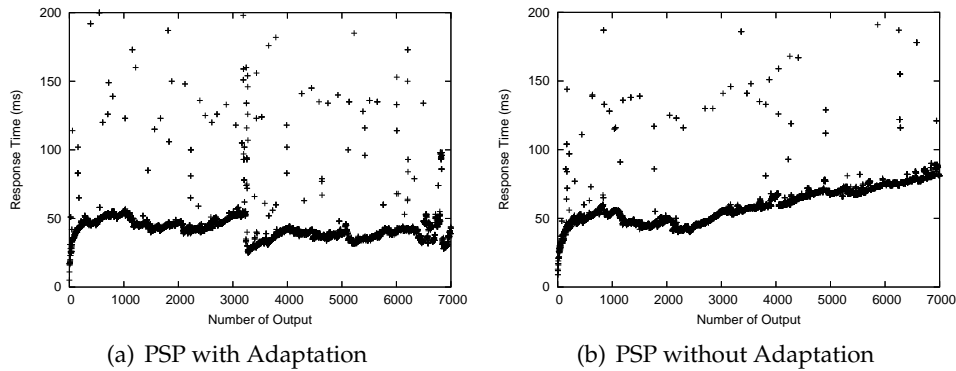
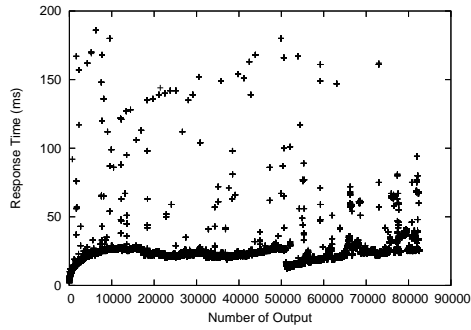


Figure 16.6: Experimental Results of Adaptation

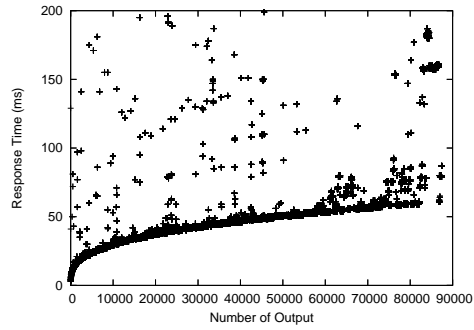
In the following experiments, we compare the response time when runtime adaptation is turned on and off in the PSP scheme. In the middle of processing, we set the arrival rate to increase from 50 tuples/sec to 100 tuples/sec. Figures 16.6(a) and 16.6(b) compare the performance of PSP under this change. The query used is a 3-way join, with the middle join cost and the join selectivity being 0.05. Clearly the runtime adaptation can make the system more stable and robust to environmental changes.

Figures 16.7(a) and 16.7(b) show the performance of a 3-way join, with small join cost, join selectivity 0.2 and the arrival rate increase from 50 tu-

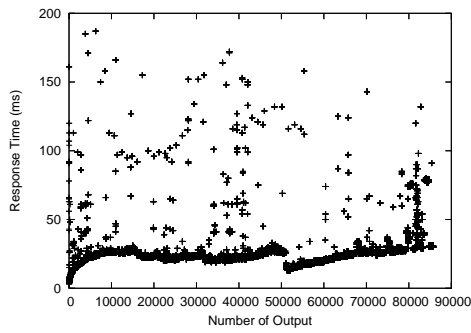
ples/sec to 100 tuples/sec. Figures 16.7(c) and 16.7(d) show the performance of the same query with similar parameters except that the arrival rate increases from 50 tuples/sec to 150 tuples/sec.



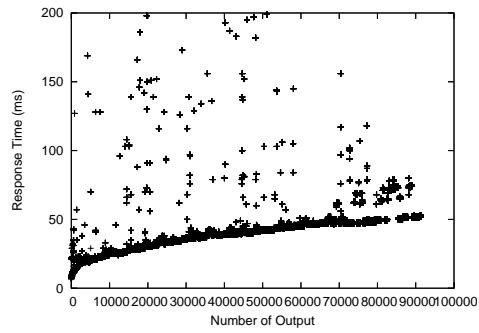
(a) PSP with Adaptation



(b) PSP without Adaptation



(c) PSP with Adaptation



(d) PSP without Adaptation

Figure 16.7: Experiments Results of Adaptation

Chapter 17

Related Work

Parallel and distributed query processing has been the focus of both academia and industry for a long time [DG92, GHK92, Kun00, Val93]. Two main categories of parallelism are employed in the literature: *pipelined parallelism* and *partitioned parallelism*. The proposed PSP scheme belongs to the pipelined parallelism. Superior to the traditional query plan based pipelining, the PSP scheme has the advantage of being able to employ an optimal length pipelining at the optimizer's will.

Distributed stream processing has been considered in recent years for distributed Eddies [TD03], Borealis [Ac04, ABcea05] and System S [JAA⁺06] and DCAPE [LZJ⁺05]. For distributed processing of stateful stream queries, state partitioning [SHea03] has been proposed. State partitioning has the major limitation of only supporting equi-joins, while our target is generic joins. For generic joins, duplicated data partitions in multiple machines are required for hash-based solution.

For distributed processing of generic joins with arbitrary join predi-

cates, a recent project [GYW07] has proposed two state replication based approaches. As indicated in Section 15, such state duplication may abuse large amounts of memory resources, possibly also causing increased data shipping and processing costs. Section 15 provides a detailed comparison between our PSP scheme and their solutions.

There are several existing works for finding optimal join orderings for multi-way join operators [VNB03, BMM⁺04]. Our PSP scheme is clearly orthogonal to this issue. The optimal join orderings identified by such algorithms can thus be directly utilized for processing in our proposed PSP distribution schemes.

Load-shedding [TZ⁺03] and approximated query processing [SW04] and spilling data to disk [UF00] are alternate solutions for tackling continuous query processing with insufficient resources. Approximated query processing [SW04] is another general direction for handling such situations. Different from these, we aim to guarantee accurate high-performance processing and thus focus on distributed processing in a cluster. Those works are clearly orthogonal to our work, and can be applied on our solution if the total computation resources of the cluster are found to be insufficient.

Part III

Distributed Multiple Multi-way Join Query Optimization

Chapter 18

Introduction

In Parts I and II, we have discussed the sharing among different queries and distributed multi-way join query processing. In the third part of the dissertation work, we will integrate these two solutions to tackle the problem of multiple query optimization in a distributed system. The common state slicing concept behind these two parts makes the seamless integration possible.

18.1 Research Motivation

Stream applications may issue queries on the same set of input streams, with various window constraints and selection predicates on the streams. Sharing the computation of multiple multi-way stream joins can potentially save huge memory and CPU resources.

Although the shared multi-way join query plan reduces the memory and CPU consumption to answer all the stream queries in the given query

set, the requirements may still be beyond the resource limitations in a single processing node. Distributed processing of the shared query plan then is necessary. It can improve the scalability of the stream engine, in terms of stream arrival rates, number of input streams and size of the given query workload.

18.2 Proposed Strategies

To share the multi-way join operators in a distributed system, we propose to use a two-phase approach. We first form a logical shared query plan for a given query workload and then deploy it in the cluster of processing nodes.

Given a set of continuous queries over the same set of input streams, a logical shared query plan is formed in the first phase. To form the shared query plan, the selections on the streams in each individual query plan are pulled up first to make the multi-way joins “sharable” in the sense that all the join operators share exactly the same join input streams and join conditions, but may have distinct window constraints. Then the states of each input stream are sliced according to the window constraints of the stream queries in the workload. Next the selections are pushed into the ring of sliced joins to stipulate that every joined result generated is used to answer at least one query in the query workload. Lastly, one routing operator is employed to dispatch the joined results to each individual stream query application.

After forming the logical shared query plan, it can be deployed phys-

ically in the cluster. Workload balancing at each processing node is the optimization goal of the deployment. To achieve this, further state slicing may be needed to break the state sliced joins that have large states into smaller ones such that each can be hosted in one processing node. Also, a physical node may host multiple small state sliced joins of the logical query plan. Each processing node also has a copy of the routing operator in the logical plan to dispatch the joined results generated in this processing node to each application.

The logical and physical state slicing in the two-phase approach are orthogonal since the optimization goals of each phase are independent. When the logical query plan is built, the state slicing is conducted according to the query semantics, including the window constraints and the selections on the input streams. After pushing selections into the state sliced join ring, the query plan is to ensure that no unnecessary join probing will happen. In the physical state slicing phase, the logical plan is deployed in the cluster with consideration of workload balance and tradeoff between number of nodes and the transmission costs. The state slicing during plan deployment in the second phase does not affect the optimization goal of the first phase in the sense that the deployment does not change the join probing cost. Also the state slicing in the second phase is independent from the logical state slicing in the first phase in the sense that a processing node can hold multiple logical state sliced joins, determined by the second phase.

In part III, we tackle the following issues related to the distributed processing of multiple stream joins.

- *Pushing Selections into State Sliced Join Ring.* Pushing selection into the chain has been discussed in Chapter 7 for state sliced binary joins. For the same optimization reason, namely to minimize the join cost, we now want to push the selections into the ring for multi-way state sliced joins. Compared to the selection push down in Chapter 7, the new challenge is that we need to ensure all the intermediate results generated are necessary. That is, no extra join cost arises.
- *Adding Router to Dispatch the Joined Results.* Unlike in the binary state sliced join chain, the joined results generated by every processing node may serve multiple stream queries. Thus the joins down the chain will serve less and less queries and the last one only serves one query. The reason is that in the ring the intermediate results will be propagated along the ring and probe the corresponding states in all the operators. The routing of the joined result will be much more complex than the binary counterpart in the sense of routing logic. A fast routing strategy is thus critical.
- *Deploying the Logical Query Plan.* A new cost model, considering the selections, needs to be developed in a cluster to optimally deploy the logical plan, in terms of response time and workload balancing.

In this part, we assume that the joined queries share the same join orderings among the input streams. Otherwise no sharing is possible since the multi-way join does not reserve intermediate results in the states. Multiple query sharing aware join order optimization, as a promising future work, is beyond the scope of this dissertation. We also assume the queries

share the same join conditions, since otherwise no join probing costs can be shared even when the states memory may.

18.3 Road Map

The rest of this part is organized as follows. Chapter 19 presents the selection push down approach in the multi-way state sliced join ring. Chapter 20 illustrates the routing strategy using bitmaps to dispatch the joined results. Chapter 21 presents the cost based logical plan allocation for deployment in the cluster.

Chapter 19

Selection Pushdown for Multi-way Join

In this chapter, we discuss how to push the selections into the ring to build logical shared query plan with a ring of state sliced joins. For ease of illustration, we limit our discussion to the case that all the stream queries join the same set of input streams on the same join conditions. They can have different selection predicates on the input streams and may use different window constraints. We also assume the same join orderings are employed. In this chapter, we discuss how to push selections into the ring, while in the next chapter the routing strategy is described to dispatch the joined results to each query.

19.1 Selection Pull Up and Window based State Slicing

Consider a workload of N stream queries registered in the DSMS, where each query performs a sliding window join on data streams S^1, S^2, \dots, S^m with possibly different window constraints on each input stream. Each stream query may also have selections on each input stream.

Without loss of generality, we focus on one of the input streams, S^j , and the selections on stream S^j . Given a set of continuous queries, the queries are sorted by their window lengths on stream S^j in ascending order, denoted as q_1, q_2, \dots, q_n . That is, $W_{i-1} < W_i$ holds where W_{i-1} denotes the window size of q_{i-1} on stream S^j and W_i denotes the window size of q_i on S^j . Here we only discuss the cases that all the windows are distinct from one another, since the case of same window constraints is trivial. Note that for input streams other than S^j , the order may be different. We also denote the corresponding selection of q_i on Stream S^j as σ_i .

To share the join computation involving the stream S^j , we first pull the selections up in the query plan for each query q_i , according to the following equation.

$$\dots \bowtie \sigma_i(S^j) \bowtie \dots = \sigma_i(\dots \bowtie S^j \bowtie \dots)$$

The next step is to slice the state of stream S^j according to the window constraints, with increasing window lengths. Thus, we have a sequence of sliced states $S^j[W_0, W_1], S^j[W_1, W_2], \dots, S^j[W_{n-1}, W_n]$, where $W_0 = 0$ and

W_i denotes the window size of query q_i over stream S^j .

The same process can be conducted for the other input streams. Eventually we will have a set of sequences of sliced states for each input stream.

Since the logical window-based state-slicing for multi-way join without the selections corresponds to actually a PSP query plan deployed on one single node, so the correctness of this slicing is stipulated by Theorem 8.

Following Theorem 5, we have below theorem:

Theorem 10 *The total state memory used in the states $S^j[W_0, W_1]$, $S^j[W_1, W_2]$, ..., $S^j[W_{n-1}, W_n]$ is equal to the state memory used for stream S^j in the multi-way join $S^1 \bowtie S^2 \bowtie \dots \bowtie S^j \bowtie \dots \bowtie S^m$, where $W_0 = 0$ and W_i denotes the window size of query q_i over stream S^j .*

The proof is similar to the one for Theorem 5 and is not repeated here, since after selection pullup, the window-based multi-way join state-slicing is in fact an extended “Mem-Opt” state slicing.

19.2 Selection Push Down

Similar with selection push down into the chain discussed in Chapter 7, we can also push down the selections into the ring of sliced joins. We now propose the following theorem, which is an extended version of Theorem 4 for multi-way joins.

Theorem 11 *The select operator, which has predicates on stream attributes except the timestamps, can be pushed down into the ring without affecting the query semantics. That is, when the selection σ is pushed into the ring between adjacent*

sliced join J_i and J_{i+1} , the union of the join results of the m -way sliced window joins in a ring $\sigma(S^1[0, W_1] \bowtie \dots \bowtie S^m[0, W_1]), \dots, \sigma(S^1[W_{i-1}, W_i] \bowtie \dots \bowtie S^m[W_{i-1}, W_i]), \sigma, S^1[W_i, W_{i+1}] \bowtie \dots \bowtie S^m[W_i, W_{i+1}], \dots, S^1[W_{N-1}, W_N] \bowtie \dots \bowtie S^m[W_{N-1}, W_N]$ is equivalent to the results of a regular sliding window join $\sigma(S^1[W] \bowtie \dots \bowtie S^m[W])$, where $W = W_N$.

The proof of Theorem 11 is similar to the proof of Theorem 4, since the σ operator between J_i and J_{i+1} will suppress down-stream select operators. All the down-stream select operators thus can be safely removed.

Consider the input stream S^j , let us assume the queries in the workload are sorted by their window lengths on stream S^j in ascending order, denoted as q_1, q_2, \dots, q_n . Let us also denote the corresponding selections of q_i over stream S^j as σ_i^j . Reusing the techniques discussed in Chapter 7, the selections can be pushed down into the ring between adjacent state sliced joins. We denote the selection on stream S^j before sliced join J_i as $\widehat{\sigma}_i^j$. The predicate of the selection $\widehat{\sigma}_i^j$ corresponding to the disjunction of the selection predicates from σ_i^j to σ_n^j is denoted as:

$$\widehat{\sigma}_i^j = \sigma_i^j \vee \sigma_{i+1}^j \vee \dots \vee \sigma_n^j$$

Obviously the predicates of $\widehat{\sigma}_i^j$ overlap partially with $\widehat{\sigma}_{i+1}^j$ and so on. In implementation, each stream tuple is marked to show which σ_i^j has already been evaluated to avoid re-evaluation of the same predicates multiple times.

Similar to Theorem 6, we have the following theorem.

Theorem 12 *The state slicing sharing with selection push-down consumes the*

minimal state memory for the multi-way joins in a given workload.

The proof is straightforward since: (1) at any time, the contents in the state memory of all sliced joins are pairwise disjoint with each other. Thus no duplication exists in the states; and (2) any state tuple is needed to answer at least one query in the workload.

We have only discussed the selections of the input stream tuples above. Next, we now show that the intermediate results are also subject to selections to avoid unnecessary join costs.

To illustrate the necessity of filtering the intermediate results, we use the following example 3-way join. Assume query q_1 is $A[w_1^A] \bowtie B[w_1^B] \bowtie C[w_1^C]$ and q_2 is $A[w_2^A] \bowtie B[w_2^B] \bowtie C[w_2^C]$. We also assume $w_1^B < w_2^B$ but $w_1^C > w_2^C$. That is, the window size comparisons of q_1 and q_2 are just opposite on streams B and C . According to the state slicing for multi-way joins, the state of stream B will be sliced into two parts, denoted as $B[0, w_1^B]$ and $B[w_1^B, w_2^B]$. Similarly the C state is sliced as $C[0, w_2^C]$ and $C[w_2^C, w_1^C]$. Assume the incoming a tuple from stream A will probe B states first and then C states. We can see that the intermediate result $a \bowtie B[w_1^B, w_2^B]$ only needs to join with the first sliced state of C : $C[0, w_2^C]$. The join probing of $a \bowtie B[w_1^B, w_2^B] \bowtie C[w_2^C, w_1^C]$ should be avoided since it will not serve any of the queries. This situation also exists in case of selection push down for the input streams.

Without loss of generality, consider the input stream S^j . Let us assume the queries in the workload are sorted by their window lengths on stream S^j in ascending order, denoted as q_1, q_2, \dots, q_n . Assuming the correspond-

ing sliced states of S^j are denoted as $S^j[0, w_1^j]$, $S^j[w_1^j, w_2^j]$, ..., $S^j[w_{n-1}^j, w_n^j]$.

After the selection push down into the ring, we have:

Lemma 9 *The intermediate results (or final results) generated from the probing of the sliced state $S^j[w_{i-1}^j, w_i^j]$ is used to serve the queries q_i, q_{i+1}, \dots, q_n .*

Proof: According to the way that the ring is formed, we know the window sizes of the queries q_1, q_2, \dots, q_n are increasing monotonously. Thus Lemma 9 holds since the tuples in state $S^j[w_{i-1}^j, w_i^j]$ are inside the windows of the queries q_i, q_{i+1}, \dots, q_n . ■

Since the intermediate results can be generated in any of the state sliced joins in the ring and only are propagated forward along the ring, we can not filter the intermediate results out before they have been propagated for one complete round along the ring. That is, all the intermediate results are going to be transmitted along the ring for one round, just like in the basic PSP scheme. For this, we change the step 3-2 in Figure 12.3 to check the timestamps before doing the join probing. For state sliced join J_i , if the timestamps of the incoming intermediate result satisfies any of the queries from q_i to q_n , then the intermediate result is used to probe the state $S^j[w_{i-1}^j, w_i^j]$ in J_i . Otherwise the probing is omitted.

Theorem 13 *The state-slicing sharing with selection push-down and filtering of intermediate results before probing consumes the minimal join probing cost for the multi-way joins in a given workload.*

Proof: Proof by induction. Assume the join ordering is $S^1 \rightarrow S^2 \rightarrow \dots \rightarrow S^m$. Let's consider the processing of the arrival tuple s^1 from stream S^1 .

Basis: With the selection push-down into the ring for the input stream tuples, all the intermediate results generated by s^1 probing the states of S^2 at J_i satisfy at least one of the queries q_i, q_{i+1}, \dots, q_n . No extra unnecessary intermediate result is generated.

Induction: Assume the intermediate results probing states of S^j satisfy at least one of the queries in the workload. Then with the filtering of intermediate results before probing, the new generated intermediate results (or final joined results) at $J_{i'}$ satisfy at least one of the queries $q_{i'}, q_{i'+1}, \dots, q_n$. No extra unnecessary intermediate result is generated.

■

Chapter 20

Routing the Joined Results

In this chapter, we discuss how to route the joined results to output them to serve the different queries. From the previous chapter, we already know that each state-sliced multi-way join operator can produce joined results for possibly many queries in the workload. The reason is that in the ring the intermediate result will be propagated along the ring and will probe the corresponding states in all the operators. The routing of the joined results will be much more complex than the binary counterpart. A fast routing strategy is thus critical.

In this chapter, we assume that the joined queries share the same join orderings among the same set of input streams. Without loss of generality, we also assume each query in the workload has distinct selection predicates and window constraints on the input streams.

20.1 Routing Bitmaps for the Logical Window Slices

Instead of using a large query plan to dispatch the joined results to each of the queries, we use a routing operator to achieve fast dispatch of the joined results.

We use the following state sliced 3-way joins with ring length equals to 3 for serving a workload of three queries as the example to illustrate our bitmap routing approach. Note that the ring length here is equal to the number of queries in the workload since all the window constraints are distinct.

For ease of illustration, we ignore the selections in this section and will discuss the selections in the next section.

Example: Assume the 3-way join is $A \bowtie B \bowtie C$ and the join ordering for the arrival tuple from stream A is $A \rightarrow B \rightarrow C$. We assume that the states of the input streams are sliced into three slices each and denote them as: A_1, A_2, A_3 , and so on for input streams B and C . Accordingly, we define the state IDs for each of the input streams. For example, the three sliced states of stream A are assigned IDs as: 001 for A_1 , 010 for A_2 , and 100 for A_3 .

Similar as Chapter 19, given a set of continuous queries q_1, q_2, q_3 , the queries are first sorted by their window lengths on streams A, B, C individually in ascending order. Without loss of generality, we assume the order is: q_2, q_3, q_1 for stream A , q_1, q_2, q_3 for stream B , and q_3, q_1, q_2 for stream C . We also define IDs for each of the queries in the workload as 001 for q_1 , 010 for q_2 , and 100 for q_3 . In the state sliced operator, it holds a matrix O to record the query orders on each of the input streams. In this example, we

have:

$$O = \begin{pmatrix} 010 & 001 & 100 \\ 100 & 010 & 001 \\ 001 & 100 & 010 \end{pmatrix}$$

The first column denotes the ordering of queries on the window sizes of stream A , the second column for stream B , and the third column for stream C .

We attach a bitmap onto each of the joined results to identify the state slices generating it. To do this, each sliced join operator needs to incorporate the state IDs to the joined intermediate result or final result during join probings. For example, we have a joined result generated by the probing of an incoming tuple from stream A against one tuple in B_2 and another one in C_3 . Then the bitmap for this joined tuple is set to be a vector T as defined by:

$$T = (111, 110, 010)$$

The first item T_1 for stream A is set to be 111 since the joined result is generated from an incoming tuple from stream A instead of a state tuple. The second item for stream B is set to be 110 since according to Lemma 9, the B_2 state slice is used to serve q_2, q_3 for our assumed order of window sizes on stream B . That is $T_2 = O_{2,2} \vee O_{3,2} = 010 \vee 100$. The third item for stream C is set to be 010 since the C_3 is used to serve q_2 only.

The routing operator now can use the vector T to identify the queries to

send the joined results by using bitwise boolean multiplication as follows:

$$T_1 \cdot T_2 \cdot T_3 = 111 \cdot 110 \cdot 010 = 010$$

The multiplication result 010 means only q_2 is the target query to send this joined tuple to. Note the multiple queries may be the targets. For example if the result is 101 then this means both q_1 and q_3 are the target queries.

The correctness of the routing strategy is stipulated by Lemma 9.

20.2 Bitmaps for Evaluation of the Selections

From Chapter 19, we see that every joined result tuple is guaranteed to serve at least one query in the workload after the pushdown of the selections on the input streams and intermediate results. If the routing strategy discussed in the above section ends up with one target query to send the result to, then no more check of the selection predicates is necessary.

However if multiple queries are targets from the routing strategy, then the extra check of the selection predicates is needed to guarantee the correctness. To do this, we reuse the bitmap for the evaluation of selection $\widehat{\sigma}_i^j$. That is, check the vector that recording the evaluation of σ_i^j of stream S^j to see if the selection predicate of the target query on stream S^j has been evaluated. If not, the tuple is subject to evaluation of this predicate before being sent out to the target query.

Chapter 21

Logical Query Plan

Deployment in the Cluster

After we have the logical shared query plan for multi-way joins with selections pushdown and addition of the routing operator, the logical query can be deployed in a cluster reusing the PSP scheme discussed in Part II of this dissertation. During the deployment, each processing node will have a copy of the routing operator to dispatch the joined results generated in this node to the target queries. The routing operators are identical in logic across all processing nodes. After deployment, each processing node may hold multiple logical state sliced join operators or slice of the logical state sliced join according to the cost model discussed here.

In Chapter 13, we have developed a cost model for the PSP scheme for the distributed processing of multi-way joins in a cluster. Here we extend the cost model to incorporate the case of deployment of the logical state

sliced query plan.

21.1 Extended Cost Model

We here reuse the parameters in Table 13.1 for the cost model with the addition of parameters representing the selectivities of the pushdown selections $\hat{\sigma}_i$, denoted as $S_{\hat{\sigma}_i}$.

We again assume that the network bandwidth is sufficient for our workload. The network latency then is proportional to the number of hops of the transmission. We assume the sending and receiving latency between processing nodes to be proportional to the number of tuples transmitted.

We first calculate the processing workload L_C^{share} for the join processing of one input tuple in the logical query plan and the workload L_{PSP}^{share} for the processing of the same input tuple in the PSP scheme deployment of the logical plan. Same as in Chapter 13, we assume that an in-memory nested loop join algorithm is employed. We also assume the optimal join ordering for the input tuple from stream S^1 is: $S^1 - > S^2 - > \dots - > S^m$ and the processing nodes and the network connections between them are homogeneous.

$$\begin{aligned}
 L_C^{share} &= T_i + T_p + T_j \sum_{2 \leq k \leq M} \prod_{1 \leq i \leq k-1} \lambda_i W_i S_{\bowtie i} S_{\hat{\sigma}_i} \\
 L_{PSP}^{share} &= L_C + T_s N (1 + \sum_{2 \leq k \leq M-1} \prod_{1 \leq i \leq k-1} \lambda_i W_i S_{\bowtie i} S_{\hat{\sigma}_i})
 \end{aligned} \tag{21.1}$$

21.2 Minimize Average Response Latency

To estimate the processing latency of the deployment, we consider the average latency for join results from one input tuple. The discussion in Chapter 13 is still valid for the deployment of the shared query plan. We repeat the formula below. The processing latency τ_i for node $i, 1 \leq i \leq N$ is:

$$\tau_i = (i - 1)T_n + \max\left\{\frac{L_{PSP}^{share}}{N}, MNT_n\right\} \quad (21.2)$$

There is no closed form for the average processing latency τ .

From Equation 21.2, we see the response latency is sensitive to the number of the processing nodes N . Intuitively, adding more processing nodes increases the CPU power. On the other hand, the longer the length of the ring the higher the network transmission cost.

Same as in Chapter 13, we have that the processing latency for each node is decreasing with larger N , while the network latency is increasing. Both facts need to be considered for the optimal ring length with minimal processing latency.

21.3 Workload Balancing

Since the PSP deployment is a pipelined execution model, workload balance must be achieved for optimal performance to avoid bottleneck node.

From Equation 21.1, the dominant CPU cost for each node is the join probing cost, which is proportional to the total size of the sliced states in the node. To balance the workload of each node, we keep the number of

state tuples balanced in every node, with consideration of the selectivities of the pushdown selections in the ring.

Part IV

Conclusions and Future Work

Chapter 22

Conclusions of This Dissertation

Query optimization is one of the most critical techniques for improving query performance in any database system. Among these techniques the optimization of continuous join queries, especially for the multi-way joins with arbitrary join graphs, is essential since stateful join operations tend to dominate the CPU and memory usage in a database system. For stream query optimization, the real-time query response requirement and in-memory processing of stream operators exacerbate the situation.

In this dissertation, I propose a novel solution of slicing the states in the time domain called *state slicing*, designed to split a huge stateful operators into a group of smaller stateful operators at the optimizer's will. Our proposed method is generic in the sense that the key idea of state slicing does not rely on the query semantics such as the type of predicates, attribute do-

main and attribute distribution. Our solution is versatile and generic for arbitrary join predicates with minimal extra cost. Based on the state slicing concept, we show solutions of two important problems, namely, computation sharing among multiple stream queries with overlapping window constraints and distributed query processing of generic stateful join queries.

In the first part of this dissertation, we focus on the problem of sharing window join operators across multiple continuous queries. The window constraints may vary according to the semantics of each query. In order to efficiently share computations of window-based join operators, we propose a new paradigm for sharing join queries with different window constraints and filters. The two key ideas of the approach are *state-slicing* and *pipelining*. The window states of the shared join operator are sliced into fine-grained pieces based on the window constraints of individual queries. Multiple sliced window join operators, with each joining a distinct pair of sliced window states, can be formed. Selections now can be pushed down between the appropriate sliced window joins to avoid unnecessary computation and memory usage. Based on the state-slice sharing paradigm, two algorithms are proposed for the chain buildup, one that minimizes the memory consumption and the other that minimizes the CPU usage. The algorithms are guaranteed to always find the optimal chain with respect to their targeted resource of either minimizing memory or CPU costs, for a given query workload. Chains in the “middle” can also be built considering tradeoffs between the system memory consumption and CPU usage. The experimental results show that our strategy achieves respected opti-

mization goals for memory or CPU costs over a diverse range of workload settings among alternate solutions in the literature. The proposed techniques are implemented in an actual DSMS (*CAPE*). Results of performance comparison of our proposed techniques with state-of-the-art sharing strategies are reported. Our solution has been shown to be more efficient than other sharing strategies for various workloads of stream queries.

In the second part of this dissertation, we focus on distributed processing of generic MJs with arbitrary join predicates, especially of multi-way joins with large window constraints. Generic stream joins occur in many practical situations, from simple range (or band) join queries to complicated scientific queries with equation-based predicates. Such join operators tend to be complex and CPU intensive. Our goal is to minimize the query response time to meet the real-time response requirement of the stream applications. A novel MJ operator distribution scheme called Pipelined State Partitioning (PSP) is proposed in this part of the dissertation. We propose a novel solution to separate a macro MJ operator into a series of smaller state-sliced MJ operators. Different from value-based partitioning, the PSP scheme is join predicate agnostic and thus general. Beyond this basic PSP scheme, we design two extensions. One, PSP-I (with I for Interleaving) introduces a delayed purging technique for the states to enable interleaved processing of multiple stream tuples with asynchronous processor coordination. Such interleaved processing is used to avoid idle processors which exist in the synchronized basic PSP scheme. Two, beyond interleaved processing, PSP-D (with D for Dynamic) further incorporates a dynamic state ring structure to avoid repeated maintenance cost of sliced states, which

comes from the standard tuple insertion and state purging routines. A cost model is developed to achieve the optimal state slicing and allocation, in terms of query response latency. The tradeoff between employing more processing nodes and having more transmission hops is considered. Runtime adaptive state relocation are also employed for achieving load balancing and re-optimization in a fluctuating environment by smoothing the sliced state size and adding/removing processing nodes dynamically. We have implemented the proposed PSP scheme within the *D-CAPE* DSMS. A series of experimental studies are conducted to illustrate the performance of the PSP scheme (in term of response time and state memory usage) under various workloads. The experimental results show that our strategy provides significant performance improvements under diverse workload settings.

In Part I and Part II we have discussed the state slicing based binary stream join query sharing and distributed multi-way join query processing. In the third part of dissertation work, we integrate these two solutions to tackle the problem of multiple query optimization in a distributed system. The common state slicing concept behind these two parts makes the seamless integration possible. We propose a two-phase query plan generation to share the computation of multiple multi-way stream joins in a cluster. In the first phase, the selections are pushed into the ring and the state sliced joins based on the selection predicates are formed. In the second phase, the ring query plan is deployed in the processing nodes with consideration of balanced workload in each node. To achieve a balanced workload, the state sliced joins generated in the first phase may be further sliced. Also

one processing node may host multiple state sliced joins together with the selections between them. A cost based deployment is used to achieve the balanced workload. To achieve fast routing of the joined results, we propose a bitmap based routing strategy. Since the number of distinct sub-joins between sliced states may be huge for multi-way join sharing, we use one routing operator to dispatch all the joined results instead of using one routing operator for the joined results from each sub-joins. Based on the bitmap in the joined result, it can be routed to the corresponding query user.

Chapter 23

Future Work

23.1 State Slicing Aware Continuous Query Optimization

So far all our discussions in this dissertation separate continuous query plan optimization and state slicing techniques. That is, we assume these two kinds of optimizations are independent of each other. For example, we assume throughout this work that the join ordering optimization has been finished before considering the state slicing optimization.

Intuitively, these two kinds of optimizations may relate to each other in some cases. For example, when we consider the sharing of the multi-way join queries, the optimizer for join ordering may pick different candidate join orderings in case of the sharing.

However, considering both kinds of optimizations together will greatly enlarge the search space for the optimal solutions in general. Thus it will be

a challenging optimization task. New heuristics may be needed to obtain a sub-optimal solution in practice.

Beyond the join orderings, in the case of multi-way join query sharing, we also need to consider all the possible tree shapes with consideration of state sliced sharing. This is also a hard optimization task.

23.2 Computation Sharing for Complex Event Query Processing

Recently the emergence of stream data processing has been extended to complex event processing on event streams. The early work in this direction includes the Berkeley HiFi project [RJK⁺05, WDR06], Siemens RFID middle-ware [WL05, WLLB06] and Cornell expressive publish/subscribe system (Cayuga) [DGH⁺06]. This research is generally called *Complex Event Processing*(CEP) on event streams.

A CEP system may need to process multiple sequential event patterns with different window constraints. To reduce the memory and CPU consumptions, it is natural to extend the *State Slice* concept to the NFA-based PathStack evaluation [WDR06]. The purpose here is to push down the selections as deep as possible into the automaton.

Similar to the state slice join in Part I, the selection operator can be pushed down between the PathStacks. The execution of the selection operator and the processing of new incoming events from the streams can be scheduled arbitrarily. That is, it is not necessary that the input queue of the select operator must be empty all the time. However, before the sequence

construction the selection operator between the PathStacks must be scheduled until the input queue is consumed. That is, for lazy evaluation, the sequence construction is the only time that the automaton and the selection operator need to be synchronized.

In multiple event patterns, common sub-sequential patterns often exist. How to share the computations among the common sub-sequences is a new issue for multiple event query optimization. The sharing of common sub-event sequences can be divided into two categories, the sharing of prefix sub sequences and sharing of suffix sub sequences.

Sharing of the computation of the common prefix can be achieved using a cache, assuming that the memory is sufficient. The content of the cache is the enumeration of the event sequences according to the common prefix pattern. The cache is inserted when a new enumeration is invoked and is deleted when the event instances expire.

Since the nature of the lazy evaluation, the multiple event patterns with a common suffix can be constructed at the same time and no catch is necessary for the sharing of the computation. In order to achieve simultaneously sequence construction, a mix typed stack is employed in PathStack.

23.3 Approximate Continuous Query Processing

Providing query answers to the end user with low latency is always desired, even with approximated answers. For complex continuous queries, a fast response time might be more important than a precise answer, given that the continuous query results are always changing. To catch up with

the stream speed, approximate processing must be light weighted and can be improved for accuracy with extra work during off-peak time.

Load-shedding [TZ⁺03] and approximated query processing [SW04] are both general directions for handling system overflow. These ideas can be applied to our solution as well whenever the available memory and CPU resources are insufficient even after applying our state-slice sharing optimization for multi-queries.

Overload Detection. The system overload can happen when (1) the main memory is not large enough to hold the state of the operators; or (2) the processing requirements for the operators exceed the capacity of the CPU power. The detection of these two kinds of overload is different.

To detect the state memory overload, we need to monitor the average input rate of the streams over the duration of the maximum window constraint. When the CPU is overloaded in a DSMS, the tuples waiting for the processor will accumulate in the queues and the total memory used for the queues will increase indefinitely.

To detect CPU overload, we generally set an upper bound for the total queue memory in the system. Whenever the total queue size is beyond the threshold, we say the DSMS is CPU overloaded.

Selective Dropping of the Workload. As pointed out in [TZ⁺03], insertion of drop boxes into the query plan is an effective solution to shrink the workload and thus solve the problem of overload. For using drop boxes in a shared query plan for multi-queries, we need to consider the following

additional issues beyond those in [TZ⁺03]:

- **Combined Loss/Gain Ratios for Multi-queries.** In a shared state-sliced query plan, the insertion of drop boxes needs to minimize the total loss for all the concurrently running queries in the DSMS.
- **Interaction between State Split/Merge and Insertion of Drop Boxes.** State split and merge may change the state memory and CPU consumption. How to achieve the optimal location for the insertion of drop boxes with the consideration of state split and merge is a challenging problem.

Semantic Load Shedding. Different queries may be interested more in one part of the query result than the other parts. One possible example is that a query may be more interested in the joined result of streams A and B when the difference of the timestamps is small. We call such a user-specified interest over window constraints I-QoS. By utilizing the I-QoS of each query, we can semantically shrink the workload in a DSMS.

The basic idea of such semantic load shedding depends on the amount of probing skip in the sliced chain of join operators. The probing step in the victim sliced join operator will be shortcut. Thus the processing time of the victim join operator is largely reduced, since the probing cost is the main cost for a join operator.

Note that the semantic load shedding cannot reduce the state memory usage, unless the victim sliced join operator is the last one in the chain.

Bibliography

- [AAB⁺05a] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Çetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stanley B. Zdonik. The design of the borealis stream processing engine. In *CIDR*, pages 277–289, 2005.
- [AAB⁺05b] Mohamed H. Ali, Walid G. Aref, Raja Bose, Ahmed K. Elmagarmid, Abdelsalam Helal, Ibrahim Kamel, and Mohamed F. Mokbel. NILE-PDT: A phenomenon detection and tracking framework for data stream management systems. In *VLDB*, pages 1295–1298, 2005.
- [ABcea05] Yanif Ahmad, Bradley Berg, Ugur Çetintemel, and et. al. Distributed operation in the borealis stream processing engine. In *SIGMOD*, pages 882–884, 2005.
- [ABW06] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The cql continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, 2006.
- [Ac04] Yanif Ahmad and Ugur Çetintemel. Networked query processing for distributed stream-based applications. In *VLDB*, pages 456–467, 2004.
- [ACC⁺03] D. Abadi, 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*, 12(2):120–139, August 2003.
- [AH00] Ron Avnur and Joseph M. Hellerstein. Eddies: continuously adaptive query processing. In *Proceedings of the 2000 ACM SIG-*

- MOD 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.
- [Ata99] Mikhail J. Atallah. *Algorithms and theory of computation handbook*, 1999.
- [AW04] A. Arasu and J. Widom. Resource sharing in continuous sliding-window aggregates. In *VLDB*, pages 336–347, Aug/Sep 2004.
- [BBD⁺02] 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.
- [BBDW05] Pedro Bizarro, Shivnath Babu, David DeWitt, and Jennifer Widom. Content-based routing: Different plans for different data. In *VLDB*, pages 757–768, 2005.
- [BBMW02] B. Babcock, S. Babu, R. Motwani, and J. Widom. Models and issues in data streams. In *PODS*, pages 1–16, June 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.
- [BMM⁺04] Shivnath Babu, Rajeev Motwani, Kamesh Munagala, Itaru Nishizawa, and Jennifer Widom. Adaptive ordering of pipelined stream filters. In *SIGMOD*, pages 407–418, 2004.
- [BMWM05] Shivnath Babu, Kamesh Munagala, Jennifer Widom, and Rajeev Motwani. Adaptive caching for continuous queries. In *ICDE*, pages 118–129, 2005.
- [BPSM97] Editors: T. Bray, J. Paoli, and C.M. Sperberg-McQueen. Extensible Markup Language (XML), 1997. <http://www.w3.org/TR/PR-xml-971208>.

- [BW01] S. Babu and J. Widom. Continuous queries over data streams. In *ACM SIGMOD*, Sep 2001.
- [CA93] R. G. G. Cattell and T. Atwood, editors. *The Object Database Standard, ODMG-93*. M. Kaufmann, 1993.
- [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 *VLDB*, pages 215–226, August 2002.
- [CCea03] D. Carney, U. Cetintemel, and A. Rasin et al. Operator scheduling in a data stream manager. In *VLDB*, pages 838–849, 2003.
- [CDN02] Jianjun Chen, David J. DeWitt, and Jeffrey F. Naughton. Design and evaluation of alternative selection placement strategies in optimizing continuous queries. In *ICDE*, pages 345–356, 2002.
- [CF02] S. Chandrasekaran and M. Franklin. Streaming queries over streaming data. In *VLDB*, pages 203–214, August 2002.
- [Cha98] Surajit Chaudhuri. An overview of query optimization in relational systems. In *PODS*, pages 34–43, 1998.
- [DBBM03] M. Dalar, B. Babcock, S. Babu, and R. Motwani. Chain: Operator scheduling for memory minimization in stream systems. In *Proceedings of ACM-SIGMOD*, pages 253–264, 2003.
- [DG92] David DeWitt and Jim Gray. Parallel database systems: the future of high performance database systems. *Communications of the ACM*, 35(6):85–98, 1992.
- [DGH⁺06] Alan J. Demers, Johannes Gehrke, Mingsheng Hong, Mirek Riedewald, and Walker M. White. Towards expressive publish/subscribe systems. In *EDBT*, pages 627–644, 2006.
- [Dij59] Edsger. W. Dijkstra. A note on two problems in connexion with graphs. In *Numerische Mathematik*, volume 1, pages 269–271. 1959.
- [DMRH04] L. Ding, N. Mehta, E. A. Rundensteiner, and G. T. Heineman. Joining punctuated streams. In *EDBT Conference*, pages 587–604, March 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.
- [GHK92] Sumit Ganguly, Waqar Hasan, and Ravi Krishnamurthy. Query optimization for parallel execution. In *Proceedings of ACM SIGMOD*, pages 9–18. ACM Press, 1992.
- [GO03a] L. Golab and M. Tamer Ozsu. Processing sliding window multi-joins in continuous queries over data streams. In *VLDB*, pages 500–511, September 2003.
- [GÖ03b] Lukasz Golab and M. Tamer Özsü. Issues in data stream management. *SIGMOD Rec.*, 32(2):5–14, 2003.
- [Gra93] Goetz Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–170, 1993.
- [GYW07] Xiaohui Gu, Philip S. Yu, and Haixun Wang. Adaptive load diffusion for multiway windowed stream joins. In *ICDE*, pages 146–155, 2007.
- [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.
- [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*, 2003.
- [HH99] P. J. Hass and J. M. Hellerstein. Ripple joins for online aggregation. In *Proceedings of ACM-SIGMOD Conference*, pages 287–298, 1999.
- [HXcZ07] Jeong-Hyon Hwang, Ying Xing, Ugur Çetintemel, and Stanley B. Zdonik. A cooperative, self-configuring high-availability solution for stream processing. In *ICDE*, pages 176–185, 2007.
- [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.

- [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.
- [Ioa96] Yannis E. Ioannidis. Query optimization. *ACM Comput. Surv.*, 28(1):121–123, 1996.
- [JAA⁺06] Navendu Jain, Lisa Amini, Henrique Andrade, Richard King, Yoonho Park, Philippe Selo, and Chitra Venkatramani. Design, implementation, and evaluation of the linear road benchmark on the stream processing core. In *SIGMOD*, pages 431–442, 2006.
- [JCE⁺94] Christian S. Jensen, J. Clifford, R. Elmasri, S. K. Gadia, P. Hayes, S. Jajodia, C. Dyreson, F. Grandi, W. Kafer, N. Kline, N. Lorentzos, Y. Mitsopoulos, A. Montanari, D. Nonen, E. Peressi, B. Pernici, J. F. Roddick, N. L. Sarda, M. R. Scalas, A. Segev, R. T. Snodgrass, M. D. Soo, A. Tansel, P. Tiberio, and G. Wiederhold. A consensus glossary of temporal database concepts. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 23(1):52–64, 1994.
- [KBZ86] R. Krishnamurthy, H. Boral, and C. Zaniolo. Optimization of non-recursive queries. In *Proceeding of VLDB*, pages 128–137, 1986.
- [KDY⁺06] Sailesh Kumar, Sarang Dharmapurikar, Fang Yu, Patrick Crowley, and Jonathan Turner. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In *SIGCOMM*, pages 339–350, 2006.
- [KFHJ04] Sailesh Krishnamurthy, Michael J. Franklin, Joseph M. Hellerstein, and Garrett Jacobson. The case for precision sharing. In *VLDB*, pages 972–986, 2004.
- [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.
- [KRB85] Won Kim, David S. Reiner, and Don S. Batory, editors. *Query Processing in Database Systems*. Springer, 1985.

- [Kun00] H. Kuno. Surveying the E-Services Technical Landscape. In *International Workshop on Advance Issues of E-Commerce and Web-Based Information Systems (WECWIS)*, pages 94 – 101, 2000.
- [KWF06] Sailesh Krishnamurthy, Chung Wu, and Michael J. Franklin. On-the-fly sharing for streamed aggregation. In *SIGMOD*, pages 623–634, 2006.
- [L. 00] L. Bouganim and F. Fabret et al. Dynamic query scheduling in data integration systems. In *ICDE*, pages 425–434, 2000.
- [LZJ⁺05] Bin Liu, Yali Zhu, Mariana Jbantova, Bradley Momberger, and Elke A. Rundensteiner. A dynamically adaptive distributed system for processing complex continuous queries. In *VLDB*, 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 *SIGMOD*, pages 347–358, 2006.
- [MRSR01] Hoshi Mistry, Prasan Roy, S. Sudarshan, and Krithi Ramamritham. Materialized view selection and maintenance using multi-query optimization. In *SIGMOD*, pages 307–318, 2001.
- [MS96] Sherry Marcus and V. S. Subrahmanian. Foundations of Multimedia Database Systems. *Journal of ACM*, 1996.
- [MSHR02] S. Madden, M. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *ACM SIGMOD*, pages 49–60, June 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.
- [Pc05] Olga Papaemmanouil and Ugur Çetintemel. Semcast: Semantic multicast for content-based data dissemination. In *ICDE*, pages 242–253, 2005.

- [PdBG94] Jan Paredaens, Jan Van den Bussche, and Dirk Van Gucht. Towards a theory of spatial database queries. In *Symposium on Principles of Database Systems*, pages 279–288, 1994.
- [PSR03] B. Pielech, T. Sutherland, and E. A. Rundensteiner. Adaptive scheduling framework for a continuous query system. In *Submission*, 2003.
- [RDS⁺04] E. A. Rundensteiner, L. Ding, T. Sutherland, Y. Zhu, B. Pielech, and N. Mehta. Cape: Continuous query engine with heterogeneous-grained adaptivity. In *VLDB Demo*, pages 1353–1356, 2004.
- [RG00] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill Higher Education, 2000.
- [RJK⁺05] Shariq Rizvi, Shawn R. Jeffery, Sailesh Krishnamurthy, Michael J. Franklin, Nathan Burkhart, Anil Edakkunni, and Linus Liang. Events on the edge. In *SIGMOD*, pages 885–887, 2005.
- [RRWM07] Venkatesh Raghavan, Elke A. Rundensteiner, John P. Woycheese, and Abhishek Mukherji. Firestream: Sensor stream processing for monitoring fire. In *ICDE*, 2007.
- [RSSB00] Prasan Roy, S. Seshadri, S. Sudarshan, and Siddhesh Bhobe. Efficient and extensible algorithms for multi query optimization. In *SIGMOD*, pages 249–260, 2000.
- [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.
- [Sel88] Timos K. Sellis. Multiple-query optimization. *ACM Trans. Database Syst.*, 13(1):23–52, 1988.
- [SHea03] Mehul A. Shah, Joseph M. Hellerstein, and et. al. Flux: An adaptive partitioning operator for continuous query systems. In *ICDE*, pages 25–36, 2003.
- [SLJR05] Timothy M. Sutherland, Bin Liu, Mariana Jbantova, and Elke A. Rundensteiner. D-cape: distributed and self-tuned continuous query processing. In *CIKM*, pages 217–218, 2005.

- [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.
- [SW04] Utkarsh Srivastava and Jennifer Widom. Memory-limited execution of windowed stream joins. In *VLDB*, pages 324–335, 2004.
- [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] Nesime Tatbul, Ugur Çetintemel, Stanley B. Zdonik, Mitch Cherniack, and Michael Stonebraker. Load shedding in a data stream manager. In *VLDB*, pages 309–320, 2003.
- [TcZ07] Nesime Tatbul, Ugur Çetintemel, and Stanley B. Zdonik. Staying fit: Efficient load shedding techniques for distributed stream processing. In *VLDB*, pages 159–170, 2007.
- [TD03] Feng Tian and David J. DeWitt. Tuple routing strategies for distributed eddies. In *VLDB*, pages 333–344, 2003.
- [TMSF03] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras. Exploiting punctuation semantics in continuous data streams. *TKDE*, 15(3):555–568, May/June 2003.
- [TZ⁺03] Nesime Tatbul, Ugur etintemel, Stanley B. Zdonik, Mitch Cherniack, and Michael Stonebraker. Load shedding in a data stream manager. In *VLDB*, pages 309–320, 2003.
- [UF00] T. Urhan and M. Franklin. XJoin: A reactively scheduled pipelined join operator. *IEEE Data Engineering Bulletin*, 23(2):27–33, 2000.
- [Val93] Patrick Valduriez. Parallel database systems: Open problems and new issues. *Distributed and Parallel Databases*, 1(2):137–165, 1993.
- [VN02] S. D. Viglas and J. F. Naughton. Rate-based query optimization for streaming information sources. In *Proceedings of ACM-SIGMOD*, pages 37–48, 2002.

- [VNB03] S. Viglas, J. Naughton, and J. Burger. Maximizing the output rate of multi-way join queries over streaming information. In *VLDB*, pages 285–296, Sep 2003.
- [WA93] 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.
- [WDR06] Eugene Wu, Yanlei Diao, and Shariq Rizvi. High-performance complex event processing over streams. In *SIGMOD*, pages 407–418, 2006.
- [WL05] Fusheng Wang and Peiya Liu. Temporal management of rfid data. In *VLDB*, pages 1128–1139, 2005.
- [WLLB06] Fusheng Wang, Shaorong Liu, Peiya Liu, and Yijian Bai. Bridging physical and virtual worlds: Complex event processing for rfid data streams. In *EDBT*, pages 588–607, 2006.
- [WRGB06] Song Wang, Elke A. Rundensteiner, Samrat Ganguly, and Sudeept Bhatnagar. State-slice: New paradigm of multi-query optimization of window-based stream queries. In *VLDB*, pages 619–630, 2006.
- [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.
- [XHcZ06] Ying Xing, Jeong-Hyon Hwang, Ugur Çetintemel, and Stanley B. Zdonik. Providing resiliency to load variations in distributed stream processing. In *VLDB*, pages 775–786, 2006.
- [ZKOS05] Rui Zhang, Nick Koudas, Beng Chin Ooi, and Divesh Srivastava. Multiple aggregations over data streams. In *SIGMOD*, pages 299–310, 2005.
- [ZR07] Yali Zhu and Elke A. Rundensteiner. Adapting partitioned continuous query processing in distributed systems. In *ICDE Workshops*, pages 594–603, 2007.
- [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.