Metadata-Aware Query Processing over Data Streams

by

Luping Ding

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 29th, 2008

APPROVED:

Professor Elke A. Rundensteiner Advisor	Professor George Heineman Committee Member
Professor Murali Mani Committee Member	Professor Leonidas Fegaras Univ. of Texas at Arlington External Committee Member
Professor Michael Gennert Head of Department	

Abstract

Many modern applications need to process queries over potentially infinite data streams to provide answers in real-time. This dissertation proposes novel techniques to optimize CPU and memory utilization in stream processing by exploiting metadata on streaming data or queries. It focuses on four topics: 1) exploiting stream metadata to optimize SPJ query operators via operator configuration, 2) exploiting stream metadata to optimize SPJ query plans via query-rewriting, 3) exploiting workload metadata to optimize parameterized queries via indexing, and 4) exploiting event constraints to optimize event stream processing via run-time early termination.

The first part of this dissertation proposes algorithms for one of the most common and expensive query operators, namely join, to at runtime identify and purge no-longer-needed data from the state based on punctuations. Exploitations of the combination of punctuation and commonly-used window constraints are also studied. Extensive experimental evaluations demonstrate both reduction on memory usage and improvements on execution time due to the proposed strategies.

The second part proposes herald-driven runtime query plan optimization techniques. We identify four query optimization techniques, design a lightweight algorithm to efficiently detect the optimization opportunities at runtime upon receiving heralds. We propose a novel execution paradigm to support multiple concurrent logical plans by maintaining one physical plan. Extensive experimental study confirms that our techniques significantly reduce query execution times.

The third part deals with the shared execution of parameterized queries instantiated from a query template. We design a lightweight index mechanism to provide multiple access paths to data to facilitate a wide range of parameterized queries. To withstand workload fluctuations, we propose an index tuning framework to tune the index configurations in a timely manner. Extensive experimental evaluations demonstrate the effectiveness of the proposed strategies.

The last part proposes event query optimization techniques by exploiting event constraints such as exclusiveness or ordering relationships among events extracted from workflows. Significant performance gains are shown to be achieved by our proposed constraint-aware event processing techniques.

Acknowledgments

I wish to thank my advisor, Prof. Elke A. Rundensteiner, for her inspiration, encouragement, guidance, patience and all kinds of help. I wouldn't arrive here without her. I wish to thank my committee members, Prof. George T. Heineman, Prof. Murali Mani and Prof. Leonidas Fegaras, for many useful inputs. I wish to thank my colleagues of CAPE project, Yali Zhu, Nishant Mehta, Timothy Sutherland, Bradford Pielech, Rimma Nehme, and Natasha Bogdanova, for working together with me on such a wonderful project. I wish to thank my intern mentors, Songting Chen at NEC Labs America, Kevin Wilkinson at HP Labs, and Paul Larson at Microsoft Research, for bringing new horizon to me. I wish to thank prof. Dan Dougherty for the help in the classic SAT/IMP problem. I wish to thank my DSRG friends, Hong Su, Li Chen, Song Wang, Bin Liu, Mingzhu Wei, and many others, for their friendship and help.

My deepest appreciation goes to my Mom and Dad, my husband Qingguang and my little angel Ada. They are my forever support and source of energy.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Research Motivation

1.1.1 Stream Processing Applications

As sensor network and online processing technologies mature, more and more modern applications need to process streaming data instead of persistently stored data. Below we list some of such applications.

- **Network analysis applications** [87] process streams of network packets to monitor usage and to detect intrusions. A network analysis application might execute a query that continuously monitors the source-destination pairs in the top 5 percentile in terms of total traffic in the past 20 minutes over a backbone link.

- **Monitoring applications** [93, 87] process data streams from sensor networks to monitor storehouse temperature, road traffic or environmental conditions. A storehouse temperature monitoring application might execute a query that reports the maximum temperature once every hour in a warehouse.

- **Transaction management applications** [63] process streams of transactions to control real-time inventory, recommend on-line discount policies, etc. These applications might execute queries over transaction streams to report total sales of items or to provide purchase recommendations to customers.

- **Online auction applications** [93, 94] process data streams of selling items, bids and registered users to answer queries such as to continuously report the item(s) with the maximum number of bids or monitor the average closing price once an hour.

In response, a lot of research efforts [12, 15, 29, 49, 78, 79] in data management have recently focused on query processing over real-time data streams.

1.1.2 Motivation of Exploiting Metadata on Streaming Data

In the applications listed above, data is in the form of continuous streams that are generated on the fly during query execution and thus are not available in its entirety until the end of query execution. Moreover, users often ask long-running queries (called *continuous queries* [15]) and expect the results to be delivered in real-time. This renders traditional query evaluation techniques ineffective because they tend to assume one-time queries over finite persistent data with pre-built indexes and materialized views. The main challenges are summarized as follows:

1). The data streams are potentially infinite. Hence the *stateful* operators, such as join, duplicate elimination, and aggregate operators, may need to maintain unboundedly growing states of all historical data in order to produce exact results [16, 93]. This potentially requires infinite storage resources.

2). Most stream applications put stringent requirements on real-time response. This requires stateful operators to maintain their state in main memory. The available memory may be quickly used up when faced with large volumes of fast-arriving data streams. Therefore, strategies for shrinking operator states while assuring correctness of query results are required.

3). Data streams are continuously generated on the fly. Hence the meta knowledge about streaming data, such as data value arrival patterns, is largely unknown at query compilation time. Rather it may become available only at query execution time [93]. Worse yet the arrival patterns of metadata is quite likely unpredictable. Therefore, the query execution strategies determined at query compilation time may become suboptimal at runtime. Accordingly, adaptive query execution approach and adaptive usage of metadata are highly desired.

To tackle these challenges, new metadata semantics, named *punctuations* [93], have been proposed. Stream providers may insert dynamic metadata, such as *punctuations*, into data streams. A punctuation in a data stream signals that tuples with certain attribute values will *no longer* occur in this stream. Such information can be used by the stateful operators to detect and then discard no-longer-useful data from the state.

An optimization example. We now use an example query in an online auction application to briefly illustrate how the punctuations can be used to optimize the query execution. Figure 1.1(a) shows two streams generated by an online auction application. Each auction is represented by a tuple in the Auction stream. Each bid placed by a bidder is represented by a tuple in the Bid stream. In both streams, tuples arrive in the order of their timestamp, which represents their open auction time or bidding time, respectively [1]. When the open duration for a particular auction expires, a punctuation can be derived and inserted into the Bid stream to signal the end of bids for that auction. The punctuation $<180, *, *>$ in the Bid stream in Figure 1.1(a) indicates that no more Bid tuples arriving after this punctuation will have "item_id=180". Notice that punctuations also have timestamps indicating the time when they are announced. In addition, since the $item\_id$ attribute is the key of the Auction stream, a punctuation can be derived on this attribute following each tuple in this stream.

Let's consider the query in Figure 1.1(b). This query asks for the total number of bids for each auction. One execution plan for this query is shown in Figure 1.1(c). It contains an equijoin operator which joins streams Auction and Bid on *item_id*, and a group-by operator that groups tuples in the output stream (Out_1) of the join by *item_id* and then evaluates the aggregate function count($*$) on each group.

Without additional knowledge on when the bidding for each auction

[1]We follow the general assumption that all input tuples have their timestamp generated by stream sources.

Auction Stream

item_id	seller_id	start_price	timestamp
180	jsmith	130.00	May-10-06 9:00:00
180	*	*	***May-10-06 9:00:20***
182	melissa	20.00	May-10-06 9:10:00
182	*	*	***May-10-06 9:10:30***

Tuple (in regular font)
Punctuation (in bold and italic font)

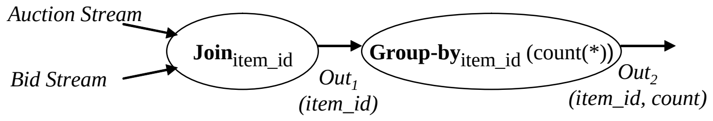
Bid Stream

item_id	bidder_id	bid_price	timestamp
180	pclover	175.00	May-14-06 8:27:00
182	smartguy	30.00	May-14-06 8:30:20
180	richman	177.00	May-14-06 8:57:00
180	*	*	***May-14-06 8:58:00***

(a) Punctuated Streams – Auction and Bid

```
Select     A.item_id, Count (*)
From       Auction A, Bid B
Where      A.item_id = B.item_id
Group by  A.item_id
```

(b) Continuous Query



(c) Query Plan

Figure 1.1: Example Streams and Queries in Online Auction Application.

finishes (thus no more bids for this auction will be recorded), the join state will have to maintain the data it has received for indefinitely long. For high-speed input streams, the state of the join operator may quickly become too huge to fit in the memory. To guarantee the precision of the result, part of the state would need to be moved to secondary storage. As more and more data are relocated, the real-time join processing efficiency may be severely affected and potentially put into jeopardy due to the expensive I/O operations.

If we consider punctuations provided in data streams, a smaller state can be achieved instead, as illustrated below. First, according to punctuations in the Auction stream, each Bid tuple can only match at most one

Auction tuple. Hence, a Bid tuple can be discarded immediately after it joins with a matching Auction tuple. Second, when a punctuation on the *item_id* attribute is obtained from the Bid stream, the Auction tuple in state that contains the same *item_id* value can then be purged. This way the Auction tuples can be removed from the state once the auction has been closed. Meanwhile, a punctuation regarding this *item_id* value can be propagated to the Out_1 stream for the *group-by* operator to produce a result for this specific item. We can see that the state of the Auction stream only maintains tuples that represent the open auctions. The state of the Bid stream only maintains the tuples that haven't joined with the matching Auction tuple yet. Both states become very concise.

The following advantages can be brought to query execution by such concise operator states. First, it helps to avoid the expensive I/O operations as the state shrunk by punctuations may fit into memory. Second, it reduces the memory requirements for the query so that the saved memory can be used for other important purposes, e.g., for evaluating other queries. Third, it reduces CPU overhead as now fewer tuples in the state need to be examined during the probing.

1.1.3 Motivation of Exploiting Metadata on Streaming Queries

While long-running continuous queries are common in some stream applications such as monitoring applications, in many other applications where streaming data are generated such as transaction management, a large number of concurrent one-time user queries may be experienced. In these applications, not only the data are streaming, but also the queries form a high-speed stream. Below we show such an example.

```
Query 1
SELECT      categoryID, buyer_state, buyer_job, COUNT(*)
FROM        Bid_info
WHERE       buyer_state = 'MA' and categoryID = electronic
GROUP BY    categoryID, buyer_state, buyer_job
WINDOW      24 Hours
```

```
Query 2
SELECT      categoryID, buyer_state, buyer_job, COUNT(*)
FROM        Bid_info
WHERE       buyer_state = 'CA' and categoryID = homegoods
GROUP BY    categoryID, buyer_state, buyer_job
WINDOW      24 Hours
```

```
Query Template for Queries 1 and 2

  SELECT      categoryID, buyer_state, buyer_job, <agg-func-list>
  FROM        Bid_info
  WHERE       <selection-predicates>
  GROUP BY    categoryID, buyer_state, buyer_job
  WINDOW      <window-length>
```

Figure 1.2: Example Queries and Corresponding Query Template.

Consider an online purchase application where user transactions are recorded as data streams. It may provide purchase recommendations based on user interests. For example, a Massachusetts user that plans to buy a TV may want to receive recommendations based on Query 1 in Figure 1.2, while a California user who wants to buy some home goods may need recommendations based on Query 2 in Figure 1.2. Both are one-time queries based on the *transaction* stream generated so far. Since many concurrent users may require recommendations at the same time, the query system may experience a high-speed stream of queries.

If each of these queries is processed individually, for data streams of large volume, the query system may face scalability problems [71] as each query needs to maintain separate operator states and no processings of possibly common sub-tasks are shared. We observe that queries in these applications are often similar although not identical as they may be generated through a common user interface. For example, the two example queries in Figure 1.2 may be submitted by users through a GUI interface for "request recommendation". We can use a query template as an abstraction for these similar queries. Each individual query is then treated as a customized instantiation of the template. For example, the two example queries can be instantiated from the query template in Figure 1.2 by filling in the selection predicate in the WHERE clause, the aggregate function in the SELECT clause and the window specification in the WINDOW clause.

By employing such a query template, we can use a single query plan to achieve the shared execution of a large number of parameterized queries instantiated from a query template. This way the operator state and inter-operator queues can be shared to avoid data duplication.

Since the actual filter predicates, aggregate functions and even the historical data to be queried (i.e., window specification) may vary significantly among the parameterized queries, one important problem that must be solved is how to organize the historical data to best serve the query workload. On one hand, multiple access paths via indexes should be provided to speed up data lookup based on varied or even disjoint sets of attributes. On the other hand, to handle streaming data, the index maintenace costs upon frequent data insertions due to arrival of new data and deletions due to data expiration should be minimized.

1.1.4 Motivation of Exploiting Metadata in Event Stream Processing

Besides the applications that produce *relational data streams*, as automated business processes become ubiquitous, many applications such as business activity monitoring, supply chain management and anomaly detection generate *continuous event streams*. Unlike relational streams that consist of homogeneous data tuples, the data objects (i.e., events) in event streams can be of different types and hence have different sets of attributes. Query processing over event streams [101, 35] aims at detecting interesting *event patterns* in event streams for quick detection and reaction to critical business situations.

The event patterns specify complex *temporal* and *logical* relationships among events. Consider the example event pattern *EP1* below, in which "SEQ" represents the *temporal* relationship between two events and [totalPrice>200] is the predicate on the GenerateQuote event. This pattern monitors the cancelled orders that involve the participation of both suppliers and remote stocks, with quote's price > \$200. Frequent occurrences of such patterns may indicate the need for an immediate inventory management, for example.

Event Pattern EP1:

SEQ((SEQ(OrderFromSupplier,GenerateQuote[totalPrice > 200])

AND SEQ(UseRemoteStock,GenerateInvoice)),CancelOrder)

We observe that in practice, many business events do not occur randomly. Instead they follow pre-defined business logic or rules, typically called a workflow model [54]. As consequence, various constraints may exist among events in these event processing applications. In particular, *occurrence* constraints, such as mutually exclusive events, and *order* constraints, such as one event must occur *prior* to the other event, can be observed in all the applications listed above.

The availability of these constraints enables us to predict the non-occurrences of future events from the observed events. Such predictions would help us to identify which partial query matches are guaranteed not to lead to final results. Further efforts in maintaining and evaluating these partial matches can be prevented, resulting in significant savings in memory and CPU. Example 1 below illustrates such optimization opportunities that remain unexplored in the literature.

An optimization example. Assume the event stream is generated by the online order transactions [80, 97] that follow a predefined workflow. We assume each task in the workflow, if performed, will submit an event to the event stream. Suppose the *UseLocalStock* and the *UseRemoteStock* events are mutually exclusive. Also, suppose that any *GenerateQuote* event, if it occurs, must be before the *SendQuote* event in a transaction.

Consider the example event pattern *EP1* again. By exploiting the event constraints, whenever a UseLocalStock event occurs, this transaction is guaranteed to not match the query because the UseRemoteStock event will never occur in this transaction. Also, once a SendQuote event is seen in a transaction, and no GenerateQuote event with totalPrice>200 has been observed so far, the transaction will not match the query because no Generate-

Quote event will happen after the SendQuote event. In either case, any partial matches by these transactions need not be maintained and evaluated further as they are guaranteed to never lead to a final result. If the query processing of large numbers of transactions could be terminated early, a significant amount of CPU and memory resources would be saved.

1.1.5 State-of-the-Art in Metadata-Aware Stream Processing

Using metadata to optimize queries has been extensively studied in traditional database, where it is called *semantic query optimization* [24, 30, 59, 70, 102]. Existing work focuses on employing integrity constraints and index information to rewrite a query plan into another equivalent plan yet with lower cost. The optimization techniques include join/select introduction, join/select elimination and detection of unsatisfiable conditions. These optimizations are all conducted before query execution commences.

Metadata have also begun to be considered in the stream processing context. The *k-constraint-exploiting algorithm* [16] exploits one-to-many join cardinality and clustered data arrival patterns to detect and purge no-longer-useful data to shrink the state of stateful operators. These clustered patterns are statically specified, and hence only characterize restrictive cases of the real-world data. If the actual data fails to obey these static constraints, the precision of the join result may suffer due to the incorrect purge of tuples.

The punctuation model covers a wide class of constraints, including the well-known static ones such as the unique key and the clustered arrival of attribute values [16]. [93] provides punctuation-based pass, purge and propagation invariants for algebra operators. However, no research work has been done to design query execution strategies by exploiting punctuations, which is the focus of the first two parts of this dissertation.

The *index selection problem* has been extensively studied in static databases [4, 28, 51, 65], in which data updates are rare compared to queries. Index selection tools take a query workload as input and suggest a set of indexes that can maximally benefit the given workload. Index adaptation due to changes in workloads means inserting a new index or deleting an existing index.

Indexing in stream contexts has not yet received much attention, possibly due to the dynamic nature of the streaming data. [56] studies methods for indexing a single attribute for individual streaming algebra operators under the sliding window semantics. Index selection driven by workload metadata of streams of queries has so far not yet been tackled in stream contexts, which now is the focus of the third part of this dissertation.

For event stream processing, existing work [35, 101] focuses on query model/language design and query algebra development. None of them considers exploiting the common event constraints to optimize the memory and CPU utilization during event query execution, which is the focus of the last part of this dissertation.

1.2 Research Focus of This Dissertation

The research goal of this dissertation is to investigate the techniques for exploiting metadata, either on streaming data or on queries, to optimize the CPU and memory utilizations in query processing over data streams.

1.2.1 Constraint-Aware Stream Query Operators

Research challenges. Punctuations are dynamic constraints that become available only at runtime. Typically no prior knowledge on punctuation arrival patterns is available at static query optimization phase. This requires the punctuation-aware execution techniques to be lightweight and runtime-adjustable in order to handle evolving punctuation arrival behavior.

In addition, sliding windows are essential constraints in stream contexts because they help to bound the size of the operator states and also instruct the queries to provide results on recent data. The *sliding window* constrains the query to only consider the "recent" portions of the streams [12]. Sliding windows and punctuations are constraints about different aspects of the streaming data, i.e., the timestamp and the application-specific attributes, respectively. Both can be used to identify and purge no-longer-useful data from operator states. In many cases the data sets invalidated according to these two types of constraints respectively may overlap with each other. An ill-designed operator execution algorithm that exploits both constraint types may achieve minor gains in memory but incur a possibly doubled probing overhead compared to the algorithms that only utilize one constraint type. Overall it may thus yield worse performance than exploiting only one or none of the constraint types. Correspondingly, we must carefully design the algorithm.

We thus put forth the following goals for our design:

1). The query operators should be able to react to punctuations and conduct appropriate optimizations. The operators should also be adaptive to handle fluctuating data and punctuation statistics.

2). If both punctuation and sliding window constraints are present, the operator execution algorithm should achieve better performance regarding both memory overhead and result output rate compared to the algorithms that exploit only one of the constraint types.

3). If no punctuations are provided for the data stream, the punctuation-aware operator should achieve equivalent performance as punctuation-unaware operators. That is, we wish to avoid any penalty of a potentially more complex solution in the cases when the techniques cannot lead to any gains.

Dissertation Contributions. In this dissertation task, we target SPJ (Select-Project-Join) queries since SPJ operations are the core operations in both continuous and static query languages. We focus on the design of Join operators since Join is the only stateful operation in SPJ operations and hence benefits from exploiting punctuations. We also equip other operators with constraint-aware abilities. Hence our approach support complete SPJ queries. We made the following contributions in this dissertation task.

We develop the PJoin algorithm for the join operator to exploit punctuations. We propose alternate strategies for the join operator to purge the state and to propagate punctuations, including eager/lazy purge and eager/lazy propagation. We equip the PJoin algorithm with configurable execution logic to dynamically apply appropriate purge and propagation strategies so to achieve runtime-adjustable join solutions. We also explore the trade-off between different purge strategies regarding the memory overhead and the data output rate experimentally, and trade-off between different strategies with regard to the punctuation output rate.

We recognize the optimization opportunities enabled by punctuations and by the interactions between the constraints of different dimensions, i.e., the sliding window in the time dimension and the punctuation in the attribute value dimension. We design the PWJoin algorithm that is able to exploit not only punctuations but also sliding window constraints. For PWJoin, we design a novel index structure for the join state and corresponding state purge strategies to facilitate the optimizations based on the two constraint types. We also propose early propagation technique by exploiting synergy of the two constraint types.

We design cost models for estimating the memory and the CPU costs of the PJoin and the PWJoin solutions. We compare the performance of these algorithms with corresponding metadata-unaware algorithms based on these cost models.

We have implemented the PJoin and the PWJoin algorithms in the CAPE stream processing system [83, 76] (see Section 1.3 for details). We report on the extensive experimental studies we have conducted to explore the effectiveness of these metadata-exploiting join solutions.

The PJoin and the PWJoin approaches can be applied to existing stream processing systems [7, 1] to equip the equi-join operator with the ability of exploiting punctuations to shrink the runtime join state.

1.2.2 Constraint-Driven Runtime Stream Query Optimization

Research challenges. Dynamic constraints on streaming data values can be utilized to optimize not only individual operator implementation but also the query plan structure. Several challenges must be tackled when exploiting dynamic constraints to optimize query plan structure due to the following observations.

First, since constraints become available only at runtime, the query optimization must be conducted frequently at runtime upon receipt of each constraint to assure prompt reaction to constraints.

Second, constraints may have their *lifespans*, i.e., the properties described by a constraint may only be satisfied by a particular substream. Therefore, respective optimizations driven by a constraint will be applicable only to the corresponding substream and thus only for limited periods at a time.

Finally, one constraint defined for one stream may enable several distinct optimizations in collaboration with constraints from other streams. Hence multiple distinct query plans optimized by constraints may be valid at a time with *partially overlapped scopes* (i.e., the substreams that these plans are applicable to may overlap with each other).

Therefore, first, the algorithm employed to find the optimized plans given a set of constraints must be efficient so as to identify all beneficial optimization opportunities. Second, the algorithm must be lightweight so as to minimize the runtime optimization overhead. For execution, a query execution paradigm must be designed so that 1) it supports the concurrent execution of multiple logical plans on overlapping input substreams without duplication of data storage and costs, and 2) it can adaptively phase in and out logical plans on substreams with negligible physical plan switching costs [38, 104].

Dissertation contributions. In this dissertation task, we identify four semantic query optimization opportunities that can be enabled by heralds (a constraint model extended from punctuation [93], as will be defined in Section 11.1). The corresponding optimization techniques parallel the SQO techniques found in traditional databases [31, 70].

To minimize the optimization overhead, we develop an efficient constraint reasoning algorithm named $PredSAT$ based on classic satisfiability reasoning theory. PredSAT is guaranteed to identify all four herald-driven optimization opportunities incrementally at runtime.

Multiple concurrent SQO plans may be enabled by heralds for processing different, potentially overlapping stream partitions. We propose a versioned minimum range model for generating multiple concurrent logical plans based on the result of PredSAT.

To achieve multiple concurrent logical plans with one single physical plan, we propose a novel query execution paradigm employing multi-modal operators with runtime configuration logic. This paradigm eliminates any replication of operator states or inter-operator queues, guarantees instantaneous application of herald-driven query optimizations, requires zero plan migration effort, and naturally supports highly flexible adaptive execution.

We conduct an extensive experimental study in the CAPE system [83]. The experimental results confirm that our herald-driven optimization techniques significantly reduce query execution time, up to 60% in our tested scenarios.

Our techniques can be incorporated into the existing stream processing systems [7, 1] to optimize SPJ queries with inequality select and/or join predicates when dynamic constraints like heralds present.

1.2.3 Index Tuning for Parameterized Streaming Queries

Research challenges. In many stream applications, not only data is streaming, but also queries form high-speed streams. As an example, for pull-based continuous queries [9, 26], the requests for pulling the query results may form a high speed query stream. To provide customized results when they are pulled, the query stream may contain a large number of concurrent parameterized queries that are instantiated from a pre-defined query template. Since different uses may have significantly different interests on the results, these parameterized queries may have selective predicates concerning diverse sets of attributes. The ability to efficiently select the data to be applied more expensive operations such as join or aggregation to is essential to achieving good query execution performance.

It has been well recognized that a proper index mechanism is needed to speed up the data lookups [4, 28, 56]. The new challenge we face here is that the data to be maintained for the query template is of large volume and quickly evolving due to streaming data. In addition, the parameterized queries may specify widely varied window sizes. Hence the history data pertinent to each of these parameterized queries may be different. Furthermore, both data and query workload may fluctuate due to highly dynamic streaming environments, thus requiring frequent index tuning.

Therefore, the design of the index mechanism must meet the following goals: First, the index should benefit the processing of a large number of rather diverse queries. Second, the index structure should require minimal maintenance effort when processing data updates. Third, the index structure should be memory-efficient to be maximally maintained in main memory. Lastly, the index should be lightweight to be easily migratable when the workload experiences significant changes.

To withstand the fluctuations in data and query workloads, the query system should be able to quickly observe the changes at runtime and then tune the index accordingly. First, efficient index selection algorithms are needed to identify the optimal or near-optimal index configurations within realtime. Second, method for identifying the needs for index tuning must be designed. Finally, effective index migration strategies need to be designed to migrate from the current index configuration to a new configuration deemed to be more efficient in an online fashion.

Dissertation contributions. In this dissertation task, we focus on parameterized groupby queries due to the following reasons. First, groupby queries are popular in our targeted applications for analytical purposes. Second, groupby (aggregate) operations are expensive. Therefore, efficiently selecting data to be grouped and aggregated on is the first important step leading to efficient execution of parameterized groupby queries. We made the following contributions in this dissertation task.

First, we propose the PSGB query as an abstraction of a large number of runtime instantiated queries. This formulation leads to efficient optimization of these runtime queries, i.e., a single PSGB operator can be designed to achieve resource sharing among the queries instantiated from the PSGB template without having to analyze them individually as they are instantiated.

Second, we employ a lightweight *IMP index* solution to manage the PSGB state for supporting efficient data lookups required by PSGB instantiations with diversified selection conditions. While existing work in streaming databases uses one-level hash-based indices for efficient state management, we show in our experimental study that our proposed solution beats existing solutions by a 9-fold performance improvement (without usage of any additional memory space) for large window sizes.

Our key contribution lies in being the first to tackle the index tuning problem in the streaming context. We design the *EPrune* index selection algorithm that is guaranteed to find the optimal IMP configuration. By properly pruning candidates, the complexity and hence the execution time of EPrune can be significantly reduced compared to exhaustive search, sometimes more than ten-fold.

To meet the efficiency needs that are more important for online index tuning than its guaranteed optimality, we also design a time-efficient greedy index selection algorithm named *RGreedy* and equip it with three alternative search heuristics. RGreedy is shown to find the near-optimal IMP configuration with observed polynomial complexity even in large search spaces.

Our experimental study conducted in the CAPE system [83, 76] shows that the IMP index always wins over the state-of-the-art index methods. RGreedy with PCL and Hybrid heuristics finds the optimal IMP configurations in all of our extensive test cases. For large search spaces, when EPrune takes hours to finish, RGreedy always terminates within seconds. Moreover, the PSGB operator with runtime index tuning outperforms the operator with a fixed index configuration.

Our techniques can be applied to the stream processing systems [9, 26] that support the pull-based query execution to optimize the execution of the group by queries. The techniques can also be applied to traditional database systems to optimize parameterized groupby queries when data updates are frequent.

1.2.4 Runtime Semantic Query Optimization for Event Stream Processing

Research challenges. Detecting complex patterns in event streams has become increasingly important for modern enterprises to react quickly to critical business situations [37, 54, 80]. In many practical cases business events are generated based on pre-defined business logics, such as a workflow model [54]. Hence constraints, such as occurrence and order constraints, often hold among such events. For example, in online order applications [80, 97], if UseLocalStock event occurs, the UseRemoteStock event will not occur (i.e., occurrence constraint) in the same online order transaction. Also, the GenerateQuote event, if it occurs, it must occur before the SendQuote event (i.e., the order constraint) in the same online order transaction. We have observed that reasoning using these known constraints enables us to predict the non-occurrences of certain future event types, thereby helping us to identify and then terminate long running query processes that are guaranteed to eventually not lead to successful matches.

Several key challenges must be tackled to achieve efficient constraint-aware query processing over event streams. One critical question we must answer is how to identify unsatisfiable partial query matches at *runtime*. As we will show in Section 26.3, unlike the query unsatisfiability analysis in the static case where the occurrence and order constraints are independent (Section 26.2), constraints of different types may be chained together to infer new constraints at runtime. Hence it is non-trivial to identify all possible optimization opportunities in a timely manner. In addition, there may be thousands or even millions of concurrent business transactions that generate events to be matched by the query. To assure the efficiency and scalability, the runtime reasoning for each individual transaction must be *lightweight*. Otherwise, the overhead of constraint reasoning may outweigh its benefits.

Dissertation contributions. We made the following contributions in this dissertation task. First, we identify the optimization opportunities in complex event processing to terminate unsatisfiable query processing early by exploiting occurrence and ordering event constraints.

Second, we propose a polynomial time, runtime query unsatisfiability (RunSAT) checking procedure for detecting the unsatisfiable query processing. The RunSAT checking is based on a formal logic reasoning using the combination of event query, event constraints and partial event history.

To improve the RunSAT performance, first, we apply abductive reasoning [46, 47] to pre-compute query failure conditions. Second, we exploit the incremental properties at runtime for RunSAT reasoning.

To facilitate the integration of our techniques into state-of-the-art event processing architectures [35, 101], we augment the event query with failure conditions. We identify three common constraints that enable constant RunSAT checking costs.

We conducted an extensive experimental study in a prototype event processing system. The experimental results demonstrate that significant performance gains, i.e., memory savings up to a factor of 3.5 and CPU savings at a factor of 2, are achieved through our approach, with small overhead spent on optimization itself .

Our technique can be easily plugged into existing event processing systems [101, 36] as a runtime semantic query optimization module to optimize the memory usage and to improve the query execution time.

1.3 Overview of the CAPE System

The techniques we have designed to achieve the above research goals have been implemented and tested in a prototype stream processing system named CAPE [83, 76] which is built at WPI as a team effort to serve as the testbed for our research designs for data stream processing. CAPE stands for Constraint-Aware Adaptive Stream Query Processing Engine. The CAPE system is a prototype stream processing system to evaluate queries over data streams in highly dynamic stream environments. The system has been demonstrated in VLDB 2004 (centralized version) and 2005 (distributed version) conferences [83, 76]. The proposed strategies and algorithms in the first three parts of this dissertation have been built into the CAPE system to equip it with the ability of exploiting constraint on data and queries to optimize query execution.

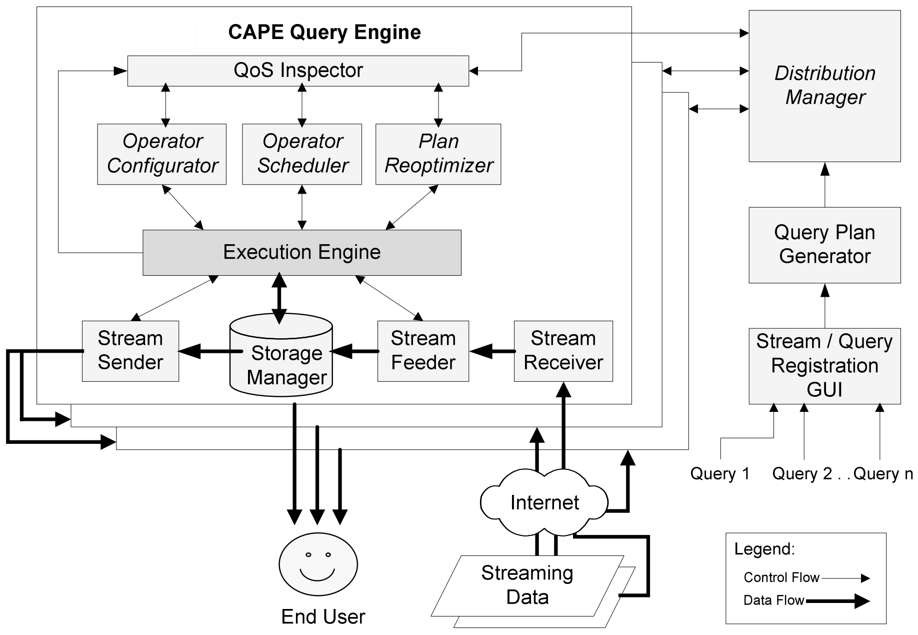


Figure 1.3: CAPE System Architecture.

The CAPE system architecture is depicted in Figure 1.3. The system is built to be run on a single machine (centralized version) as well as across multiple machines (distributed version) [2]. Each machine (processor) can run an instance of the query engine named the *CAPE engine*. If the system is run on multiple machines, a distributed manager overlooks these multiple CAPE query engines and collects statistics from all of them to make system-wide adaptation decisions. The key adaptive components in CAPE

[2]The proposed techniques in this dissertation are only applicable to the centralized version of 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 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.

As mentioned earlier, this dissertation focuses on investigating constraint-exploiting query optimization technologies in four aspects, including 1) query operator optimization using streaming data constraints, 2) query plan optimization using streaming data constraints, 3) query optimization using query workload metadata, and 4) event query optimization using event constraints from workflow. Among them, 1), 2) and 3) together correspond to the Operator Configurator and Plan Reoptimizer components in the CAPE architecture shown in Figure 1.3. All new designs and algorithms in Part I, Part II and Part III of this dissertation are implemented and experimented within the CAPE system. The optimization of event processing using event constraints is developed in a separate event processing prototype system, as will be detailed in Part IV.

1.4 Dissertation Organization

The rest of this dissertation is organized as follows: The four research topics are discussed in detail in Part I, Part II, Part III and Part IV in this dissertation respectively. The discussions of each of the four research topics include the relevant research motivation, problem introduction, background, solution description, experimental evaluation and discussion of related work respectively. Chapter 31 concludes this dissertation and Chapter 32 discusses possible future work.

Most materials in this dissertation have been published as conference papers. The materials in Part I have been presented in [40, 44, 41]. The materials in Part III have been presented in [43]. The materials in Part IV have been presented in [39]. In addition, the materials in Part II have been presented in a technical report [42].

Part I

Punctuation-Aware Stream Query Operators

Chapter 2

Introduction

2.1 Stream Join Processing and Constraints

The processing of *stateful* operators such as join over data streams is resource intensive. These operators need to maintain the already-processed data in their state to evaluate the data to be arriving in the future. Since data streams are potentially infinite, the state of these operators may grow indefinitely, thus requiring potentially unbounded storage. Many well-known stream-oriented join solutions aim to tackle this problem, including symmetric hash join [100], ripple joins [62], XJoin [95, 96], and hash-merge join [81]. These operators may quickly consume a fairly large portion of memory once execution starts. To avoid memory overflow, XJoin and hash-merge join choose to move part of the state to secondary storage. As the state continuously grows, the operator execution efficiency may drop dramatically due to expensive I/O operations. This is unacceptable for most streaming applications where real-time response is required.

It is clearly not practical to compare every tuple in one potentially infinite stream with all tuples in another also possibly infinite stream [8]. Recent work on window joins [2, 57, 68] has concluded that a significant portion of queries in stream applications are only interested in joining data from two or more streams that arrive relative to each other within a bounded time period. We will show an example of such window joins in Section 1.1.4. Under the window join semantics, the join state can be bounded by only keeping the data that reside in the current window.

However, windows cannot always be established for user queries as they clearly affect the semantics of the query. This raises the question whether there are alternate methods to aid us in bounding the size of the operator state. One such alternative is based on the observation that data streams may conform to some semantic constraints that can be utilized to detect and thus purge no-longer-needed data from the runtime join state [16, 93]. For example, in an online auction application [94], once an auction for an item is closed, it is guaranteed that no more bids will be received on this item in the Bid stream. If $itemID$ is the join attribute, such cluster-arrival constraint can be utilized by the join operator to discard no-longer-needed data in a timely manner. [93] propose to embed metadata, referred to as *punctuations*, inside data streams to explicitly announce such termination points of attribute values. Data streams that carry punctuations are referred to as *punctuated streams*.

2.2 Motivation Examples for Optimizing Joins over Punctuated Streams

Let us examine the advantages to be gained in join processing by exploiting punctuations. In a natural join over two relational streams S_1 and S_2 on a common attribute att, when a punctuation on a join value val from S_1 has been received, the tuples from S_2, either already-processed ones or future-incoming ones, that contain the join value val will thus no longer be joining with any future tuples from S_1. Hence they can be removed from the state. The same tuple purge rule applies to punctuations from S_2. The join operator may also be able to propagate the punctuations it received to benefit downstream operators [93], e.g., for them to purge their states respectively.

In addition, either *windows* or *punctuations* can be exploited to optimize the join execution. As will be shown by the example below, in some cases, these two types of constraints may be available simultaneously to the join operator, e.g., when we evaluate a window join over punctuated streams. The interaction of these two types of constraints may enable further optimizations. Below we use an example query from an online auction application [94] to illustrate the optimization opportunities enabled by the coexistence of punctuations and sliding windows.

Consider the auction application described in Section 1.1.4 and the query below. This query is similar to the query in Figure 1.1 (Section 1.1.4). The only difference is that there is a 24-hour sliding window applied to the Auction stream. Accordingly, a *window join* operator is applied for joining the Auction and the Bid stream on the item_id attribute. Therefore, regarding tuples from the Auction stream, only the ones whose open auction time is within 24 hours prior to the bid time of the latest-received Bid tuple need to be maintained. However, if many auctions are opened concurrently and for each auction there are a large number of bids, the state of the join operator may still be huge.

```
Select     A.item_id, Count(*)
From       Auction [Range 24 Hours] A, Bid B
Where      A.item_id = B.item_id
Groupby by A.item_id
```

If we exploit punctuations as described in Section 1.1.4, the state size can be further reduced from the state of the pure window join. In addition, for those Auction tuples that represent the auctions whose open period is longer than 24 hours, when it moves out of the 24-hour window, no more join results will be produced for this auction, though further Bid tuples for this auction may still arrive. Hence, any future Bid tuples for this auction can be directly dropped without even being processed.

From the above example, we obtain the following observations that motivate our join algorithm that utilizes punctuations.

1). For potentially infinite data streams, the join state will grow unboundedly. Even for window join, for relatively long-lasting windows or rapid data arrivals, the join state would typically contain a large number of tuples. In either case, by exploiting punctuations, the join state can be effectively shrunk, thus improving the probing efficiency.

2). A newly-received tuple may never need to be inserted into the state if the punctuations have indicated that this tuple will no longer join with any future-arriving tuples from the other stream. Then this tuple can be simply discarded after it has been used to produce the result based on the current state.

3). A join operator may be able to help downstream operators by propagating punctuations, for instance, to unblock blocking operators such as group-by. [93] defines formal punctuation propagation rules for typical algebra operators.

4). Further optimization is achievable due to the interaction between these two types of constraints. For example, with tuples being invalidated by windows, some punctuations can be propagated much earlier.

While these potentially huge benefits exist, to the best of our knowledge, no prior work has considered the design of join operators to effectively exploit the combined constraints of punctuations and sliding windows.

2.3 Our Approach: Punctuation-Exploiting Join Algorithms

In this dissertation task, we explore the join algorithms that exploit punctuations. We first propose an algorithm named PJoin (for Punctuation-exploiting Join) for joins without window specifications. Secondly, we propose the Punctuation-exploiting Window Join (PWJoin) algorithm. The two algorithms have been published in [44, 40] and [41] respectively. Our contributions are summarized as follows:

1). We recognize the optimization opportunities enabled by punctuations and by the interactions between the constraints of different dimensions, i.e., the sliding window in the time dimension and the punctuation in the attribute value dimension. We develop the PJoin and the PWJoin algorithms that are able to exploit punctuations in non-windowed and windowed cases respectively.

2). For PJoin, we propose alternate strategies for the join operator to purge the state and to propagate punctuations, including eager/lazy purge and eager/lazy propagation. We explore the trade-off between different purge strategies regarding the memory overhead and the data output rate experimentally, and trade-off between different strategies with regard to the punctuation output rate.

3). For PWJoin, we design a novel index structure for the join state and execution strategies to facilitate the exploitation of the combined constraints. We also provide a generalized PWJoin solution to handle multiway joins.

4). We provide the cost models for estimating the memory and the CPU costs of the PJoin and the PWJoin solutions. We compare the performance of these algorithms with corresponding constraint-unaware algorithms based on these cost models.

5). We have implemented the PJoin and the PWJoin algorithms in the CAPE system. We report on the extensive experimental studies we have conducted to explore the effectiveness of these constraint-exploiting join solutions.

Chapter 3

Background

3.1 Pipelined Join

The join algorithms we propose in this work extend the popular stream-oriented join algorithm, namely symmetric pipelined join [100]. As shown in Figure 3.1, the pipelined join maintains a state for each of its input streams. As a tuple t arrives from S_1, it is first *inserted* into the state of S_1. Then it is used to *probe* the state of S_2 that has been constructed. For any match found, a result tuple is produced. The processing of tuples from S_2 is similar. In summary, the join operator continuously performs the *insert-probe* operation sequence for each input tuple. Each tuple is processed to completion before the processing of the next tuple starts.

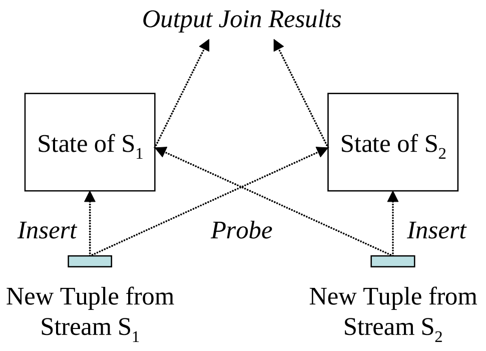


Figure 3.1: Pipelined Join.

3.2 Punctuation

Punctuations [93] are dynamic constraints that are interleaved with the streaming data. Under the relational data model, each data stream is associated with a fixed schema, i.e., a set of attributes. All tuples and punctuations in a stream conform to the schema of this stream. According to [93], a punctuation in a stream is expressed as a list of patterns, with each pattern corresponding to an attribute in the schema of the stream. A tuple t is defined to $match$ a punctuation p if the value of every attribute of t matches the corresponding pattern specified in p. The punctuation semantics specify that no tuple arriving *after* a punctuation will match this punctuation. The following five patterns along with the values they match are defined in [93].

- A *wildcard*, denoted as $*$, matches all values.
- A *constant* c, matches only c.

- A *range*, denoted with [a, b] for inclusive ranges or (a, b) for exclusive ranges, matches those values that fall in the given range.
- A *list*, denoted as {a, b, c}, that matches values in the list.
- The *empty pattern*, denoted as $\emptyset$, that does not match any value.

For example, given the Bid stream with schema <item_id, bidder_id, bid_price>, the punctuation <{1001, 2004}, $*$, $*$> describes that no bids on items 1001 or 2004 will occur in this stream after this punctuation.

Table 3.1: Notation and Functions.

Notation	*Meaning*
t	A relational tuple
p	A punctuation
$tset_i$	Set of all tuples received from stream S_i so far
$pset_i$	Set of all punctuations received from stream S_i so far
$vpset_i$	Set of all join values specified by punctuations received from stream S_i so far
Function	*Return Value*
$\pi_{att}(t)$	Value of attribute att of tuple t
$\pi_{att}(p)$	Value of attribute att of punctuation p
$\Pi_{att}(pset)$	Set of values from applying $\pi_{att}(p_i)$ to all p_i in $pset$
match(t, p)	True if t matches p
setMatch(t, pset)	True if t matches any p in $pset$
setNomatchPset(tset, pset)	All punctuations in $pset$ that have no match in $tset$

[93] proposes the *pass*, *keep* (or the inverse, *purge*) and *propagate* invariants for query operators to produce partial results, purge state and propagate punctuations to the output stream. Below we show the purge and propagate invariants for the natural join operator. The definitions of these invariants and the invariants we propose later in Section 7.2 use the functions listed in Table 3.1. Table 3.1 also lists the notation that will be used throughout this work.

Lemma 1 (Purge Invariant). *In a natural join $S_1 \bowtie S_2$ with a common attribute att, the following tuples are no longer useful and can be purged from the state.*

$$[t \mid t \in tset_1 \wedge setMatch(\pi_{att}(t), \Pi_{att}(pset_2))] \cup [t | t \in tset_2 \wedge setMatch(\pi_{att}(t), \Pi_{att}(pset_1))]$$

Lemma 2 (Propagate Invariant). *In a natural join $S_1 \bowtie S_2$ with a common attribute att, the punctuations on the following join values can be propagated to the output stream. It is assumed that all input punctuations contain wildcards for all non-join attributes [93].*

$$[p | p \in setNomatchPset(tset_1, pset_1) \vee p \in setNomatchPset(tset_2, pset_2)]$$

Besides the above invariants [93], we have derived the following punctuation propagate invariant for regular joins (i.e., no window specified) based on the punctuation semantics. We name it *regular propagate invariant* to distinguish it from the propagate invariant proposed in [93] (Lemma 2). Both propagate invariants are designed for regular joins. They are also applicable to window joins.

Lemma 3 (Regular Propagate Invariant). *In a natural join $S_1 \bowtie S_2$ with a common attribute att, the punctuations on the following join values can be prop-*

agated to the output stream. Again, it is assumed that the input punctuations contain wildcards for all non-join attributes.

$$[v | v \in vpset_1 \wedge v \in vpset_2]$$

Proof. If the punctuation on a join value v has been received from both input streams, then no more future-arriving tuples will contain this join value. Hence no more join results will contain this join value. Therefore, a punctuations on join value v can be safely sent to the output stream to announce this fact.

According to the above invariants, the following optimizations can be achieved in the evaluation of a binary natural join on a common attribute.

1). *In-state purge.* A new punctuation received from one input stream can be used to purge the matching tuples from the state of the other stream.

2). *On-the-fly purge.* A new tuple received from one stream doesn't need to be inserted into the state if it matches a punctuation that has been received from the other stream.

3). *Propagation.* A punctuation on a join value val can be propagated to the output stream if a punctuation regarding this join value is received from one stream and no tuple currently in the state of this stream matches this punctuation. The propagated punctuations have the same attribute set as the join results, and may hence be different from the attribute set of the input punctuations. For example, in a join S_1<A, B> $\bowtie$ S_2<B, C>, the input punctuations from stream S_1 and those from S_1 both have two attributes (A and B, and B and C, respectively). However, the propagated punctuations will have three attributes, i.e., A, B and C.

3.3 Sliding Window Join

The *sliding window* constrains the query to only consider the "recent" portions of the streams [12]. There are two types of sliding windows – time-based windows and count-based windows. Below we provide the window join semantics with the more commonly used window type, i.e., the time-based window. The count-based window join will be described in Section 7.5.

The join $S_1 \bowtie S_2$ with time-based sliding windows W_1 on S_1 and W_2 on S_2 is defined as follows: each tuple from S_1 (S_2) with timestamp ts_1 (ts_2) can only join with tuples from S_2 (S_1) that arrived within the last W_2 (W_1) time units prior to ts_1 (ts_2).

The sliding window join algorithm extends the pipelined join algorithm by discarding the tuples that have expired from the current window in a timely manner. As a tuple t_1 with timestamp ts_1 arrives from S_1, it is first *inserted* into the state of S_1. Second, t_1 is used to *remove* tuples from the state of S_2 whose timestamp is less than (ts_1–W_2). Then t_1 is used to *probe* the state of S_2 and the corresponding join results are produced for any matches. In this algorithm, the expired tuples from one stream are purged according to the timestamp of the new tuple from the other stream. This *cross invalidation* strategy guarantees that no spurious duplicate results are produced and no valid result is missing. In summary, the join operator continuously performs the *insert-invalidate-probe* operation sequence to process each input tuple.

Chapter 4

PJoin: Punctuation-Aware Streaming Join Operator

4.1 PJoin Execution Logic

4.1.1 Components and Join State

Components. Join algorithms typically involve multiple subtasks, including: (1) probing the in-memory join state using a new tuple and produce result for any match being found (*memory join*), (2) moving part of the in-memory join state to disk when running out of memory (*state relocation*), (3) retrieving data from disk into memory for join processing (*disk join*), (4) purging no-longer-useful data from the join state (*state purge*) and (5) propagating punctuations to the output stream (*punctuation propagation*).

The frequencies of executing each of these subtasks may be rather different. For example, *memory join* runs on a per-tuple basis, while *state* *relocation* executes only when memory overflows and *state purge* is activated upon receiving one or multiple punctuations. To achieve a fine-tuned, adaptive join execution, we design separate components to accomplish each of the above subtasks. Furthermore, for each component we explore a variety of alternate strategies that can be plugged in to achieve optimization in different circumstances, as further elaborated upon in Section 4.1.2 through Section 4.1.5. To increase the throughput, several components may run concurrently in a multi-threaded mode. Section 4.1.6 introduces our event-based PJoin framework.

Join state. Being a pipelined join operator (Section 3.1), PJoin maintains a separate state for each input stream. All the above components operate on this shared data storage. For each state, a *hash table* holds all tuples that have arrived but have not yet been purged. Similar to XJoin [95], each hash bucket has an in-memory portion and an on-disk portion. When memory usage of the join state reaches a *memory threshold*, some data in the memory-resident portion will be moved to the on-disk portion. A *purge buffer* contains the tuples which should be purged based on the present punctuations, but cannot yet be purged safely because they may possibly join with tuples stored on disk. The purge buffer will be cleaned up by the disk join component. The punctuations that have arrived but have not yet been propagated are stored in a *punctuation set*.

4.1.2 Memory Join and Disk Join

Due to the memory overflow resolution explained in Section 4.1.3 below, for each new input tuple, the matching tuples in the opposite state could possibly reside in two different places: memory and disk. Therefore, the join operation can happen in two components. The *memory join* component will use the new tuple to probe the memory-resident portion of the matching hash bucket of the opposite state and produce the result, while the *disk join* component will fetch the disk-resident portion of some or all the hash buckets and finish the left-over joins due to the state relocation (Section 4.1.3). Since the disk join involves I/O operations which are much more slower than in-memory operations, the policies for scheduling these two components are different. The memory join is executed on a per-tuple basis. Only when the memory join cannot proceed due to the slow delivery of the data or when punctuation propagation needs to finish up all the left-over joins, will the disk join be scheduled to run. Similar to XJoin [95], we associate an *activation threshold* with the disk join to model how urgent it is to be scheduled for execution.

4.1.3 State Relocation

PJoin employs the same memory overflow resolution as XJoin, i.e., moving part of the state from memory to secondary storage (disk) when the memory becomes full (reaches the *memory threshold*). The corresponding component in PJoin is called *state relocation*. Readers are referred to [95] for further details about the state relocation.

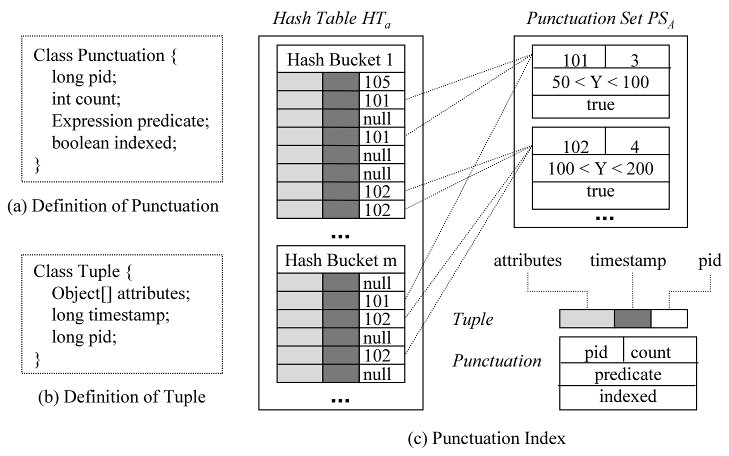
4.1.4 State Purge

The state purge component removes data that will no longer contribute to any future join result from the join state by applying the purge rules described in Section 3.2. We propose two state purge strategies, *eager (immediate) purge* and *lazy (batch) purge*. *Eager purge* starts to purge the state whenever a punctuation is obtained. This can guarantee the minimum memory overhead caused by the join state. Also by shrinking the state in an aggressive manner, the state probing can be done more efficiently. However, since the state purge causes the extra overhead for scanning the join state, when punctuations arrive very frequently so that the cost of state scan exceeds the saving of probing, eager purge may instead slow down the data output rate. In response, we propose a *lazy purge* which will start purging when the number of new punctuations since the last purge reaches a *purge threshold*, which is the number of punctuations to be arriving between two state purges. We can view eager purge as a special case of lazy purge, whose purge threshold is 1. Accordingly, finding an appropriate purge threshold becomes an important task. In Chapter 5 we experimentally assess the effect on PJoin performance posed by different purge thresholds.

4.1.5 Punctuation Propagation

Besides utilizing punctuations to shrink the runtime state, in some cases the operator can also propagate punctuations to benefit other operators downstream in the query plan, for example, the *group-by* operator in Figure 1.1 (c). According to the propagation rules described in Section 3.2, a join operator will propagate punctuations in a *lagged* fashion, that is, before a punctuation can be released to the output stream, the join must wait until all result tuples that match this punctuation have been safely output. Hence we consider to initiate propagation periodically. However, each time we invoke the propagation, each punctuation in the *punctuation sets* needs to be evaluated against all tuples currently in the same state. Therefore, the punctuations which were not able to be propagated in the previous propagation run may be evaluated against those tuples that have already been compared with last time, thus incurring duplicate expression evaluations. To avoid this problem and to propagate punctuations correctly, we design an incrementally maintained *punctuation index* which arranges the data in the join state by punctuations.

Punctuation index. To construct a punctuation index (Figure 4.1 (c)), each punctuation in the punctuation set is associated with a unique ID (pid) and a *count* recording the number of matching tuples that reside in the same state (Figure 4.1 (a)). We also augment the structure of each tuple to add the pid which denotes the punctuation that matches the tuple (Figure 4.1 (b)). If a tuple matches multiple punctuations, the pid of the tuple is always set as the pid of the first arrived punctuation found to be matched. If the tuple is not valid for any existing punctuations, the pid of this tuple is null. Upon arrival of a new punctuation p, only tuples with pid field being $null$ need to be evaluated against p. Therefore the punctuation index is constructed incrementally so to avoid the duplicate expression evaluations. Whenever a tuple is purged from the state, the punctuation whose pid corresponds the pid contained by the purged tuple will decrement its $count$ field. When the *count* of a punctuation reaches 0 which means no tuple matching this punctuation exists in the state, according to Lemma 2 in Section 3.2, this punctuation becomes propagable. The punctuations being propagated are immediately removed from the punctuation set.



```
Class Punctuation {
    long pid;
    int count;
    Expression predicate;
    boolean indexed;
}
```

(a) Definition of Punctuation

```
Class Tuple {
    Object[] attributes;
    long timestamp;
    long pid;
}
```

(b) Definition of Tuple

(c) Punctuation Index

Figure 4.1: Data Structures for Punctuation Propagation.

Algorithms for index building and propagation. We can see that punctuation propagation involves two important steps: *punctuation index building* which associates a punctuation with each tuple in the join state, and *propagation* which outputs the punctuations with the *count* field being zero. Clearly, propagation relies on the index building process. Algorithm 1 below shows the algorithm for constructing a punctuation index for tuples from stream B (Lines 1-14) and the algorithm for propagating punctuations from stream B to the output stream (Lines 16-21).

Eager and lazy index building. Although our incrementally constructed punctuation index avoids duplicate expression evaluations, it still needs to scan the entire join state to search for the tuples whose *pid*s are null each

Algorithm 1 Index-Build-B Procedure

```
ArrayList pIndexSet = new ArrayList();
/* Select all punctuations from B punctuation set $PS_B$ not used for indexing tuples. */
for $p_i$ in $PS_B$ do
  if ! ($p_i$.indexed) then
    pIndexSet.add($p_i$);
  end if
end for
/* Index all tuples in the B hash table $HT_b$ that have not yet been indexed. */
for $bucket_k$ in $HT_b$ do
  for $t_j$ in $bucket_k$ do
    if $t_j$.pid == null then
      for $p_i$ in pIndexSet do
        if match($t_j$, $p_i$) then
          $t_j$.pid = $p_i$.pid; /* assign pid to matching tuple. */
          continue;
        end if
      end for
    end if
  end for
end for
```

Algorithm 2 Propagate-B Procedure

```
/* Output and remove all punctuations whose count field is 0 in $PS_B$. */
for $p_i$ in $PS_B$ do
  if $p_i$.count == 0 then
    output($p_i$); /* Send $p_i$ to output stream. */
    remove($PS_B$, $p_i$); /* Remove $p_i$ from $PS_B$ */
  end if
end for
```

time it is executed. We thus batch the index building for multiple punctuations in order to share the cost of scanning the state. Accordingly, instead of triggering the index building upon the arrival of each punctuation, which we call *eager index building*, we run it only when the punctuation propagation is invoked, called *lazy index building*. However, eager index building is still preferred in some cases. For example, it can help guarantee the steady instead of bursty output of punctuations whenever possible. In the eager approach, since the index is incrementally built right upon receiving each punctuation and the index is indirectly maintained by the state purge, some punctuations may be detected to be propagable much earlier than the next invocation of propagation.

Propagation mode. PJoin is able to trigger punctuation propagation in either *push* or *pull* mode. In the *push* mode, PJoin actively propagates punctuations when either a fixed time interval since the last propagation has gone by, or a fixed number of punctuations have been received since the last propagation. We call them *time propagation threshold* and *count propagation threshold* respectively. On the other hand, PJoin is also able to propagate punctuations upon the request of the down-stream operators, which would be the beneficiaries of the propagation. This is called the *pull* mode.

4.1.6 Event-driven Framework of PJoin

To implement the PJoin execution logic described above, with components being tunable, a join framework which incorporates the following features is desired.

1). The framework should keep track of a variety of runtime parameters that serve as the triggering conditions for executing each component, such as the size of the join state, the number of punctuations that arrived since the last state purge, etc. When a certain parameter reaches the corresponding threshold, such as the purge threshold, the appropriate components should be scheduled to run.

2). The framework should be able to model the different coupling alternatives among components and easily switch from one option to another. For example, the lazy index building is coupled with the punctuation propagation, while the eager index building is independent of the punctuation propagation strategy selected by a given join execution configuration.

To accomplish the above features, we have designed an *event-driven* framework for PJoin as shown in Figure 4.2. The memory join runs as the main thread. It continuously retrieves data from the input streams and generates results. A *monitor* is responsible for keeping track of the status of various runtime parameters about the input streams and the join state being changed during the execution of the memory join. Once a certain threshold is reached, for example the size of the join state reaches the memory threshold or both input streams are temporarily stuck due to network delay and the disk join activation threshold is reached, the monitor will invoke the corresponding event. Then the listeners of the event, which may be either disk join, state purge, state relocation, index build or punctuation propagation component, will start running as a second thread. If an event has multiple listeners, these listeners will be executed in an order specified in the event-listener registry described below.

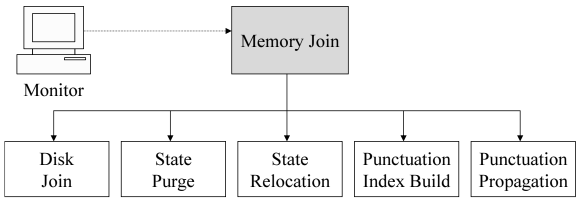


Figure 4.2: Event-Driven Framework of PJoin.

The following events have been defined to model the status changes of monitored *runtime parameters* that may cause a component to be activated.

1). *StreamEmptyEvent* signals both input streams run out of tuples.

2). *PurgeThresholdReachEvent* signals the *purge threshold* is reached.

3). *StateFullEvent* signals the size of the in-memory join state reaches the *memory threshold*.

4). *NewPunctReadyEvent* signals a new punctuation arrives.

5). *PropagateRequestEvent* signals a propagation request is received from down-stream operators.

6). *PropagateTimeExpireEvent* signals the *time propagation threshold* is reached.

7). *PropagateCountReachEvent* signals the *count propagation threshold* is reached.

PJoin maintains an *event-listener registry*. Each entry in the registry lists the event to be generated, the additional conditions to be checked and the listeners (components) which will be executed to handle the event. The registry while initiated at the static query optimization phase can be updated at runtime. All parameters for invoking the events, including the *purge*, *memory* and *propagation threshold*, are specified inside the monitor and can also be changed at runtime.

Table 4.1 gives an example of this registry. This configuration of PJoin is used by several experiments shown in Chapter 5. In this configuration, we apply the *lazy purge* strategy, that is, to purge state whenever the purge threshold is reached. Also the *lazy index building* and the *push* mode propagation are applied, that is, when the count propagation threshold is reached, we first construct the punctuation index for all newly-arrived punctuations since the last index building and then start propagation.

Events	*Conditions*	*Listeners (Activated In Order)*
StreamEmptyEvent	Activation threshold is reached.	Disk Join
PurgeThresholdReachEvent	none	State Purge
StateFullEvent	C1*	State Purge
StateFullEvent	C2*	State Relocation
PropagateCountReachEvent	none	Index Build, Propagation
C1*: There exists punctuations which haven't been used to purge the state. C2*: No punctuations exist that haven't been used to purge the state.		

Table 4.1: Example Event-Listener Registry.

Chapter 5

Experimental Evaluation for PJoin Operator

5.1 Experimental Setup

We have implemented the PJoin operator in the CAPE system [83]. Below we describe the experimental study we have conducted to explore the effectiveness of our punctuation-exploiting stream join optimization. The test machine has a 2.4GHz Intel(R) Pentium-IV processor and a 512MB RAM, running Windows XP and Java 1.4.1.01 SDK. We have created a benchmark system to generate synthetic data streams by controlling the arrival patterns and rates of the data and punctuations. In all experiments shown in this section, the tuples from both input streams have a Poisson inter-arrival time with a mean of 2 milliseconds. All experiments run a many-to-many join over two input streams, which, we believe, exhibits the most general cases of our solution. In the charts, we denote the PJoin with purge threshold n (i.e., maximum number of punctuations allowed between two consecutive purges) as PJoin-n. Accordingly, PJoin using eager purge is denoted as PJoin-1.

5.2 PJoin vs. XJoin

First we compare the performance of PJoin with XJoin [95], a stream join operator without a constraint-exploiting mechanism. We are interested in exploring two questions: (1) how much memory overhead can be saved and (2) to what degree can the tuple output rate be improved. In order to be able to compare these two join solutions, we have also implemented XJoin in our system and applied the same optimizations as we did for PJoin.

To answer the first question, we compare PJoin using the eager purge with XJoin regarding the total number of tuples in the join state during the length of the execution. The input punctuations have a Poisson inter-arrival with a mean of 40 tuples/punctuation. From Figure 5.1 we can see that the memory requirement for the PJoin state is almost insignificant compared to that of XJoin.

As the punctuation inter-arrival increases, the size of the PJoin state will increase accordingly. When the punctuation inter-arrival reaches infinity so that no punctuations exist in the input stream, the memory requirement of PJoin becomes the same as that of XJoin.

In Figure 5.2, we vary the punctuation inter-arrival to be 10, 20 and 30 tuples/punctuation respectively for three different runs of PJoin accord-

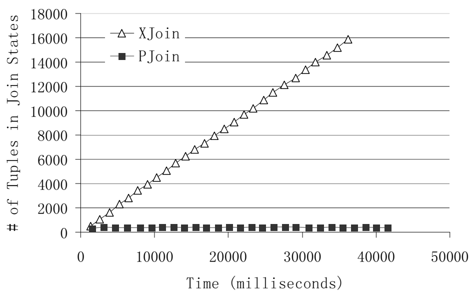


Figure 5.1: PJoin vs. XJoin, Memory Overhead, Punctuation Inter-arrival: 40 tuples/punctuation.

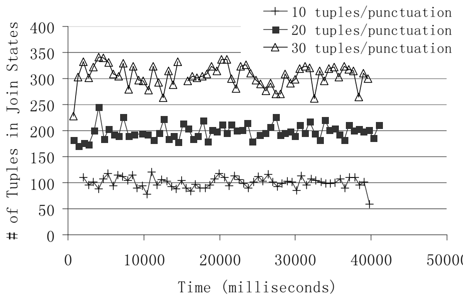


Figure 5.2: PJoin Memory Overhead, Punctuation Inter-arrival: 10, 20, 30 tuples/punctuation.

ingly. We can see that as the punctuation inter-arrival increases, the average size of the PJoin state becomes larger correspondingly.

To answer the second question, Figure 5.3 compares the tuple output rate of PJoin to that of XJoin. We can see that as time advances, PJoin maintains an almost steady output rate whereas the output rate of XJoin drops. This decrease in XJoin output rate occurs because the XJoin state increases over time thereby leading to an increasing cost for probing state. From this experiment we conclude that PJoin performs better or at least equivalent to XJoin regarding both the output rate and the memory resource consumption.

5.3 State Purge Strategies for PJoin

Now we explore how the performance of PJoin is affected by different state purge strategies. In this experiment, the input punctuations have a Poisson inter-arrival with a mean of 10 tuples/punctuation. We vary the *purge*

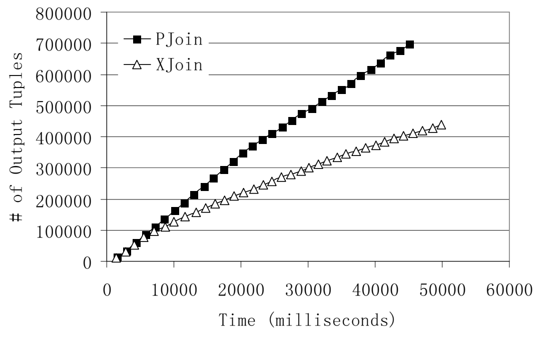
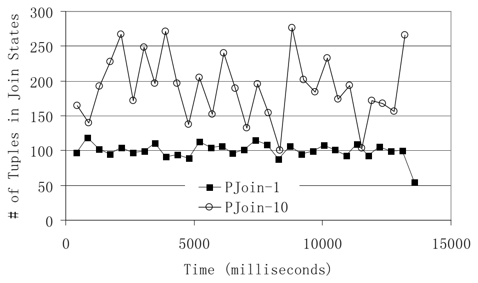


Figure 5.3: PJoin vs. XJoin, Tuple Output Rates, Punctuation Inter-arrival: 30 tuples/punctuation.

threshold to start purging the state after receiving every 10, 100, 400, 800 punctuations respectively and measure its effect on the output rate and memory overhead of the join.

Figure 5.4 shows the state requirements for the eager purge (PJoin-1) and the lazy purge with purge threshold 10 (PJoin-10). The chart confirms that the eager purge is the best strategy for minimizing the join state, whereas the lazy purge requires more memory to operate.



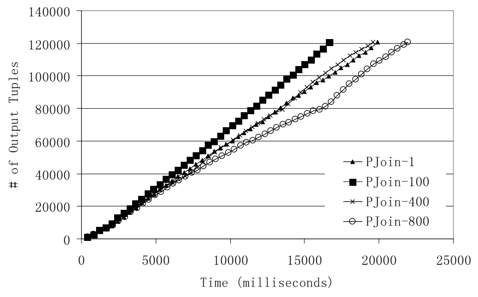


Figure 5.4: Eager vs. Lazy Purge, Memory Overhead, Punctuation Inter-arrival: 10 tuples/punctuation.

Figure 5.5: Eager vs. Lazy Purge, Tuple Output rates, Punctuation Inter-arrival: 10 tuples/punctuation.

Figure 5.5 compares the PJoin output rate using different purge strategies. We plot the number of output tuples against time summarized over four experiment runs, each run with a different purge threshold (1,100,400 and 800 respectively). We can see that up to some limit, the higher the *purge threshold*, the higher the output rate. This is because there is a cost associated with purge, and thus purging very frequently such as using the eager strategy leads to a loss in performance. But this gain in output rate is at the cost of the increase in memory overhead. When the increased cost of probing the state exceeds the cost of purge, we start to lose on performance, such as the case of PJoin-400 and PJoin-800. This is the same problem as encountered by XJoin, that is, every new tuple enlarges the state, which in turn increases the cost of probing the state.

5.4 Asymmetric Punctuation Inter-arrival Rate

Now we explore the performance of PJoin in terms of input streams with asymmetric punctuation inter-arrivals. We keep the punctuation inter-arrival of stream A constant at 10 tuples/punctuation and vary that of stream B. Figure 5.6 shows the state requirement of PJoin using eager purge. We can see that the larger the difference in the punctuation inter-arrival of the two input streams, the larger will be the memory requirement. Less frequent punctuations from stream B cause the A state to be purged less frequently. Hence the A state becomes larger.

Another interesting phenomenon not shown here is that the B state is very small or insignificant compared to the A state. This happens because

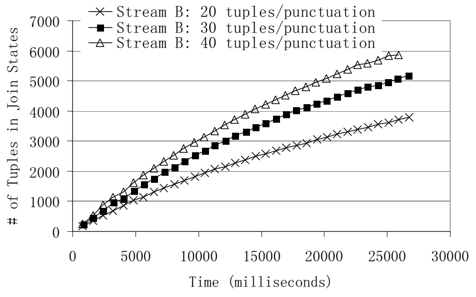


Figure 5.6: Memory Overhead, Asymmetric Punctuation Inter-arrival Rates, A Punctuation Inter-arrival: 10 tuples/punctuation, B Punctuation Inter-arrival: 20, 30, 40 tuples/punctuation.

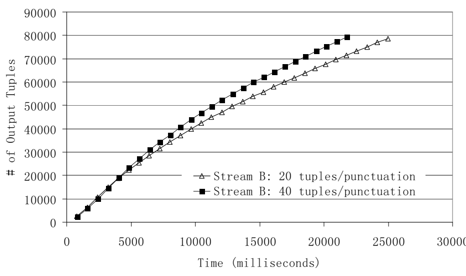


Figure 5.7: Tuple Output Rates, Asymmetric Punctuation Inter-arrival Rates, A Punctuation Inter-arrival: 10 tuples/punctuation, B Punctuation Inter-arrival: 20, 40 tuples/punctuation.

punctuations from stream A arrive at a faster rate. Thus most of the time when a B tuple is received, there already exists an A punctuation that can drop this B tuple on the fly [44]. Therefore most B tuples never become a part of the state.

Figure 5.7 gives an idea about the tuple output rate of PJoin for the above cases. The slower the punctuation arrival rate, the greater is the tuple output rate. This is because the slow punctuation arrival rate means a smaller number of purges and hence the less overhead caused by purge.

Figure 5.8 shows the comparison of PJoin against XJoin in terms of asymmetric punctuation inter-arrivals. The punctuation inter-arrival of stream A is 10 tuples/punctuation and that of stream B is 20 tuples/punctuation. We can see that the output rate of PJoin with the eager purge (PJoin-1) lags behind that of XJoin. This is mainly because of the cost of purge associated with PJoin. One way to overcome this problem is to use the lazy purge together with an appropriate setting of the *purge threshold*. This will make the output rate of PJoin better or at least equivalent to that of XJoin. Figure 5.9 shows the state requirements for this case. We conclude that if the goal is to minimize the memory overhead of the join state, we can use the eager purge strategy. Otherwise the lazy purge with an appropriate purge threshold value can give us a significant advantage in tuple output rate, at the expense of insignificant increase in memory overhead.

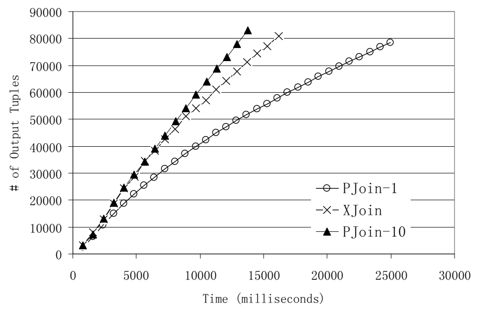


Figure 5.8: Eager vs. Lazy Purge, Output Rates, Asymmetric Punctuation Inter-arrival Rates, A Punctuation Inter-arrival: 10 tuples/punctuation, B Punctuation Inter-arrival: 20 tuples/punctuation.

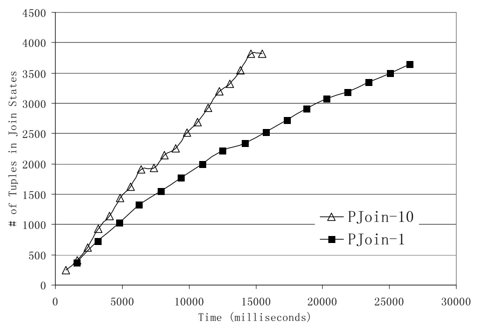


Figure 5.9: Eager vs. Lazy Purge, Memory Overhead, Asymmetric Punctuation Inter-arrival Rates, A Punctuation Inter-arrival: 10 tuples/punctuation, B Punctuation Inter-arrival: 20 tuples/punctuation.

5.5 Punctuation Propagation

Lastly, we test the punctuation propagation ability of PJoin. In this experiment, both input streams have a punctuation inter-arrival with a mean of 40 tuples/punctuation. We show the ideal case in which punctuations from both input streams arrive in the same order and of same granularity, i.e., each punctuation contains a constant pattern. PJoin is configured to start propagation after a pair of equivalent punctuations has been received from both input streams.

Figure 5.10 shows the number of punctuations being output over time. We can see that PJoin can guarantee a steady punctuation propagation rate in the ideal case. This property can be very useful for the down-stream operators such as *group-by* that themselves rely on the availability of input punctuations.

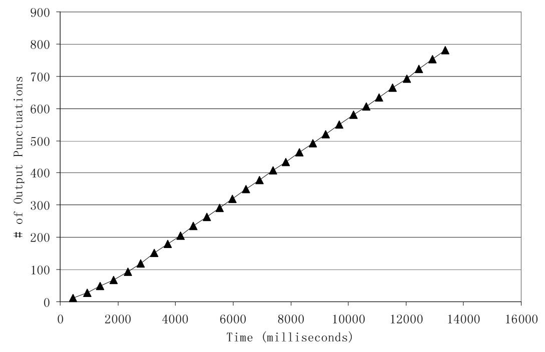


Figure 5.10: Punctuation Propagation, Punctuation Inter-arrival: 40 tuples/punctuation

Chapter 6

PWJoin: Exploiting Combined Constraints in Join Processing

As the starting point, we consider evaluating an equijoin $S_1 \bowtie S_2$ over two relational, possibly punctuated streams S_1 and S_2, in which S_1 and S_2 have exactly one common attribute att, and two time-based sliding windows W_1 and W_2 ($0 \leq W_i < \infty$, i=1,2) are specified on S_1 and S_2 respectively. Later in Chapter 7 we will explain how to extend our algorithm to handle multiway sliding window joins.

6.1 Assumptions

For the discussion of PWJoin, we assume all punctuations have a $constant$ (single-value) pattern on the join attribute and a $wildcard$ pattern on the other attributes [93]. We focus on the single-value punctuations for the ease of exposition of the core concepts. In addition, this is the most general type in terms of discrete domains. The other types can be viewed as shortcuts. For instance, punctuations with a list pattern or a range pattern can be represented by a sequence of punctuations with a single-value pattern. We also assume that no duplicate punctuations occur in a single stream because they would certainly be redundant.

The following two assumptions utilized extensively in the literature [14, 57] will be used in this work. First, besides application-specific attributes, such as *item_id*, every tuple has a timestamp field ts that records the time when this tuple is inserted into the stream. Second, all tuples (from all input streams) have a global ordering on their timestamp and they are processed according to this order.

For ease of presentation, we assume that the join state can always fit into main memory. If this assumption does not hold, we would have to either drop some input tuples (*load shedding*) [68, 91] or move part of the state to the secondary storage (*state relocation*) [75, 77, 98]. However, these techniques, which are being studied extensively in the recent literature, are orthogonal to the problem we solve in this work.

6.2 Design Goals

Sliding window and punctuation are constraints about different aspects of the data, i.e., the timestamp and the application-specific attributes respectively. In many cases the data sets invalidated according to these two types of constraints respectively may overlap with each other. An ill-designed join algorithm that exploits both constraint types may achieve minor gains in memory but incur a possibly doubled search overhead compared to the algorithms that only utilize one constraint type. Overall it may yield even worse performance. Correspondingly, we must carefully design the algorithm. We thus put forth the following goals for our design:

1). If both punctuation and sliding window constraints are present, the join algorithm should achieve better performance regarding both memory overhead and result output rate compared to the algorithms that exploit only one of the constraint types.

2). If no punctuations are available for the stream, our join algorithm should achieve equivalent performance as pure sliding window joins. That is, we wish to avoid penalty of a potentially more complex solution in the cases when the techniques cannot lead to any gains.

In the following, we present our PWJoin approach, the first punctuation-exploiting window join algorithm.

6.3 Optimizations Enabled by Combined Constraints

Either punctuations or sliding windows by themselves can be exploited to shrink the runtime state of the join operator, as explained in Sections 3.2 and 3.3 respectively. The example we discussed in Section 2.2 also shows that if these two types of constraints are simultaneously available, their interactions can enable further optimization opportunities for join processing.

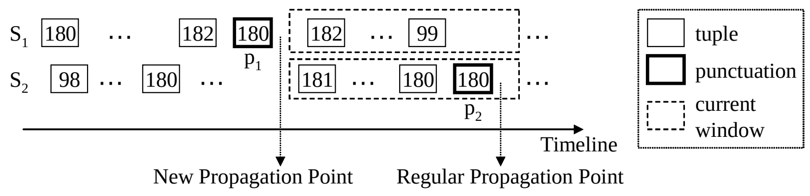


Figure 6.1: Early Punctuation Propagation.

Potentially early punctuation propagation and tuple dropping. Consider the example join execution in Figure 6.1. Punctuations p_1 and p_2 both announce the end of the join value 180. Assume p_1 is received before p_2. According to the propagate invariant for non-windowed joins (Lemma 2), a punctuation on join value 180 can only be propagated after p_1 and p_2 have both been received. However, by in addition considering sliding windows, once the last S_1 tuple containing join value 180 moves out of the window, no more such tuple will appear in the state of S_1. Even if p_2 has not been received yet, we would know that no more result with join value 180 will be generated in the future. Therefore a punctuation on 180 can be propagated at this point, with no need to wait for the arrival of p_2. This new propagation point would be earlier than the one that would be propagated solely based on punctuations (by regular propagate invariant). Due to such early propagation, the downstream operators may in turn be able to make their optimization decisions earlier.

We have derived the following theorem that our proposed window-assisted propagation is based on.

Theorem 1 *Let t be the last tuple from stream S_i (i=1,2) that contains join value*

val. Assume that t has expired from the sliding window of S_i at time T. Then no result tuples that contain join value val will be generated after time T.

Proof. We know that t is the last tuple from S_i with join value val and it has expired from the window at time T. Hence no tuple in the current state nor future arriving tuples of S_i will have the same join value. Therefore, no result with join value val will be generated after T.

The *window-assisted propagate invariant* for PWJoin is stated as follows.

Lemma 4 (Window-Assisted Propagate Invariant). *In a natural join $S_1 \bowtie S_2$ with a common attribute att, the punctuations on the following join values can be propagated at time T.*

$$[p.att \,|\, p \in pset_i \wedge p.ts < T \wedge (\forall t \in tset_i, match(t,p) \rightarrow t.ts < T - W_i),\ i = 1,2]$$

Let's now examine all cases in which a punctuation p_{out} regarding a join value val may be propagated. Assume that two punctuations regarding join value val have been received from streams S_1 and S_2 at time T_1 and T_2 respectively, with $T_1 < T_2$.

1). If no tuple in S_1 ever contained val, p_{out} can be propagated at T_1.

2). If at least one tuple in S_1 contains val and assume the last tuple with this join value in S_1 arrives at T. Apparently T<T_1. There are three cases to consider.

a. If T+W_1<T_1, p_{out} can be propagated at T_1.

b. If T+$W_1 \geq T_1$ and T+$W_1 < T_2$, p_{out} can be propagated at T+W_1.

c. If T+$W_1 \geq T_1$ and T+$W_1 \geq T_2$, p_{out} can be propagated at T_2.

Cases (1), (2)(a) and (2)(c) use the propagate invariant based solely on punctuations, assuming in-state purge and on-the-fly purge are conducted properly. Case (2)(b) is based on the window-assisted propagate invariant. To achieve this propagation, we simply keep track of the expiration of the last tuple that contains each distinct punctuated join value.

In addition, when a window-assisted propagation occurs (suppose it is due to the expiration of a tuple from S_1), tuples containing this join value may still arrive from stream S_2. However, these tuples can then be directly dropped without being processed because they will not find any matches in the state of S_1. This way the join workload can be reduced. The invariant for dropping tuples driven by the window-assisted punctuation propagation is defined below.

Lemma 5 (Tuple Drop Invariant). *In a natural join $S_1 \bowtie S_2$ with a common attribute att, the following tuples can be dropped given that punctuation p is propagated by the window-assisted propagate invariant (Lemma 4) at time T due to the expiration of a tuple from stream S_i.*

$$[t \mid t \in tset_j \wedge j \neq i \wedge match(t, p) \wedge t.ts > T]$$

6.4 PWJoin State Design

To exploit punctuations and sliding windows, the PWJoin algorithm must include three search-based operations: (1) $probe$, that searches the state for matching tuples to produce join results, (2) $purge$, that removes no-longer-joining tuples according to punctuations and (3) $invalidate$, that discards expired tuples based on sliding window semantics. To distinguish the purge of tuples by sliding windows from that by punctuations, henceforth we will use the terms $invalidate$ and $purge$ to name these two operations respectively.

Traditional data structures for maintaining the state for natural joins, such as a chronologically linked list or a hash table, only favor the search in one dimension. For example, maintaining tuples in a chronological list is good for finding expired tuples by window semantics, while managing tuples in a hash table is effective for detecting no-longer-useful tuples by punctuations. Neither of them is appropriate for the exploitation of both constraints. Therefore, we now propose a state design that will effectively serve the needs of PWJoin.

Two-dimensional storage structure. Figure 6.2 shows the storage structure of the PWJoin state. For space reasons we only show the time list for stream S_1. We use a linear list to link all tuples from one stream in chronological order (newest tuple at the end), named *time list*. The head and the tail of the time list are indicated by the *WindowBegin* and the *WindowEnd* pointers respectively. As the window moves, tuples are in turn removed from the head of the time list. Moreover, each set of tuples containing the

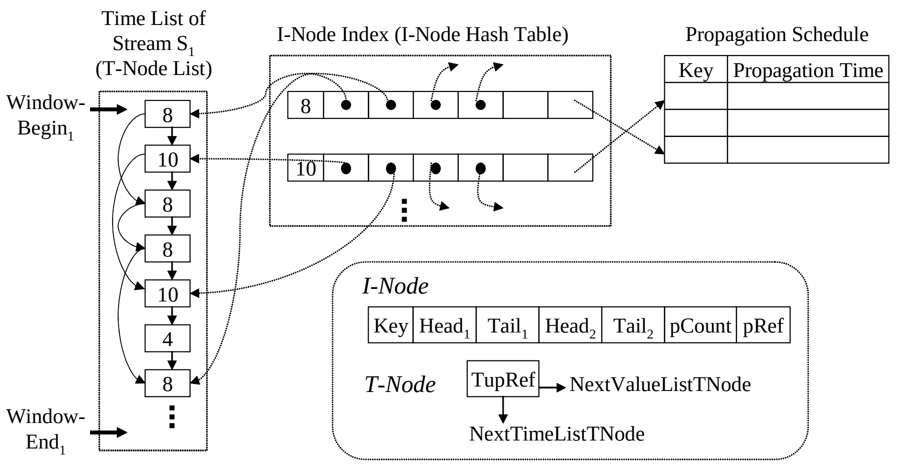


Figure 6.2: PWJoin State Structure.

same join value are linked into a *value list* (also in chronological order). In short, all tuples in the state form a single time list and multiple value lists. Each tuple participates in the time list and exactly one of the value lists. For example, in Figure 6.2, tuples from input stream S_1 compose a single time list <8, 10, 8, 8, 10, 4, 8>, and three value lists <8, 8, 8, 8>, <10, 10> and <4>. A linked list node, which we call the *T-Node* (for Tuple-Node in short), is employed to contain the reference (*tRef*) to the tuple inserted into the state. All tuples in the state are stored in a central storage named *tuple pool*. To optimize for the storage, each tuple in the tuple pool only contains necessary fields for later computations. For example, the join value of each tuple is removed since it can be found in the corresponding I-Node (to be explained below). To keep the figure less cluttered, in Figure 6.2 we didn't show the tuple pool. Instead we display the join value of each tuple in the corresponding T-Node. Each T-Node contains two additional pointers: *NextTimeListTNode* that points to the next T-Node in the same time list, and *NextValueListTNode* that points to the next T-Node in the same value list. This way data are organized along both time dimension and attribute value dimension.

Inter-stream cluster index. To facilitate the search by the join value, we create an index node, named *I-Node*, for each distinct join value to cluster the corresponding value lists from both input streams. Each I-Node contains the following fields:

1). *Key*: join value represented by the I-Node;

2). $Head_i$, $Tail_i$ (i=1,2): pointers to the head and the tail T-Nodes of the value list for stream S_i respectively;

3). *pCount*: number of streams from which the punctuation on Key has been received;

4). *pRef*: pointer to the punctuation on Key in the *propagation schedule* (will be explained later);

For a particular punctuation type, we can use a customized indexing method for organizing I-Nodes. For single-valued or list-valued punctuations, we use a hash-based index while for range-valued punctuations we usually employ a tree-structured index. Since we only focus on single-valued punctuations in this work, the I-Nodes in Figure 6.2 are maintained in a hash table, which we call the *I-Node index*. Most importantly, as will be seen later in Chapter 7, this index structure is easily extendible for effectively handling more generalized join operations. In particular, it is memory-efficient for the multiway join cases since only one index node is created for each distinct join value, regardless of the number of input streams.

Under the above state design, the *probe* and the *purge* operations will search the I-Node index to find the matching value list while the *invalidate* operation will check the head of the time list to detect the expired tuples. Therefore, all three operations perform efficiently because they directly obtain the tuples that they are interested in while the access of irrelevant tuples is avoided. In addition, only two pointers (*Head* and *Tail*) are maintained in the I-Node for each distinct value per stream. Thus little cost for maintaining the index structure is incurred as tuples dynamically enter and leave the state.

To achieve window-assisted propagation, the PWJoin state also maintains a *propagation schedule*. Each item in the schedule contains two fields:

1). Key: the join value announced by an already-received punctuation.

2). $PropTime$: the time a punctuation on Key can be propagated based on the window-assisted propagation invariant.

All items in the propagation schedule are sorted in ascending order of their propagation time. This is to facilitate the window-assisted propagation. When a punctuation is received from a stream S_i (i=1,2), if no item with Key=val exists in the propagation schedule, a new scheduling item with Key=val will be created and inserted into the propagation schedule. The position of this item in the schedule depends on the $PropTime$

value this item, which is computed based on the timestamp of the last tuple t with join value val received from S_i and the window length W_i of S_i ($PropTime = t.ts + W_i$). The last-received tuple with join value val from S_i can be located at the tail of the value list corresponding to S_i that is associated with the I-Node on val. Otherwise, if the propagation schedule has contained a scheduling item with Key=val, a punctuation on the same join value must have been received from the other stream previously. Then the punctuation on this join value can be propagated according to the regular propagate invariant (Lemma 3). The corresponding scheduling item can then be removed from the schedule. An existing propagation scheduling item can also be deleted when the window-assisted propagation condition is satisfied.

In Table 6.1, we list the functions we have designed for accessing all information found in the PWJoin state. These functions will be used in the pseudo-code for the PWJoin algorithms described later on.

6.5 PWJoin Algorithm

Being aware of punctuations, the PWJoin algorithm needs to process two types of objects – regular streaming data (tuples) and punctuations. Since PWJoin conducts symmetric execution logic, we illustrate the processing of data and punctuations from one input stream, say S_1. The pseudo-code is shown in Algorithms 3–6.

Processing data. Once a new tuple t is received from S_1, its timestamp is first used to *invalidate* expired tuples from the time list of stream

Table 6.1: PWJoin State Management Functions.

Functions for PWJoin State	
GetWindowBegin(i):	Get beginning T-Node of time list from stream S_i
GetWindowEnd(i):	Get ending T-Node of time list from stream S_i
CreateINode(key):	Create an I-Node with key key
DeleteINode(key):	Delete an I-Node with key key
GetINode(key):	Get I-Node with key key
GetPropSchedule():	Get propagation schedule
InsertTuple(t, i, inode):	Insert tuple t from stream S_i into corresponding value list associated with $inode$ and time list
Functions for I-Node	
GetHeadTNode(i):	Get T-Node at head of value list from stream S_i
GetTailTNode(i):	Get T-Node at tail of value list from stream S_i
PurgeValueList(i):	Purge the value list corresponding to stream S_j (j≠i)
ClearValueLists():	Delete all tuples from every associated value list
GetPropScheduleItem()	Get propagation scheduling item referenced by I-Node
Functions for T-Node	
GetTuple():	Get tuple referenced by $tupleRef$ field
GetNextTimeListTNode():	Get T-Node next in same time list
GetNextValueListTNode():	Get T-Node next in same value list
Functions for Propagation Schedule	
GetItemAtPos(pos):	Get scheduling item at position pos
AddNewItem(p):	Add a new schedule item based on punctuation p
DeleteItem(key):	Delete schedule item with key key
DeleteItemAtPos(pos):	Delete scheduling item at position pos

S_2 (Algorithm 4, Lines 3–14). This process stops when the first unexpired tuple is encountered, which then becomes the new beginning of this time list. Second, the timestamp of t is used to check whether any values in the propagation schedule can be propagated (Algorithm 4, Lines 15–23). The corresponding propagation scheduling items are then deleted from the list. Meanwhile, the $pRef$ pointer of the I-Node representing this value will be set to *null*. This is for dropping tuples later on.

After the invalidation is done, the join value of t is used to *probe* the I-Node index. If the matching I-Node is not found, a new I-Node will be created for this join value and t will be associated with this I-Node (Algorithm 5, Lines 3–8).

If the matching I-Node $inode$ is found, the value of $pCount$ and $pRef$ will be checked to achieve optimizations enabled by punctuations (Algorithm 5, Lines 9–15). There are three cases to consider:

1). $pCount$=0, which means that no punctuation regarding this join value has been received from either stream. Then t is inserted into the state of S_1.

2). $pCount$>0 and $pRef \neq$ null, which means that a punctuation regarding this join value has been received from stream S_2. But this punctuation has not been propagated yet. The join results are then produced by joining t with all tuples in the value list pointed by the $Head_2$ pointer of $inode$ (Lines 16–21). t is discarded afterwards (on-the-fly purge).

3). $pCount$>0 and $pRef$=null, which means that a punctuation regarding this join value has been received from stream S_2, and it has already been propagated by the window-assisted propagation (Lemma 4). In this case, t is dropped without being processed.

Processing punctuations. When a new punctuation p on a join value val is retrieved from S_1, punctuation-related optimizations are conducted (Algorithm 6). First, p is used to probe the I-Node index. If the matching I-Node $inode$ is not found, no tuple ever in either of the input stream has contained the join value val. A punctuation regarding join value val is propagated. Then a new I-Node $inode$ is created with Key=val and $pCount$=1. This is used to drop the future arriving S_2 tuples containing join value val.

If the matching I-Node $inode$ is found, all tuples in the value list for S_2 are deleted. This is based on the purge invariant (Lemma 1) defined in Section 3.2. Then the $pCount$ field of $inode$ is checked and the following cases are considered.

1). If $pCount$=0, a punctuation on the same join value hasn't been received from the other stream yet. Then $pCount$ is incremented by 1. A new propagation scheduling item with Key=val is created. We retrieve the timestamp of the tail tuple in the associated value list for S_1, assuming it's ts_1. We set the $PropTime$ of the new scheduling item to be (ts_1+W_1). Then this item is placed in the proper position in the propagation schedule. The $pRef$ pointer of $inode$ is pointed to this item.

2). If $pCount>0$ and $pRef \neq$null, the regular propagate invariant (Lemma 3) is satisfied. A punctuation regarding join value val is propagated and the item in propagation schedule pointed to by the $pRef$ of $inode$ is deleted. $inode$ is deleted from the I-Node index afterwards because no more tuple containing this join value will arrive from either input stream.

3). If $pCount>0$ and $pRef$=null, the regular propagate invariant is satisfied. However, a punctuation regarding this join value has already been propagated by the window-assisted propagation. So we simply delete $inode$.

Algorithm 3 BINARY-PWJOIN

```
1: if a tuple $t$ is received from stream $S_i$ then
2:    INVALIDATE(t, i)
3:    PROBE(t, i)
4: else if a punctuation $p$ is received from stream $S_i$ then
5:    PURGE(p, i)
6: end if
```

Algorithm 4 BINARY-PWJOIN-INVALIDATE

```
1: Input: Tuple $t$, Number $sid$
2:
3: for every stream ID $i$ in $\{1, 2\}$ and $i \neq sid$ do
4:    tnode := state.GetWindowBegin(i)
5:    while tnode $\neq$ null do
6:       tHead := tnode.GetTuple()
7:       if tHead.ts $+ W_i <$ t.ts then
8:          tnode.tRef := null
9:          tnode := tnode.GetNextTimeListTNode()
10:      else
11:         break
12:      end if
13:   end while
14: end for
15: propschedule := state.GetPropSchedule()
16: item := propschedule.GetItemAtPos(0)
17: while item.propTime $<$ t.ts do
18:    propagate a punctuation on item.key
19:    inode := state.GetINode(item.key)
20:    inode.pRef := null
21:    PropSchedule.DeleteItemAtPos(0)
22:    item := propschedule.GetItemAtPos(0)
23: end while
```

In this algorithm, the purge operation is triggered by the arrival of punctuations. For data streams carrying no punctuations, the purge operation will never be performed, thus causing zero overhead. In addition, the cost of on-the-fly purge is minimized because it is accomplished as a side effect of the probe operation, i.e., by checking $pCount$ and $pRef$ fields of the matching I-Node. Therefore, we expect that our design enables the

Algorithm 5 BINARY-PWJOIN-PROBE

```
Input: Tuple $t$, Number $sid$

inode := state.GetINode(t.att)
if inode = null then
    inode := state.CreateINode(t.att)
    state.InsertTuple(t, sid, inode)
    return
end if
if inode.pCount $>$ 0 then
    if inode.pRef = null then
        return
    end if
else
    state.InsertTuple(t, sid, inode)
end if
tnode := inode.GetHeadTNode(k) /* k $\neq$ $sid$ */
while tnode $\neq$ null do
    s := tnode.GetTuple()
    join t and s and send the result to output stream
    tnode := tnode.GetNextValueListTNode()
end while
```

Algorithm 6 BINARY-PWJOIN-PURGE

```
Input: Punctuation $p$, Number $sid$

inode := state.GetINode(p.att)
if inode = null then
    inode := CreateINode(p.att)
    inode.pCount ++
    propagate a punctuation on p.att
    return
end if
if inode.pCount $>$ 0 then
    if inode.pRef = null then
        state.DeleteINode(inode.key)
    else
        propagate a punctuation on p.att
        state.GetPropSchedule().DeleteItem(inode.key)
        state.DeleteINode(inode.key)
    end if
else
    inode.pCount ++
    inode.PurgeValueList($sid$)
    state.GetPropSchedule().AddNewItem(p)
end if
```

PWJoin to achieve almost the same performance as the pure window join for non-punctuated streams. In dealing with punctuated streams, on-the-fly purge may provide huge gains by avoiding unnecessary tuple insertions and deletions. In-state purge can also help to effectively shrink the state and hence to improve the probe efficiency. So PWJoin is expected to perform better than the pure window join in most cases, as shown later by our experimental study (Chapter 8).

6.6 State Maintenance

Similar to other stream join algorithms, the PWJoin algorithm involves frequent operations for inserting tuples into and deleting tuples from the state. These operations must guarantee that both the time list and the value lists are updated correctly.

Tuple insertion. Inserting a new tuple t into the state of S_1 follows two steps:

1). A new T-Node $tnode$ is created to contain the reference to tuple t and $tnode$ is appended to the end of the time list (pointed to by the $WindowEnd_1$ pointer). Then the $WindowEnd_1$ pointer is adjusted to point to the new window end, i.e., $tnode$. This can be done within constant time.

2). The join value of t is used to probe the I-Node index. If the matching I-Node $inode$ exists, the $NextValueListTNode$ pointer of the tail T-Node on the value list for S_1 is pointed to $tnode$. The $Tail_1$ pointer of

$inode$ is now updated to point to $tnode$, the new tail of the value list. Else if $inode$ does not exist, a new I-Node is created and both $Head_1$ and $Tail_1$ pointers of this I-Node point to $tnode$.

Tuple deletion. To delete a tuple t from the state of S_1, two cases must be considered: (a) the tuple is deleted by the invalidate operation, (b) the tuple is deleted by the purge operation. Assume that t is represented by a T-Node $tnode$. In case (a), we first remove $tnode$ from the head of the time list by pointing the $WindowBegin_1$ pointer to the next T-Node in the time list. Then we need to adjust the $Head_1$ pointer (sometimes also the $Tail_1$ pointer) with the corresponding I-Node that is currently pointing to $tnode$. However, this will incur an extra probe on the I-Node index to locate the I-Node. And it may become a significant overhead because it happens for every tuple that is invalidated from the time list.

In response, we propose a *lazy* T-Node deletion strategy. After a tuple is removed from the time list by the invalidate operation, we don't immediately adjust the pointers of the corresponding T-Node. Instead we only set $tRef$ of the T-Node to null. Next time when the probe operation accesses the value list associated with this I-Node, all the T-Nodes that contain a null $tRef$, which can all be located at the head of the value list, will be removed from the value list.

Similarly, in case (b), when the purge operation deletes all the T-Nodes from a value list, we only set $tRef$ of these T-Nodes to null. Next time when the time list is probed by the invalidate operation, all the T-Nodes containing a null $tRef$ will be removed from the time list.

In our Java-based implementation, we employ a *T-Node recycle bin* to recycle the T-Nodes that have been deleted both from the time list and from the value list.

Tuple dropping. When a window-assisted propagation is initiated (assuming it is due to a punctuation from S_1 being invalidated from the window), we will not remove the corresponding I-Node from the state of S_1 immediately. Instead, we simply set the $pRef$ pointer of this I-Node to null. Since the tuples with this join value may still arrive from S_2 but they are guaranteed to not be joinable with any future S_1 tuples, we keep this I-Node in order to drop these S_2 tuples. This I-Node will be removed only when the matching punctuation arrives from S_2, i.e., when the regular propagation condition is satisfied (Lemma 3). According to the FcR property to be described in Section 6.7, an I-Node will be removed when the lifespan of the corresponding join value ends.

6.7 Cost Analysis

By exploiting punctuations in addition to sliding windows, the PWJoin algorithm can achieve a more compact state than the pure window join. This is useful when processing queries with large windows over rapid data streams. Now we apply the *unit-time-basis* cost model [68] to derive the formulas for estimating the memory and CPU costs of PWJoin. We then use these formulas to compare the execution cost of PWJoin with the pure window join algorithm.

Before explaining our formulas, we introduce an important property, named *Finite Occurrence Range (FcR) property*, that as we discuss below is assumed by most punctuation-based stream processing applications. In Definition 1, the *lifespan* of a value v for an attribute *att* in a stream S_i, denoted as $L_{v,i}$, is defined to be the time range [$T_{v,i}^{start}$, $T_{v,i}^{end}$]. $T_{v,i}^{start}$ is the time when a tuple with *att*=v occurs in S_i for the first time and $T_{v,i}^{end}$ is the time when a punctuation on *att*=v is announced for S_i. The *unioned lifespan* of two lifespans L_1=[$T_{v,1}^{start}$, $T_{v,1}^{end}$] and L_2=[$T_{v,2}^{start}$, $T_{v,2}^{end}$], i.e., $L_1 \cup L_2$, is defined to be a lifespan [T_v^{start}, T_v^{end}] with T_v^{start}=min($T_{v,1}^{start}$, $T_{v,2}^{start}$) and T_v^{end}=max($T_{v,1}^{end}$, $T_{v,2}^{end}$).

Definition 1 Finite Occurrence Range (FcR) Property. *Let SS=*$\{S_1, ..., S_n\}$ *be a set of streams that have a common attribute att. The attribute att is said to have the finite occurrence range property over SS if every value* v *of att has a finite unioned lifespan over all streams in SS. That is, for every value* v *of att, (*max*(*$T_{v,1}^{end}$*,...* $T_{v,n}^{end}$*) –* min*(*$T_{v,1}^{start}$*, ...* $T_{v,n}^{start}$*)) is a finite value.*

The Auction and the Bid streams in the auction application described in Section 1.1.4 have the FcR property on the *item_id* attribute because the unioned lifespan of each *item_id* value over the two streams equals the lifespan of the corresponding auction. Most transaction-based applications, in which punctuations typically arise, have the FcR property in their streams because the lifespan of a transaction is usually finite.

In response, to simplify our discussion, our cost formulas described below assume the FcR property for the input streams of the join operator. Moreover, to assure predictive behavior, like prior work, we assume an *input-limited mode* [100], i.e., the join operator can always keep up with the workload. Hence at any time T, the number of tuples that have been processed equals the number of tuples that have arrived thus far. This is a reasonable assumption because otherwise the query system would be in an unstable state [11]. We will use the notations defined in Table 6.2 in our cost analysis.

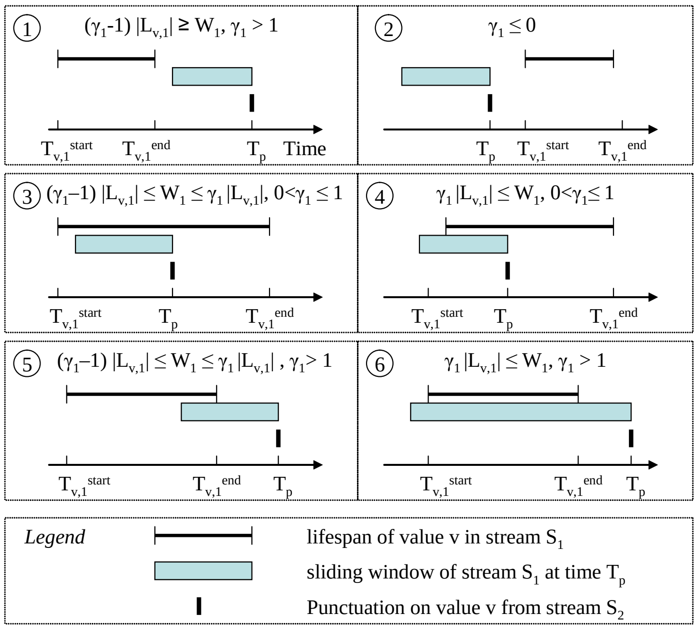
Table 6.2: Additional Notations Used in Cost Analysis.

$Notation$	$Meaning$
λ_i	# of tuples arriving from stream S_i within a time unit (i=1,2)
λ_{pi}	# of punctuations arriving from stream S_i within a time unit (i=1,2)
W_i	time window for stream S_i
$L_{t,i}$	state lifespan of a tuple from stream S_i
$L_{v,i}$	lifespan of a distinct join value in stream S_i
$\|B\|$	# of hash buckets in hash table
M	# of tuples from all streams that have same join value in a window
C_E	cost of conducting an equality match
C_I^t	cost of inserting a T-Node into a linked list
C_D^t	cost of deleting a T-Node from a linked list
C_I^p	cost of inserting a punctuation into propagation schedule
C_D^p	cost of deleting a punctuation from propagation schedule
C_J	average cost of producing join result using a new tuple

Memory overhead. We now analyze the memory required by the PWJoin state in terms of the number of tuples to be maintained in the join state (also called the *state size*). Since PWJoin has a symmetric execution logic, without loss of generality, here we show the state size corresponding to stream S_1, denoted as $|JS_1|$. The state size corresponding to stream S_2 can be estimated similarly.

First we define the *average state lifetime* of a tuple t from a stream S_i (i=1,2), denoted as $|L_{t,i}|$, to be the average time duration a tuple spends in the state. $|JS_1|$ then equals $\lambda_1|L_{t,1}|$. In sliding window joins, $|L_{t,i}|$ equals W_i (for i=1,2) because each tuple will expire from the window after W_i units.

Hence $|JS_1|$ equals $\lambda_1 W_1$.



① $(\gamma_1-1)|L_{v,1}| \geq W_1, \gamma_1 > 1$
$T_{v,1}^{start}$ $T_{v,1}^{end}$ T_p Time
② $\gamma_1 \leq 0$
T_p $T_{v,1}^{start}$ $T_{v,1}^{end}$
③ $(\gamma_1-1)|L_{v,1}| \leq W_1 \leq \gamma_1|L_{v,1}|, 0<\gamma_1 \leq 1$
$T_{v,1}^{start}$ T_p $T_{v,1}^{end}$
④ $\gamma_1|L_{v,1}| \leq W_1, 0<\gamma_1 \leq 1$
$T_{v,1}^{start}$ T_p $T_{v,1}^{end}$
⑤ $(\gamma_1-1)|L_{v,1}| \leq W_1 \leq \gamma_1|L_{v,1}|, \gamma_1 > 1$
$T_{v,1}^{start}$ $T_{v,1}^{end}$ T_p
⑥ $\gamma_1|L_{v,1}| \leq W_1, \gamma_1 > 1$
$T_{v,1}^{start}$ $T_{v,1}^{end}$ T_p
Legend
lifespan of value v in stream S_1
sliding window of stream S_1 at time T_p
Punctuation on value v from stream S_2

Figure 6.3: Cases of Localized Occurrence.

By considering punctuations, the window-based $|L_{t,i}|$ may be shortened because a tuple may be removed from the state by punctuations before it expires from the window. There are six different cases to consider regarding the relationship between the arrival time of the punctuation from stream S_2 and the lifespan of the matching join value [1] in stream S_1, as illustrated in Figure 6.3. In the figure, we use T_p to represent the arrival time of the corresponding punctuation from stream S_2. We show the window of stream S_1 at time T_p, with length W_1. We use $|L_{v,1}|$ to denote the length

[1]Under the FcR property, each join value has a limited lifespan. If this property doesn't hold, not all join values will correspond to a punctuation. Then in the worst case the state size equals the state size of the pure window join.

of lifespan $L_{v,1}$. $|L_{v,1}|$ then equals ($T_{v,1}^{end}$–$T_{v,1}^{start}$). We use γ_1 to denote the *punctuation lagging rate*, which is defined to be $\frac{T_p - T_{v,1}^{start}}{|L_{v,1}|}$.

Equation 6.1 computes $|L_{t,1}|$ for the six cases. Cases 1 and 2 are the worst and the best cases respectively. In the best case, no S_1 tuple ever needs to be maintained in the state. In the worst case, the state size equals the state size of the pure window join. i.e., $\lambda_1 W_1$. In the other four cases, $|L_{t,1}| < W_1$, thus resulting in certain memory cost reduction by exploiting punctuations. We can see that the state size has a positive correlation with γ_1 and a negative correlation with W_1. This is because the earlier the punctuation is received, the less time the matching tuples from the other stream need to spend in the state. Moreover, the bigger the window is, the more memory cost may be potentially saved.

$$|L_{t,1}| = \begin{cases} W_1, & (\gamma_1 - 1)|L_{v,1}| \geq W_1, \gamma_1 > 1 \\ 0, & \gamma_1 \leq 0 \\ \gamma_1 W_1 - \frac{W_1^2}{2 \cdot |L_{v,1}|}, & (\gamma_1 - 1)|L_{v,1}| \leq W \leq \gamma_1 |L_{v,1}|, 0 < \gamma_1 \leq 1 \\ \frac{\gamma_1 |L_{v,1}|}{2}, & \gamma_1 |L_{v,1}| \leq W, 0 < \gamma_1 \leq 1 \\ \gamma_1 W_1 - \frac{1}{2}(\frac{W_1^2}{|L_{v,1}|} - (\gamma_1 - 1)^2 |L_{v,1}|), & (\gamma_1 - 1)|L_{v,1}| \leq W \leq \gamma_1 |L_{v,1}|, \gamma_1 > 1 \\ (\gamma_1 - \frac{1}{2})|L_{v,1}|, & \gamma_1 |L_{v,1}| \leq W, \gamma_1 > 1 \end{cases} \tag{6.1}$$

Another interesting observation is that the state size is independent of the punctuation arrival rate. This is because even if punctuations arrive at high speed, if all tuples they can purge have already expired from the window, then no memory can be saved.

CPU cost. Next, we estimate the *unit time CPU cost* of the PWJoin algorithm, i.e., the CPU time used to process tuples and punctuations that arrive within a time unit. Again, due to the symmetric execution logic, we only show the cost formula related to stream S_1.

We first compute the cost for processing a single tuple from stream S_1, denoted as $C_{t,1}^{PW}$. It includes the cost for probing the state for matching tuples from stream S_2, and the insertion and the deletion cost of the S_1 tuple. The state probing incurs a hash lookup in the I-Node index, which is a hash table that contains I-Nodes. Since each I-Node corresponds to a distinct join value, if they are $|B|$ hash buckets in the I-Node index and in a window, on average M tuples from all streams having same join value, the average hash bucket size of the I-Node index is $\frac{|JS_1|+|JS_2|}{|B|M}$. Since the hash lookup can stop once the matching I-Node is found, the average number of I-Nodes being accessed for each hash lookup is $\frac{|JS_1|+|JS_2|}{2|B|M}$.

In addition, each T-Node representing a tuple participates in two linked lists, i.e., the time list and the value list. Hence double insertion and deletion costs are incurred for tuples that must be inserted into the state, i.e., the tuples that don't satisfy the on-the-fly purge condition. We use α_1 to denote the probability for a tuple from stream S_1 to be purged on the fly. α_1 equals max(0, 1-γ_1) when $\gamma_1 \geq 0$, i.e., when the punctuation from stream S_2 arrive after the lifespan of the matching join value in stream S_1 has starts. α_1 equals 1 otherwise.

Finally, the cost for producing joined results using the matching tuples should be included. Since this cost remains the same for different join algorithms, we simply represent it by C_J. Equation (6.2) computes the single tuple processing cost of PWJoin related to stream S_1.

$$C_{t,1}^{PW} = \frac{|JS_1| + |JS_2|}{2|B|M} C_E + 2(1 - \alpha_1)(C_I^t + C_D^t) + C_J \tag{6.2}$$

Secondly, we consider the cost for processing a single punctuation from stream S_1, denoted as $C_{p,1}^{PW}$. The processing of a punctuation incurs a hash lookup in the I-Node index. Among punctuations received from both input streams on each distinct join value, only one of them incurs the insertion and deletion effort. We use β_1 to represent the probability for a punctuation from stream S_1 to cause the insertion and deletion costs. Since we assume that a single stream contains no duplicate punctuations (Section 6.1), we have $0 \leq \beta_1 \leq 1$. Equation (6.3) computes the unit punctuation processing cost of PWJoin related to stream S_1.

$$C_{p,1}^{PW} = \frac{|JS_1| + |JS_2|}{2|B|M} C_E + \beta_1(C_I^p + C_D^p) \tag{6.3}$$

Equation (6.5) computes the unit time CPU cost of PWJoin, i.e., C_1^{PW}.

$$\begin{aligned} C_1^{PW} &= \lambda_1 C_{t,1}^{PW} + \lambda_{p1} C_{p,1}^{PW} \\ &= (\lambda_1 + \lambda_{p1}) \frac{|JS_1| + |JS_2|}{2|B|M} C_E + \lambda_1(2(1 - \alpha_1)(C_I^t + C_D^t) + C_J) \\ &\quad + \lambda_{p1}\beta_1(C_I^p + C_D^p) \qquad (6.4) \\ & \qquad (6.5) \end{aligned}$$

Now let's compute the unit time CPU cost of a hash-based window join operator [68]. Again we consider the popular window join algorithms that employ a separate hash table to maintain the state for each input stream. To facilitate the invalidation of tuples based on sliding windows, we assume that tuples in each hash bucket are linked in chronological order. The hash lookup for processing a tuple from stream S_1 needs to access all tuples in a hash bucket of the state for stream S_2. Since the state for stream S_2 will contain $\lambda_2 W_2$ tuples on average, the cost is $\frac{\lambda_2 W_2}{|B|} \cdot C_E$, assuming that the hash table contains $|B|$ buckets. In addition, every tuple incurs a single insertion and deletion cost, and the cost for producing joined results. Equation (6.6) computes the CPU cost of hash window join for processing tuples and punctuations from stream S_1 that arrive within a time unit, i.e., C_1^{HW}.

$$C_1^{HW} = \lambda_1 \cdot (\frac{\lambda_2 W_2}{|B|} \cdot C_E + C_I^t + C_D^t + C_J) \tag{6.6}$$

When comparing the cost of PWJoin with the cost of the hash window join, we note that the PWJoin has an additional cost for processing punctuations. In addition, for each tuple that doesn't satisfy the on-the-fly purge condition, more insertion and deletion costs are incurred. However, punctuations usually occur much less frequent than regular tuples. If γ_i is less than or equal to 0.5, e.g., as could be achieved by cases 2, 3 and 4 in Figure 6.3, the insertion and deletion costs of PWJoin will be no greater than the hash window join. More importantly, the dominating cost for processing tuples is the hash lookup cost. When M is large and $|JS_i|$ is small, i.e., when the number of distinct join values is small in a window, the hash lookup cost of PWJoin would be much lower than that of the hash window join. Therefore, in most cases, we expect PWJoin to yield better performance than hash window join. Our experiment results reported in Chapter 8 using various tuple/punctuation workloads indeed confirm the behavior estimated here.

Chapter 7

Generalized PWJoin Algorithm

Chapter 6 dealt with the base case, i.e., a binary join with time-based windows. We now generalize our PWJoin solution to handle a large variety of streaming join queries. In particular, we will generalize our solution in the following two aspects:

1). Instead of limiting the number of input streams to be 2, we now explore constraint-exploiting strategies for n-way joins with $n \geq 2$.

2). Besides time-based windows, we want to incorporate the support for other important window types, in particular, count-based windows [12].

We describe our multiway PWJoin solution in Sections 7.1–7.4, and our solution to support count-based windows in Section 7.5.

7.1 Multiway Join Operator

In many cases a continuous query may contain a multiway join [14, 57, 98]. Consider the following query from the online auction application. For each category, it reports the total number of bids on all the items that belong to this category within 24 hours of each item's opening. This query contains a three-way join, Auction $\bowtie$ Bid $\bowtie$ Category, on a common attribute *item_id*.

```
Select     C.category_id, C.category_name, count(*)
From       Auction A [Range 24 Hours], Bid B, Category C
Where      A.item_id = B.item_id And B.bidder_id = C.item_id
Group by   C.category_id
```

There are many ways of evaluating an n-way join query. The two extreme choices are (1) using a tree of binary joins and (2) employing a single n-way join operator. Clearly, query plans can also be designed to be composed of a mixture of both binary and multiway join operators based on a cost model. Prior research [98] has shown that in certain cases a multiway pipelined join operator produces outputs sooner than any trees composed of binary joins. This is because the multiway join operator treats its inputs symmetrically. Hence, a new tuple from any input stream can be used to generate and propagate results in a single step, without having to pass the intermediate results through a multi-stage binary execution pipeline. This symmetric execution also enables flexible join ordering, thus reducing the need for expensive runtime plan reorganization [104]. In response, we now extend our design of the constraint-exploiting strategies to cover not only binary joins but also multiway joins. With more streams involved in the join, additional care needs to be taken to guarantee the safe state purge and correct punctuation propagation. We also show that state purge can be done more efficiently in such a mega operator than being applied separately to several binary joins.

In summary, we consider evaluating the join $S_1 \bowtie ... \bowtie S_n$ over n relational, possibly punctuated streams S_1, ... S_n, in which the n input streams have exactly one common attribute att, and n time-based sliding windows W_1, ..., W_n ($0<W_i<\infty$, $1\leq i\leq n$) are specified on the n streams respectively. We consider this join scenario because it covers the most commonly occurring join queries, i.e., joining on a common key or a foreign key, such as the example provided above. In addition, we have observed that a great optimization opportunity exists for this type of queries, as will be shown in Section 7.2. If a multi-join query contains joins on different attributes, we can always group joins based on the common join attributes and then use the proposed multiway PWJoin operator to process each of the join groups.

7.2 Issues and Solutions for Evaluating Multiway Joins

The generalization of the PWJoin algorithm includes the solutions to several new problems that arise in the evaluation of multiway sliding window joins over punctuated streams.

Improving probing efficiency. The existing multiway join approaches [57, 98] employ a hash table or a linear list as their state structure to maintain tuples from each input stream. Thus compared to a binary join, an n-way join may incur (n-1) times the probing cost for finding matching tuples in processing a new input tuple. When the join selectivity is low, a lot of accesses to irrelevant tuples may occur.

To gain an insight into this problem, we now show an example execution of a 4-way hash join in Figure 7.1. A new tuple with join value 8 from stream S_4 will probe the hash tables for streams S_1, S_2 and S_3. It then joins with the matching tuples in the corresponding hash buckets in these three hash tables. In this example let's assume it is hash bucket k. Hence the processing of this tuple costs 3 hash lookups (for locating the hash bucket k in the hash tables for streams S_1, S_2 and S_3) and 12 tuple accesses, including 8 accesses of irrelevant tuples. This can clearly be inefficient.

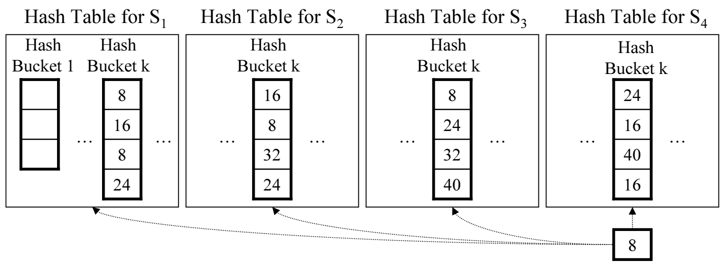


Figure 7.1: MultiWay Hash-Based Join.

The cluster index we have proposed for the binary PWJoin in Section 6.4 helps to solve this problem. To handle an n-way join, we extend the I-Node structure to contain n $Head$ pointers and n $Tail$ pointers that are used to locate the head and the tail of the n value lists (for n input streams) associated with this I-Node. This way the processing of each input tuple only incurs one hash lookup since the matching tuples from all input streams can be located by the same I-Node.

Early failing stragegy. In addition, we add a $vCount$ field into each I-Node to record the number of non-empty value lists associated with this I-Node. This field enables the *early failing strategy* that avoids further probes that won't produce any result. This strategy works as follows. When a new tuple t is received from stream S_i, it is first inserted into the state of S_i. Second, the join value of t is used to retrieve the matching I-Node from the I-Node index. Then the $vCount$ field of this I-Node is checked. If its value is less than n, i.e., the number of input streams, this means that at least one input stream S_j (j≠i, because S_i has tuple t currently in the join state) does not have matching tuples currently in the state. In this case, the probe operation can be skipped because we know that at this time, no join result can be produced using t_i. The $vCount$ value is updated whenever an empty value list associated with this I-Node becomes non-empty or vice versa.

If the number of input streams is large, the probe cost of the PWJoin using the cluster index can be much lower than using a separate index for each stream. The probe efficiency can thus be significantly improved. Moreover, the $vCount$ field helps us to immediately detect the probes that won't produce any result. Otherwise, in the processing of a new tuple, a significant effort may have already been made in assembling partial result tuples before the join detects that the answer set will be empty (because some later-probed streams don't have matching tuples in their current states). Hence, the $vCount$ field helps to completely avoid the probing that will lead to an empty answer set. Accordingly, it eliminates the need for determining the probe sequence based on the join selectivity [57, 98].

Generalizing punctuation-based purge and propagate invariants. To exploit punctuations in multiway join processing, the purge and the propagate invariants need to be generalized. For example, in a binary join, a punctuation received from one input stream can be used immediately to purge the matching tuples from the state of the other input stream. However, in an n-way join where n>2, a punctuation received from one input stream cannot be used alone to safely purge any tuples from other streams. Additional conditions have to be satisfied to ensure a safe purge. Otherwise some join results may be missed. We have designed the following purge and propagate invariants for multiway joins. They are the respective generalizations of the three invariants described in Section 3.2 for binary joins, i.e., we can derive those invariants for binary joins by setting n=2. These invariants hold regardless of the presence of the window.

Lemma 6 (Purge Invariant for N-Way Join). *In an n-way join $S_1 \bowtie ... \bowtie S_n$ with a common attribute att, the following tuples are no longer useful and can be discarded from the state.*

$$[t \mid t \in tset_i \wedge \forall j \in [1,n] \wedge j \neq i, setMatch(t, pset_j)]$$

Lemma 7 (Propagate Invariant for N-Way Join). *In an n-way join $S_1 \bowtie ...$ $\bowtie S_n$ with a common attribute att, the punctuations on the following join values can be propagated.*

$$setNomatchPset(tset_1, pset_1) \cup ... \cup setNomatchPset(tset_n, pset_n)$$

Lemma 8 (Regular Propagate Invariant for N-Way Join). *In an n-way join $S_1 \bowtie ... \bowtie S_n$ with a common attribute att, the punctuations on the following join values can be propagated.*

$$[v|\, \forall i \in [1, n], v \in vpset_i]$$

In addition, similar to the binary PWJoin, if sliding windows are specified on punctuated streams, further optimizations including potentially early punctuation propagation and tuple dropping can be achieved by multiway joins. The window-assisted propagate invariant for multiway joins is same as for binary joins. We can also define the following Purge-and-Drop invariants for multiway joins.

Definition 2 Purge-and-Drop Invariant for Multiway Join. *If at time T, a punctuation p is propagated by the window-assisted propagate invariant due to the expiration of a tuple from stream S_i, the following tuples can be purged from the state.*

$$[t \mid t \in tset_j \wedge j \neq i \wedge match(t, p) \wedge t.ts \leq T]$$

In addition, the following tuples can be dropped with no probing and insertion attempts being necessary.

$$[t \mid t \in tset_j \wedge j \neq i \wedge match(t, p) \wedge t.ts > T]$$

Conducting timely window-assisted punctuation propagation. In binary joins, for a particular join value val, window-assisted punctuation propagation can only happen when (1) only one punctuation on this value has been received from an input stream and (2) the last tuple t that contains join value val from this stream has expired from the window. When a second punctuation on join value val is received from the other stream before t expires from the window, the regular punctuation propagation condition will be satisfied (Lemma 3). Then window-assisted punctuation propagation for this join value will not occur. Therefore, an item in the propagation schedule is created when the first punctuation on this join value is received and its propagation time will remain unchanged during its lifetime in the schedule.

However, in a multiway join, this is not always the case. Before a window-assisted propagation can happen for a join value val, more than one punctuation on val may have arrived from different streams. Again, a propagation scheduling item for val will be created upon receiving the first punctuation on val. However, its propagation time may be brought forward upon receiving a punctuation on val from another stream, depending on the arrival time of the last tuple containing val and the length of the sliding window (recall that sliding windows on different streams may last for different time durations). For example, suppose the last tuple t_i from stream S_i that contains join value val is received at time T_i and the last tuple t_j from stream S_j (j$\neq$i) that contains the same join value val is received at time T_j. Then t_i and t_j will expire at times T_i+W_i and T_j+W_j respectively. Given different values of T_i, W_i, T_j and W_j, either one of these two tuples may expire earlier than the other one. If we keep the first computed propagation time unchanged, the opportunity for conducting propagation and tuple dropping earlier may be missed. As a consequence, more processing overhead may be incurred.

In order to conduct window-assisted propagation and tuple dropping at the earliest possible time, we adjust the processing of punctuations in the multiway PWJoin algorithm as follows. As a punctuation p on a join value val is received from stream S_i, we first find the I-Node with key val in the I-Node index. Then we check this I-Node. If 0$<pCount<$n-1 and $pRef$ is not null, the punctuation regarding this join value has arrived from at least one of the other streams. But the regular propagate invariant (Lemma 8) is not satisfied. We get the last tuple in S_i with join value val by following the $Tail_i$ pointer of the I-Node and compute its expiration time as the new potential propagation time of this join value. Then we check the recorded propagation time of this join value by following the $pRef$ pointer of the I-Node. If the recorded propagation time is later than the newly computed propagation time, we update it by the new time and move this punctuation to the appropriate place in the propagation schedule. Recall that items in propagation schedule are ordered by their propagation time. In all the other cases, the operations remain same as in the binary PWJoin algorithm.

The inter-stream cluster index also facilitates the optimizations enabled by the constraints, i.e., to purge and to drop useless tuples. When the window-assisted propagation is conducted for a specific join value val, we obtain the corresponding I-Node and delete all the tuples associated with this I-Node. Then the $vCount$ value of this I-Node is set to 0 and the $pRef$ pointer is set to null. These help to drop any future-arriving tuples containing this join value.

Managing I-Nodes. In the inter-stream cluster index structure, each I-Node represents a distinct join value. Tuples that contain the same join value are associated with the same I-Node. The creation of an I-Node is triggered by the arrival of the first tuple that contains a particular join value. If the join attribute distributes over a wide domain, the I-Node index will maintain a large number of I-Nodes. However, in many cases, only a subset of these I-Nodes are active at a time. For example, some I-Nodes may be detected to be no longer useful since no more tuples containing those join values will arrive in the future. Then keeping these I-Nodes will cost extra memory and also affect the probe efficiency of the index. Therefore, we aim to detect and eliminate the inactive I-Nodes from the I-Node index in a timely manner.

The current implementation of the multiway PWJoin applies a conservative I-Node deletion policy. According to the Propagate Invariant II for Multiway Join (Definition 8), when the punctuation on a particular join value has been received from all input streams, it is guaranteed that no more tuples containing this join value will arrive in the future. Then the corresponding I-Node will be removed from the I-Node index. We can see that this condition is fairly restrictive since it requires that all input streams contain punctuations on the join value. There are other approaches available too. For example, as tuples are being expired from sliding windows, some I-Nodes may be associated with zero tuples. We call such I-Node the *loner*. We can keep track of the loners. If they keep being loners for a sufficiently long time, it may be beneficial to delete them from the I-Node index. Studying alternate I-Node deletion policies remains as future work.

7.3 Multiway PWJoin Algorithm

The pseudo code of the multiway PWJoin algorithm is shown in Algorithms 7 and 8. We only present the procedures that contain changes to the binary PWJoin algorithm. These changes are underlined in the code.

Algorithm 7 MULTIWAY-PWJOIN-PROBE

```
Input: Tuple $t$, Number $sid$

inode := state.GetINode(t.att)
if inode = null then
    inode := CreateINode(t.att)
    state.InsertTuple($t$, $sid$, inode)
    return
else if $\underline{\text{inode.vCount} < \text{n}}$ then
    if inode.pCount $>$ 0 and inode.pRef = null then
        return
    else
        state.InsertTuple($t$, $sid$, inode)
        return
    end if
end if
output results from joining t with all tuples associated with inode
if $\underline{\text{inode.pCount} < \text{n-1}}$ then
    $\underline{\text{state.InsertTuple}(t, sid, \text{inode})}$
end if
```

Algorithm 8 MULTIWAY-PWJOIN-PURGE

```
Input: Punctuation $p$, Number $sid$

inode = state.GetINode(p.att)
if inode = null then
    inode := state.CreateINode(p.att)
    inode.pCount ++
    propagate a punctuation on p.att
    return
end if
if inode.pCount = n-1 then
    if inode.pRef = null then
        state.DeleteINode(inode.key)
    else
        propagate a punctuation on p.att
        state.GetPropSchedule().DeleteItem(inode.key)
        state.DeleteINode(inode.key)
    end if
else
    inode.pCount ++
    t := inode.getTailTNode($sid$).getTuple()
    newPropTime := t.ts + $W_{sid}$
    ScheduleItem it := inode.GetPropScheduleItem()
    if it.PropTime > newPropTime then
        it.PropTime := newPropTime
    end if
end if
```

7.4 Cost Analysis

As described in Section 7.2, the punctuation-driven purge in the multiway PWJoin is different than in the binary PWJoin. Specifically, in a binary join, a punctuation can be used to purge tuples from the other stream as soon as it has been received. However, in multiway joins, the purge of tuples from a stream needs to wait until either a matching punctuation expires from the window (purge scenario 1), or the matching punctuation has been received from all the other streams (purge scenario 2).

Equation 6.1 can still be used to compute the tuple lifespan for the multiway PWJoin. However, the definition of γ_i needs to be modified. In terms of the purge scenario 1, T_p in Figure 6.3 should be replaced by the *earliest* time a punctuation on the corresponding join value (with lifespan $L_{v,1}$) expires from the window. That is, T_p = $A_{pi} + W_i$, with A_{pi} being the arrival time of the expired punctuation (assuming it is from stream S_i and i $\neq$ 1).

In terms of purge scenario 2, T_p in the figure should be replaced by the arrival time of the *last* punctuation on the corresponding join value (with lifespan $L_{v,1}$).

The unit time CPU cost of the multiway PWJoin can be derived in a similar manner as the binary PWJoin, as shown in Equation 7.1. Note that α_1 needs to be computed using the new γ_1 value.

$$\begin{aligned} C_1^{PW} &= \lambda_1 C_t^{PW} + \lambda_{p1} C_p^{PW} \\ &= (\lambda_1 + \lambda_{p1}) \frac{\sum\limits_{i=1}^{n} |JS_i|}{2|B|M} C_E + \lambda_1 (2(1-\alpha_1)(C_I^t + C_D^t) + C_J) \\ &\quad + \lambda_{p1} \beta_1 (C_I^p + C_D^p) \end{aligned} \tag{7.1}$$

7.5 Handling Count-Based Windows

Besides time-based windows, another type of window, namely count-based windows [12], also appear commonly in the streaming applications. For example, the following query continuously reports the total number of bids for each of the last 10 opened auctions.

```
Select      A.item_id, count(*)
From        Auction A [Rows 10], Bid B
Where       A.item_id = B.item_id
Group by    A.item_id
```

The count-based sliding window join is defined as follows. The join S_1 $\bowtie$ ... $\bowtie$ S_n with count-based sliding windows C_i on S_i (0≤C_i<∞, 1≤i≤n) is defined as follows: a tuple t from S_i can only join with the last C_j tuples from S_j (1≤j≤n, j≠i) that arrived prior to t.

In a time-based window join, whether a tuple belongs to the current window depends on the relationship between its timestamp and the timestamp of the latest tuple received from other streams. Hence the number of tuples that reside in a window may vary over time under varying data arrival rates. In count-based windows, however, the number of tuples in a window remains constant. Due to this difference, the following changes need to be made to the time-based PWJoin algorithm in order to correctly process count-based window joins.

Invalidation. The time-based PWJoin algorithm uses cross invalidation (see Section 3.3), i.e., a new tuple from one input stream is used to invalidate tuples from other streams that have expired from the current window. To deal with the count-based window, *self purge* should be applied instead. That is, whenever a new tuple is received from an input stream, the oldest tuple in the time list of the same stream is removed.

Avoiding window sliding disorder problem. When considering punctuations in addition to sliding windows, some tuples may be purged from the state before they expire from the window. This doesn't cause any problem for the time-based window join because these tuples won't contribute to join results anyway. However, this may introduce a new problem for the count-based window join. This is because the tuple count within the state may be affected prematurely by such a punctuation-driven purge, thereby producing spurious join results.

As illustrated in Figure 7.2, assume both streams have a count-based window of 10 tuples and at time T_{12}, tuples b_4 and b_7 have been purged by punctuations. The correct window of stream S_b at this time should contain tuples b_3 through b_{12}. However, if we don't remember the fact that b_4 and b_7 have been purged and blindly maintain 10 tuples for each of the streams, tuples b_1 and b_2 will be mistakenly kept in the current window. This may produce incorrect results. We call this the *window sliding disorder problem*.

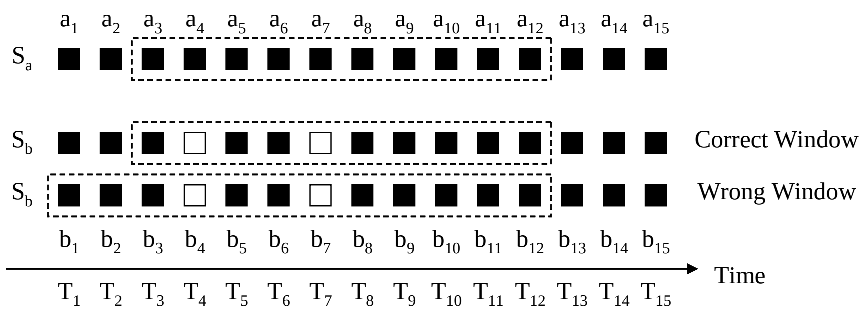

a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8 a_9 a_{10} a_{11} a_{12} a_{13} a_{14} a_{15}
S_a
S_b
Correct Window
S_b
Wrong Window
b_1 b_2 b_3 b_4 b_5 b_6 b_7 b_8 b_9 b_{10} b_{11} b_{12} b_{13} b_{14} b_{15}
Time
T_1 T_2 T_3 T_4 T_5 T_6 T_7 T_8 T_9 T_{10} T_{11} T_{12} T_{13} T_{14} T_{15}

Figure 7.2: Example of Window Sliding Disorder With Count-Based Window.

To avoid this problem, we adjust the PWJoin algorithm as follows. Each time a new tuple is received, it is used to invalidate tuples from the time list of the same stream. As explained before, exactly one tuple from this stream, which is referenced by the T-Node pointed to by the $WindowBegin$ pointer, is invalidated. There are two cases to be considered.

1). If $tRef$ of this T-Node is not null, which means that the tuple being referenced hasn't been purged by any punctuation yet. We will delete this tuple and set $tRef$ to null. Then we point $WindowBegin$ to the next T-Node in the time list. We still keep this T-Node due to the lazy T-Node deletion policy described in Section 6.6.

2). If $tRef$ of this T-Node is null, which means the tuple being referenced has already been purged by a punctuation, we will put this T-Node to the T-Node recycle bin (Section 6.6) and then point the $WindowBegin$ pointer to the next T-Node in the time list.

In addition, if a newly-received tuple satisfies the *on-the-fly purge* condition, it won't be inserted into the state. However, we still keep a T-Node in the time list for this tuple as a place holder. This T-Node does not reference to any tuple and it won't be inserted into any value list.

Chapter 8

Experimental Evaluation for PWJoin Operator

8.1 Experimental Setup

We have implemented the PWJoin algorithm in the CAPE system [83] to execute the sliding window join. We now report on an extensive experimental study we have conducted to explore the effectiveness of the constraint-exploiting join optimization strategies. Below we show some of the main results obtained from this study. Our testing machine has a 733 MHz Intel(R) Celeron(TM) processor and a 512MB RAM, running WindowsXP and Java 1.4.2.05 SDK. We compare the PWJoin algorithm with other stream join solutions in the literature, including PJoin [40] that exploits solely punctuation constraints, binary sliding window join [68] and multiway sliding window join [57]. We have implemented the hash-based version of all these algorithms in our system. According to their design, these algorithms employ a separate hash table to maintain the current state of each input stream. For ease of presentation, we will use the terms PJoin, WJoin and MWJoin to refer to these three algorithms respectively. In addition, we use the term PWJoin to refer to the binary PWJoin and use MPWJoin to denote our new multiway PWJoin solution.

8.2 Experimental Study of Binary PWJoin

First, we compare the performance of PWJoin with WJoin. In this experiment, we explore (1) how much the memory overhead is saved by PWJoin in comparison with WJoin, (2) how much the throughput of PWJoin is improved and (3) how the performance of PWJoin is affected when it faces with useless punctuations. We evaluate the join operators over a pair of punctuated streams, with $\frac{\lambda}{\lambda_p} = 100$. That is, there are on average 100 tuples between any two consecutive punctuations. The inter-arrival time of tuples from each stream conforms to a Poisson distribution with a mean of 2 milliseconds. We vary the window size for input streams in different runs and record the total number of tuples in the join state and the total number of tuples output up to each sampling point. Within the same experimental run, we apply the same window size to both input streams.

PWJoin vs. WJoin, memory overhead and tuple output rate. In Figure 8.1 we show the result of three runs regarding the total number of tuples in the PWJoin state and in the WJoin state, both with the time window being 1, 5, and 15 seconds respectively. Accordingly, we denote PWJoin in these 3 runs as PWJoin-1, PWJoin-5 and PWJoin-15 in the figure. The same notation applies to WJoin. We can see that as the window becomes larger, the memory savings by PWJoin become more and more significant. This is consistent with the memory cost estimation in Section 6.7.

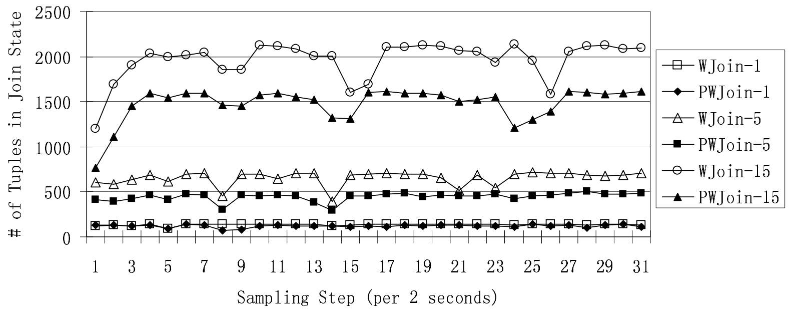


Figure 8.1: Memory Overhead, PWJoin vs. WJoin.

For these experimental runs, we also plot the number of output tuples of PWJoin and WJoin for each run. Figure 8.2 shows the number of output tuples of these two join solutions up to each sampling point in 2 runs, with a 5-second window and a 15-second window on both input streams respectively. We observe that when the window is small, since the number of tuples purged by each punctuation is small, the cost by maintaining PWJoin state and by punctuation-driven operations exceeds the saving in probing. Therefore, WJoin performs slightly better. As the window becomes larger, the gains in probing by employing constraint-exploiting techniques gradually take over. Thus the PWJoin begins to perform better than WJoin.

PWJoin vs. WJoin, useless punctuations. Now we turn to observe the overhead required for the PWJoin operator to handle useless punctuations,

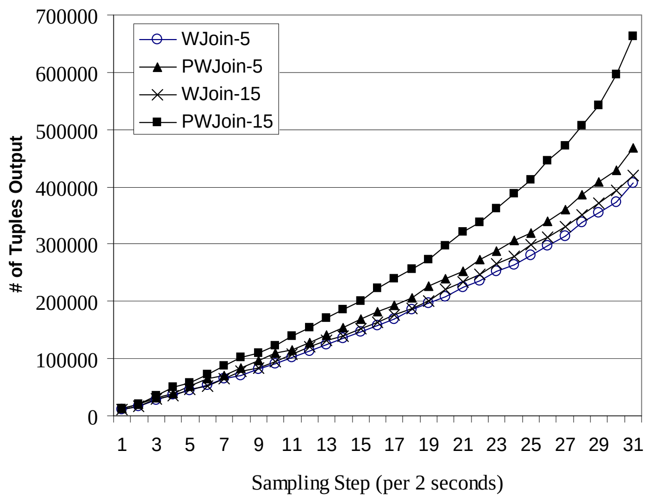


Figure 8.2: Tuple Output Rate, PWJoin vs. WJoin, Window: 5 secs, 15 secs.

i.e., punctuations that enable no optimizations. In terms of data streams without punctuations, the cost of PWJoin is almost the same as WJoin because the operations caused by purge and propagation are only triggered by the arrival of punctuations. If punctuations never happen, no extra cost would be incurred in PWJoin.

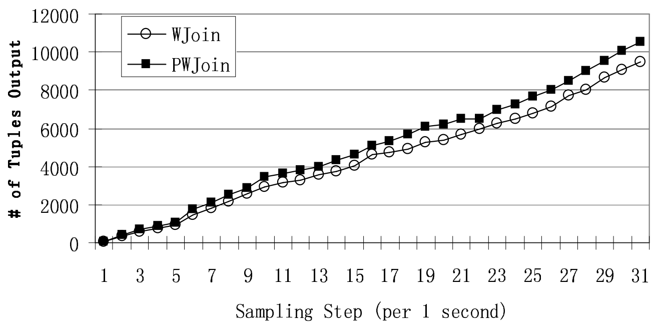


Figure 8.3: Tuple Output Rate, PWJoin vs. WJoin, Useless Punctuations.

Let us now consider the worst case in which all punctuations are useless, i.e., the punctuation does not match any tuples in state so that no tuples would ever be purged. However, PWJoin still tries to search for tuples that can be purged for each newly-arriving punctuation. This will cost extra time. Figure 8.3 shows the number of output tuples by PWJoin and WJoin over two punctuated streams ($\frac{\lambda}{\lambda_p}$=30), however, with all punctuations being useless instead. We can see that even in this case, the PWJoin still performs better than the WJoin. This is due to the following reasons. First, the cost for processing a useless punctuation equals the cost of probing a hash table for a matching I-Node. This is even less than the cost of processing a tuple because it does not incur the overhead of forming join results. Moreover, punctuations normally arrive much more infrequently than the actual tuples, in this case, 30 times less frequent. In addition, the PWJoin achieves more performance gains by exploiting constraints. Hence we conclude that in most cases, the cost of handling punctuations is trivial compared to the potential advantages it may offer.

Synergy of punctuation and window constraints. Finally, we consider the synergy of punctuations and windows, i.e., the optimization enabled by the interactions of the two constraint types. As we discussed in Section 6.3, early punctuation propagation and early tuple dropping can be potentially achieved. We run PWJoin and PJoin over two punctuated streams ($\frac{\lambda}{\lambda_p}$=30). Both streams have Poisson tuple inter-arrival time with a mean of 2 milliseconds. For PWJoin, a 1-second window applies to both input streams. We can see from Figure 8.4 that PWJoin has an overall higher punctuation output rate. This is due to the window-assisted early propagation and hence the tuple droppings that occur in the PWJoin execution. Notice that such semantic tuple droppings do not affect the precision of the join result (see Section 6.3).

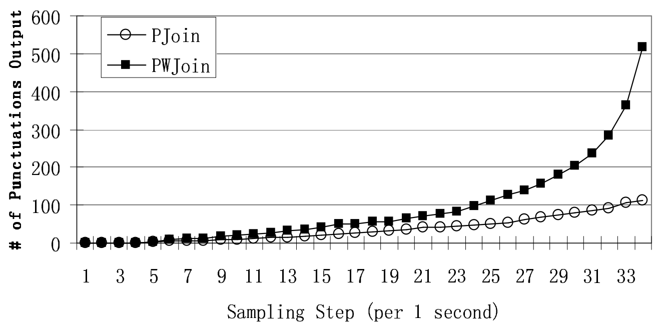


Figure 8.4: Punctuation Output Rate, PWJoin vs. PJoin, Window: 1 sec.

8.3 Experimental Study of Multiway PWJoin

We now show some results from the experiments that explore the effectiveness of the multiway PWJoin (*MPWJoin*) algorithm. In all the experiments shown in this section, we execute a 4-way join query over four streams S_1, ..., S_4 on a common attribute att. We compare the performance of MPWJoin with MWJoin.

We first consider the case in which no tuple dropping enabled by the window-assisted propagation occurs. We begin with a 4-way join query over four punctuated streams. λ_i=100 tuples/second and $\frac{\lambda}{\lambda_p}$=10 for i=1..4. In the query, a 10-second sliding window is applied to each stream. Therefore, a window of each stream contains on average 1000 tuples. In addition,

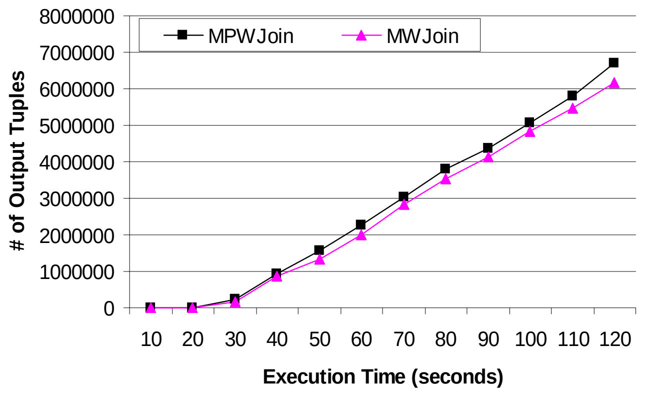


Figure 8.5: MPWJoin vs. MWJoin, Tuple Output Rate, Relatively High Join Selectivity.

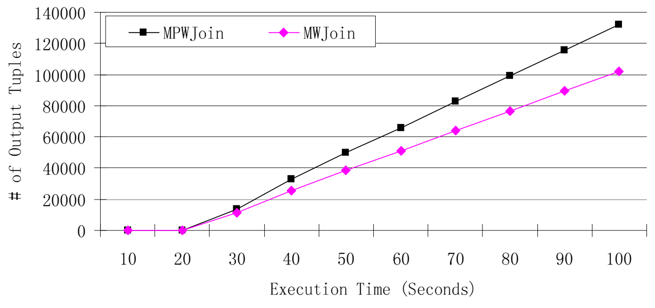


Figure 8.6: MPWJoin vs. MWJoin, Tuple Output Rate, Relatively Low Join Selectivity.

a tuple from any input stream will join with on average 10 tuples in each of the other three streams. This is a relatively high join selectivity compared to the join selectivity used in existing work on multiway joins [98].

Figure 8.5 shows the total number of join results output by the MP-WJoin and the MWJoin respectively up to each sampling point (per 10 seconds). We can see that the MPWJoin performs slightly better than the MWJoin.

We then reduce the join selectivity such that a tuple from any one input stream will join with on average 2 tuples in each of the other three streams. As shown in Figure 8.6, MPWJoin yields a much higher output rate than MWJoin. This is because in processing each new tuple, MPWJoin only conducts one hash lookup to find the matching I-Node. Afterwards, if no join result is expected, the $vCount$ field of the I-Node helps prevent further probe operations. All tuples being accessed will contribute to the join result. However, to process a new tuple, MWJoin needs to conduct n hash lookups if the join is successful. Otherwise, the hash lookup will stop whenever a failed lookup occurs. So the number of hash lookups is at least 1. Therefore, under the lower join selectivity, the efforts made by MWJoin on probes that lead to no join result will become more significant. Then MPWJoin should provide better output rates than MWJoin.

In addition, with an increase in the number of input streams, the more performance gains can be achieved by the inter-stream cluster index of MPWJoin because it only needs one hash probe for retrieving the corresponding I-Node and hence to locate matching tuples from all streams. Since highly selective join predicates are very common such as the key to foreign key join, the MPWJoin state design becomes a preferred solution for such cases.

Next, we study the performance gains from tuple dropping that are enabled by the interaction of the window and punctuation constraints for multiway joins (see Section 6.3). We evaluate the MPWJoin and the MWJoin over four input streams. One of them is a punctuated stream with $\frac{\lambda}{\lambda_p}$=10. The other three streams don't contain punctuations. The tuple arrival rate

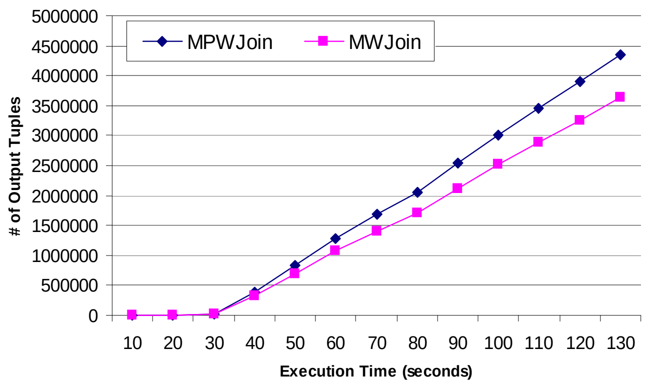


Figure 8.7: MPWJoin vs. MWJoin, Tuple Output Rate, with 40% Tuple Dropping.

of all the four streams is 100 tuples/second and a 10-second window is applied to all streams. In addition, approximately 40% of the input tuples satisfy the tuple dropping condition.

Figure 8.7 shows the tuple output rate of the MPWJoin and the MWJoin. We can see that due to the workload reduction caused by tuple dropping, MPWJoin achieves a much better output rate than the MWJoin. In addition, the MPWJoin consumes nearly 35% less memory than the MWJoin due to the more compact state, which is not shown here.

Chapter 9

Related Work

9.1 Related Work

As query evaluation over continuous data streams receives increasing attention, several data stream management systems have been built to explore the solutions for tackling the challenges arising in this new context. These systems include Aurora [2], CAPE [83, 84], STREAM [82], TelegraphCQ [25], NiagaraCQ [29], to name a few.

Specific to join processing, the first well-known pipelined join solution is the *symmetric hash join* [100]. *XJoin* [95, 96], *hash-merge join* [81] and *RPJ (Rate-based Progressive Join)* [90] are extended pipelined joins designed for special optimization purposes, i.e., to produce the first results as soon as possible and to finish the remaining joins at a fast rate. However, all these join algorithms face the problem of potentially unbounded runtime join state as data continuously streams in.

To bound the memory requirement by the stateful operator, a lot of work has been done in designing or detecting constraints that can help the stateful operators, include the join, to discard no-longer-needed data from the runtime operator state in a timely manner. [68] investigates the binary join algorithms under the time-based constraints, i.e., sliding windows. It also propose strategies for maximizing the join results in various scenarios. In addition, it also provide a unit-time-basis cost model for analyzing the performance of these algorithms. We adopt this cost model in analyzing the memory overhead of our PWJoin algorithm. The storage structures and indexing methods for sliding windows to improve the execution efficiency of operators, including the join, are also studied [56]. The state design of PWJoin operator extends from this work. [64] researches the shared execution of multiple window join operators. It provides alternative strategies that favor different window sizes. [33] tackles the problem of approximating sliding window joins over data streams in a stream processing system with limited resources. Different from these work, we focus on the execution strategy of a single join operator and we aim to deliver exact query results.

Value-based constraints have also been considered in the stream query optimization. The *k-constraint-exploiting algorithm* [16] exploits clustered data arrival patterns to detect and purge expired data to shrink the join state. These clustered patterns are statically specified in most cases, and hence only characterize restrictive cases of real-world data. If the actual data fails to obey these static constraints, the precision of the join result may suffer due to the incorrect purge of tuples. Moreover, this work mainly focuses on utilizing value-based constraint instead of exploring the interaction between window and value-based constrains, as done in our work.

Punctuations [93] are dynamic constraints embedded inside data streams. Static constraints such as unique key and clustered arrival of attribute values can also be modeled by punctuations. Therefore, punctuation covers a wide class of constraints that may help with continuous query optimization. [93] provide pass, purge and propagate rules enabled by punctuations for algebra operators. [74] employs punctuations to assist in the execution of window aggregate queries. We are the first to develop the punctuation-exploiting join algorithms [40, 41, 44].

Due to the existence of a large number of multi-join queries, the design of multiway join operators has received increased attention. Viglas et al. [98] propose an multiway join algorithm, namely MJoin, as well as the strategies for handling memory overflow and for choosing the optimal probing sequence. [57] studies the sliding window multiway join operator and proposes join ordering heuristics to minimize the processing cost per unit time. [14] provides the algorithms for adaptively caching the intermediate results for multiway join queries. The multiway PWJoin algorithm we propose in this work completely prevents the unnecessary join probings, hence eliminating the need for choosing the optimal probing sequence. In addition, the inter-stream cluster index we have designed for the PWJoin state lessens the requirements for materializing intermediate results.

Part II

Herald-Driven Runtime Stream Query Plan Optimization

Chapter 10

Introduction

10.1 Exploiting Metadata in Stream Processing

Many modern applications, including supply chain management [52], Web data search [95] and IP network management [33], need to process long-running (lasting for hours or days) or even continuous queries over large volumes of real-time data. In these applications, data arrives as high-speed streams. Queries over those streams need to be processed in an online fashion to enable real-time responses. Since data in these applications are generated on the fly, no meta knowledge about cardinality and data value arrival patterns may be available at query compilation time to be used for query optimization. Also, no pre-built index is available to be exploited for query processing. Therefore, traditional query optimization strategies, which heavily rely on these information, become largely inapplicable. The query system may face scalability problems when processing hundreds or even thousands of concurrent queries over high-speed streams, as often experienced by the above applications [64, 91].

It has been observed that in these applications meta knowledge on data values may become available as streaming data are generated [16]. As an example, consider the online sales management application [52]. Sales records may be periodically propagated to the central management system by each store. Therefore, the transaction records of each store within a reporting period will tend to be clustered together in the merged data stream. Correspondingly, metadata regarding such clustered data arrival, e.g., declaring that the next 5000 tuples will have *storeID=101*, could be provided along with the actual data received by the query system.

Besides being provided directly by the applications, such metadata on data values could also be derived by a query system itself [86]. As an example, query systems often employ a buffer to collect input data. Data in the buffer may be processed for a variety of reasons, including being sorted to correct out-of-order arrivals [86] or to perform load shedding [91]. Such data pre-processing could easily annotate the data with relevant metadata. While such pre-processing incurs overhead, the metadata it provides could thereafter potentially benefit a large number of queries.

Motivating example. We now use an example to illustrate how such runtime metadata on data values could be exploited to optimize query execution. Consider our sales management application again in which the sales of each store are reported to the central information system through a store hierarchy at an hourly basis. Each store reports sales data to the state information center, which reports to the regional center, which reports to the national center, which finally reports to the central system. Therefore, in the final $sales$ stream, data may be clustered based on storeID, stateID, regionID and countryID. A metadata tuple could then be easily inserted immediately before each such cluster to indicate the attribute values satisfied by tuples in the cluster. We refer to such metadata as *herald* because it serves as a "messenger" indicating the properties that a particular chunk of data (i.e., a substream) following it must satisfy.

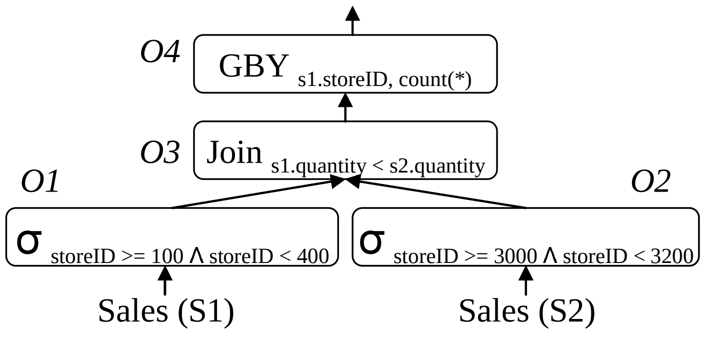

O4
GBY s1.storeID, count(*)
O3
Join s1.quantity < s2.quantity
O1
σ storeID >= 100 Λ storeID < 400
O2
σ storeID >= 3000 Λ storeID < 3200
Sales (S1)
Sales (S2)

Figure 10.1: Query Plan for Example Query Q1.

Consider the example query *Q1* below defined on the *Sales* stream with schema <*storeID, date, category, quantity*>. Suppose the stores in each state have IDs within a particular range. For example, the stores in Massachusetts have IDs within the range of [200, 400) and in California have IDs within the range of [3000, 3200). Query Q1 compares the hourly sale quantities of MA stores with CA stores. For each MA store, the query reports the number of CA stores such that the sales quantity of the MA store is less than the quantity of the CA store. The query execution plan is shown in Figure 10.1, consisting of two Select, one Join and one Group By operator.

Example Query Q1:

SELECT s1.storeID, count(∗)

```
FROM      Sales s1, Sales s2
WHERE     s1.storeID >= 200 and s1.storeID < 400
          and s2.storeID >= 3000 and s2.storeID < 3200
          and s1.quantity < s2.quantity
GROUP BY  s1.storeID
```

Assume a herald *<storeID = [200, 400), date = $*$, category = $*$, quantity = $*$; count, 5000>* is received by operator O1, which indicates that the next 5000 sales tuples all satisfy the condition *storeID* $\in$ *[200, 400)*, i.e., they are all MA store sales data. Then these 5000 tuples could bypass the evaluation against the predicate of operator O1 since they are guaranteed to satisfy this predicate. Let's further assume that before reporting data, each store partitions the hourly sales data into three partitions with 0≤quantity<1000, 1000≤quantity<5000 and quantity>5000 respectively. Given a herald *<storeID = [200, 400), date = $*$, category = $*$, quantity = [5000, ∞); count, 5000>* received by operator O1, the data conforming to this herald can bypass O1. Second, at operator O3, these data don't even need to be joined with any data tuples from the other stream that belong to the partitions 0≤quantity<1000 or 1000≤quantity<5000 since they are guaranteed not to produce any join results. That is, the processing of these partition pairs could use a much more efficient query plan (i.e., with the join eliminated) than the one selected without considering heralds. Clearly, such herald-driven optimization could result in significant savings in query execution costs.

10.2 Challenges in Herald-Driven Optimization

Several important observations can be made regarding herald-driven query optimization. First, heralds become available only at *runtime*. Hence query optimization must be conducted at runtime.

Second, heralds have their *lifespans*, i.e., the properties described by a herald are only satisfied by a particular substream. In other words, respective optimizations driven by a herald will be applicable only to the corresponding substream and thus only for limited periods at a time.

Finally, one herald defined for one stream may enable several distinct optimizations in collaboration with heralds from other streams. For example, the herald $<$storeID = [200, 400), date = $*$, category = $*$, quantity = [1000, 5000)$>$ from stream S_1 enables us to eliminate the join if combined with the herald $<$storeID = [3000, 3200), date = $*$, category = $*$, quantity = [5000, ∞)$>$ from S_2, and makes the join unsatisfiable if combined with the herald $<$storeID = [3000, 3200), date = $*$, category = $*$, quantity = [1000, 5000)$>$ from S_2. Hence multiple distinct query plans optimized by heralds may be valid at a time, with *partially overlapped scopes* (i.e., the substreams that these plans are applicable to may overlap with each other).

The above properties raise serious challenges in designing new query optimization and execution techniques for exploiting heralds. For query optimization, first, the algorithm employed to find the optimized plans given a set of heralds must be efficient so to identify all beneficial optimization opportunities. Second, the algorithm must be lightweight so to minimize the runtime optimization overhead. For execution, a query execution paradigm must be designed so that 1) it supports the concurrent execution of multiple logical plans on overlapping input substreams without duplication of data storage and costs, and 2) it can adaptively phase in and out logical plans on substreams with negligible physical plan switching costs [38, 104].

10.3 State-of-the-Art in Stream Processing

Semantic query optimization (SQO) [31, 70], which utilizes semantic information about the input data for query optimization, has been considered in *static* database systems where data updates are infrequent, ranging from relational [31], object-oriented [59], to XML databases [88]. The proposed techniques include join/select elimination, join/select introduction, and detection of empty answer sets. They are based on schema-level information, i.e., integrity constraints, assumed to be known at query compilation time in such static systems. Query optimization itself is thus also conducted at query compilation time. In all cases, one single optimized plan is produced. We clearly face a different problem in that 1) the metadata to be exploited have scopes (i.e., may be valid only for a subsequence of the input data) and 2) they only become available at query execution time. Clearly, lightweight constraint reasoning techniques that can react to runtime metadata changes in an incremental fashion are needed.

The punctuation [93] metadata model has been proposed for streaming data to declare that the data that follows will *not* satisfy certain properties. One class of existing work focuses on the optimization of execution logic of individual operators such as join or aggregate [40, 41, 74]. Our work now is the first to target the query plan level, i.e., to optimize the overall plan structure to minimize execution costs.

Present works on runtime query optimization for streaming data [13, 57] focus on the traditional query rewriting problem of reordering joins or filters using selectivity statistics. No semantic knowledge has been considered thus far.

In summary, existing work on SQO and on stream query optimization have been separate efforts. We are the first to combine them to conduct herald-aware query optimization.

10.4 Our Contributions: Herald-Driven SQO

Our contributions of this work are summarized as follows:

1). We are the first to explore stream query optimization at query plan level by exploiting dynamic metadata on data values, namely $heralds$ (Section 11.1). In particular, we identify four semantic query optimization opportunities that can be enabled by heralds, which parallel the SQO techniques found in traditional databases [31, 70].

2). To minimize the optimization overhead, we develop an efficient constraint reasoning algorithm named $PredSAT$ based on classic satisfiability reasoning theory. PredSAT is guaranteed to identify all four herald-driven optimization opportunities incrementally at runtime.

3). Multiple concurrent SQO plans may be enabled by heralds for processing different, potentially overlapping stream partitions. We propose a versioned minimum range model for generating multiple concurrent logical plans based on the result of PredSAT.

4). To achieve multiple concurrent logical plans with one single physical plan, we propose a novel query execution paradigm employing multi-modal operators with runtime configuration logic. This paradigm eliminates any replication of operator states or inter-operator queues, guarantees instantaneous application of herald-driven query optimizations, requires zero plan migration effort, and naturally supports highly flexible adaptive execution.

5). We conduct an extensive experimental study in the CAPE stream processing system [83]. The experimental results confirm that our herald-driven optimization techniques significantly reduce query execution time, up to 60% in our tested scenarios.

Chapter 11

Background

11.1 Herald Model

Heralds are metadata interleaved within streaming data. A herald describes constraints on the attribute values of a sequence of tuples immediately following it. We adopt the punctuation model [93] (Section 3.2) to represent attribute constraints and adapt it for modeling heralds. Same as the punctuation, a herald contains 1) a sequence of *attribute patterns*, each corresponding to an attribute in the stream schema, indicating a range in the domain of that attribute, and 2) a *timestamp* that records the time when the herald becomes effective. In addition, each herald is associated with a *lifespan* which denotes the scope of validity of the herald. The lifespan can be count-based (i.e., covering a certain number of tuples following the herald) or time-based (i.e., covering a time range starting from the herald's timestamp). Similar to the punctuation model, we assume that all tuples and heralds in the same input stream are received in their timestamp order.

Tuples arriving within a herald's lifespan must conform to it, i.e., they must match the attribute patterns specified by the herald. We also assume that heralds in the same stream don't have overlapped lifespans, i.e., no contradicting constraints are specified. Thus logically, for any input stream, at most one herald is valid at any time. A tuple not described by any herald is called a *free tuple*.

Two important differences exist between the punctuation [93] and our herald model. First, a punctuation signals that the data following it *will not* match the given attribute patterns (i.e., negative predicates), while a herald indicates that the data following it *will* match the attribute patterns (i.e., positive predicates). Second, the punctuation model is prohibitively strict, i.e., every punctuation has an implicit (and in fact infinite) lifespan extending from its announcement time to the end of the stream. Instead, a herald has an explicitly specified lifespan, either bounded (which is the more realistic scenario in practical applications) or unbounded (specified as ∞), starting from its announcement.

In the following, we will denote a herald as below. ptn_i ($1 \leq i \leq n$) is an attribute pattern for the i^{th} attribute A_i. *lifetype* is the type of the lifespan, with keyword $count$ denoting the count-based lifespan and $time$ denoting the time-based lifespan. *lifeval* is the value of the lifespan. The time-based lifespan is in *milliseconds*.

$$(< ptn_1, ptn_2, ..., ptn_n >; < lifetype, lifeval >)$$

Below is an example substream with a herald (in bold) and subsequent tuples that conform to this herald. The herald indicates that for the next 300 tuples following it, the values of *storeID* will be within the range of [200, 400) and the value of *category* will be *fruit*.

schema: storeID, date, category, quantity

(<[200, 400), *, fruit, *>: count, 300) – *herald*

(<201, 2008-02-01, fruit, 40>)

(<202, 2008-02-01, fruit, 20>)

...

11.2 Our Targeted Query

Definition 3 Targeted Query. *We consider single block conjunctive queries specified as follows* [1].

```
SELECT <select-list>
FROM   <list-of-streams>
WHERE  <where-conditions>
[WINDOW <window-specification>]
```

The where-conditions are $p_1 \wedge p_2 \wedge ... \wedge p_n$ *in which 1)* $p_i = z_1 \theta z_2$ *(*$1 \leq i \leq n$*); 2)* z_j *(j = 1, 2) is either an attribute or a constant, but* z_1 *and* z_2 *cannot both be constants; 3)* θ *is =, < or* $\leq$*. Each predicate* p_i *is called an inequality predicate [60], though* θ *could be "=".*

[1]Henceforth we assume queries have no WINDOW specified. We discuss issues for handling windowed queries in Chapter 13.

We differentiate between *selection* predicates (i.e., with one constant operand) and *join* predicates (i.e., with two attribute operands) in a query. The query Q1 in Section 10.1 is an example of our targeted query and it has selection predicates such as *(S1.storeID > 200)* and join predicates such as *(s1.quantity < s2.quantity)*.

Chapter 12

PredSAT: Predicate Satisfiability Reasoning Algorithm

12.1 Herald-Driven Optimization Strategies

We have identified four query optimization cases enabled by heralds that are guaranteed to always lead to a reduction in query execution costs, as described below. This implies that no cost-based decision needs to be made on whether to apply our proposed optimizations.

Select data skipping (SDS). If any selection predicate cannot be satisfied by the input data, the entire conjunctive query cannot be satisfied by the data. Expression (12.1) below shows an example unsatisfiable selection predicate.

$$(A > 1000 \wedge A < B)_q \wedge (A < 800)_d \models False \tag{12.1}$$

In the expression, we denote the ranges signaled by a herald, i.e., the inequalities that are already satisfied by the data, by subscript d. The query predicates (thus yet to be enforced by the query) are denoted by subscript q.

If any unsatisfiable selection predicate is identified based on the currently valid heralds, the corresponding data can be skipped by the select operation.

Join data skipping (JDS). Similarly, whenever an unsatisfiable join predicate is detected based on heralds, the corresponding data can be skipped by the join operation. Expression (12.2) below shows an example with a join query over two input streams A and B.

$$(A > 800 \wedge A < B)_q \wedge (B < 800)_d \models False \tag{12.2}$$

Select elimination (SE). If a selection predicate is known to always evaluate to true over the input data that conforms to a herald, it is a redundant predicate regarding the data. These data can then be directly passed through corresponding part of the query that was to evaluate this now redundant predicate. Expression (12.3) shows an example in which $A > 1000$ is a redundant selection predicate.

$$(A > 1000 \wedge A \leq B)_q \wedge (A > 1200)_d \models (A \leq B)_q \tag{12.3}$$

Join simplification (JS). Similar to the select elimination, any redundant join predicate regarding a certain herald need not be evaluated over each individual tuple that conforms to this herald. Instead, a Cartesian product can replace the join predicate to simply combine the tuples without first having to execute this redundant join predicate. Expression (12.4) shows an example in which $A < B$ is a redundant join predicate.

$$(A < 800 \wedge A < B)_q \wedge (B > 1000)_d \models True \tag{12.4}$$

12.2 Extent of Query Optimization

The above four optimization strategies are based on predicate satisfiability as will be formally defined in Section 12.3. Select/Join data skipping are due to *unsatisfiable* Select/Join predicates, while Select elimination and Join simplification are due to *satisfied* Select/Join predicates respectively.

	Evaluation Result		
	True	*False*	*Unknown*
Select	Select Elimination	Query Pause	Regular Eval.
Join	Join Simplification	Query Pause	Regular Eval.

Table 12.1: Query Optimization Opportunities.

According to Definition 3, predicates in our targeted queries can only be selection or join predicates. For each predicate, the result from evaluating an input must be either true (satisfied), false (unsatisfiable) or unknown (yet to be evaluated), as shown in Table 12.1. Therefore, the above four optimization cases compose a complete set of semantic query optimizations based on predicate satisfiability. Most importantly, each of these optimizations, once identified, ensures performance gains. This eliminates the need for a complex cost-based query optimizer. Instead, a lightweight mechanism for identifying these optimizations can be employed.

Among the four optimization techniques, select/join data skipping and select elimination parallel the cases covered by existing SQO techniques, namely, detecting an empty answer set and predicate elimination respectively [31, 70]. Join simplification is the runtime version of join elimination (JE) [31, 70]. They both are based on the identification of a redundant join predicate. Join is composed of the functionality of Cartesian product and predicates. JE identifies the join predicates for which both the Cartesian product and the predicates are redundant. Since Cartesian product determines the schema of the intermediate results, its necessity would not be affected by metadata on attribute values such as heralds. Therefore, at runtime join elimination becomes join simplification, which simply concatenates tuples from the two inputs without evaluating the predicates.

12.3 Satisfiability Reasoning

The identification of the applicability of any of the four herald-driven optimization strategies outlined above can be mapped to the classic satisfiability (denoted as SAT) and implication (denoted as IMP) problems [60]. Below we first introduce the definitions of the classic SAT and IMP problems (Definitions 4 and 5) and the corresponding reasoning algorithms. Based on these we then define our herald-driven SAT and IMP problems (Definition 7) and derive our *incremental* reasoning algorithm for identifying the four herald-driven SQO opportunities (Section 12.4).

Definition 4 Satisfiability Problem (SAT). *Given a conjunctive formula S composed of a set of inequality predicates, the SAT problem is to check whether at least one assignment for S exists such that S evaluates to true under the assigned values. If yes, S is said to be* **satisfiable***. Otherwise, S is said to be* **unsatisfiable***, denoted as $S \models False$.*

Definition 5 Implication Problem (IMP). *Given two conjunctive formulae S and T, both composed of a set of inequality predicates, the IMP problem is to check whether every assignment that satisfies S also satisfies T. If yes, S is said to* **imply** *T, denoted as $S \rightarrow T$.*

For both integer and real domains, one of the most effective SAT/IMP reasoning algorithms proposed in the literature is the *real minimum range algorithm* [60], henceforth referred to as the $RMin$ algorithm.

The RMin algorithm has $|S|$ time complexity for solving the satisfiability problem and $|S|^2 + |T|$ time complexity for the implication problem for our targeted queries (Definition 3 in Chapter 11). Here $|S|$ and $|T|$ denote the number of predicates in formulas S and T respectively.

We now review RMin as our work extends RMin to make it employable for runtime query optimization (Section 12.4). RMin utilizes the *inequality graph* defined below (Definition 6) for representing the set of predicates.

Definition 6 Inequality Graph. *An inequality graph for a conjunctive inequality formula S, denoted as G_S = (V_S, E_S), is a directed graph. Each node X in V_S*

one-to-one corresponds to a distinct attribute X in S. Each directed edge from node X to node Y in E_S, labeled with $\otimes$ and denoted as (X, Y, $\otimes$), one-to-one corresponds to an inequality (X $\otimes$ Y) $\in$ S. The label $\otimes$ is either $<$ or $\leq$.

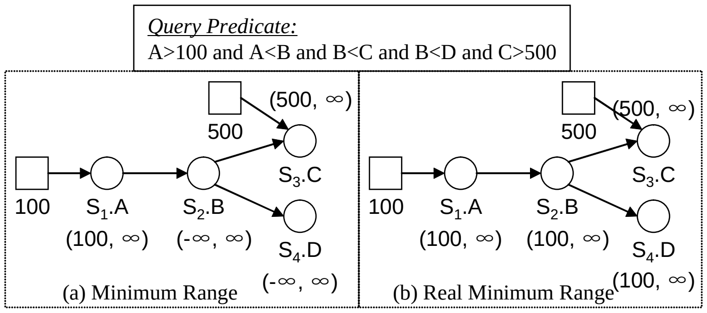


Figure 12.1: Computing Real Minimum Ranges.

Figure 12.1(a) shows an example of the inequality graph for predicate *S1.A>100 and S1.A<S2.B and S2.B<S3.C and S2.B<S4.D and S3.C>500.* We use a circle to denote a variable and a square to denote a constant. The label of the edge, if not shown, is assumed to be $<$. Otherwise it's $\leq$.

We call a node Y a *parent node* of a node X (and X a *child* of Y) if X can reach Y via a directed edge. We denote the set of all parent nodes of a node X as *parents(X)* and the set of all children nodes of a node X as *children(X)*.

For two nodes X and Y in G_S, if X can reach Y via directed path and Y can also reach X via directed path, X=Y is implied by S by transitivity. All such variables and the edges among them are said to form a *strongly connected component*, or *SCC*. As an example, predicate *A≤B and B≤C and C≤A* corresponds to an SCC. By transitivity, it equals to *A=B and B=C and C=A*. We use G_{S_c} to denote the *collapsed inequality graph* after collapsing each SCC in G_S into a single node. S_c denotes the *collapsed inequality formula* from S.

The RMin algorithm then works as follows. For an attribute X, let C_{up}^{X} = min(C_j) for all constants C_j such that X $\leq C_j \in S_c$. And let C_{low}^{X} = max(C_i) for all constants C_i such that X $\geq C_i \in S_c$. The closed range [C_{low}^{X}, C_{up}^{X}] is called the *closed minimum range* for X. These minimum ranges can be derived from the query in a straightforward manner. For example, consider a query with predicate *A>100 and A<B and B<C and B<D and C>500*, the minimum ranges for the attributes are shown in Figure 12.1(a).

The minimum range can be further refined to be the *real minimum range* [A_{low}^{X}, A_{up}^{X}] in which A_{low}^{X} and A_{up}^{X} denote the *real lower bound* and the *real upper bound* of the attribute X respectively, computed as below. First, attributes in S_c are sorted in their topological order. Then attributes are selected in S_c one by one according to their topological order in S_c. For an attribute X, A_{low}^{X} = max(C_i, C_{low}^{X}) for all C_i such that C_i = $A_{low}^{X_i}$. Here X_i is X's child, if the edge from X_i to X is labeled with $\leq$; or C_i = $A_{low}^{X_i} + 1$ if the edge for X_i to X is labeled with $<$.

Next, we select attribute X one by one according to the inverse topological order of S_c. A_{up}^{X} = min(C_j, C_{up}^{X}) for all C_j such that C_j equal to $A_{up}^{X_j}$. Here X_j is X's parent, if the edge from X to X_j is labeled with $\leq$; or C_j = $A_{up}^{X_j}$-1 if the edge from X to X_j is labeled with $<$.

As shown in Figure 12.1(b), the real lower bounds of attributes S_2.B and S_4.D are refined to be 100 instead of -∞ because these attributes are forced to be greater than S_1.A, whose real lower bound is 100.

The reasoning of the classic SAT and IMP problems using the real minimum ranges are based on Theorem 2. The proof of this theorem can be found in [60].

Theorem 2 *S is satisfiable iff 1) no SCC in G_S contains an edge labeled $<$ and 2) for each attribute X in S, $A^X_{low} \leq A^X_{up}$.*

In addition, if S is satisfiable, $S \rightarrow T$ iff 1) for any $(X \leq Y) \in T$ there exists a path from X to Y in G_{S_c}, or $A^X_{up} \leq A^Y_{low}$; 2) for any $(X < Y) \in T$ there exists a path from X to Y in G_{S_c} with at least one edge of the path labeled with $<$, or $A^X_{up} < A^Y_{low}$; 3) for any $(X \leq C) \in T$, $C \geq A^X_{up}$; and 4) for any $(X \geq C) \in T$, $C \leq A^X_{low}$.

In the example in Figure 12.1, all predicates are satisfiable and no predicate can be implied by other predicates.

12.4 PredSAT Algorithm

Herald-driven query optimization can be abstracted to the following satisfiability and implication problems.

Definition 7 Herald-Driven Satisfiability (H-SAT) and Implication (H-IMP) Problems. *Given a set of inequality query predicates P_Q expressed by a query Q and a set of inequalities P_D satisfied by input data D,*

1). H-SAT*: if $\bigwedge\limits_{p_i \in (P_Q \cup P_D)} \models False$, select/join data skipping by p_i can be applied for data D;*

2). H-IMP_S*: for a selection predicate p_s in P_Q,*

if $\bigwedge\limits_{p_k \in (P_Q \cup P_D - p_s)} \rightarrow p_s$, select elimination can be applied to the predicate p_s;

3). H-IMP_J*: for a join predicate* p_j *in* P_Q*,*

if $\bigwedge_{p_k \in (P_Q \cup P_D - p_j)} \rightarrow p_j$*, join simplification can be applied to the predicate* p_j.

Here $\cup$ *and* $-$ *denote the set-based union and difference operations respectively.*

Heralds are interleaved with streaming data. Hence they dynamically become valid at runtime. To identify possible optimization opportunities enabled by heralds, the RMin algorithm needs to be invoked each time a new herald is received. Since the number of inequalities increases as more heralds are received, a significant reasoning overhead may be incurred. However, we observe that each herald only refers to a single attribute. Thus running the RMin algorithm over all inequalities may trigger redundant reasoning. Therefore, we now propose an incremental SAT/IMP reasoning algorithm based on RMin, which we call the PredSAT (for Predicate SATisfiability reasoning) algorithm, that limits the reasoning scope to only relevant inequalities. The *relevant* inequalities are the ones that could enable any of the four optimization strategies (Section 12.1) if combined with the given new herald.

We now describe the PredSAT algorithm. When a continuous query is registered into the system, the PredSAT algorithm constructs the inequality graph with real minimum ranges as induced by the query, same as the RMin algorithm [60]. During query execution, the PredSAT algorithm further refines the real minimum ranges according to the current heralds. The refined minimum ranges are called *herald minimum ranges*.

If a herald h with predicate $X \leq C$ is received, it's obvious that only the real upper bound of the node X and all the nodes that are descendants of X may possibly be affected, i.e., further tightened. The PredSAT algorithm starts from node X. If $A_{up}^{X} \leq C$, the algorithm stops. Otherwise, a refined upper bound C can be computed for X. The algorithm then proceeds to check X's children nodes in a breadth-first manner. The traversal terminates when an examined node has a real upper bound already smaller than C.

When computing herald minimum ranges, the algorithm checks the optimization opportunities as follows:

1). *Unsatisfiable query.* P_Q is unsatisfiable if there exists a variable X in S_c, $A_{up}^{X} < A_{low}^{X}$.

2). *Redundant selection predicate.* For p_s: $X < C_q$, $\bigwedge\limits_{p_i \in (P_Q - \{p_s\}) \cup \{h\}} \rightarrow p_s$ if $\bigwedge\limits_{P_Q - \{p_s\}}$ is satisfiable and $A_{up}^{X} > C$ (i.e., $C < C_q$). Therefore, p_s is redundant.

3). *Redundant join predicate.* For p_j: X < Y, $\bigwedge\limits_{p_i \in (P_Q - \{p_j\}) \cup \{h\}} \rightarrow p_j$ if $P_Q - \{p_j\}$ is satisfiable and Y $\in$ parent(X)) and $A_{low}^{Y} > C$. Therefore, p_j is redundant

The proof is straightforward,

Similarly, given a herald h: $X \geq C$, we determine

1). P_Q is unsatisfiable if $A_{up}^{X} < A_{low}^{X}$.

2). For p_s: $X > C_q$, $\bigwedge\limits_{p_i \in (P_Q - \{p_s\}) \cup \{h\}} \rightarrow p_s$ if $P_Q - \{p_s\}$ is satisfiable and $A_{low}^{X} < C$ (i.e., $C_q < C$).

3). For p_j: $X > Y$, $\bigwedge_{(P_Q-\{p_j\})\cup\{h\}} \rightarrow p_j$ if $P_Q - \{p_j\}$ is satisfiable and Y $\in$ children(X)), $A_{up}^{Y} < C$.

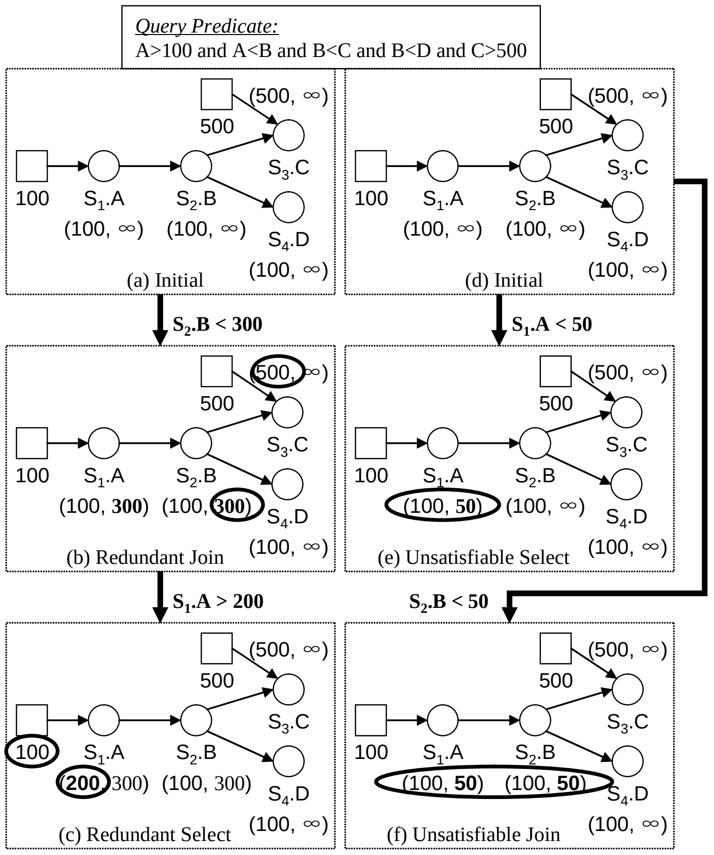


Figure 12.2: Example of PredSAT Reasoning.

For query predicate *A>100 and A<B and B<C and B<D and C>500*, the real minimum ranges computed from the predicate are shown in Figure 12.2(a). When a herald indicating S_2.*B* < *300* is received from S_2, the real upper bounds of attributes S_1.A and S_2.B are updated from ∞ to 300. Since the real upper bound of S_2.B (300) is now less than the real lower bound of

S_3.C (500) (Figure 12.2(b)), based on PredSAT algorithm, the join predicate $S_2.B < S_3.C$ now becomes redundant for processing substreams described by this herald and the output of evaluating selection on stream S_3. As shown in Figure 12.2(e), from the original query predicate, when a herald indicating $S_1.A < \mathit{50}$ is received, the real upper bound of attribute S_1.A is updated to be 50, which is lower than its real lower bound (100). This indicates that the predicate $S_1.A > \mathit{100}$ is unsatisfiable for the substream described by this herald. Similarly, Figures 12.2(c) and (f) show the cases of redundant select predicate and unsatisfiable join predicates respectively, based on the PredSAT algorithm.

12.5 Mapping Reasoning Results to a Logical SQO Plan

Once an optimization opportunity is identified for a set of substreams during the reasoning, an SQO plan can be generated to process these particular substreams. The remaining substreams would continue to be serviced by the default plan. Figure 12.3 shows such an example. When the herald h with $A<500$ is received, the join predicate $S_1.A < S_2.B$ is identified to be redundant regarding the substream described by h. Therefore, an SQO plan with the join operator replaced with a Cartesian product operator is produced to process the substream described by h and the S_2 stream. The results of these two plans are then merged to produce the complete final result for the query.

The SQO plan generation follows the rules described in Equations 12.5 – 12.8. In the equations, $\cup$ represents *union all*. S_h, S_{h1} and S_{h2} represent the

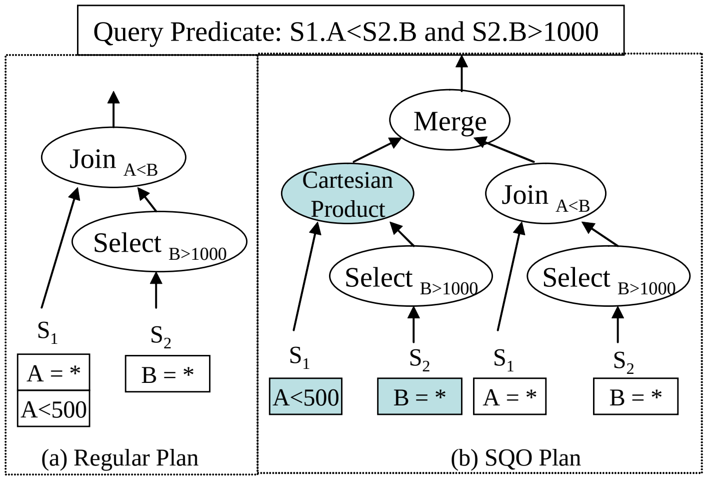


Figure 12.3: Mapping Reasoning Result to SQO Plans.

substreams described by heralds h, h1 and h2 respectively. σ_p, $\bowtie_p$ and $\times$ denote the select operation with predicate p, the join operation with predicate p, and the Cartesian product respectively.

Select elimination for S_h:

$$\sigma_p(S_1 + S_h) = \sigma_p(S_1) + S_h \; if \; \forall t \in S_h, \sigma_p(t) \models TRUE \tag{12.5}$$

Select data skipping for S_h:

$$\sigma_p(S_1 + S_h) = \sigma_p(S_1) \; if \; \forall t \in S_h, \sigma_p(t) \models FALSE \tag{12.6}$$

Join simplification for (S_{h1}, S_{h2}):

$$\begin{aligned}(S_1 + S_{h1}) \bowtie_p (S_2 + S_{h2}) &= S_1 \bowtie_p S_{h2} + S_2 \bowtie_p S_{h1} \\ &\quad + S_1 \bowtie_p S_2 + S_{h1} \times S_{h2} \\ &\quad if\ \forall(t_1 \in S_{h1}, t_2 \in S_{h1}), \bowtie_p (t_1, t_2) \models TRUE\end{aligned} \tag{12.7}$$

Join data skipping for (S_{h1}, S_{h2}):

$$\begin{aligned}(S_1 + S_{h1}) \bowtie_p (S_2 + S_{h2}) &= S_1 \bowtie_p S_{h2} + S_2 \bowtie_p S_{h1} \\ &\quad + S_1 \bowtie_p S_2 \\ &\quad if\ \forall(t_1 \in S_{h1}, t_2 \in S_{h1}), \bowtie_p (t_1, t_2) \models FALSE\end{aligned} \tag{12.8}$$

Chapter 13

Generating Logical Plans

During query execution, multiple heralds on the same attribute of a stream may be received over time. In addition, a single herald may enable multiple optimizations by being combined with different heralds from other input streams. Thus we must handle possibly overlapping validity scopes of multiple concurrent SQO plans.

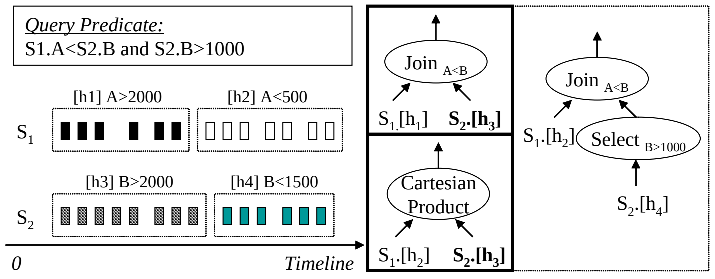


Figure 13.1: Query Plans with Scopes.

Figure 13.1 shows an example. Consider the query over two input streams S_1 and S_2 with predicates $S_1.A < S_2.B$ *and* $S_2.B > 1000$. Assume that the data received from the input stream S_1 are described by two heralds in sequence – h_1: $A > 2000$ and h_2: $A < 500$. Also, the data received from stream S_2 are described by two heralds in sequence – h_3: $B > 2000$ and h_4: $B < 1500$. The lifespans of these heralds are marked by the rectangles enclosing the substream of tuples directly below the herald predicate (in left bottom of Figure 13.1). We denote the substreams that conform to these heralds as S_1.[h_1], S_1.[h_2], S_2.[h_3] and S_2.[h_4] respectively. We observe that the following optimizations are applicable to the given input substreams: select elimination (B>1000) of S_2.[h_3] and join simplification (S_1.A<S_2.B) for (S_1.[h_2], S_2.[h_3]), join data skipping for (S_1.[h_1], S_2.[h_4]). For the substream pair (S_1.[h_2], S_2.[h_4]), no herald-driven optimization is applicable. Therefore, for the input streams received so far, potentially four distinct query plans may need to be constructed to best serve each of these four cases. We call these plans the *SQO logical plans*. We define the *scope* of an SQO logical plan to be the set of substreams that need to be processed by the plan.

We can see that herald h_1 contributes to the formation of two SQO logical plans, one by itself and one by being combined with another herald h_2. To capture all possible herald-driven optimization opportunities based on all currently valid heralds, we propose the notion of a *versioned real minimum range*.

We call the real lower and upper bounds for each attribute X computed solely based on query predicates *query lower bound* and *query upper bound* respectively, denoted as Q^X_{low} and Q^X_{up}. Obviously, each attribute has a single query minimum range. The query upper and lower bounds of an attribute may be further refined by heralds received at runtime. Since multiple heralds may be received for a single attribute, an attribute may be associated with multiple lower and upper bounds respectively. They are called *herald lower bounds* and *herald upper bounds* respectively. We denote the i^{th} herald lower and upper bounds as $A^X_{low,i}$, $A^X_{up,i}$ respectively. The herald lower and upper bounds are maintained in two lists respectively associated with the attribute.

During query execution, each time a new herald is received, if a lower or upper herald bound can be tightened due to the herald, a new lower or upper bound is created and appended to the end of the corresponding list. Whenever such change occurs, the optimization reasoning is conducted. During the reasoning, all herald bounds of the attributes that are being visited are examined and multiple SQO logical plans may be achieved.

Handling windowed query. For query with windows, the PredSAT algorithm and the mapping strategy can still be applied. Moreover, PredSAT can be further optimized to purge the minimum ranges once the lifespans of the associated heralds have moved out of the window.

Chapter 14

Runtime-Configurable Execution Paradigm

Given that input streams containing heralds, multiple SQO logical plans may be concurrently applicable to different sets of substreams. That is, multiple query plans with different scopes may exist at a given time. In addition, the scopes of different SQO plans may share common substreams. Therefore, we cannot default to a traditional single-plan solution, which employs an online plan migration technique [104] to switch from the current plan to one other (more efficient) plan in the middle of the query execution,

Consider the example in Figure 13.1 (Chapter 13). The join simplification is applicable to substreams ($S_1.[h_2]$, $S_2.[h_3]$). However, we can only guarantee correctness of the plan with join simplification when it is applied to the S_1 state containing only tuples from $S_1.[h_2]$. This is impossible because tuples in S_1 that arrive before $S_1.[h_2]$, i.e., $S_1.[h_1]$, have already been inserted into the S_1 state and may remain there until the end of the query execution. This causes a dilemma.

Therefore, we must support multiple concurrent query plans. However, to physically maintain multiple query plans and route tuples to the corresponding query plans may incur significant data duplication in both operator input queues and operator states since different plans may share many common input substreams and intermediate substreams.

In view of this, we now propose a new query execution paradigm that supports multiple concurrent logical query plans by physically maintaining only one single plan. Our new execution paradigm consists of five key components.

1). *Data partitioning.* We partition data based on heralds so that different substreams can logically be served by the same execution logic.

2). *Multi-modal operators.* We design powerful query operators with configurable execution logic so that they apply customized algorithms to process data from different stream partitions, as guided by the query optimizer.

3). *Lightweight control table.* We design a control structure that enables the application of customized execution logic for particular partitions by simply toggling a flag in the control table.

4). *Isolated operator tuning.* The configuration of the operator logic is internal to the operator itself, without affecting other operators in the query plan. So it is complementary to other optimizations that may be applied to the query plan.

5). *Partition propagation.* Each operator is equipped with the ability to propagate data partition information so to make any downstream operators configurable without requiring any re-partitioning effort.

Our fine-tuned execution paradigm has significant advantages. First, it avoids data duplication by physically maintaining a single plan. Second, it avoids duplicate computations for tasks such as state insertion or purging or due to multiple logical plans working on overlapping input substreams. Third, it reduces system overhead by avoiding context switching among the otherwise much larger set of operators and even between different plans. Lastly, it is flexible to revert back to the default herald-unaware plan. All it needs is to toggle a flag. No plan structure change is needed.

14.1 Herald-Driven Data Partitioning

Our data partitioning scheme partitions streams based on heralds. A partition could be a source partition or an intermediate partition. Source partitions are obtained by partitioning source streams. There are two types of source partitions: 1) the *herald partition* that contains tuples described by a single herald, and 2) the *anonymous partition* that contains tuples not described by any herald. Intermediate partitions are produced as output of select or join operators. An intermediate partition generated by a join may contain tuples from two source partitions, or from a source partition and an intermediate partition, or from two intermediate partitions. Therefore, an intermediate partition can also be a herald partition, containing tuples described by m (m≥1) herald(s), or an anonymous partition.

Each source partition is assigned a stream-wise unique ID. The default partition ID 0 is reserved for the anonymous partitions, while each new herald source partition is assigned the next available partition ID. The partition ID of an intermediate partition is the concatenation of the partition IDs of its component partitions.

14.2 Multi-Modal Operators

To assure agility of operators, we equip our herald query operators with multiple distinct execution modes that are configurable at runtime. That is, the operator processes every batch of data described by a herald in its most efficient manner as determined by the optimizer. This achieves multiple SQO logical plans within one *single* physical plan.

To configure its execution logic at runtime, each operator is equipped with a *control table* containing instructions on how to process the received herald partitions. Each time a new herald partition is received, the operator will use the herald associated with the partition to probe the control table and get the corresponding instruction. Based on the instruction, the operator will apply the appropriate execution strategy to the current data partition. For instance, a select operator may either directly output the partition (select elimination), drop the partition (select data skipping) or evaluate the partition using regular predicate checking.

The control table is probed each time a new herald partition is received. Thus it needs to be probe-efficient. We implement the control table as a hash table with the partition ID as hash key, as further explained below. This way, if instructions for a given partition are available, they can be retrieved with a single lookup.

The control table is dynamically updated at runtime by the *operator configurator*, one of the key components of the herald-driven semantic query optimizer (Section 15.1). The new entries are added into the control table as new herald-driven SQO opportunities are identified. To prevent the control table from growing unboundedly, existing entries are removed when corresponding partitions have been processed.

14.2.1 Multi-Modal Select Operator

The select operator differentiates between three types of partitions: 1) *Pass partition* in which all tuples are guaranteed to satisfy the selection predicate; 2) *Skip partition* in which all tuples are guaranteed to *not* satisfy the selection predicate; and 3) *Unknown partition* for which it is unknown if any of its tuples will satisfy the selection predicate or not.

The select operator directly propagates any *pass* partition to its output stream (due to select elimination) and discards any *skip* partition (due to query pause) without evaluating any of their tuples. Tuples in the *unknown* partitions will be evaluated against the selection predicate as done by the regular select operator. This design enables one single operator to achieve three distinct query plans by applying three distinct logics to process its input data.

In the control table of the select operator, each hash entry contains a list of <PartitionID, ActionFlag> pairs. The action flag can be one of three values: 0 means to pass (for Pass partitions), 1 to skip (for Skip partitions) and 2 means to evaluate (for Unknown partitions).

During query execution, when an input partition is received, the select operator first checks whether it is an anonymous partition (i.e., with partition ID 0). If yes, which means there is no herald associated with this partition, the select operator evaluates tuples in this partition as done by the regular select operator. Otherwise, the partition ID is used to probe the control table. If a match is found, the corresponding action flag will be used to trigger the suitable execution logic to be applied to the tuples in the partition.

14.2.2 Multi-Modal Join Operator

The multi-modal join operator is associated with two control tables corresponding to its two input streams respectively. Similar to the control table of select, each control table for the join operator is hashed on the partition ID. Each hash entry contains a list of <LeftPID, RightPID, ActionFlag> triples indicating whether the corresponding pair of partitions should be passed (due to join simplification), skipped (due to join data skipping), or evaluated.

When a new partition p is received from the left input stream, the join operator first checks its partition ID to see whether it is an anonymous partition. If yes, the join operator will process tuples in this partition as the regular join does (i.e., no optimization is done). Otherwise, the partition ID is used to probe the control table for the right input. If a match is found, the <LeftPID, RightPID, ActionFlag> triples in the list are enumerated. For each triple, the join logic indicated by $action$ is applied to the left-side partition with ID *LeftPID* and the right-side partition with ID *RightPID*. The processing of partitions received from the right input stream is similar.

Consider the example shown in Figure 13.1. Suppose the partitions corresponding to heralds h_1, h_2, h_3 and h_4 have partition IDs 1, 2, 3 and 4 respectively. Then the control table of the select operator has one entry (3, Pass). The control table for the left input of the join operator, which is S_1, has two entries with keys being partition IDs 1 and 2 respectively. The entry with partition ID 1 contains a list with one element (1, 4, DROP). The entry with partition ID 2 contains a list with one element (2, 3, PASS). Correspondingly, the control table of the right input of the join operator, which is the output of the select operator, has two entries with keys being partition ID 3 and 4 respectively. The entry with partition ID 3 contains a list with one element (3, 2, PASS) and the entry with partition ID 4 contains a list with one element (4, 1, DROP).

14.3 Partition ID Propagation

The herald-driven data partitioning is initially conducted for source input streams. For the proposed semantic optimization to be applied to non-leaf operators as well, the partition IDs associated with source stream partitions need to be propagated through the query plan.

We first discuss the propagation rules for the select operator. Each Pass partition is sent to the output stream of the select operator with its current partition ID. For each Unknown partition, if at least one tuple satisfies the selection predicate, a result partition is created with the current partition ID and it will contain all tuples in the input partition that satisfy the selection predicate.

The join operator each time processes a new partition from one of its inputs, joining it with all existing partitions in the state of the other input. For each pair of partitions that may produce join results, the operator creates a result partition with the partition ID being the combination of the partition IDs of the two input partitions.

Chapter 15

Experimental Evaluation

15.1 System Implementation

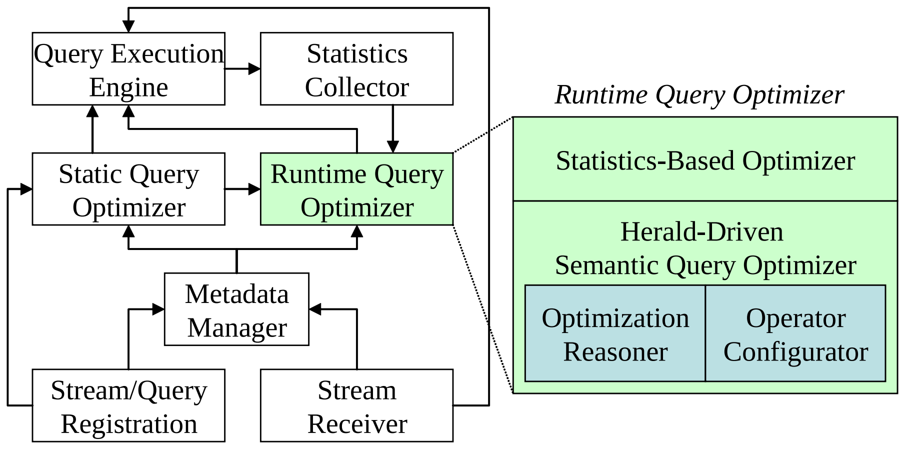


Figure 15.1: Herald-Aware Stream Engine.

Figure 15.1 shows the framework of our herald-aware stream processing engine. The arrows in the figure represent the communications between the corresponding components. When a query is registered into the stream engine, first, the *static query optimizer* is invoked to find the optimal or near-optimal query execution plan without considering any heralds. Then the execution plan is sent to the *query execution engine* to be executed. The *run-time query optimizer* dynamically optimizes the query during execution. The runtime optimizer includes both the *statistics-based optimizer* and the *herald-driven semantic query optimizer*. The statistics-based optimizer adjusts the query plan shape based on statistics about operator selectivities [57] gathered by the *statistics collector*.

The *herald-driven optimizer*, which is the focus of this work, is composed of two modules – *optimization reasoner* and *operator configurator*. Each time a herald is received, the *herald-driven optimizer* is invoked. During each of its runs, the *reasoner* is invoked to identify any newly applicable optimization opportunities and figures out the (potentially multiple) SQO logical plans. Then the *operator configurator* configures the control table of the corresponding operators to realize the SQO logic plans.

The stream receiver feeds the data streams to the *query execution engine*, and forwards the heralds to the *metadata manager*. The *metadata manager* maintains both static constraints such as integrity constraints and runtime heralds. It is consulted by the static and the runtime query optimizers.

15.2 Experimental Setup

We have implemented the techniques proposed in this work in our Java-based continuous query system named CAPE [83]. We have conducted an extensive experimental study to explore the effectiveness of herald-driven query optimization. The test machine has a 2.66GHz Intel(R) Pentium 4 processor and a 448MB RAM, running Windows XP and Java 1.6.1_01 SDK. We have created a benchmark system to generate synthetic data streams by controlling the data distributions and arrival rates.

The experiments to be presented in the following compare the query execution time when using herald-aware optimization (called the *herald* approach) with herald-unaware techniques (called the *regular* approach). We configure 1GB virtual memory for JVM. This is large enough to keep all data structures, including operator control tables for the herald approach and join states, in memory for all experiments presented here.

In our experiments, we vary the following parameters to evaluate the performance of our herald-driven techniques in a wide range of circumstances.

1). *Average partition size.* This is defined to be the average number of tuples in each herald partition. The partition size follows Uniform distribution.

2). *Operator selectivity.* The selectivity of the select operator is defined to be $\frac{N_{out}}{N_{in}}$ in which N_{out} and N_{in} denote the total number of output and of input tuples respectively. The selectivity of the join operator is $\frac{N_{out}}{N_1 * N_2}$ in which N_{out}, N_1 and N_2 denote the total number of output tuples, of left input tuples and of right input tuples respectively.

3). *Partition pass/drop rate.* The partition pass (or drop) rate of the select operator is defined to be $\frac{P_{pass}}{P_{in}}$ (or $\frac{P_{drop}}{P_{in}}$). Here P_{pass} (or P_{drop}) denotes the number of partitions that produce full (or no) select results. P_{in}

denotes the number of partitions. The partition pass (or drop) rate of the join operator is defined to be $\frac{P_{pass}}{P_1 * P_2}$ (or $\frac{P_{drop}}{P_1 * P_2}$). P_{pass} (or P_{drop}) denotes the number of partitions pairs that produce full (or no) join results. P_1 and P_2 denote the number of partitions from the left input and from the right input respectively.

In our experiments, we use the example query shown in Figure 13.1. The query plan contains a select and a join operator. The select operator takes one source stream as input and the join operator takes the output of the select operator and another source stream as inputs, and outputs the result for the query. Such a query plan is a common component of a large number of real application query plans [64, 94].

We compare the total query execution time of using the herald and the regular approach respectively. Each input stream contains 50 herald partitions, with varying partition sizes. The regular approach is executed against the same input streams, however, with heralds removed. Therefore, the regular approach has no herald-related costs. The execution time of using the herald approach includes the overhead of runtime optimization and herald-aware execution.

15.3 Evaluating Join Optimizations

We first evaluate the join optimization strategies, i.e., query pause by join and join simplification.

Varying partition drop rate. In the first experiment, we evaluate how the join data skipping would affect the query execution time. We vary the partition drop rate to control the frequency of join data skipping. In the result shown in Figure 15.2, the partition drop rate of the join ranges from 0 to 1 by 0.1. We set the average partition size to be 50 tuples, the selectivities of the select and the join operators to be 0.2 and 0.1 respectively.

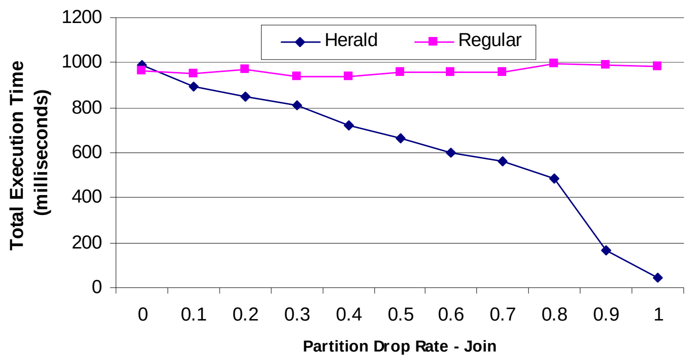


Figure 15.2: Par. Size:50, σ_{Select}=0.2, σ_{Join}=0.1.

The result in Figure 15.2 shows that as the partition drop rate increases, the execution time using the herald approach quickly decreases while no significant change in execution time is observed when using the regular approach. When the drop rate reaches 0.9, the herald approach has more than 80% reduction on execution time compared to the regular approach. This result is promising because with just a small partition size (i.e., 50 tuples) and low selectivity of the underlying select operator (i.e., when only 10% of its input data actually reaches the join), the herald approach already achieves significant performance gains.

Varying partition sizes. Next, we investigate what role the average partition size plays in affecting the performance gains by herald-driven SQO by studing scenarios with different partition sizes, namely, 20, 50 (as done already above) and 100. We first set the average partition size to be 20 tuples. All the other configurations remain the same as in the previous experiment. From the results shown in Figure 15.3, we can see that with such a reduced partition size, the herald-driven approach has only 10% to 15% better performance than the regular approach when drop rate is lower than 0.4 (recall this includes optimization overhead). However, when drop rate becomes relatively high, the herald-driven approach again achieves significant performance gains, for example, more than 50% reduction on execution time when the drop rate is 0.9.

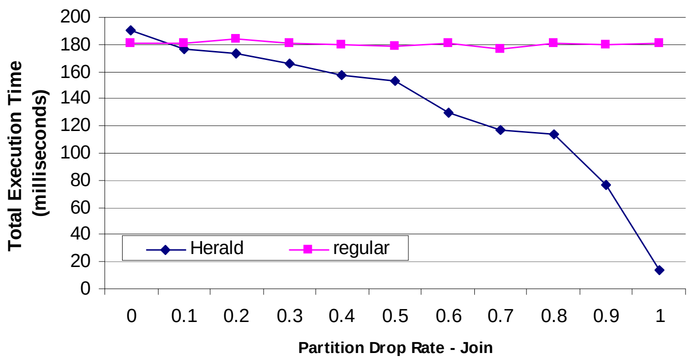


Figure 15.3: Par. Size:20, σ_{Select}=0.2, σ_{Join}=0.1.

Second, we now set the average partition size to be higher, i.e., 100 tuples. The result is shown in Figure 15.4. This time, we can see that the trend is similar to the case when the average partition size is 50 tuples. However, the gains at each point are significantly larger. This is because the amortized optimization overhead is reduced by batching more data each time.

Varying operator selectivity. In the third set of experiments, we study how the selectivity of the underlying operator affects the performance gains

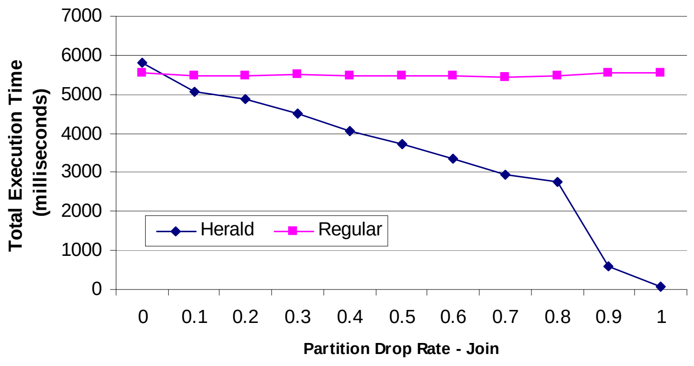


Figure 15.4: Par. Size:100, σ_{Select}=0.2, σ_{Join}=0.1.

by herald-driven approach. The selectivity of the underlying operator, which in our experiment is the select operator, determines the number of tuples to be processed by the join operator. In the previous experiments, the selectivity of the select operator is set to be 0.2. In this experiment, we set the selectivity of the select operator to be 0.1 and 0.4 respectively. The average partition size is 50 tuples and the join selectivity is 0.1. The results are shown in Figures 15.5 and 15.6 respectively.

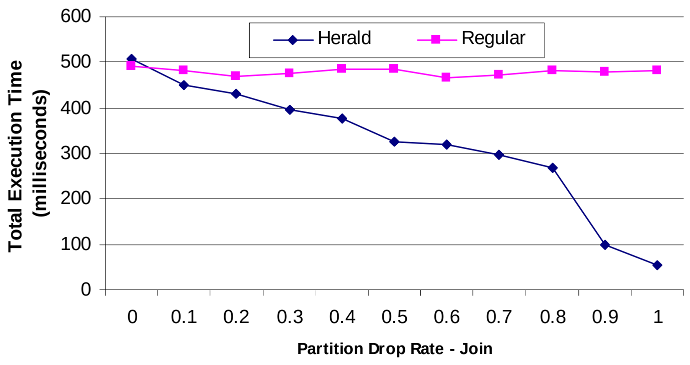


Figure 15.5: Par. Size:50, σ_{Select}=0.1, σ_{Join}=0.1.

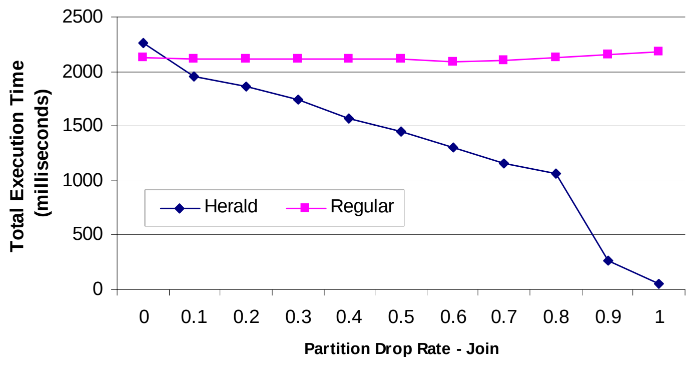


Figure 15.6: Par. Size:50, σ_{Select}=0.4, σ_{Join}=0.1.

We can see that as the selectivity of the select operator increases, the performance gains by the herald-driven SQO also increases. This is because when the selectivity of the select operator increases, more data needs to be processed by the join operator. Bigger performance gains can thus be achieved by employing herald-driven optimization.

Performance impact by pass rate. Finally, we evaluate the performance impact of the pass rate. We fix the average partition size to be 50 tuples, the selectivities of the select and the join operators to be 0.2 and 0.1 respectively. We then vary the partition pass rate from 0 to 1 by 0.1.

The result shown in Figure 15.7 shows that similar to the drop rate, the herald approach has significantly increased performance gains as the partition pass rate increases.

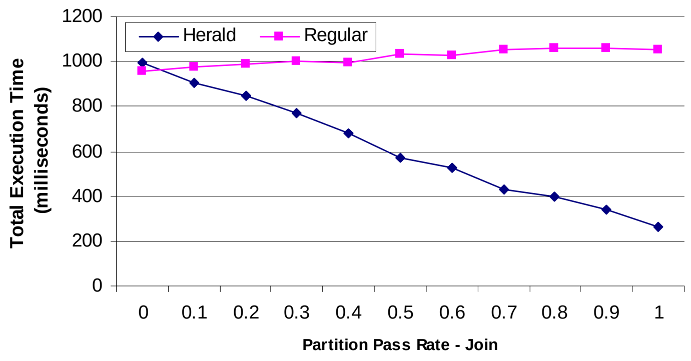


Figure 15.7: Par. Size:50, σ_{Select}=0.2, σ_{Join}=0.1.

15.4 Evaluating Select Optimizations

Next, we evaluate the performance impact achievable by the select optimizations. We now show the experimental results from varying the partition drop rate, i.e., to test the optimization of select data skipping.

We first set the average partition size to be 50 tuples, the selectivity of the select operator and the join operator to be 0.1 and 0.1 respectively. No performance gains can be observed by the herald approach. We then increase the average partition size to be 100 tuples. Again, the curve of the herald approach meets the curve of the regular approach at each point. We further increase the average partition size to be 200 tuples, and also reduce the selectivity of both the select and the join operators to be 0.01. This time we observe a reduction in execution time by the herald approach, as shown in Figure 15.8.

The reason for no significant performance gains by select data skipping for relatively small partition sizes and moderate select and join selectivities

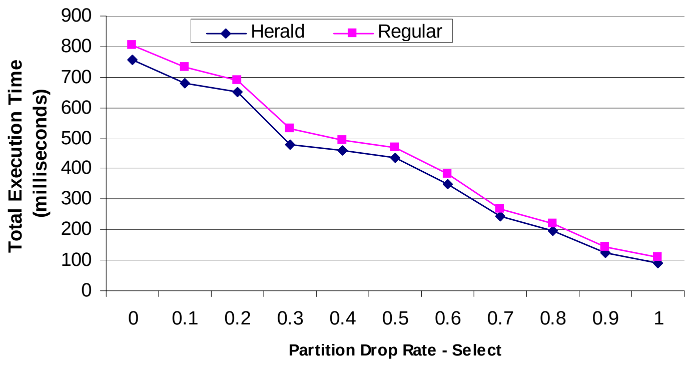


Figure 15.8: Par. Size:200, σ_{Select}=0.01, σ_{Join}=0.01.

is because in these cases, the cost of the join operations is dominant. Hence the performance gains by eliminating some of the operations by the select operator is shadowed by the dominating join operations.

Next, we proceed to measure the execution time of the select operator in isolcation. This allows us to observe that the reduction in select execution time by using the herald approach is proportional to the partition drop rate. Also, the performance gains of the select operator by select elimination are proportional to the partition pass rate.

Summary of observations. We observe significant performance gains by using our herald-driven optimization techniques when partition drop/pass rate is medium or high. Also, the performance gains by the herald approach increase as the partition drop/pass rates increase, as the selectivities of the underlying operators increase, and as the partition sizes increase. In addition, all experiments include the optimization reasoning overhead which is shown to be negligible.

Chapter 16

Related Work

Using metadata to optimize queries has been extensively studied for traditional databases [31, 70], commonly called semantic query optimization (SQO). Existing work focuses on employing schema knowledge and integrity constraints that are available at query compilation time to select the most efficient query plan for execution. In these works, optimization is conducted at query compilation time. The direct equivalent in the stream contexts [16] is to employ the integrity constraints to optimize the memory usage by purging operator states in a timely manner. Rather than integrity constraints, we instead focus on metadata about attribute values. In most stream applications, such metadata is largely unavailable during compilation time. The new challenge that we thus must tackle is to conduct efficient query optimization at runtime. In addition, since the metadata we consider is interleaved with the actual data and has diversified lifespans, the optimized query plans may have different yet overlapped application scopes. Therefore, instead of using a single execution plan during the entire query execution, we must devise a scheme to support multiple concurrent SQO plans, each employing differently tuned execution logic for different data substreams.

Our herald metadata model (Section 11.1) extends from the punctuation model [93]. Heralds can be practically obtained in many cases, e.g., by employing a stream buffer and a preprocessor, without relying on the applications to provide them. Existing work utilizing punctuations focuses on the design of execution algorithms of query operators such as joins [40, 41] or the compile-time detection of the "unsafe" queries that may need to maintain unbounded operator states [74]. [74] exploits punctuations to mark the sliding window boundary to handle disorder. We instead focus on query optimization at the query plan level and propose the configurable execution logic for the common SPJ operators.

As stream applications proliferate, much research has focused on runtime query optimization. [13, 57, 104] exploit runtime statistics on operator selectivities to adaptively reorder the operators in the query execution plan. Unlike our work, at any point in the query execution, only a single logical plan is set up for execution.

In the execution paradigm of Eddies [10, 38], individual tuples are adaptively routed through the operator network instead of using one or even a few query paths determined at compilation time. This achieves fine-grained query execution adaptation. Our work differs from Eddies in two aspects. First, the adaptation of Eddies is selectivity driven while our adaptation is semantics driven. Second, we achieve adaptations by employing embedded operator control logic while Eddies would instead require to implement three different (join) operators with three customized logics and then the physical routing would end up actually mimicing the multiple logical plans. Thus a tuple may need to be routed to three versions of a join operator. In this sense, we achieve a more lightweight plan adaptation.

Similar to [18], we employ different plans for different data. In [18], for different data, only the operator execution orders are different. This follows the traditional idea of (syntactic) query optimization. In our work, to process a particular batch of data, some operators may be skipped while other operators may be executed in a more efficient way.

Part III

Index Tuning for Parameterized Streaming Groupby Queries

Chapter 17

Introduction

17.1 Groupby Queries in Stream Applications

Groupby queries with aggregate functions are extensively used in data stream applications to provide statistical summaries for monitoring and real-time analysis.

Network monitoring applications run groupby queries over network packet data to monitor network traffic or to measure network performance [87, 103]. As an example, Query 1 in Figure 17.1 monitors the total traffic of source-destination pairs in the past 20 minutes over link B.

Online transaction systems conduct groupby queries over transaction records to provide real-time recommendations to users. Query 2 (Figure 17.1) in online auctions [87, 94] computes the number of bids placed in the past 24 hours grouped by auction category, buyer's state and occupation.

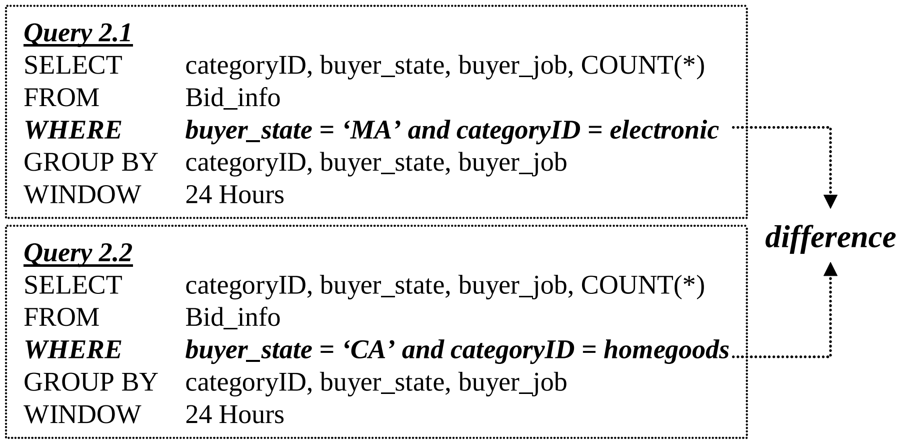
Real-time business intelligence (BI) applications [50] where data arrives in a data warehouse or OLAP system via a trickle feed, continuously

```
Query 1 (Network Traffic Monitoring)
SELECT      srcIP, destIP, SUM(len)
FROM        Packets
WHERE       collectorID = 'B'
GROUP BY    srcIP, destIP
WINDOW      20 Minutes
```

```
Query 2 (Auction Recommendation)
SELECT      categoryID, buyer_state,
            buyer_job, COUNT(*)
FROM        Bid_info
GROUP BY    categoryID, buyer_state,
            buyer_job
WINDOW      24 Hours
```

Figure 17.1: Example Streaming Groupby Queries.

or every few minutes, also use groupby queries extensively.



```
Query 2.1
SELECT      categoryID, buyer_state, buyer_job, COUNT(*)
FROM        Bid_info
WHERE       buyer_state = 'MA' and categoryID = electronic
GROUP BY    categoryID, buyer_state, buyer_job
WINDOW      24 Hours
```

```
Query 2.2
SELECT      categoryID, buyer_state, buyer_job, COUNT(*)
FROM        Bid_info
WHERE       buyer_state = 'CA' and categoryID = homegoods
GROUP BY    categoryID, buyer_state, buyer_job
WINDOW      24 Hours
```

Figure 17.2: Similar (Not Identical) Groupby Queries.

These applications often experience a large number of concurrent users that desire the results of *similar but not identical* groupby queries [71]. For example, a Massachusetts user that plans to buy TV may want to receive recommendations based on Query 2.1 in Figure 17.2, while a California user who wants to buy home goods may need recommendations based on Query 2.2 in Figure 17.2. These two queries differ only in their WHERE clauses. To execute large numbers of such similar but different queries separately faces scalability and performance problems because the selection operations cannot be shared and overlapping states may be maintained.

In addition, the user preferences are potentially changing with users' current needs. A user may at one time require recommendations on home goods because she is moving to a new house, but at another time needs recommendations on toys because she is expecting a child. For large numbers of users, such changing needs may result in the frequent addition of new queries and removal of existing queries, thus incurring significant and repetitive query optimization costs.

It has been observed that in many applications including the ones mentioned above and the applications where users use resource-limited devices, such as PDAs, users prefer to see the aggregate results *on demand* rather than being continuously interrupted and overwhelmed by continuous query result updates triggered by the arrival of every new tuple in the input stream [9, 26]. For example, users may want to see the recommendations based on queries in Figure 17.2 whenever they log into the auction system or when they synchronize their PDA. Such on-demand output model is resource-efficient, potentially saving significant CPU time on continuously updating the aggregate results, and the bandwidth and server load on transmitting those possibly never-accessed results. These resources can instead be devoted to provide high-quality and quick responses upon user requests.

Such on-demand query output model also provides a natural way to specify diversified user-specific query parameters on a common groupby query. In other words, a query template would enable users to conveniently specify, for instance, the group selection when they request results.

17.2 Parameterized Streaming Groupby Query

It has been recognized that an abstraction in the form of a parameterized query not only facilitates multi-query optimization but also fosters the optimized execution by maximal sharing of computations and operator state [27]. In this spirit, we propose the notion of a **Parameterized Streaming Groupby Query** (or **PSGB query**) that represents a potentially infinite number of (i.e., non-parameterized) groupby queries that are instantiated at runtime by user requests. We design a *PSGB operator* to achieve the shared execution of a large number of PSGB instantiations without having to analyze them when they are instantiated by runtime user requests. In particular, by employing the PSGB operator, the memory for maintaining the groupby state and the CPU time for organizing the groupby state to facilitate the construction and the retrieval of the groups can be shared among all PSGB instantiations.

PSGB Query that incorporates Query 1 in Figure 1

```
SELECT     srcIP, destIP, <agg-func-list>: SUM(len)
FROM       Packets
WHERE      collectorID = ‘B’ and <predicates-on-grouping-attr>
GROUP BY   srcIP, destIP
WINDOW     <window-length>: 60 Minutes
```

User Request: [srcIP=216.239.37.4; 20 Minutes]

Figure 17.3: Example PSGB Query and User Request.

A PSGB query has a basic structure to be specified at query registration time, and *dynamic parameters* to be specified at runtime by user requests. Figure 17.3 shows an example PSGB query with dynamic aggregate functions (SUM(len) by default), selection predicates and window lengths (60 minutes by default), and a user request specifying the additional selection predicate *srcIP=216.239.37.4*, and a *20-minute* suffix window that overrides the default window length.

17.3 Issues with PSGB Query Processing

The PSGB operator differs from the regular groupby operator in that it needs to additionally conduct the *selection* operation that picks up the user-desired data based on dynamic selection predicates. In our targeted applications, large and quickly-evolving groupby states (caused by rapid data arrivals), and high volumes of user requests with selective predicates concerning diverse sets of attributes can be observed. Hence, to efficiently perform the selection operation is essential to achieving good PSGB query performance.

Since we focus on equality selections in PSGB instantiation, a hash-based index appears to be a good fit for organizing the groupby state [58]. A traditional one-level hash index, be it on a single attribute or on multiple attributes, are only appropriate in the special case when the hash keys form a subset of the search keys, i.e., the attributes involved in selection predicates. Otherwise a full scan of the groupby state is required. Hence it is not flexible enough to support widely-varying dynamic selection predicates. A more powerful yet lightweight index method is needed.

In traditional databases, to expedite the aggregation, certain aggregate results may be pre-computed and materialized [61, 66]. Prior work in stream contexts [103] has studied how to share the aggregate results among statically specified groupby queries differing only in their grouping attributes. In our problem, due to the quickly-evolving groupby state, and large diversity of selection predicates and aggregate functions potentially involved in user requests, most aggregate results tend to be not used frequently enough to justify the pre-computation. Even if they are chosen to be pre-computed, an index is still needed to expedite the search over the pre-computed results based on selection predicates.

In summary, an index mechanism that can effectively organize the quickly evolving groupby state to support efficient lookup of the data based on dynamic selection predicates is a critical technology to be developed for PSGB query processing. Moreover, to withstand the fluctuations in data and query workloads, the groupby operator should be able to quickly observe the changes at runtime and then tune the index accordingly. In streaming data processing, the index solution must be amendable to enable efficient on-line migration [104] from one index structure to another. To the best of our knowledge, no existing work has focused on index design and tuning for streaming groupby operators. In this work, we propose the first solution to this problem.

17.4 Our Approach: Groupby Index Tuning

Our **index tuning** approach involves the interleaved execution of three modules – *index selection*, *index assessment* and *index migration*. During query execution, the index assessment module is periodically executed. In each run, it first invokes the index selection module to derive the optimal or near-optimal index configuration based on the up-to-date workload statistics collected at runtime [34]. Then the cost of using the new configuration is compared with the cost of using the current one, both based on the new workload. If the new cost plus the potential migration cost is lower than the current cost, the index migration module is invoked to migrate the groupby state to the new index structure.

In this work, we focus on the problem of **index selection**, the first and most critical step in index tuning. As observed by our experiments (Chapter 22), the query performance may degrade significantly if an inappropriate index strategy or an inefficient index configuration algorithm is used. Our contributions of this work are:

1). We propose the PSGB query as an abstraction of a large number of runtime instantiated queries. This formulation leads to efficient optimization of these groupby queries, i.e., a single PSGB operator can be designed to achieve resource sharing among these queries without having to analyze them individually as they are instantiated.

2). We employ a lightweight *IMP index* solution to management PSGB state for supporting efficient data lookups required by PSGB instantiations with diversified selection conditions (Section 19.2). While existing work in streaming databases uses hash-based indices for efficient state management, we show that our proposed solution beats existing solutions by a 9-fold performance improvement (without usage of any additional memory space) for large window sizes.

3). Our key contribution lies in tackling index tuning in streaming context. We design the *EPrune* index selection algorithm that is guaranteed to find the optimal IMP configuration. By properly pruning candidates, the complexity of EPrune can be significant reduced compared to exhaustive search, sometimes more than ten-fold (Section 20.3).

4). To meet the efficiency needs that are more important for online index tuning than guaranteed optimality, we also design a time-efficient greedy index selection algorithm named *RGreedy* and equip it with three alternative search heuristics. RGreedy is shown to find the near-optimal IMP configuration with observed polynomial complexity even in large search spaces (Section 20.4).

5). Our experimental study conducted in the CAPE stream processing system [83] shows that the IMP index always wins over the state-of-the-art index methods. RGreedy with PCL and Hybrid heuristics finds the optimal IMP configurations in all of our extensive test cases. For large search spaces, when EPrune takes hours to finish, RGreedy always terminates within seconds. Moreover, the PSGB operator with runtime index tuning outperforms the operator with a fixed index configuration (Chapter 22).

Chapter 18

Background

18.1 PSGB Query

In this work, we consider the processing of PSGB queries as shown in Figure 18.1. The dynamic parameters are underlined, while all other query constructs are statically specified. The WINDOW clause specifies a *suffix* window that ends at the current time [9]. It indicates the length of the data history to be queried. If a user request doesn't specify a window, the default window which is the largest allowed window is assumed [9]. The predicates in the WHERE clause are selection predicates. The dynamic selection predicates are conjunctive *equality* conditions on the *grouping attributes*, i.e., they select the groups to be produced aggregate results. If the user request doesn't specify the grouping attributes, a default set of grouping attributes will be used. Any user-specified grouping attributes must belong to the default grouping attributes.

Consider the example PSGB Query in Figure 17.3. Suppose at time T

SELECT	***<group-attr-list>***, ***<agg-func-list>****: default-agg-funcs*
FROM	<stream-name>
WHERE	<static-preds> and ***<dynamic-preds>***
GROUP BY	***<group-attr-list>****: default-group-attr-list*
WINDOW	***<window-length>****: default-window-length*

Figure 18.1: PSGB Query Specification.

during the execution, tuples in the current groupby state belong to three groups: (1) *(srcIP = 216.239.37.4, destIP = 216.239.2.3)*, (2) *(srcIP = 216.239.37.4, destIP = 216.239.5.8)*, and (3) *(srcIP = 216.239.23.10, destIP = 216.239.6.1)*. Assume a user request [– ; srcIP = 216.239.37.4; –] is received at time T (– denotes that no values are specified). Since it specifies no condition on destIP, the two aggregate results respectively on groups (1) and (2) will be returned. That is, the above user request selects two aggregate results from the complete result set of Query 1 (Figure 17.1) at time T that match the dynamic predicates specified in the request.

The *PSGB* operator is used to handle 1) the *dynamic* selection predicates to dynamically filter out the groups not of interest to the user, and 2) the groupby operations to produce the aggregate results for the selected groups. Our goal is to propose the index design and index selection approach for the PSGB operator to minimize its processing costs for a given query workload.

18.2 Execution of PSGB Operator

The PSGB operator uses the following execution logic. It maintains a state to hold all tuples residing in the current suffix window. There are two tasks for the operator: 1) process input *data tuples* and 2) process *user requests*. As a new tuple t is received, it is first inserted into the state. Then, if the window is time-based, the timestamp of t is used to purge the tuples in the state that have expired from the window (we assume the input tuples are received in the order of their timestamps). If the window is count-based, the oldest tuple in the state will be removed. As a user request is received, the groupby state is probed and the aggregate results are produced based on matching tuples. If the aggregate results had been pre-computed, those pre-computed values would be retrieved and output. In the following, we first assume the typical setting that no aggregate results are pre-computed. Later (Section 21.2) we discuss how our proposed techniques can be naturally generalized to support the scenario of pre-computed aggregate results.

Chapter 19

Index Design for PSGB Operator

19.1 Requirements on Index Design

As discussed in Section 17.3, traditional one-level hash index is not capable of handling selections that involve varied or even disjoint sets of attributes. On the other hand, to build multiple indexes over this shared state for different subsets of attributes also suffers from serious difficiencies. First, each tuple would require multiple references, causing potentially high memory overhead. The groupby state should be maximally kept in memory to achieve real-time query responses [12]. Thus memory is a highly precious resource. Also, the maintenance cost of multiple indexes regarding streaming data would likely be considerable. Therefore, we need a new index solution that carefully balances query processing with index maintenance costs, i.e., that meets the contradictory demands of being lightweight (for index maintenance and evolution) yet efficient (for query processing). The following design guidelines apply:

1). The index should benefit the processing of a large number of rather diverse requests.

2). The index structure should require minimal maintenance effort when processing data updates.

3). The index structure should be memory-efficient to be maximally maintained in main memory.

4). The index should be lightweight to be easily migratable when the workload experiences significant changes.

19.2 IMP: Importance-Based Partition Index

In view of these requirements, we employ a multi-level prioritized hash index as shown in Figure 19.1. We name it *IMP index* (for IMportance-based Partition index).

The IMP index divides tuples in the groupby state into 2^B partitions and uses a B-bit string to represent the address of each partition. B is derived from memory constraints on the index structure. Each attribute A_i (1≤A_i≤N) is assigned b_i contiguous bits such that 0≤b_i≤B and $\sum_i b_i$=B. Then A_i corresponds to 2^{b_i} partitions. Henceforth, we use the vector $< b_1, b_2, ..., b_N >$ to represent the bits assigned to the N attributes A_1, ... A_N.

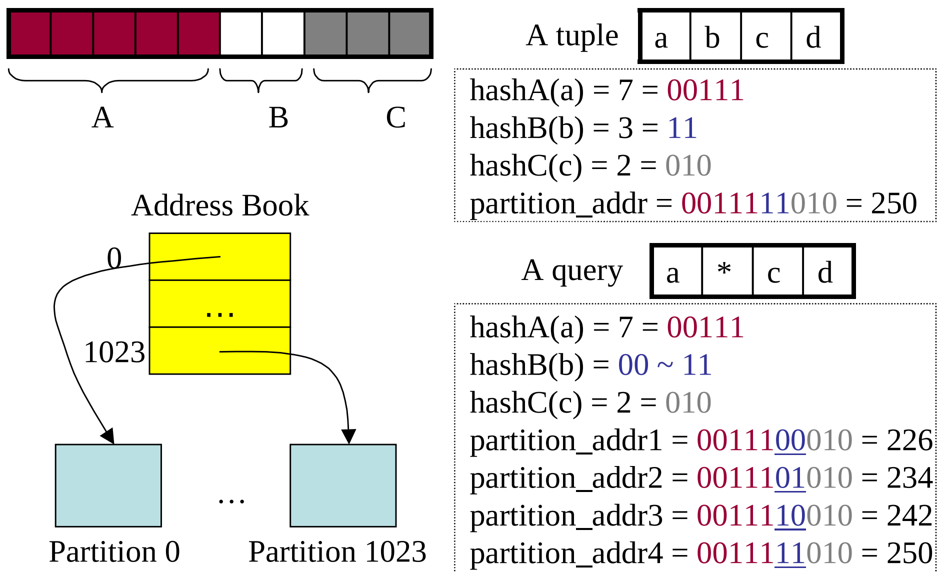


Figure 19.1: IMP Index.

We name it the **IMP configuration**. The attributes corresponding to non-zero bits are called *indexed attributes*.

A hash function is used to map the values of each indexed attribute into a bit string of the desired length. For an input tuple, its values of all the indexed attributes are used to compute the address of the *single* partition it should be placed into. Within each partition, tuples are ordered chronologically to facilitate the invalidation of tuples based on sliding window semantics (i.e., tuples that have expired from the largest window should be removed). For a user request, without any wildcards, i.e., all attributes are specified, only a *single* partition needs to be probed to obtain all tuples that match the query. If it contains one or more wildcards, while several partitions need to be probed, it still tends to be a small subset of partitions compared to a full scan. Figure 19.1 shows an example IMP configuration and the partition address computation for a tuple and a request respectively. This request doesn't specify attribute B. Since attribute B occupies 2 bits, 2^2=4 partitions need to be probed to answer the request.

19.3 Advantages of IMP Index

First, the IMP index is lightweight. It only stores the addresses to all partitions in a hash table. The partition addresses are computable with a very simple formula. Hence it is memory-efficient and easy-to-maintain, especially compared to tree-structured indexes [58]. It is also easily-migrated, with no need to rebuild any auxiliary structures.

Second, the IMP index simplifies the index selection process since it unifies the two decisions on which attributes to index and then how to index them into a single decision on how many address bits to give to each attribute. If an attribute is assigned 0 bits, this attribute is not indexed.

The most important feature of the IMP index is that the address bit allocation among the indexed attributes affects query performance non-trivially. The more bits are assigned to an attribute, the smaller the number of partitions to be probed to answer the requests that specify conditions on this attribute. Therefore, intuitively we should assign more bits to frequently queried attributes (i.e., important attributes) to reduce the overall processing cost.

We use an example to illustrate how the IMP index configuration affects the query performance. Assume a PSGB query with two attributes A and B. Suppose 60% of the requests only specify a selection condition on attribute A and 40% of the requests only specify a selection condition on attribute B. Assume we use a 10-bit partition address (2^{10}=1024 partitions in total). Let's compare two IMP configurations. The first configuration assigns 6 bits to attribute A and 4 bits to B. Then we need to examine 16 partitions to answer the requests involving A and 64 partitions to answer the requests involving B. On average $0.6 \cdot 16 + 0.4 \cdot 64$=36 partitions need to be examined to answer a single request. The second configuration assigns 4 bits to attribute A and 6 bits to B. On average $0.6 \cdot 64 + 0.4 \cdot 16$=45 partitions need to be checked to answer a single request. Since attribute A is queried more frequently than B, assigning more bits to A achieves a smaller overall probe cost.

19.4 IMP Index Cost Analysis

We use *unit processing cost (UPC)* [68] as the measurement of IMP configurations. UPC is defined to be the cost for processing the data tuples and the user requests received within a time unit. Clearly the IMP configuration with the minimum UPC for the given workload is the optimal index configuration. Henceforth, we use the terms processing cost and unit processing cost interchangeably.

Equation 19.1 computes the UPC. It includes the tuple processing cost C_T and the request processing cost C_R. C_T includes the cost for hashing the indexed attributes ($C_{hash,T}$), inserting tuples into the index (C_{insert}) and deleting tuples from the index (C_{delete}). C_R includes the cost for hashing the indexed attributes specified in the requests ($C_{hash,R}$), probing the resultant partitions (C_{probe}) and generating aggregate results (C_{agg}).

$$C = C_{hash,T} + C_{insert} + C_{delete} + C_{hash,R} + C_{probe} + C_{agg} \tag{19.1}$$

Among these costs, C_{insert}, C_{delete} and C_{agg} are independent of the IMP configuration. Hence the index selection should only consider the IMP-relevant cost (denoted as C_V). C_V is the sum of $C_{hash,T}$, $C_{hash,R}$ and C_{probe}, as computed by Equation 19.2 using the notations defined in Table 19.1.

Notation	*Meaning*
r_k	a user request
p_i	a selection pattern
λ_d	# of tuples received within a time unit
λ_r	# of user requests received within a time unit
C_h	average cost for computing a hash function
C_c	average cost for conducting a value comparison
N_A	# of indexed attributes
N_{A,r_k}	# of indexed attributes specified in request r_k
W_{r_k}	window length (in # of time units) of request r_k
B_{r_k}	# of bits assigned to all attr. specified in request r_k
N_{A,p_i}	# of indexed attributes specified in pattern p_i
W_{p_i}	window length of pattern p_i
B_{p_i}	# of bits assigned to all attr. specified in pattern p_i
F_{p_i}	frequency of p_i

Table 19.1: Notations.

$$\begin{aligned} C_V &= C_{hash,T} + C_{hash,R} + C_{probe} \\ &= \lambda_d N_A C_h + \lambda_r \sum_{r_k \in R} (N_{A,r_k} C_h + \frac{\lambda_d W_{r_k}}{2^{B_{r_k}}} C_c) \end{aligned} \tag{19.2}$$

For an incoming tuple t, N_A hash operations need to be conducted to find the exact partition to insert t. Hence $C_{hash,T}$ equals $\lambda_d N_A C_h$. For a user request r_k, its associated hash cost depends on the number of indexed attributes specified in r_k. As r_k ranges over R, the set of all user requests that arrived within a time unit, $C_{hash,R}$ equals $\lambda_r \sum_{r_k \in R} N_{A,r_k} C_h$. The probing cost associated with r_k equals the total number of partitions probed, $2^{B-B_{r_k}}$, times the average number of tuples to be probed in each partition, $\frac{\lambda_d W_{r_k}}{2^B}$, and then times the value comparison cost C_c. As r_k ranges over R, C_{probe} equals $\lambda_r \sum_{r_k \in R} \frac{\lambda_d W_{r_k}}{2^{B_{r_k}}} C_c$. Since in our targeted applications, the streaming data arrival rate (λ_d) and the request window length (W_{r_k}) are both usually high, then $\sum_{r_k \in R} N_{A,r_k} C_h << \sum_{r_k \in R} \frac{\lambda_d W_{r_k}}{2^{B_{r_k}}} C_c$. Hence C_V can be approximated by Equation 19.3.

$$C_V \approx \lambda_d N_A C_h + \lambda_r \sum_{r_k \in R} \frac{\lambda_d W_{r_k}}{2^{B_{r_k}}} C_c \tag{19.3}$$

19.5 Selection Pattern

We can see that the parameters that affect the processing cost and vary between different workloads are λ_d, λ_r and W_{r_k}. Also, which attributes are frequently queried by user requests instead of what particular values are specified for these attributes is relevant to the processing cost. We thus define the concept of *selection pattern* as an abstraction of all dynamic selection predicates (or *DSP*) that involve the same set of attributes. For ease of presentation, we assume an order for all attributes in the stream schema. In a selection pattern, the attributes involved in DSP are denoted by their respective names. They are also called *specified attributes*. Any attribute not involved in DSP is represented by a wildcard $*$. Consider the PSGB query in Figure 17.3 and two user requests with DSP being "srcIP = 216.239.37.4" and "srcIP = 207.46.250.19" respectively. Both DSPs match selection pattern <srcIP, $*$>. Given N attributes, they are in total 2^N distinct selection patterns. P denotes the set of all possible selection patterns.

For cost analysis purposes, we define the *frequency* and the *window length* of selection patterns.

Definition 8 *The* **frequency of a selection pattern** p_i *in a workload* D*, denoted as* F_{p_i}*, equals* $\frac{L}{|R|}$*, with* L *being the number of user requests in* D *with selection pattern* p_i *and* $|R|$ *being the total number of user requests in* D*.*

Definition 9 *The* **window length of a selection pattern** p_i *in a workload* D*, denoted as* W_{p_i} *equals* $\frac{\sum W_{r_k}}{|R_{p_i}|}$*, with* r_k *ranging over all requests in* D *and* $|R_{p_i}|$ *being total number of user requests with selection pattern* p_i*.*

Based on these definitions, we derive that $\sum\limits_{r_k \in R} W_{r_k} = \sum\limits_{p_i \in P} W_{r_i} F_{p_i}$, and $B_{r_k} = B_{p_i}$ if the user request r_k's selection predicate matches pattern p_i. Given statistics on selection pattern frequencis and window lengths in a workload D, C_V can instead be approximated by Equation 19.4.

$$C_V \approx \lambda_d N_A C_h + \lambda_r \sum_{p_i \in P} \frac{\lambda_d W_{p_i} F_{p_i}}{2^{B_{p_i}}} C_c \tag{19.4}$$

Chapter 20

Index Selection Algorithms

20.1 Index Selection Algorithms

20.2 Index Selection Problem Definition

The appropriate configuration of the IMP index is the key to achieving good PSGB processing performance. We hence define our index selection problem in Definition 10.

Definition 10 *Given a query with N attributes A_i (1≤i≤N) and B address bits (i.e., 2^B partitions) assumed by memory constraints, the* **Index Selection Problem** *is to find an* IMP configuration *that minimizes the cost for processing the given groupby workload with the following parameters (all are average values):*

- *data arrival rate λ_d;*
- *request arrival rate λ_r;*
- *selection pattern frequencies F_{p_i} ($1 \leq i \leq 2^N$);*

- *selection pattern window length W_{p_i} ($1 \leq i \leq 2^N$).*

According to Equation 19.4, for each selection pattern p_i, F_{p_i} and W_{p_i} together affect the IMP index cost. For simplicity, thereafter we assume W_{p_i} is a constant for all p_i, i.e., all PSGB instantiations use the default window. Also we assume the grouping attributes are fixed in the PSGB query. If the aggregate results are not pre-computed, C_V is not affected by grouping attributes in PSGB instantiations. The extensions for handling pre-computed aggregate results are discussed in Section 21.2.

Search space. Given N attributes and B address bits, any IMP configuration that has ≤Min(N, B) attributes to share B bits is a potential index solution. To index k (1≤k≤Min(N, B)) attributes, the total number of distinct IMP configurations equals the total number of ways to place (k-1) marks among (B-1) positions to divide a B-bit string into N pieces, which is $\begin{pmatrix} B-1 \\ k-1 \end{pmatrix}$. There are $\begin{pmatrix} N \\ k \end{pmatrix}$ ways to select k attributes out of N attributes. Hence the search space can be computed by Equation 20.1. As N and B grow, the search space will grow exponentially. For example, when B=16 and N=4, there are in total 969 candidates. However, as N approaches to 10, the search space contains 2,042,975 candidates.

$$SearchSpace(N, B) = \sum_{k=1}^{Min(N,B)} \begin{pmatrix} N \\ k \end{pmatrix} \begin{pmatrix} B-1 \\ k-1 \end{pmatrix} \tag{20.1}$$

Next, we propose two index selection algorithms that are suitable for different index selection and tuning scenarios.

20.3 EPrune: Pruned Exhaustive Search

The *EPrune* algorithm is guaranteed to find the optimal IMP configuration by conducting a cost-based search. The algorithm is composed of two tasks: (1) *candidate construction* that constructs candidate IMP configurations, and (2) *cost evaluation* that evaluates the given candidates and returns the one with the minimum processing cost. To be memory-efficient, the algorithm iteratively generates each IMP configuration and directly feeds it to the cost evaluation module, i.e., in a pipelined fashion. In other words, these two tasks are interleaved.

We construct all possible IMP configurations by progressively including more attributes. Figure 20.1(a) depicts the process. The y axis represents the attributes included and the x axis represents the number of bits assigned to the respective attribute sets. Entry (i, j) contains all possible IMP configurations $< b_1, b_2, ..., b_N >$ satisfying the following conditions: 1) 0≤b_k≤B for 1≤k≤j; 2) b_m=0 for j<m≤N; and 3) $\sum_{z=1}^{j} b_i$=i. Therefore, all IMP configurations over attribute set <A_1, A_2, ..., A_N> using B bits can be found at entries (B, k) with 1≤k≤N. Candidates in these entries achieve the finest partition by using all B bits. This is a prerequisite to achieving minimum processing cost under the B-bit constraint. Hence, only these candidates will be evaluated by the cost evaluation module.

The arrow in the figure indicates the order in which these candidates are generated. To construct candidates for (B, j) (0<j≤N), all candidates of (p, j-1) (0≤p≤B-1) are iterated over and b_j in each of them is set to be (B–p), as shown by the shaded blocks in Figure 20.1(a).

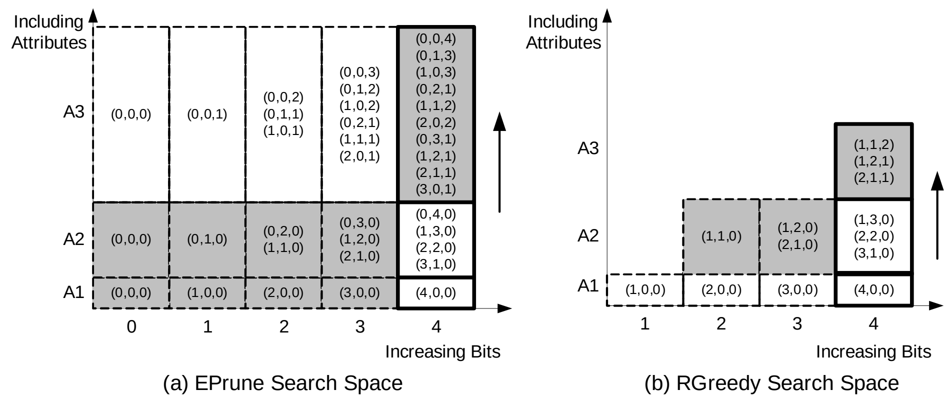


Figure 20.1: Algorithm Search Spaces.

The cost evaluation module evaluates the processing cost for each constructed IMP configuration using Equation 19.4 and keeps the optimal IMP configuration found thus far.

Pruning strategy. If an attribute is queried infrequently enough so that the overhead for indexing it exceeds the gained probe cost reduction, this attribute will not be included into the index. An extreme case is that an attribute never appears in the selection predicate of any user request. If those attributes can be detected and pruned before the search starts, the search space may be significantly shrunk. We achieve this goal by employing a benefit function.

By indexing an attribute A, the maximum probe cost reduction is $\lambda_r \sum\limits_{p_i \in P_A} \lambda_d W_{p_i} F_{p_i} (1 - \frac{1}{2^N}) C_c$ (assuming A is assigned N bits). P_A represents the set of patterns in which A is specified. The increased hash cost is $(\sum\limits_{p_i \in P_A} F_{p_i} \lambda_r + \lambda_d) C_h$. Hence the maximum benefit of A, denoted as MB(A), can be computed by Equation 20.2. The attributes with negative MB values can be pruned.

$$MB(A) = \lambda_r \sum_{p_i \in P_A} \lambda_d W_{p_i} F_{p_i} (1 - \frac{1}{2^N}) C_c - \sum_{p_i \in P_A} (F_{p_i} \lambda_r + \lambda_d) C_h \qquad (20.2)$$

Theorem 3 *The IMP configuration output by the EPrune algorithm achieves minimum UPC (i.e., optimal).*

Proof. The *MB* funtion estimates the maximum pure gains from indexing an attribute. If the MB value of an attribute is negative, it can not be beneficial to index this attribute. Hence the pruning won't prune any attribute that will eventually be indexed by the optimal IMP configuration. Since EPrune conducts exhausive search after pruning, it is guaranteed to find the optimal IMP configuration.

The pseudo code of EPrune is shown in Algorithm 9.

Algorithm 9 EPrune Algorithm

```
Input: Attribute set $S_A$, integer B, request pattern set $S_R$
Output: An IMP configuration with minimum cost

optimal_config := $< 0, ..., 0 >$; /* no attr. is indexed. */
optimal_cost := Cost(optimal_config, $S_R$);
$S_A$.Remove_NonBeneficial_Attr();
Initialize candidate construction module cc by $S_A$ and B;
while cc.Has_More_Configs() do
   new_config := cc.Get_Next_Config();
   new_cost := Cost(new_config, $S_R$);
   if new_cost < optimal_cost then
      optimal_config := new_config;
      optimal_cost := new_cost;
   end if
end while
Return optimal_config;
```

Complexity. In this work, we define the **complexity** of an index selection algorithm to be the total number of IMP candidates examined by the algorithm. By employing the pruning strategy, with M attributes being pruned, the complexity of EPrune can be computed by Equation 20.3.

$$\sum_{k=1}^{Min(N-M,B)} \left(\begin{array}{c} N-M \\ k \end{array}\right) \left(\begin{array}{c} B-1 \\ k-1 \end{array}\right). \tag{20.3}$$

The pruning can lead to huge search savings. For example, when B=16 and N=10, the complexity of EPrune is 2,042,975. Suppose M=2, the complexity is reduced to 245,157, by a factor of 10.

For a fixed B value (e.g., assume B=16), when N_r=N-M is small, the optimal IMP configuration can be quickly found by EPrune. However, as the value of N_r increases, the complexity increases precipitously. Then EPrune may take hours or even days to finish. In streaming environments, data/request statistics may often experience unpredictable changes. Therefore, it is not practical to spend major time to search for an optimal index configuration that may become sub-optimal shortly after. Rather time-efficient algorithms are needed to quickly find a near-optimal configuration even in large search spaces, as introduced next.

20.4 RGreedy: Greedy Algorithm

It is clearly acceptable to trade the optimality for timeliness in many stream applications for which real-time answers are critical. We thus design RGreedy, a heuristic-based greedy algorithm, that doesn't guarantee the optimality but is shown to be useful for a huge variety of practical cases.

The basic idea of RGreedy is to first rank each attribute by the benefit that may be obtained from indexing that attribute (i.e., the *importance* of the attribute). Then the algorithm progressively considers the attribute with the next highest ranking for inclusion into the index. At each step, RGreedy constructs the new IMP candidates, assuming all attributes considered at this step are being indexed. If the best IMP configuration among these new candidates achieves less cost than the best IMP configuration derived from all previous steps, the algorithm continues. Otherwise, the algorithm terminates and the IMP configuration found to be best thus far is returned. The intuition is that if it is not beneficial to index an attribute, to index a less important attribute is unlikely to achieve cost savings. Algorithm 10 shows the pseudo code of RGreedy.

Algorithm 10 RGreedy Algorithm

```
Input: Attribute set $S_A$, integer B, request pattern set $S_R$,
heuristic H
Output: An IMP configuration with enumerated minimum cost

optimal_config := <0, ..., 0>;
optimal_cost := Cost(optimal_config, $S_R$);
$L_A$ := Rank(H, $S_A$); $S_I$ := $\emptyset$;
while $L_A$.Has_More_Attr() do
  $S_I$ := $S_I$ + $L_A$.Get_Next_Attr();
  $S_C$ := Generate_Candidates_Greedy($S_I$, B);
  while $S_C$.Has_More_Configs() do
    new_config := $S_C$.Get_Next_Config();
    new_cost := Cost(new_config, $S_R$);
    if new_cost < optimal_cost then
      optimal_config := new_config; optimal_cost := new_cost;
    else
      Return optimal_config;
    end if
  end while
end while
Return optimal_config;
```

Complexity. The RGreedy algorithm progressively considers attributes. It only searches through the IMP configurations in which all considered attributes are indexed. Hence RGreedy has a significantly lower complexity than EPrune, as shown in Figure 20.1(b). Equation 20.4 shows the worst case complexity of RGreedy, with M denoting the number of attributes being pruned by the pruning strategy proposed for EPrune. This is exponential only when (N-M) is significantly smaller than B. In addition, since RGreedy stops whenever no further cost reduction can be achieved by considering one more attribute, we find that in practice the complexity of RGreedy is usually much lower than the worst case complexity (see Section 22.3).

$$WorstCase\_Complexity(RGreedy) = \sum_{i=1}^{Min(N-M,B)} \begin{pmatrix} B-1 \\ i-1 \end{pmatrix} \tag{20.4}$$

Measuring attribute importance. The effectiveness of the RGreedy algorithm clearly relies on the order in which the attributes are being considered. Such order is determined by a function for ranking the attribute importance. This order determines how fast the algorithm terminates. Further, since the hash cost increases with more attributes being included into the index, an ill-designed importance measure may cause the algorithm to stop before the important attributes are being considered. Therefore, the order can affect the quality of the configuration found by the algorithm. We have designed several ranking heuristics to estimate the attribute importance. Below we introduce two single-criterion heuristics and a hybrid heuristic.

20.4.1 Occurrence Weight Leading Heuristic

Our first heuristic ranks the importance of each attribute by its occurrence weight, as defined in Definition 11. Hence it is named the *occurrence weight leading (OWL)* heuristic.

Definition 11 *The* **occurrence weight of a selection pattern** p_i*, denoted as OW(*p_i*), is defined to be* $W_{p_i}F_{p_i}$ *in which* W_{p_i} *and* F_{p_i} *represent the window length and the frequency of* p_i *respectively. The* **occurrence weight of an attribute** A*, denoted as OW(A), is defined to be the sum of the occurrence weights of all selection patterns in which A is specified.*

According to Equation 19.4 (Section 19.4), if we assign relatively more bits, i.e., large B_{p_i} values, to the selection patterns with large $W_{p_i}F_{p_i}$ values, the usually dominating probe cost will be reduced. Hence, to index attributes with high occurrence weights is likely more beneficial.

Since the probe cost is usually the dominating cost, in all examples we show henceforth we only compare the probe costs of candidate IMP configurations. We use the average number of partitions to be probed for processing a single request as the indicator of the probe cost of each IMP configuration. For ease of exposition, we assume all user requests use the default window length W.

Example 1. Consider the request statistics in Table 20.1. The occurrence weights of attributes A, B and C are 0.5W, 0.4W and 0.2W respectively. As shown in Table 20.2, attribute A is considered in the first iteration since it has the highest occurrence weight. A single IMP configuration is available. In the second iteration, attribute B is included. Considering both A and B, three configurations are produced and <3,1,0> has the lowest probe cost. In the third iteration, attribute C is included. Again, three candidates arise and <2,1,1> is the best. Now that all the three attributes have been considered, the algorithm stops and returns configuration <2,1,1>. It corresponds to the optimal configuration found by the EPrune algorithm. While EPrune needs to examine 15 candidates, the RGreedy with OWL finds the optimal configuration by only checking 7 candidates,

Selection Pattern	*Frequency*	Occurrence Weight
<A, ∗, ∗>	0.4	0.4W
<A, B, ∗>	0.1	0.1W
<∗, B, ∗>	0.3	0.3W
<∗, ∗, C>	0.2	0.2W

Table 20.1: Example 1 – Request Statistics.

Step	*Next Attr.*	*IMP Config.*	*Probe Cost*
1	A	<4,0,0>	8.5
2	B	<1,3,0>	7.1
		<2,2,0>	6.14
		<3,1,0>	6.5
3	C	<1,1,2>	6.8
		<1,2,1>	6.2
		<2,1,1>	**5.8**

Table 20.2: Example 1 – OWL Execution.

However, OWL may miss the optimal configuration in some special situations when some frequently queried attributes such as A_j always co-occur with other frequent attributes say A_i in the user requests. In this case, if A_i has been included into the index, to additionally include A_j won't benefit more queries, while instead raising the hash cost. Then the algorithm would terminate before considering other attributes that in this case could possibly be more beneficial. We call this the *frequent correlation effect*. Below we construct such a worst case example.

Example 2. Consider the statistics in Table 20.3. Attribute A has the highest occurrence weight. So it is included first and the only configuration is <4,0,0>. Then B is considered and the optimal configuration for including A and B is <2,2,0>. Since B *completely* correlates with A in user requests, including it will cause A to be assigned less bits than before. As a consequence, the processing of requests with pattern <A, $*$, $*$> will now need to probe more partitions. The probe cost for other requests remains unchanged. Moreover, the hash cost becomes higher by indexing more attributes. Hence the algorithm will stop and return <4,0,0>. The optimal configuration <3,0,1> is missed.

Request Pattern	*Frequency*	Occurrence Weight
<A, $*$, $*$>	0.3	0.3W
<A, B, $*$>	0.4	0.4W
<$*$, $*$, C>	0.3	0.3W

Table 20.3: Example 2 – Request Statistics.

20.4.2 Pattern Coverage Leading Heuristic

In view of the above shortcoming of the OWL heuristic, we propose another heuristic, namely *pattern coverage leading (PCL)*, that instead considers the pattern coverage of the attributes, as defined in Definition 12. The intuition is that the index is likely to be beneficial by indexing the attributes that together cover a majority of requests.

Definition 12 *A selection pattern p is said to be* **covered** *by an index if at least* *one attribute specified in p is included in the index. The* **remaining pattern coverage** *of an attribute A, denoted as RPC(A), is defined to be the sum of the occurrence weighs of all patterns that are not covered by the index and specify a non-wildcard value on A.*

While the attribute occurrence weights are static, the remaining pattern coverage of an attribute must be recomputed as additional attributes are included into the index.

Table 20.4 shows the execution of the RGreedy algorithm using PCL given the statistics in Example 2. We see that while OWL missed the optimal configuration, PCL finds it.

Step	RPC	*Next Attr.*	*IMP Config.*	*Probe Cost*
1	A: 0.7W, B: 0.4W, C: 0.3W	A	<4,0,0>	5.5
2	C: 0.3W, B: 0	C	<1,0,3>	6.2
			<2,0,2>	4
			<3,0,1>	**3.8**

Table 20.4: Example 2 – PCL Execution.

The PCL heuristic works well for the cases when frequently queried attributes are completely correlated with each other. Hence it overcomes the shortcomings of OWL. However, it still does not guarantee to always find the optimal IMP configuration. It fails when a frequent attribute correlates with both frequent and infrequent attributes and is shadowed by the infrequent attributes in terms of pattern coverage. Example 3 below illustrates this scenario.

Example 3. Consider the statistics in Table 20.5. Attribute B is a frequent attribute. It co-occurs with both A (frequent) and C (infrequent). During the search, after A is included, RPC(B) is reduced from 0.65W to 0.25W so it becomes less than RPC(C), which is 0.3W. Therefore, C will be considered next with best configuration <3,0,1>. After C is included, RPC(B) is reduced to zero. Then B is never considered by the algorithm. However, the optimal configuration is <2,2,0> since including B instead of C actually benefits more queries. Interestingly, OWL is able to find this optimal configuration that PCL missed.

Request Pattern	*Frequency*
<A, $*$, $*$>	0.3
<A, B, $*$>	0.4
<$*$, B, C>	0.25
<$*$, $*$, C>	0.05

Table 20.5: Example 3 – Request Statistics

20.4.3 Hybrid Heuristic

It can be seen that the OWL and the PCL heuristics complement each other. We hence propose a heuristic that combines them, named *Hybrid* heuristic. The basic idea is to run the greedy search once using the OWL heuristic and then using the PCL heuristic. After that we compare the final configurations suggested by these two runs. The one with the smaller cost will be selected as final decision. By applying the Hybrid heuristic, we are able to find the optimal configurations for Examples 2 and 3, while either OWL or PCL will miss one of them respectively.

Optimization by reusing cost computations. If we blindly run the algorithm with OWL once and PCL once and then compare their results, we may repeat many computations. First, the two heuristics always consider the same attribute in the first iteration because in this iteration the OW value of each attribute equals its RPC value. In later iterations, the two heuristics may also consider the same sets of attributes. Example 1 is an extreme example here since both OWL and PCL would consider the three attributes in the same order. Hence running one of them would be enough.

To reuse computations for applying the Hybrid heuristic, we first check the attribute consideration orders of OWL and PCL. If the two orders are same, we only run RGreedy with OWL. Otherwise, we first run RGreedy with OWL and keep the optimal IMP configuration for each iteration with its cost into a hash table. The hash key is the ID of the attribute set. Then we run RGreedy with PCL. In this run, in each iteration, we first check the hash table to see whether the given attribute set has been considered in the first run. If yes, we skip the evaluation and use the result directly.

Analysis. Without reusing computations, the complexity of using the Hybrid heuristic equals (Complexity(OWL) + Complexity(PCL) - 1). By reusing computations, it becomes the worst case complexity only when the two heuristics consider totally different sets of attributes. In the best case when OWL and PCL consider the attributes in the same order, the three heuristics have the same complexity.

Chapter 21

Generalizations

21.1 Dealing With Narrow Attribute Domains

The EPrune and RGreedy algorithms implicitly assumed that the number of distinct values of any indexed attribute A_k is at least equal to 2^{b_k} with b_k being the number of bits assigned to A_k. However, this is not always the case. For example, the $gender$ attribute only corresponds to two values, $male$ and $female$. Hence it should be assigned at most one bit. If the $gender$ attribute is queried very often, it may be assigned more than one bit by the algorithms described so far. This may lead to non-optimal index configurations [6]. Below is such an example.

Example 5. Consider Example 1 in Section 20.4.1 again. The optimal configuration found when assuming no partition limits is <2, 1, 1>. Now suppose the domain of attribute A only corresponds to two values. Then one address bit will never be used. Tuples in the state only occupy 8 instead of 16 partitions. The average number of tuples being probed for processing a single request becomes $c_1 = 3.8 \cdot \frac{\lambda_d W}{8} = 0.475\lambda_d W$. If the algorithm takes the narrow attribute domain into consideration, the optimal configuration then becomes <1, 2, 1> with cost $c_2 = 6.2 \cdot \frac{\lambda_d W}{16} = 0.388\lambda_d W$. Given that the groupby state usually contains large numbers of tuples (i.e., a large $\lambda_d W$ value), c_2 should be significantly lower than c_1 most of the time.

To solve this problem, we set the upper limit on the number of bits assigned to each attribute A_i (1≤i≤N) to be $b_i^{max} = \llcorner log_2 DV_i \lrcorner$. DV_i denotes the number of distinct values of A_i. We revise EPrune and RGreedy algorithms to construct the IMP candidates subject to these limits. According to Theorem 4, the optimality of the EPrune algorithm is still guaranteed. The proof of this theorem is straightforward.

Theorem 4 *Given N attributes A_1, ... A_N, B address bits and statistics on DV_i for A_i (1≤i≤N), the IMP configuration output by the revised EPrune algorithm has the lowest UPC among all possible IMP configurations.*

21.2 Supporting Pre-computed Aggregates

The algorithms described so far assume that no aggregate results are pre-computed, which is a common setting as discussed in Section 17.3. However, in the special situation that the request arrival rate is significantly higher than the data arrival rate and most requests are not selective, it may be worthwhile to pre-compute some aggregate results. Similar to the aggregate processing in prior works [66, 103], decisions on whether and what to pre-compute are based on the cost-benefit analysis. While the above is an orthogonal issue, once the pre-computation decision has been made, our approach can easily be extended to incorporate this option, as discussed below. The implementation and evaluation of this extension remains our future work.

Given N possible grouping attributes, the grouping attributes specified in PSGB instantiations could be any of the 2^N attribute combinations. Therefore, similar to the Cube queries [66], aggregate results for different subsets of grouping attributes in the PSGB query and different aggregate functions may need to be pre-computed and maintained. We use a *PSGB sub-state* to maintain the aggregate results for each distinct groupby-aggregate setting chosen for materialization and then construct the IMP index over each sub-state.

Therefore, we apply a two-step decision making based on the data/query statistics (Section 20.2). First, we decide what aggregate results to pre-compute and how many PSGB sub-states to maintain. For this, we apply the work in [66] and use our cost model extended to also include the aggregate maintenance cost for streaming data. Second, we select the IMP configuration for each PSGB sub-state. To achieve this, the probe cost computation should be changed. Each entry in hash partitions now represents the aggregate result for a particular group (more advanced data structures may be used for each entry, such as the techniques in [9]) instead of a regular data tuple. The search within each partition would stop once the matching entry is found so the average search cost per partition now is half number of the entries in the partition. Using this revised cost model, the proposed index selection algorithms remain applicable with no further modifications.

Chapter 22

Experimental Evaluation

22.1 Experimental Setup

We have implemented the PSGB operator with index tuning in the CAPE system [83]. We conduct an extensive experimental study to explore the effectiveness of the index selection and tuning. The test machine has a 3GHz Intel(R) Pentium-IV processor and a 1GB RAM, running Windows XP and Java 1.5.0_06 SDK.

In the experimental study, we focus on exploring the answers to three questions: 1) Is IMP index a suitable solution for organizing the PSGB operator state? 2) Is RGreedy an effective index selection algorithm regarding both efficiency and quality? 3) How does the index tuning affect the query performance when the query or data workload changes?

Workload. To extensively test our approach regarding the above questions, we have created a large variety of streaming data and PSGB request workloads by varying key factors. We generate two types of request workloads: 1) *random request workloads* in which the frequencies of selection patterns conform to either normal distribution or Zipf distribution, and 2) *extreme request workloads* in which the frequencies of selection patterns are specially designed for four extreme cases, namely, *Uniform*, *Asc*, *Desc* and *Exclusive*. In the Uniform workload, all possible request patterns have the same occurrence weight ($\frac{1}{2^N}$). N represents the total number of attributes. In the Asc workload, the requests that specify only the first attribute occupy 50% of requests. The other request patterns all have same occurrence weight ($\frac{1}{2\times 2^N}$). The Desc workload is the opposite to the Asc workload, i.e., the requests that specify only the last attribute occupy 50% of requests. The Asc and the Desc workloads will affect the performance of EPrune because EPrune considers the attributes in a fixed order, regardless of request statistics. In the Exclusive workload, each request specifies only one attribute and all involved request patterns have same occurrence weight ($\frac{1}{N}$). We also create the workload according to the examples used in this work (Section 20.1).

The occurrence weight of selection patterns, i.e., $F_{p_i}W_{p_i}$, can be viewed as a single factor in the IMP index cost model. Without loss of generality, in all experiments, we assume a default window length for all requests in each request workload D. We vary only the selection pattern frequencies *within* the workload. The default window lengths are varied *across* different workloads. We also vary the number of address bits, the number of attributes and the ratio of data arrival rate to request arrival rate. All factors and their values used in our experiments are listed in Table 22.1.

Factor	*Values*
B (# of address bits)	16
N (# of attributes)	3, 8, 10
$\frac{\lambda_d}{\lambda_r}$	0.01, 0.1, 1, 10, 100
Default Window Length (# of tuples)	100000, 200000, 1000000
Normal Distribution Variance	2, 4, 8, 16, 32, 64, 128
Zipf Skew Factor (α)	0.7, 0,8, 0.9, 1.0, 1.1

Table 22.1: Experiment Setup - Factor/Value Used.

22.2 Comparing Alternating Index Methods

The first set of experiments explores whether the IMP index is the most efficient solution for organizing the PSGB operator state compared to the index methods employed by the existing work on streaming data processing [56]. We run the groupby operator using three different hash index approaches – *IMP* index, *1-Attr-Hash* index with hash function $h_1(attr)$ in which $attr$ is the attribute with the highest occurrence weight, and *M-Attr-Hash* index with hash function $h_M(attr_1, ..., attr_m)$ in which $attr_1$, ..., $attr_m$ are the attributes occurring in the most frequent request pattern. We test the groupby operator using all workloads created according to Table 22.1. In these experiments, the operator using the optimal IMP index always achieves the equivalent or in more than 80% of the cases significantly better performance than the other two approaches.

We now show the results of one such experiment for $\frac{\lambda_d}{\lambda_q} = 1$ and B=16. We emphasize the ratio because we conduct the stress test (corresponding to the *CPU-limit mode* in [100]) such that the PSGB operator is never idle. For tuples, the values of each attribute conform to a uniform distribution with 2048 distinct values. We conduct 6 runs, each over a workload con-

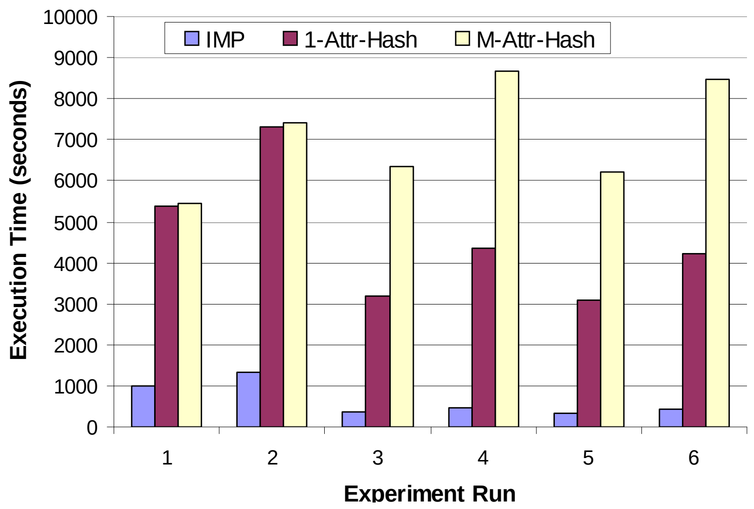


Figure 22.1: Comparing Proposed IMP Index With Traditional Hash Index Solutions.

taining 200,000 tuples and 200,000 requests. i.e., each tuple followed by a request in time. In these runs, we use 3 different request statistics that are respectively specified in Examples 1 (Table 20.1), 2 (Table 20.3) and 3 (Table 20.5) in Section 20.4. Table 22.2 summarizes the experimental parameters. We record the total execution time of the groupby operator in each of these runs (Figure 22.1).

Run #	*Request Load*	Window Size (# Tuples)
1	Example 1	100,000
2	Example 1	200,000
3	Example 2	100,000
4	Example 2	200,000
5	Example 3	100,000
6	Example 3	200,000

Table 22.2: Parameters for Experiment 1.

We can see from Figure 22.1 that in all these runs, the groupby with the

IMP index speeds up the query processing by at least 50%, in some cases even more than 90% compared to using existing index methods. The gains increase with the window size. As expected, as the groupby state becomes larger, by properly balancing the partition factors among all attributes, the IMP index tends to gain more probing cost savings compared to the other two approaches.

22.3 Comparing Index Selection Algorithms

The second set of experiments compares our proposed index selection algorithms regarding how fast each algorithm terminates (i.e., *efficiency*) and how close each of their decisions approaches the optimal configuration (i.e., *optimality*).

We run the four algorithms – EPrune, RGreedy with OWL, PCL and Hybrid heuristics respectively over all workload cases generated according to Table 22.1. We can observe that first, the execution times of both EPrune and RGreedy increase with query window length and normal distribution variance, and decrease with Zipf skew factor. This is because more skewed selection pattern frequencies provide more opportunities for pruning attributes. For the same selection pattern statistics, the larger window creates the need for indexing more attributes. Second, for large window lengths, e.g., with 1,000,000 tuples, and high N values (e.g., N=10), if no attributes can be pruned, EPrune takes hours to finish. However, RGreedy always finishes within minutes, and most of the time within seconds. Third, RGreedy with each of the three heuristics finds the optimal IMP configuration for all the extreme workloads. This is because the extreme cases fit their logic perfectly. For the random workloads, while RGreedy with PCL and hence Hybrid heuristics finds the optimal IMP configuration for all the test cases, RGreedy with OWL misses about 50% of them. This is due to the frequent correlation effect discussed in Section 20.4.1. Hence PCL appears to be a very effective heuristic. Also, in more than 50% of the test cases, by reusing computations, RGreedy employing Hybrid heuristic has moderate and in many cases even trivial extra search overhead compared to using either of the other two heuristics. This indicates that RGreedy with Hybrid heuristic should be the suggested algorithm for large search spaces since Hybrid heuristic combines the advantages of OWL and PCL.

Normal Dist. Variance	Execution Time (seconds)	Normal Dist. Variance	Execution Time (seconds)
2	2.6×10^2	32	3.67×10^3
4	5.3×10^2	64	7.0×10^3
8	9.8×10^2	128	1.3×10^4
16	1.9×10^3	256	1.3×10^4

Table 22.3: EPrune Complexity - Random WorkLoad.

We now show the result of one experiment in which the PSGB query has 8 attributes, a one-million-tuple window is assumed and $\frac{\lambda_d}{\lambda_q}$=10. Table 22.3 shows the execution time of EPrune under the random request workloads, with 8 different variances. Figure 22.2 compares the three heuristics of the RGreedy algorithm under these random workloads regarding complexity, execution time and UPC [1].

We see that while EPrune takes hours to finish for large normal variance

[1] In Figures 22.2 and 22.3 Lines that connect points are used only to reveal the trend.

values, RGreedy always terminates within seconds. Also, RGreedy with PCL and therefore Hybrid finds the optimal configuration in all 8 cases. However, OWL misses all the optimal configurations in this experiment.

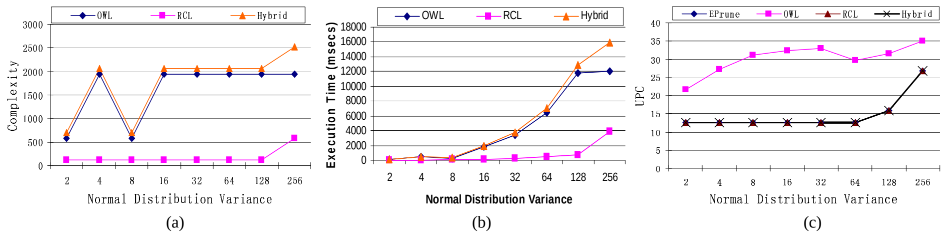


Figure 22.2: Complexity of RGreedy Heuristics - Random WorkLoad.

The comparisons of the three RGreedy heuristics under the extreme workloads are shown in Figure 22.3. They achieve the optimal IMP configuration for all extreme workload cases.

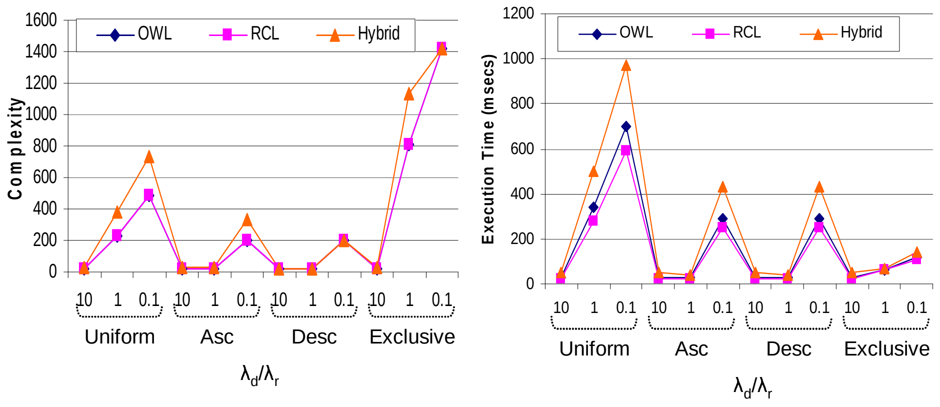


Figure 22.3: RGreedy Heuristics - Extreme WorkLoad.

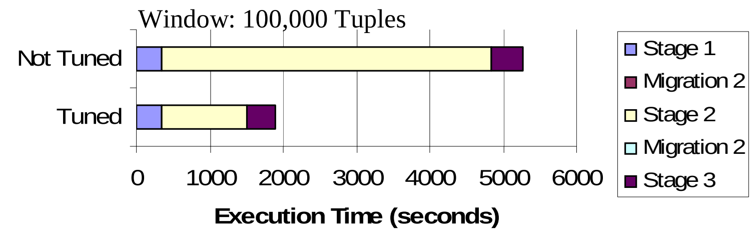
22.4 Runtime Index Tuning

Finally, we test how the runtime index tuning affects the query performance. In the experiment shown below, we use a workload with 600,000 tuples and 600,000 requests such that $\frac{\lambda_d}{\lambda_q}$=1. The first, middle and the last 200,000 requests are generated respectively according to the request statistics specified in Examples 2, 1 and 3 in Section 20.4. Hence the groupby execution is composed of three stages, each processing 200,000 tuples and 200,000 requests.

We then run the groupby operator two times, both in CPU-limit mode [100]. In the first run, the operator is forced to conduct instant migration that rehashes every tuple in the state after it finishes processing every 200,000 requests. It uses the optimal IMP configuration for each of the three stages. In the second run, the operator never conducts migration. That is, it always uses the IMP configuration that is optimal for the first stage. We then record the total execution time of the groupby operator, including the time for index assessment and migration.

Figure 22.4 shows the results for two settings, using two different window sizes respectively. In the figure, we use three different colors to mark the three execution stages.

The groupby operator with index tuning achieves more than 50% reduction on the execution time compared to the one without tuning. The migration time for the tuned groupby operator is invisible in the figure because it is too small to be seen. In both sets of experiments, the time for each migration is within seconds. This is promising because it shows



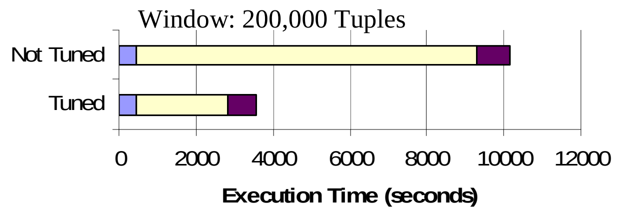


Figure 22.4: Tuned vs. Not Tuned IMP Index.

little migration overhead for a relatively large groupby state (containing 100,000 and 200,000 tuples respectively). This indicates that the runtime index tuning overall is worthwhile, as the tuning cost is small compared to the potential saving achievable by the tuning.

Chapter 23

Related Work

The *index selection problem* has been extensively studied in static databases [4, 28, 51, 65], in which data updates are rare compared to queries. Index selection tools take a query workload as input and suggest a set of indexes that can maximally benefit the given workload. Index adaptation due to changes in workloads means inserting a new index or deleting an existing index. We instead tackle the index selection problem in the stream context where not only data updates but also query requests may arrive at high rates. We maintain a single index structure to minimize the memory overhead and the index maintenance cost. Our index tuning essentially adjusts the configuration of the single index.

Indexing in stream contexts has not yet received much attention, possibly due to the dynamic nature of the streaming data. [56] studies methods for indexing a single attribute for individual streaming algebra operators under the sliding window semantics. Index selection driven by workloads, as is the focus of our work, is not tackled.

Our work relates to the work on processing groupby or aggregate queries over streaming data. Existing work mainly focuses on sharing aggregate results. Their targeted query types and assumed output models are summarized in Table 23.1. [103] studies aggregate sharing among a set of streaming groupby queries that differ only in their grouping attributes. This is a direct extension of the work in static databases [66].

[9, 71, 73] focus on computation methods for streaming aggregate queries without groupby operations, i.e., one single result is produced per window. [73] proposes techniques for sharing aggregate results among consecutive sliding windows of a single aggregate query by breaking windows into time slices. [71] generalizes this idea by slicing tuples into partitions based on window and selection predicate overlaps. The computation of aggregate functions over such partitions just puts them together into a combined aggregate value. All of these works assume 1) queries to be statically specified and 2) *continuous* output model, i.e., the result updates are produced each time a new input tuple is received. [9] investigates shared execution of aggregate queries (no groupby) using an on-demand output model. In this work, besides the window length, no other parameters in the query can be dynamically specified. We differ from these prior works in that we focus on processing groupby queries using the on-demand output model and with dynamic parameters not being limited to window lengths, but also including selection predicates, aggregate functions and grouping attributes.

[6] also employs a hash-bit method similar to our IMP index to answer partial-match selection queries over a record file. This work considers a simplistic model in which each attribute is *independently* specified in

Query	*Exec. Model*	*Dynamic Feature*	*Related Work*
Groupby	continuous	no	[103]
Aggregate	continuous	no	[71, 73]
Aggregate	on demand	dynamic window	[9]

Table 23.1: Related Work on Shared GB/AGG Exec.

queries. This simplification enables a linear time optimal bit assignment method. We instead consider a more general and practical model in which the frequencies of selection patterns are given. Our model incorporates both the independent query model [6] and the correlated query model, which makes the index selection problem much harder (exponential). The statistics on selection pattern frequencies that we work with can easily be extracted from query workload [34].

Part IV

Runtime Semantic Query Optimization for Event Stream Processing

Chapter 24

Introduction

24.1 Constraint-Aware Event Stream Processing

As automated business processes, such as Web services and online transactions [37, 54, 80], become ubiquitous, unprecedented volumes of business events are continuously generated and recorded as event streams. *Complex Event Processing (CEP)*, which aims to detect interesting *event patterns* in event streams, is gaining adoption by enterprises for quick detection and reaction to critical business situations. Common CEP applications include business activity monitoring, supply chain management, and anomaly detection. Major database vendors have recently taken significant efforts in building event-driven architectures [17, 32].

The event patterns in CEP specify complex *temporal* and *logical* relationships among events. Consider the example event pattern *EP1* below, in which "SEQ" represents the *temporal* relationship between two events and [totalPrice>200] is the predicate on the GenerateQuote event. This pattern monitors the cancelled orders that involve the participation of both suppliers and remote stocks, with quote's price > \$200. Frequent occurrences of such patterns may indicate the need for an immediate inventory management.

Event Pattern EP1:

SEQ((SEQ(OrderFromSupplier,GenerateQuote[totalPrice > 200])
AND SEQ(UseRemoteStock,GenerateInvoice)),CancelOrder)

State-of-the-art CEP systems employ automata for event pattern matching [35, 101]. When there are large numbers of concurrent business processes, many partial query matches may be kept in automata states. Events arriving later need to be evaluated against all these partial matches to produce query results. Also, event streams tend to be high-speed and potentially infinite. To provide real-time responses, as often required by applications to take prompt actions, serious challenges in CPU and memory utilization are faced by CEP.

In this work, we target an important class of event queries, namely *alert queries* [101]. Alert queries correspond to key tasks in business activity monitoring, including detection of shoplifting, large/suspicious financial transactions, or other undue business actions like orders cancelled for certain reasons (see example above). These queries detect exceptional cases to the normal business flows and are thus expected to be highly selective. Keeping large numbers of partial matches that do not lead to any query results can cause a major drain on available system resources.

We observe that in practice, many business events do not occur ran-

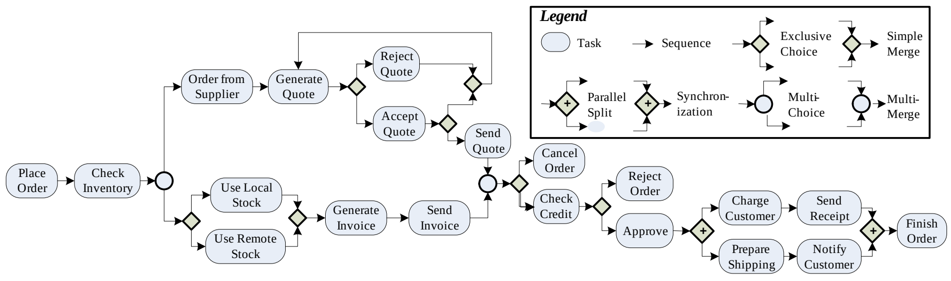


Figure 24.1: Online Order Fulfillment Workflow.

domly. Instead they follow pre-defined business logic or rules, such as a workflow model [54]. Below we list a number of such CEP applications.

1). *Business activity monitoring*: an online retailer may want to detect the anomalies from its order processing transactions. In this case, the events are generated from a BPEL workflow engine [20], a business rule engine [21] or simply a customized program.

2). *Manufacturing monitoring*: a manufacturer may want to monitor its stream-line production process [67]. The process events correspond to pre-defined procedures.

3). *ClickStream analysis*: a shopping website may want to monitor the click stream [37] to discover the user navigation pattern. Here the user click events depend on how the website is structured.

As consequence, various constraints may exist among events in these CEP applications. In particular, *occurrence* constraints, such as mutually exclusive events, and *order* constraints, such as one event must occur *prior* to the other event, can be observed in all the applications listed above. A recent survey [45] shows that the majority of the software design patterns exhibit such constraints as well.

The availability of these constraints enables us to predict the non-occurrences of future events from the observed events. Such predictions would help identify which partial query matches will definitely not lead to final results. Further efforts in maintaining and evaluating these partial matches can be terminated, thus resulting significant savings. Example 1 below illustrates such optimization opportunities that remain unexplored in the literature.

Example 1 *Assume the event stream is generated by online order transactions [80, 97] that follow the workflow in Figure 24.1. We assume each task in the workflow, if performed, will submit an event to the event stream. We can see that both occurrence and order constraints can be inferred from this workflow. For example, the UseLocalStock and the UseRemoteStock events are mutually exclusive. Also, any GenerateQuote event, if it occurs, must be before the SendQuote event in a transaction.*

Consider the example event pattern EP1 again. By exploiting the event constraints, whenever a UseLocalStock event occurs, this transaction is guaranteed to not match the query because the UseRemoteStock event will never occur in this transaction. Also, once a SendQuote event is seen in a transaction, and no GenerateQuote event with totalPrice>200 has been observed so far, the transaction will not match the query because no GenerateQuote event will happen after the SendQuote event. In either case, any partial matches by these transactions need not be maintained and evaluated further as they are guaranteed to never lead to a final result. If the query processing of large numbers of transactions could be *terminated early, a significant amount of CPU and memory resources would be saved.*

Several observations can be made from the above example. First, although the event constraints are known at query compilation time, the real optimization opportunities only emerge at $runtime$ based on the partial workflow executed so far (i.e., what events have been observed). For example, although the *UseLocalStock* and the *UseRemoteStock* events are known to be exclusive, only when one of them occurs, can we infer that the other one will not be seen in the same transaction. Second, both $occurrence$ and $order$ constraints can be exploited to short-cut query execution.

24.2 State-of-the-Art CEP Techniques

Most existing works in CEP focus on syntax and semantics of event queries [17, 35, 101]. Initial results on event query processing techniques such as event instance partitioning and predicate pushdown [101] have also been presented. However, the above identified event constraints, which can be seen as *schema* knowledge, remain unexploited in CEP.

Semantic query optimization (SQO), i.e., using schema knowledge to optimize queries, has been extensively studied for traditional databases [24, 69]. Major techniques focus on optimizing value-based filtering or matching operations, including join and predicate elimination and introduction. They remain applicable in CEP for identifying efficient query plans at compilation time. However, the relational and the object-oriented data models targeted by these techniques are unordered, thus lacking in support for expressing temporal relationships among data. Hence, these techniques have not focused on the unique optimization opportunities arising in CEP driven by the temporal event constraints. In addition, the existing SQO techiques are mainly designed for static query optimization. They may be inappropriate for runtime use. SQO has also been studied for optimizing queries over streaming XML documents [89]. In CEP, we are faced with event data from possibly thousands or millions of concurrent processes interleaved, and thus huge numbers of potential partial matches (one for each process) at runtime. Also, more types of constraints can be observed in business processes than in XML schemata. All these pose stringent requirements on scalability, generality and extensibility on exploiting constraints in CEP.

On the other hand, significant research effort has been devoted on specifying and verifying business processes, such as workflow analysis [92] and formal process verification [45]. In this existing research area, the process instances, which can be seen as *data*, are not taken into consideration.

In summary, existing work on CEP [17, 35, 101] and on business processes [45, 92] focus respectively on data and schema. No effort so far exploits the schema knowledge of business processes to optimize CEP. This now becomes the focus of our work, which is *constraint-aware CEP* or *C-CEP*.

24.3 Our Approach

Several key challenges must be tackled to exploit constraints for CEP. One critical question is how to identify unsatisfiable partial query matches at *runtime*. In addition, there may be thousands or even millions of concurrent business processes. To assure the efficiency and scalability, the runtime reasoning for each individual transaction must be *lightweight*. Otherwise, the overhead of constraint reasoning may outweigh its benefits. In this work, we propose the first general framework to address the above challenges for *constraint-aware CEP (C-CEP)*. The main contributions are summarized below:

1. We propose a polynomial time, sound and complete runtime query unsatisfiability (RunSAT) checking algorithm for detecting unsatisfiable query matches. This algorithm is based on classic logic reasoning considering the event query, the partial event history and the event constraints such as workflows (Chapter 26).

2. To improve the RunSAT performance, we propose a general pre-processing mechanism (based on abductive inference [46, 47]) to pre-compute query failure conditions. Further, we identify a set of simple yet common event constraints that allow constant time RunSAT (Chapter 27).

3. We propose to realize the above techniques based on augmenting event queries with pre-computed failure conditions. This facilitates the integration of our techniques into state-of-the-art CEP architectures [35, 101] (Chapter 28).

4. Our experimental study demonstrates that significant performance

gains, i.e., memory savings up to a factor of 3.5 and CPU savings at a factor of 2, are achieved through our approach, with a very small almost negligible overhead for optimization itself (Chapter 29).

Chapter 25

Background

25.1 Event Model

An *event* (or *event instance*), denoted as lower-case letter e_i, is defined to be an instantaneous, atomic (happens completely or not at all) occurrence of interest. We assume a discrete time domain T, and each time point represented by a non-negative integer. An *event type*, denoted as the corresponding upper-case letter E_i, defines the properties that all the event instances e_i must have. The properties of an event instance e_i include a set of attributes $e_i.A_1$, ..., $e_i.A_n$, and a timestamp $e_i.t$ of its occurrence.

We assume that the input to the CEP system is a stream of events ("*event history*") ordered by their timestamps. Out of order events can be handled by sorting the most recent K tuples [22], which is orthogonal to our problem. We assume that the event history can be partitioned into multiple sub-sequences based on certain criteria, such as transactions ids, session ids, RFIDs, etc. Thereafter we call each partition of the event history a *trace*

h.

25.2 Event Constraints

Software and workflow models exhibit certain *order* and *occurrence* constraints (Section 24.1). CEP queries also need to capture these occurrence and order between events (defined later). These constraints can be expressed using a subset of a general event language $\mathcal{L}$.

Definition 13 *An event language $\mathcal{L}$ contains a set of event types E_i, denoted as $\mathcal{E}$, a variable h denoting the event history, a binary function $<$, logic connectives ($\wedge$, $\vee$, $\neg$, $\rightarrow$), quantifiers ($\exists$ and $\forall$). A formula of $\mathcal{L}$ is either:*

1). $E_i[h]$, iff an event instance $e_i \in h$ of type E_i exists;

2). $E_i[h] < E_j[h]$, iff event instances e_i, $e_j \in h$ of type E_i and E_j, respectively, with $e_i.t < e_j.t$;

3). Any formula built upon the above two atomic formulas by means of the logical connectives and $\exists h$ and $\forall h$.

The two atomic formulas correspond to the occurrence and order properties of events. The constraint language $\mathcal{L}$ essentially corresponds to monadic logic (with monadic predicate $E_i[h]$) plus a binary function ($<$). This can easily simulate full predicate calculus, which in general is undecidable [19]. Hence, $\mathcal{L}$ and its derivatives have been used in the literature to describe the semantics of various applications. Since $\mathcal{L}$ is very general, in many practical scenarios, only subsets of $\mathcal{L}$ are considered. In this work, we focus on the following two types of constraints that allow polynomial time reasoning under both static and runtime case. These constraints may be explicitly given by the business rules or they can be extracted from a given workflow model [54]. We denote $\mathcal{C}$ as a conjunction of a set of event constraints, which contains order constraints $\mathcal{C}^t$ and occurrence constraints $\mathcal{C}^o$, defined as below.

- $\forall h_e, \neg(E_j[h_e] < E_i[h_e])$, called *order constraints*, denoted as f^t;
- Horn clauses built upon $E_i[h_e]$ and $\forall h_e$, called *occurrence constraints*, denoted as f^o.

Here h_e denotes the *entire* trace, indicating that the constraint must hold w.r.t. the *scope* of the entire trace. Such global semantics (i.e., tracewise) is common [45].

1. prior(E_i, E_j, h_e) := $\forall h_e, \neg(E_j[h_e] < E_i[h_e])$
2. exclusive(E_i, E_j, h_e) := $\forall h_e$, $E_i[h_e] \rightarrow \neg\, E_j[h_e]$
3. require(E_i, E_j, h_e) := $\forall h_e$, $E_i[h_e] \rightarrow E_j[h_e]$

Table 25.1: Constraints that Allow Constant-time Runtime Reasoning

However, even polynomial time runtime reasoning is not always satisfactory, especially if it is potentially more costly than executing the initial CEP query itself. One of our contributions is the identification of three common constraints (Table 25.2), which even allow *constant-time* runtime reasoning. This assures negligible runtime reasoning overhead and thus has the potential to significantly improve the CEP performance.

25.3 Event Query

In this work, we do not provide a new CEP language as this is already the focus of a number of existing works [17, 35]. Instead we focus on how the core common to most CEP languages can be optimized by exploiting commonly available constraints. Similar to a number of existing works [17, 35, 101], an event query is specified as follows:

```
EVENT   <event expression>
WHERE   <equal-id> [<predicates>]
```

The *EVENT* clause specifies the *event expression* that expresses the interested event pattern.

- $SEQ(E_1, E_2, ..., E_n)(t^s, t^e) := \exists t_1^s \leq t_1^e < t_2^s \leq t_2^e < ... < t_n^s \leq t_n^e$, such that $E_1(t_1^s, t_1^e) \wedge E_2(t_2^s, t_2^e) \wedge ... \wedge E_n(t_n^s, t_n^e)$. And $t_s = t_1^s$ and $t_e = t_n^e$.
- $AND(E_1, E_2, ..., E_n)(t^s, t^e) := \exists t_1^s, t_1^e, t_2^s, t_2^e, ..., t_n^s, t_n^e, E_1(t_1^s, t_1^e) \wedge E_2(t_2^s, t_2^e) \wedge ... \wedge E_n(t_n^s, t_n^e)$. And $t^s = min(t_1^s, t_2^s, ...t_n^s)$ and $t^e = max(t_1^e, t_2^e, ...t_n^e)$.
- $OR(E_1, E_2, ..., E_n)(t^s, t^e) := \exists t^s, t^e, E_1(t^s, t^e) \vee E_2(t^s, t^e) \vee ... \vee E_n(t^s, t^e)$.

We refer to the output of these operators as a *composite event*. While the event instance (called *primitive event*) has a point-in-time semantics, $e_i.t$, the composite event has an interval semantics, where t_s and t_e are the timestamp of the first and the last event in the event expression, respectively.

The above definitions adopt this interval semantics and support the arbitrary nesting of these operators. As a special case, when E_i is a primitive event type, t_s equals t_e.

The *WHERE* clause contains an equality condition on some common attributes across multiple event types in the query, which is typical for monitoring applications [17, 101]. Examples include transaction ids, session ids, RFIDs, etc. Based on the value of those ids, the event history is partitioned into subsequences. Each subsequence corresponds to one *trace* h_e that is defined previously. The query is then evaluated against each h_e. There might be additional predicates over the other attributes as well. The output of the query contains the concatenation of all matching event instances. While customized output results can be further accomplished [17, 35], this is independent of the work presented here.

For ease of presentation, we use an acyclic directed graph $G(Q) = <N, V>$ to represent an event query Q. Each node is either an event type or one of the two special types of nodes, namely, the *start (AND^S)* and *end (AND^E)* of the AND operator. Each edge represents the ordering relationship between event types in the query. Since query Q is well nested, the corresponding start and end of AND nodes are paired as well. Figure 25.1 depicts an example.

Unsatisfiability-preserving translation. The event query is translated into the formula in $\mathcal{L}$ that preserves unsatisfiability without considering the predicates on other attributes, or assuming they are always satisfiable). For any conjunctive event query Q, the corresponding formula in $\mathcal{L}$ is: $\exists h_e, \bigwedge\{E_i[h_e]\} \bigwedge\{E_j[h_e]<E_k[h_e]\}$, for any $E_i \in Q$ and for any $E_j, E_k \in Q$

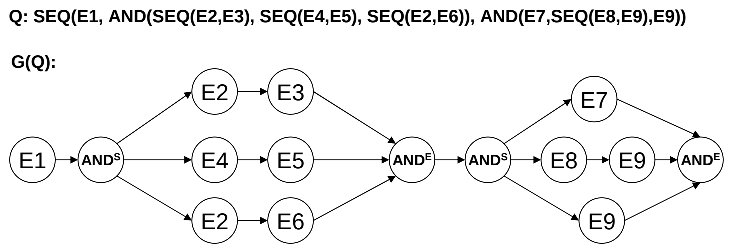

Q: SEQ(E1, AND(SEQ(E2,E3), SEQ(E4,E5), SEQ(E2,E6)), AND(E7,SEQ(E8,E9),E9))
G(Q):
E1
AND^S
E2
E3
E4
E5
E2
E6
AND^E
AND^S
E7
E8
E9
E9
AND^E

Figure 25.1: A Sample Query Graph G(Q) for A Query Q

which have a order relationship in Q. For any disjunctive event query Q, we can rewrite it into disjunctive normal form and translate each conjunctive term. Through this translation, we can reason between $\mathcal{C}$ and Q to check its unsatisfiability.

Chapter 26

Query Unsatisfiability Checking

26.1 Overview

As motivated in Example 1, given an event query Q, event constraints $\mathcal{C}$ and a partial trace h_p observed at runtime, we want to determine whether a query match may exist in the complete trace h_e with h_p being a prefix of h_e (denoted as $h_p \subseteq h_e$). We refer to this problem as the *runtime query unsatisfiability (RunSAT)* problem. There is an extreme case of this problem, i.e., given an event query Q and event constraints $\mathcal{C}$, does a query match exist in any trace h_e. We refer to this extreme case as the *static query unsatisfiability (SunSAT)* problem. In this section, we will describe these problems in detail.

26.2 Static Query Unsatisfiability

We formalize the *static query unsatisfiability (SunSAT)* problem in Definition 14.

Definition 14 Static Query Unsatisfiability (SunSAT) *Given a query Q and event constraints $\mathcal{C}$, Q is said to be statically unsatisfiable iff there does not exist a trace h_e which is consistent with $\mathcal{C}$ and matches Q.*

Static satisfiability checking is to check whether $\mathcal{C} \wedge Q \models \perp$. This involves two parts, namely, the *occurrence consistency* checking and the *temporal consistency* checking, based on the constraint-based translation of Q.

Occurrence consistency makes sure that all the event instances required in the query can indeed occur together. This is achieved by checking whether the following boolean expression is satisfiable: $\bigwedge\{E_i[h_e]\} \bigwedge \mathcal{C}^o$, for all $E_i \in Q$. When the query is conjunctive and $\mathcal{C}^o$ contains only *Horn* clauses, the checking can be done in polynomial time [85]. Unfortunately, if $\mathcal{C}^o$ contains any arbitrary constraints, then this becomes a NP-Complete problem [85].

Temporal consistency means that each event instance required in the query could occur in the desired order. This is to check $\bigwedge\{E_j[h_e] < E_k[h_e]\} \bigwedge C^t$, for all E_j, E_k that have order relationship in Q. The expression is not satisfiable iff at least one $\neg(E_j[h_e] < E_k[h_e])$ can be inferred from C^t. This involves the computation of the closure on Q and C^t, which can also be done in polynomial time.

26.3 Runtime Query Unsatisfiability

As stated before, RunSAT checking differs from SunSAT checking in that RunSAT checking considers a partial trace observed so far. In this sense, SunSAT checking can be considered as a special case of RunSAT checking, i.e., with empty partial trace. Since event data becomes available to the CEP engine in the order of occurrences, the partial trace h_p is always a *prefix* of the entire trace h_e. Definition 15 formalizes the RunSAT problem.

Definition 15 Runtime Query Unsatisfiability (RunSAT) *Given a query Q, event constraints $\mathcal{C}$ and a partial trace h_p, Q is said to be runtime unsatisfiable iff there does not exist a trace h_e that is consistent with $\mathcal{C}$ and contains a match to Q, where h_p is prefix of h_e.*

Next, we consider RunSAT for conjunctive queries, while in Section 26.4, we will discuss disjunctive queries.

Matching and Remaining Sub-Query. Given a partial trace h_p, the matching sub-query Q_m can be defined as follows. A query node E_i is contained in Q_m iff the sub-graph that contains E_i and all the nodes that can reach E_i in $G(Q)$ has a match over h_p. The *remaining query* $\overline{Q_m}$ contains all the unmatched query nodes E_i. The AND nodes are included in $\overline{Q_m}$ if not all of its branches are matched. Figure 26.1 depicts a query Q, partial trace h_p, matching sub-query Q_m and remaining sub-query $\overline{Q_m}$.

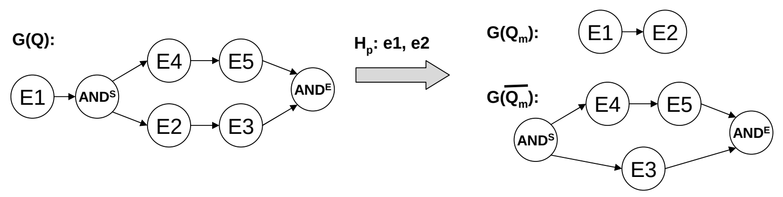


Figure 26.1: Matching, Remaining Sub-Query

Lemma 9 *Given a partial trace h_p and event constraints $\mathcal{C}$, if there does not exist a remaining trace $\overline{h_p} = h_e - h_p$ that contains a match to $\overline{Q_m}$, then Q is runtime unsatisfiable.*

Our goal is then to check the unsatisfiability of $\overline{Q_m}$, which will lead to the unsatisfiability of Q. This naturally leads to the next issue to find the constraints that must hold true for the remaining trace $\overline{h_p}$, referred to as *dynamic constraints*. To distinguish, the initially given event constraints (Chapter 25) are called *static constraints*. The dynamic constraints are derived from the static constraints and hold true for the $future$ data.

Dynamic Constraints. The constraints that the remaining trace $\overline{h_p}$ must satisfy evolve as the partial trace h_p grows. Intuitively, the event instances in h_p serve as *facts*. New constraints can be inferred based on these additional facts and the static constraints. The facts provided by h_p, denoted as $\mathcal{F}_{h_p}$, include:

- $\bigwedge \{E_i[h_p]\}$, for any $e_i \in h_p$ of type E_i
- $\bigwedge \{\neg E_j[h_p]\}$, for any $\mathcal{E} - \{E_i\}$ above

The dynamic constraints $\mathcal{C}_d(\overline{h_p})$ can be evaluated as follows.

$$\mathcal{C}_d(\overline{h_p}) = \mathcal{C} \bigwedge \mathcal{F}_{h_p} = \mathcal{C} \bigwedge \{E_i[h_p]\} \bigwedge \{\neg E_j[h_p]\} \quad (1)$$

The evaluation of Exp.(1) differs from the traditional propositional logic resolution, which basically removes two opposite literals from two clauses [85], in that first $\mathcal{C}$ also contains order constraints $\mathcal{C}^t$, and second each constraint has its own scope.

$$\text{T1}: \frac{\neg(E_i[h_e] < E_j[h_e]) \ , \ E_j[h_p]}{\neg E_i[\overline{h_p}]}$$

$$\text{O1}: \frac{f_1^o \vee \neg E_i[h_e] \ , \ f_2^o \vee E_i[h_e]}{f_1^o \vee f_2^o}$$

$$\text{O2}: \frac{f_1^o \vee \neg E_i[h_e] \ , \ f_2^o \vee E_i[h_p]}{f_1^o \vee f_2^o}$$

$$\text{O3}: \frac{f_1^o \vee E_i[h_e] \ , \ f_2^o \vee \neg E_i[\overline{h_p}] \ , \ \neg E_i[h_p]}{f_1^o \vee f_2^o}$$

Figure 26.2: Constraint Resolution

Figure 26.2 depicts the resolution rules for constraints with scopes. The constraints above the line entail the constraint below the line. We assume that each occurrence constraint f^o is in the form of a disjunction of atomic literals and negation only applies to the atomic literals. First, from rule T1, we see that while the order constraints are *independent* of the occurrence constraints in the static case, they become related in the dynamic case. That is, an occurrence constraint can be derived at runtime through an order constraint. Second, the logical resolution needs special care when each constraint has a valid scope (O1−O3). O1 states that when the two literals have the same scopes, the classic resolution rule can be applied [85]. O2 can be generalized to any h_p that is a subsequence of h_e. O3 shows that the resolution of the constraints with different scopes may need additional evidence from the partial trace h_p. Example 2 illustrates a sample scenario for applying these rules.

Example 2 *Assume two event constraints, $f_1^t = \neg(E_1[h_e]{<}E_2[h_e])$ and $f_2^o = E_3[h_e]{\rightarrow}E_1[h_e]$. When $e_2 \in h_p$, i.e, $E_2[h_p]$, we can infer $\neg E_1[\overline{h_p}]$ from f_1^t by rule T1. However, whether we can further infer $\neg E_3[h_e]$ from f_2^o depends on whether $E_1[h_p]$ is false or not (by rule O3).*

Theorem 5 *Given a query Q, static event constraints $\mathcal{C}$ and a partial trace h_p, Q is runtime unsatisfiable iff the remaining query $\overline{Q_m}$ is statically unsatisfiable w.r.t. the dynamic constraints $\mathcal{C}_d(\overline{h_p})$.*

Proof: "⇐": Follow Lemma 1. "⇒": We prove by contradiction. That is, we assume $\overline{Q_m}$ is satisfiable and there exists a sequence $h_{\overline{Q_m}}$ that matches $\overline{Q_m}$. However, for any $\overline{h_p}$ where $h_{\overline{Q_m}}$ is a subsequence of $\overline{h_p}$ (not necessarily contiguous), $\mathcal{C} \wedge \mathcal{F}_{h_e=h_p+\overline{h_p}} \models \perp$, i.e., Q is not satisfiable.

We start from considering $\overline{h_p} = h_{\overline{Q_m}}$, i.e., $\mathcal{C} \wedge \mathcal{F}_{h_e=h_p+h_{\overline{Q_m}}}$ is not satisfiable. Given the fact that $\mathcal{C} \wedge \mathcal{F}_{h_p} \wedge \overline{Q_m}$ is satisfiable, the only reason for such unsatisfiablity is due to one or more $\neg E_i[h_e]$, while $\mathcal{C} \wedge \mathcal{F}_{h_p} \wedge \overline{Q_m}$ entails that these E_i must occur in h_e. Without loss of generality, we assume that this is due to $\neg E_1[h_e]$. We show that we can find an appropriate position in $h_{\overline{Q_m}}$ where E_1 could occur while satisfying $\mathcal{C}$.

First of all, E_1 could occur, otherwise $\mathcal{C} \wedge \mathcal{F}_{h_p} \wedge \overline{Q_m} \vDash \neg E_1[h_e] \wedge E_1[h_e]$ and is thus not satisfiable. Next, E_1 could occur after h_p, otherwise by rule O_3 in Figure 26.2, $\mathcal{C} \wedge \mathcal{F}_{h_p} \wedge \overline{Q_m}$ is not satisfiable. Lastly, as long as the prior relationship graph in $\mathcal{C}^t$ does not contain cycle, we can find a position in $h_{\overline{Q_m}}$ for the occurrence of E_1 without violating the order constraints in $\mathcal{C}^t$. By repeatedly adding all the required events into $h_{\overline{Q_m}}$, we obtain an event history h_e such that it contains a match to Q and $\mathcal{C} \wedge \mathcal{F}_{h_e}$ is satisfiable. $\square$

Hence, RunSAT checking for a given prefix trace h_p involves two tasks. First, we derive the *dynamic constraints* $\mathcal{C}_d(\overline{h_p})$ that hold true for the remaining trace $\overline{h_p}$, as shown in Exp.(1). Then *RunSAT reasoning* checks whether the remaining query $\overline{Q_m}$ is unsatisfiable by $\mathcal{C}_d(\overline{h_p}) \wedge \overline{Q_m}$. Note that if Q is statically satisfiable, then only occurrence consistency needs to be checked. There is no need to re-check the temporal consistency for remaining query.

Based on the above discussion on matching sub-query and dynamic constraints, we can evaluate the following to check whether the remaining query is unsatisfiable or not. This is a SunSAT problem.

$$\mathcal{C} \wedge \mathcal{F}_{h_p} \wedge \overline{Q_m} = \mathrm{C}_d(\overline{h_p}) \wedge \overline{Q_m} \qquad (2)$$

$$\mathcal{C}_d(\overline{h_p}) \wedge \overline{Q_m} = \mathcal{C}_d(\overline{h_p}) \bigwedge \{E_j[\overline{h_p}]\},\ E_j \in \overline{Q_m} \qquad (2)$$

The evaluation of Exp.(1) and (2) both utilizes the resolution rules in Figure 26.2. Since these rules add a constant scope checking cost to the classic resolution rules, it can be done in polynomial time for Horn clauses.

Effective Dynamic Constraints. Assume that the original conjunctive query Q is statically satisfiable. Based on Exp.(2), the only dynamic constraints that can fail $\overline{Q_m}$ must be in the form of a disjunction of negated atomic literals, such as $\neg\ E_i[h_e] \vee \neg E_j[h_e]$ or $\neg E_k[\overline{h_p}]$. We refer to these constraints as *effective dynamic constraints*, $\mathcal{C}_d^r(\overline{h_p})$, where $\mathcal{C}_d(\overline{h_p}) \models \mathcal{C}_d^r(\overline{h_p})$. This leads us to *goal driven* derivation of these specific dynamic constraints (Chapter 27).

26.4 RunSAT for Disjunctive Queries

In this section, we consider RunSAT for disjunctive queries. As mentioned in Section 26.2, SunSAT for an arbitrary disjunctive query is NP-Complete. While a potentially exponential transformation into its *disjunctive normal form* may be acceptable in the static case when the size of the query is typically small, such exhaustive approach may not be appropriate to be used at runtime for the dynamic case. Rather an efficient, even if incomplete, algorithm for handling disjunctive queries is needed.

The basic idea is to break the original query into several non-overlapping *conjunctive partitions*. Starting from the OR operator that does not contain any other nested OR operator, each branch of this OR operator is marked as a conjunctive partition. This OR operator is then replaced by a single virtual node that represents a disjunction of several partitions. Similarly, we apply this mechanism to the rest of the OR operators in the query until all the OR operators are replaced.

OR-query-graph

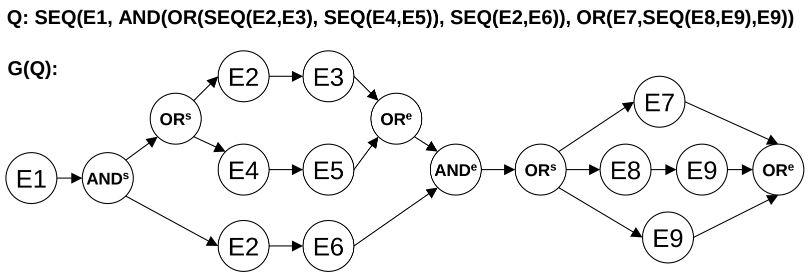

Q: SEQ(E1, AND(OR(SEQ(E2,E3), SEQ(E4,E5)), SEQ(E2,E6)), OR(E7,SEQ(E8,E9),E9))
G(Q):
E1
AND^s
OR^s
E2
E3
E4
E5
E2
E6
OR^e
AND^e
OR^s
E7
E8
E9
E9
OR^e

Figure 26.3: A Disjunctive Query and Query Graph

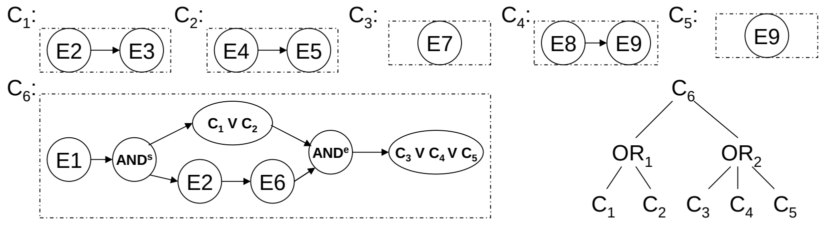


Figure 26.4: Partitioning Disjunctive Query

Figure 26.4 depicts an example for the event query in Figure 26.3. As can be seen, each OR branch is a *conjunctive partition*. These partitions form a *partition hierarchy* as also shown in the figure. The RunSAT technique described in Section 26.3 is applicable to each of these six partitions. C_6 is a *conjunctive partition* with two special nodes ($C_1 \vee C_2$ and $C_3 \vee C_4 \vee C_5$). Intuitively, if all the partitions within the same special node are unsatisfiable, the current partition is also not satisfiable. The number of conjunctive partitions generated by this method is linear in the query size. This technique however is incomplete. For example, after we fail the partitions C_1, C_3, and C_4, the query may have failed already since $C_2 \wedge C_5$ may be statically unstatisfiable already.

Chapter 27

Towards Efficient RunSAT

To achieve earliest possible detection of the runtime query unsatisfiability, RunSAT checking should be conducted each time when h_p grows, i.e., whenever a new event instance is received. In other words, the dynamic constraints derivation, Exp.(1), and RunSAT reasoning, Exp.(2), have to be performed for each event instance. Unfortunately, on first sight this appears to be much more expensive than simply processing the original query. In this section, we will address this performance issue for RunSAT. We will focus on conjunctive queries, as disjunctive queries are also handled based on their conjunctive components (Section 3.4).

27.1 Abductive Inference

As h_p grows from h_{p_1} to h_{p_2}, even an incremental method for deriving $\mathcal{C}_d(\overline{h_{p_2}})$ from $\mathcal{C}_d(\overline{h_{p_1}})$ may not be satisfactory. The reason is that first we may have to store some constraints in $\mathcal{C}_d(\overline{h_{p_1}})$ in order for incremental reasoning, and second we may derive many dynamic constraints that are not useful to fail the query at all.

Fortunately, given the fact that only the effective dynamic constraints could fail the query, we thus propose an *abduction*-based [46, 47] method to *pre-compute* the *conditions* when those effective dynamic constraints will become true. If any of the conditions are met at runtime, which presumably are cheap to monitor, we know some effective dynamic constraints begin to hold. Abductive inference can be formally defined as follows [46, 47]. For a given effective dynamic constraint f_d, $\mathfrak{p}$ is called an explanation of f_d if $\mathcal{C}$ and $\mathfrak{p}$ are consistent with each other and together entail f_d.

1) $\mathcal{C} \wedge \mathfrak{p} \vDash f_d$;

2) $\mathcal{C} \wedge \mathfrak{p}$ is satisfiable.

Here $\mathfrak{p}$ has to be a conjunction of $E_i[h_p]$ and/or $\neg E_j[h_p]$, since these are the only facts we can draw from the prefix trace h_p. Our goal is to find *all* such explanations $\bigvee\{\mathfrak{p}\}$.

To infer the non-occurrence of E_i in the remaining trace, the following three expressions compute its possible explanations.

$$\mathcal{C}^t \wedge \mathfrak{p}_1 \vDash \neg E_i[\overline{h_p}] \qquad (3)$$

$$\mathcal{C}^o \wedge \mathfrak{p}_2 \vDash \neg E_i[h_e] \qquad (4)$$

$$\mathcal{C}^o \wedge \mathcal{C}^t \wedge \mathfrak{p}_3 \vDash \neg E_i[h_e] \qquad (5)$$

First, by using order constraints $\mathcal{C}^t$ alone, we can only derive $\neg E_i[\overline{h_p}]$ from Rule T1 in Figure 26.2. Hence, $\mathfrak{p}_1$=$E_j[h_p]$ if $\mathcal{C}^t \vDash \neg(E_i[h_e] < E_j[h_e])$.

Next, from rules O1-O3 in Figure 26.2, we know that there are two alternative ways that $\neg E_i[h_e]$ can be inferred, namely, from occurrence constraints $\mathcal{C}^o$ only or from both occurrence $\mathcal{C}^o$ and order constraints $\mathcal{C}^t$. Solving Exp.(4) is the classic propositional abductive inference problem [46, 47].

Lastly, solving Exp.(5) needs aid from Rule O3 in Figure 26.2. For any order constraint $\neg(E_r[h_e] < E_s[h_e])$, given the fact that $\neg(E_r[h_e] < E_s[h_e]) \wedge E_s[h_p] \wedge \neg E_r[h_p] \rightarrow \neg E_r[h_e]$, we rewrite Exp.(5) into (6) below, which replaces the order constraint by the occurrence constraints it can possibly imply. Then $\mathfrak{p}_3 = E_s[h_p] \wedge \neg E_r[h_p] \wedge \mathfrak{p}'$.

$\mathcal{C}^o \wedge E_s[h_p] \wedge \neg E_r[h_e] \wedge \mathfrak{p}' \vDash \neg E_i[h_e]$ (6)

Although abductive inference for Exp. (5) and (6) is NP-Complete in general (details in [46, 47]), since it is a one-time cost compared to the long-running event query, the abduction cost may be still acceptable. However, note that the explanations can contain multiple positive events, such as $E_1[h_p] \wedge E_2[h_p] \wedge E_3[h_p]$ or $E_4[h_p] \wedge E_5[h_p]$. In fact, monitoring all such complex explanations could be more expensive than just executing the event query itself and thus becomes infeasible. Hence, a cost-based approach, i.e., monitoring only those explanations that will provide the best cost benefit, is necessary. This remains our future work. In this work, instead we show that when the explanations contain a single positive event for the common yet simple constraints in Table 1, they can be monitored in constant time.

27.2 Incremental RunSAT Reasoning

The second performance issue with RunSAT is that we still have to perform the RunSAT reasoning Exp.(2) for $\mathcal{C}^r_d(\overline{h_{p_1}})$ and $\mathcal{C}^r_d(\overline{h_{p_2}})$, respectively. In other words, we need to store the constraints $\mathcal{C}^r_d(\overline{h_{p_1}})$ in order to check whether they would fail the new remaining query. In fact, we find that for *monotonic* queries, this is not necessary.

Definition 16 Monotonic Query. *Assume two prefix traces h_{p1} and h_{p2} where h_{p1} is a prefix of h_{p2}. The matching sub-queries for a given query Q under these two prefix traces are Q_{m1} and Q_{m2}, respectively. Query Q is monotonic if and only if Q_{m1} is a subquery of Q_{m2}.*

Queries with SEQ, AND operators are monotonic.

Lemma 10 Incremental RunSAT Reasoning. *Assume that the prefix trace grows from h_{p1} to h_{p2}. For a conjunctive query Q, we assume that the remaining queries are $\overline{Q_{m_1}}$ and $\overline{Q_{m_2}}$, and the effective dynamic constraints are $\mathcal{C}_d^r(\overline{h_{p1}})$ and $\mathcal{C}_d^r(\overline{h_{p2}})$, respectively. If Q is a monotonic query, then $\mathcal{C}_d^r(\overline{h_{p_1}}) \wedge \overline{Q_{m_1}}$ is satisfiable $\rightarrow \mathcal{C}_d^r(\overline{h_{p_1}}) \wedge \overline{Q_{m_2}}$ is satisfiable.*

To summarize, to improve the RunSAT performance, first, the derivation of effective dynamic constraints can be precomputed through abduction. Second, when the query is monotonic, there is no need to reconsider the previously derived dynamic constraints. These two techniques pave the way for integrating RunSAT into the event query engine.

Chapter 28

Integrating RunSAT into CEP Engine

In this section, we describe how we apply the theoretical results of RunSAT checking as efficient optimization techniques for event query processing.

Our C-CEP engine employs the commonly-used automata model (i.e., NFA) since it has been shown to be a natural fit for event pattern matching [35, 53, 101]. When registering an event query into the C-CEP engine, the engine first checks whether this query is statically satisfiable w.r.t. event constraints $\mathcal{C}$. Then it uses the abductive inference to precompute the failure conditions. The original event query is augmented with these failure conditions as Event-Condition-Action rules. During query execution, these failure conditions are efficiently monitored. If any of these failure conditions are met, the current trace is unsatisfiable to the query and any partial matches are removed.

28.1 NFA Query Execution Model

For query execution, we adopt and extend the commonly-used NFA model [35, 53, 101] to also support the AND operator. Using this common execution model assures that our work can be easily integrated into existing CEP systems as a *semantic query optimization* module.

Our NFA model includes two types of states, namely, *regular states* and *logical states*, and it can be easily generated from the query graph in Figure 25.1. Each node E_i in the query corresponds to a regular state in the NFA. At runtime, the event instances that match these states are kept in the memory in order to generate the final output. The AND^E corresponds to logical state, which is activated only when all the input transitions have been triggered. There is a self-loop of $*$ transition over those nodes which have non-ϵ output transitions in order to capture the *temporal following* semantics. For example, the query in Figure 25.1 is translated into the automaton in Figure 28.1.

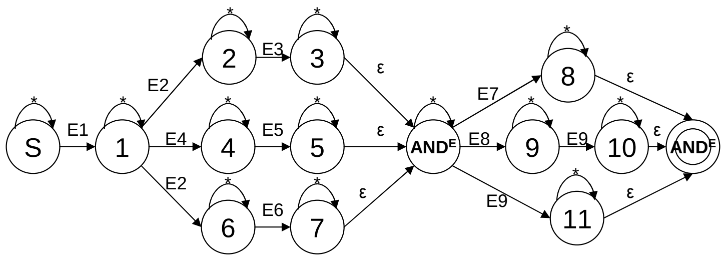


Figure 28.1: NFA for Query in Figure 25.1

28.2 Augment Query with Fail Conditions

Our query engine exploits the constraints in Table 1 for optimizing the event query. We will show that supporting these constraints does not require a cost-based optimization since the extra overhead is small. While developing a cost-based optimization framework for the more complex constraints remains our future work, our performance evaluation for these simple constraints also indicates when such optimization is beneficial, which provides the basis for cost estimation.

The *effective dynamic constraints* that could fail the query are $\neg E_i[h_e]$ and $\neg E_i[\overline{h_p}]$. $\neg E_i[h_e]$ is called *global* since it holds for the entire trace and is independent of the query matching status. $\neg E_i[\overline{h_p}]$ is called *local* since it only holds for the remaining trace. Hence whether $\neg E_i[\overline{h_p}]$ can be used to fail the query depends on whether the remaining query contains E_i or not.

28.2.1 Managing Global Failing Conditions

We first discuss how to augment the query with global failing conditions. For each E_i in the query, we derive all failing conditions for $\neg E_i[h_e]$. By solving Exp.(4), we have the failing conditions $\mathfrak{p}_2 = E_j[h_p]$ if $\mathcal{C}^o \vDash (E_j[h_e] \rightarrow \neg E_i[h_e])$. By solving Exp.(5), which is rewritten into Exp.(6), we have the failing conditions $\mathfrak{p}_3 = E_j[h_p] \wedge \neg E_k[h_p]$ if $\mathcal{C}^o \vDash (E_k[h_e] \rightarrow E_i[h_e])$ and $\mathcal{C}^t \vDash \neg(E_k[h_e] < E_j[h_e])$.

These failing conditions can be organized into a simple data structure depicted in Figure 28.2. We use an array with the size equal to the number of distinct event types. The '$+$' symbol at E_i means that $E_i[h_p]$ is a failing condition of the query. For each entry E_j marked as '$-$', we associate a bit array. For any E_k with the bit being 1 in that bit array, $E_j[h_p] \wedge \neg E_k[h_p]$ is a failing condition of the query.

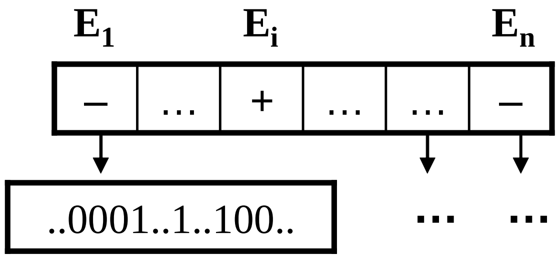


Figure 28.2: Global Failing Conditions

At runtime, given an event instance of E_i, we check if the corresponding entry in the global failing condition is marked as '$+$'. If so, we terminate the processing of this trace. Any partial results or active states for this trace can be removed. If the entry is marked as '$-$' and there is a bit array associated with it, we perform a bit-AND with a runtime bit array whose entries indicate the occurrence of E_i in h_p (1 denotes non-occurrence). If the output of this bit operation is not zero, we can fail the matching for this trace.

28.2.2 Managing Local Failing Conditions

Since the local failing conditions are tightly coupled with the particulars of the current query matching status, we build them into the NFA by introducing a special state labeled "F" (for "Failed"). All transitions triggered by local failing conditions are directed to this "Failed" state.

For each E_i in the query graph, by Exp.(3), we compute the local failing conditions $\{\mathfrak{p}_1\}$ for any E_j that is reachable from E_i in the query graph. We implement the failing conditions in NFA as the additional transitions of E_i. These failing conditions are valid only when none of these transitions out of E_i have been matched yet. Hence there is a special runtime issue, i.e., once the NFA transition from E_i to the next state is made, the local failing conditions at E_i need to be *deactivated*. Intuitively, the query matching status is changed, which breaks the assumption that none of E_i's descendant states have been matched. Such NFA state deactivation can be efficiently supported using a flag. Obviously, both global and local failing condition checking can be done in constant time. Figure 28.3 depicts the augmented query for event pattern EP1 in Section 24.1. The SendQuote event is the local failing condition.

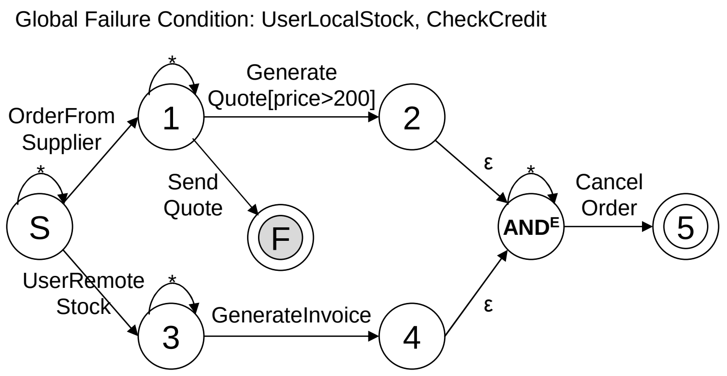


Figure 28.3: Augmented Query for EP1

28.2.3 Handling Disjunctive Queries

The forementioned failure condition management concerns conjunctive queries. For disjunctive queries, each failure condition for E_i needs to be associated with the *partition ID* to which E_i belongs. At runtime, the failing status of each partition is monitored based on the *partition hierarchy* described in Section 3.4. When a partition C_i fails, all its descendant partitions also fail. Then we check C_i's parent node. If it is a different partition, it fails as well and propagates the failure further up the hierarchy. If it is an OR node and all its other child partitions have failed already, we propagate the failure further up this OR node. The entire query for this transaction fails once the root node fails. This failure checking cost per event instance is bounded by the number of partitions in the query, which is typically much smaller than the number of events in the query. Note that for a given event instance, we may simultaneously fail multiple partitions.

Once a partition fails, its partial matches will be removed. The NFA will also associate with each state its corresponding partition ID. If the corresponding partition fails, these states will remain inactive for the rest of the transaction.

Chapter 29

Experimental Evaluation

29.1 Experimental Setup

We have implemented the proposed techniques in a Java-based CEP system. We developed an *event generator* that creates event streams based on the workflow in Figure 24.1 with the following parameters: 1) *event attributes*: 5 attributes (besides timestamp) per event, including three integer-type and two string-type; 2) *number of allowed values* of each event attribute, used to control the selectivity of the query predicates. The values conform to uniform distribution; 3) *probability distribution of exclusive choice construct*, used to control the query selectivity; and 4) *number of concurrent traces* (1000). The events of concurrent traces are interleaved in the event stream. To achieve this, we maintain a list of concurrent transactions. To generate a new event, we randomly select a transaction from the list and generate the next event in that transaction. Once a transaction is finished, we immediately activate a new transaction. Lastly, we fix the number of loops on GenerateQuote in the workflow to be 3. The test machine has an Intel(R) Pentium 1.8G processor and a 1GB RAM, running Windows XP and Java 1.5 SDK.

We compare the performance of *C-CEP*, with regular CEP, denoted as *R-CEP*. For both C-CEP and R-CEP, we apply immediate selection predicate evaluation, i.e., selection push-down. For R-CEP, each time a trace is finished, i.e., whenever a CancelOrder, RejectOrder or FinishOrder event is received, any partial matches and automata states associated with this trace can be removed. For C-CEP, we augment the query with RunSAT failing conditions. Whenever a RunSAT failing condition is satisfied, C-CEP can remove the data. We run both C-CEP and R-CEP in CPU-limit mode [100], i.e., events arrive to the CEP system at a rate such that the query processing never needs to wait for data. This way the optimization cost is also included in the total execution time. We measure 1) total number of NFA probes (for event matching), 2) total execution time for processing the given event stream, and 3) peak number of events maintained in all NFA states, which reflects the peak memory usage. This number is collected after system warm-up, i.e., after 1000 traces are processed. For C-CEP, the execution time includes the RunSAT checking cost. The input event stream contains 400K events from 20,000 traces for all the experiments below.

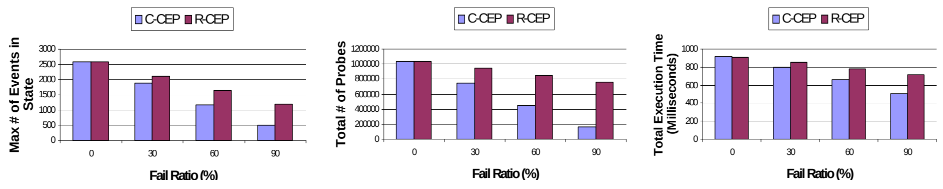
29.2 Results on Sequence Queries

We first compare the performances of C-CEP and R-CEP on sequence queries. We show the experimental results for Query Q1 below, which monitors those expensive orders that uses remote stocks (rare case). The global failing condition for this query is the UseLocalStock event, and the local failing condition for the GenerateInvoice event is the SendInvoice event.

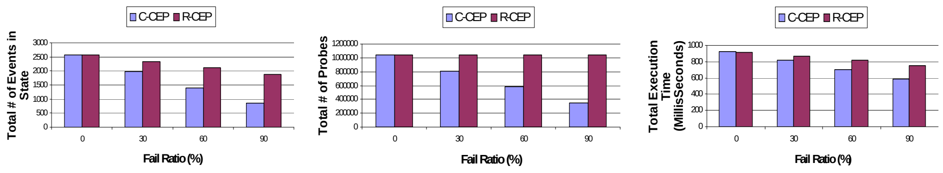
EVENT SEQ(CheckInventory,UseRemoteStock,GenerateInvoice)

WHERE GenerateInvoice.price>200

In the first experiment, we vary the matching probability of the UseRemoteStock event in the query from 0% to 90%. We achieve this by varying the probability distribution of the exclusive choices on UseLocalStock and UseRemoteStock. We define the *fail ratio* of an event E in the query to be (1–σ_E) with σ_E being the matching probability of E. The results are shown in Figure 29.1(a).



(a) Fail Query Early (at UseRemoteStock Event)



(b) Fail Query Late (at GenerateInvoice Event)

Figure 29.1: Sequence Query Performance – Query Fail Point.

Two observations are made from the results. First, as the fail ratio increases, both the total number of probes (and hence total execution time) and peak memory usage decrease. For 90% fail ratio, significant savings in memory (60%) and in execution time (32%) compared to R-CEP are achieved. This promising result suggests that C-CEP is especially attractive for those targeted *alert queries*. Note that the savings in execution time by C-CEP are not precisely proportional to the savings in NFA probes. The reason is that after a trace is determined to be unsatisfiable, for every event in the rest of the trace, a single check is needed to determine whether this event belongs to a failed trace. Second, for zero fail ratio (i.e., all traces have matches to the query), which can be seen as the worst case for C-CEP since no evaluations can be terminated early while extra cost has to be paid for RunSAT checking, the execution time of C-CEP is only negligibly higher than R-CEP. This is also promising, indicating that even in the worst case, C-CEP has comparable performance with R-CEP.

Next, we test how the query fail point affects the C-CEP performance. In the previous experiment, the query fails always due to no match for the UseRemoteStock event. We now test the case in which the query fails always due to no match for the GenerateInvoice event with price>200. We call this the "fail late" case while the previous case the "fail early" case because the UseRemoteStock event is before the GenerateInvoice event in the event query. We vary the matching probability of the GenerateInvoice event to be from 0% to 90%, while fixing the matching probability of UseRemoteStoack to 100%. We achieve this by controlling the value range of the $price$ attribute of the GenerateInvoice event. The results are shown in Figure 29.1(b).

In the "fail late" case, for 90% fail ratio, the memory saving is 54% and execution time saving is 21%. C-CEP still gains in both memory and execution time compared to R-CEP. Since failing late incurs more execution overhead, the gains are less than those achieved in the "fail early" case (Figure 29.1(a)). However, it still provides significant memory savings for alert queries and is thus useful when the memory is a stringent resource.

29.3 Results on AND Queries

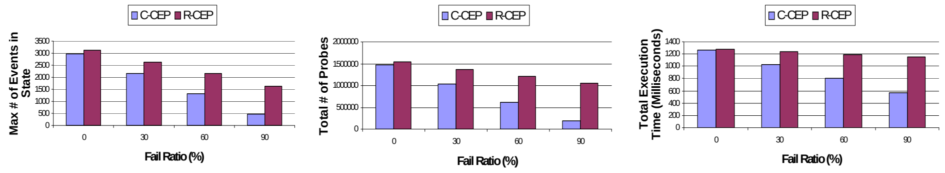
Next, we compare the performances of C-CEP and R-CEP on AND queries. We are seeking answers for two questions: 1) for relatively complex AND queries, can more performance gains be achieved compared to the sequence queries? and 2) how would the interactions between AND branches in the query affect query performance? The query is given below. The global failing conditions for this query are the UseLocalStock and the CancelOrder event, and the local failing condition for the GenerateQuote event is the SendQuote event.

EVENT SEQ(AND(SEQ(OrderFromSupplier, GenerateQuote),

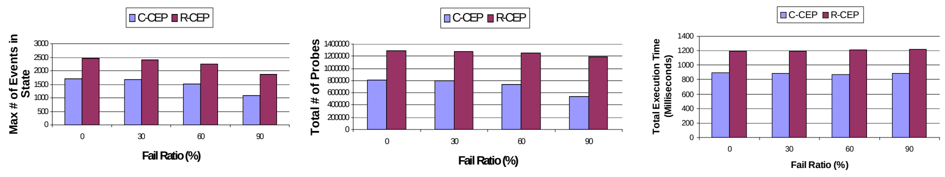
SEQ(UseRemoteStock, GenerateInvoice)),FinishOrder)

WHERE GenerateQuote.price>200

We conduct two sets of experiments. First, we fix the matching probability of the first AND branch (i.e., SEQ(OrderFromSupplier, GenerateQuote)) (more specifically, the GenerateQuote event) to be 50% and vary the matching probability of the UseRemoteStock event to be from 0% to 90%. The results are shown in Figure 29.2(a). Second, we fix the matching probability of the second AND branch (i.e., SEQ(UseRemoteStock, GenerateInvoice)) to be 50%, while varying the matching probability of the GenerateQuote event to be from 0% to 90%. Since 3 loops are involved for GenerateQuote event in the workflow, the failure on matching the first AND branch will be detected rather late compared to that for the second AND branch. This may result in performance difference between these two sets of experiments. The results are in Figure 29.2(b).



(a) Varying Fail Ratio of Branch SEQ(UseRemoteStock, GenerateInvoice)



(b) Varying Fail Ratio of Branch SEQ(OrderFromSupplier, GenerateQuote)

Figure 29.2: AND Query Performance – Interaction of AND Branches.

Two observations are made from this experiment. First, much more performance gains can be achieved compared to the sequence query Q1. As can be seen in Figure 29.2(a), for 90% fail ratio, the gains in peak memory usage and in execution time are 72% and 51% respectively. This is because Query Q2 is more complex than Query Q1, thereby rendering bigger partial matches. This causes higher event matching costs and memory overhead in R-CEP. The C-CEP on the other hand, can terminate the query execution as soon as one branch is found to be unsatisfiable. Another important observation is that the performance gains by C-CEP are determined by the

AND branch that provides the most performance gains. The second AND branch, by failing early, enables much noticeable performance gains as fail ratio increases (Figure 29.2(a)). In contrast, the first AND branch, by failing late, enables much less performance gains until the fail ratio is very high (Figure 29.2(b)).

29.4 Results on OR Queries

Finally, we test the C-CEP performance for OR queries. We modify Query Q2 above by replacing the AND operator by the OR operator and use the new query in this experiment. This query contains three conjunctive partitions: 1) SEQ(OrderFromSupplier, GenerateQuote), 2) SEQ(Use- Remote-Stock, GenerateInvoice), and 3) the entire query. We vary the fail ratios of partitions 1 and 2. We use (fr_1, fr_2) to denote that partitions 1 and 2 have fr_1 and fr_2 fail ratios respectively. Each time a query failure condition is satisfied, corresponding query partitions will be pruned. The experimental results are shown in Figure 29.3.

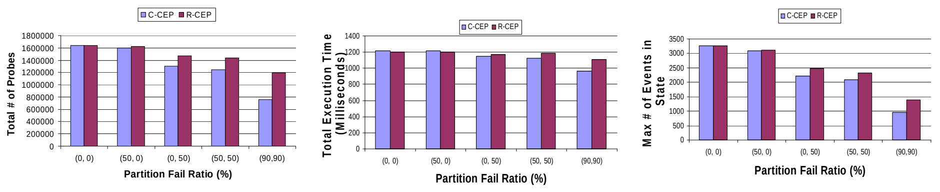


Figure 29.3: OR Query Performance – Varying Fail Ratio of Two Conjunctive Partitions.

We can see that when only one partition could possibly fail, i.e., at least one partition has 0% fail ratio, very little performance gains can be achieved by C-CEP. This is because the cost of the OR query is determined by the branch that provides the least performance gains. This is opposite to the AND query, whose cost is determined by the branch that provides the most performance gains (see Section 29.3). Hence, if a significant portion of the OR query will never fail, not much gains can be achieved by C-CEP, considering the extra RunSAT checking cost. Second, the performance gains increase with the partition fail ratios. When both partitions have high fail ratios, i.e., (90, 90), 30% gains in memory and 13% gains in execution time can be achieved by C-CEP. This is promising, indicating that even for OR queries, significant memory savings can still be achieved for anomaly detection queries.

29.5 Scalability Test

We also conduct the scalability test for the above sequence, AND queries in which the event stream contains 4M events from 200,000 traces with 10,000 concurrent traces. The results are similar to the ones presented here in terms of percentage-wise performance gains and are thus omitted. This indicates that our C-CEP techniques are also scalable.

Chapter 30

Related Work

As event processing gains popularity in many applications, an increasing effort has been devoted in developing efficient event processing systems. The existing work include *streaming databases* such as HiFi [52] that support SQL-style queries, *pub/sub systems* such as [5, 48] that support simple filtering queries, and *CEP systems* such as SNOOP [23], Amit [3], CEDR [17], Cayuga [35] and SASE [101], that support event pattern queries expressed by more powerful languages. These works focus on query model/language design and query algebra development. None of these works consider exploiting the common event constraints.

Semantic query optimization (SQO), i.e., using schema knowledge to optimize queries, has been extensively studied for traditional databases [24, 69]. Major techniques focus on optimizing value-based filtering or matching operations, including join and predicate elimination and introduction. They remain applicable in CEP for identifying efficient query plans at compilation time. These existing SQO techiques are mainly designed for static query optimization. They are inappropriate for runtime use. SQO has also been studied for optimizing queries over streaming XML documents [89]. In CEP, we are faced with event data from possibly thousands or millions of concurrent processes interleaved, and thus huge numbers of potential partial matches (one for each process) at runtime. Also, more types of constraints can be observed in business processes than in XML schema. All these pose stringent requirements on scalability, generality and extensibility on exploiting constraints in CEP.

Our work is also related to *punctuation* [72, 93]. The existing works on punctuation mainly focus on utilizing punctuations to reduce the memory usage of SQL-type of stream query. In this work, we show how to generate punctuations (effective dynamic constraints) from event constraints and how to use them to reduce both CPU and memory cost for CEP queries.

Other related areas include *workflow management* [54, 92] since the event constraints are extracted from workflows. The existing work on workflow management focuses on two problems, *workflow analysis* and *workflow verification*. Workflow analysis involves the soundness proof of a workflow and the identification of critical activities in a workflow. Workflow verification deals with the following problem. Given a finite set S of dependencies, check whether there is a workflow execution (or all executions) satisfying all the dependencies in S. This conceptually is similar to our SunSAT reasoning. Our exploitation of the order constraints relates to the work on temporal reasoning [55, 99], i.e., to detect whether a cycle exists among the order constraints in query and in event data. However, the existing works on temporal reasoning focus on the language specification and enforcement instead of utilizing temporal constraints to optimize queries.

Part V

Conclusions and Future Work

Chapter 31

Conclusions of This Dissertation

A wide range of modern applications need to process queries over large volumes of or even potentially infinite streaming data and provide real-time answers. Due to the unknown characteristics of the streaming data, traditional query processing techniques remain largely inapplicable. In addition, due to the stringent requirements on real-time responses, main memory is regarded as particularly precious resource. Many stream processing systems therefore face serious challenges on resource management, in particular memory management, when processing large numbers of concurrent stream queries, which are the typical workloads of many common applications.

This dissertation has proposed novel techniques to reduce runtime resource requirements by exploiting stream constraints and to maximally share query execution. It focused on three topics, namely constraint-aware query operator execution, shared execution of parameterized streaming group-by queries, and constraint-aware complex event processing.

The first part of this dissertation proposed punctuation-aware operator execution strategies, particularly focusing on join operators, including binary join, binary window join, and multiway window join operators. We concentrated on addressing three issues: 1) design of efficient strategies for identifying no-longer needed tuples; 2) design of strategies for effectively exploit orthogonal constraints, namely punctuation and window constraints; and 3) design of a framework for operators to adaptively tune their behaviour according to various parameters. To address the first issue, we proposed eager and lazy state purge strategies that are suitable for different cases. We also proposed eager and lazy punctuation propagation strategies for operators to benefit downstream operators. To address the second issue, we proposed an effective state organization strategy that facilitate exploitation of both punctuation and window constraints. We also designed the early propagation strategy enabled by the interactions between punctuation and window constraints. To address the third issue, we proposed an adaptive operator execution framework that equip each task with different strategies and adaptively switch between them according to runtime characteristics. We conducted extensive experiment studies that tested and validated the effectiveness of our proposed techniques.

In the second part of this dissertation, we proposed the semantic query optimization approach for data stream processing that exploits herald metadata on attribute values. We designed four herald-driven SQO techniques that, once applied, guarantee performance gains. We proposed a lightweight constraint reasoning algorithm based on classic satisfiability theory to efficiently identify optimization opportunities at runtime upon the receipt of heralds. To optimize the resource usage in supporting multiple concurrent SQO plans with different yet overlapping scopes, we proposed a novel query execution paradigm that employs data partitioning and multimodal operators to achieve multiple logical plans with one single physical plan. Our extensive experimental study confirms that our herald-driven optimization techniques help to significantly reduce query execution time.

In the third part of this dissertation, we proposed the notion of *Parameterized Streaming GroupBy query template* (*PSGB template*) that represents a potentially infinite number of groupby queries (i.e., PSGB queries) to be instantiated at runtime by user requests. We designed the *PSGB operator* to achieve shared execution of all PSGB queries instantiated from a PSGB template. This way the memory for maintaining the groupby state and the CPU time for organizing the groupby state to facilitate the construction and retrieval of groups can be shared among all these PSGB queries. We defined the index tuning problem for pull-based continuous groupby operators. We described the adaptive index tuning process that includes three key operations – index configuration, index evaluation and index migration. We employed a lightweight index structure, namely IMP index, that can be configured to benefit various frequent query patterns. Also, it is easy to migrate. We proposed algorithms for selecting the IMP index configuration that achieves the minimum or close-to-minimum processing cost for a given workload. We conducted an extensive experimental study in a continuous query system. Our experiment results validated the effectiveness of our index selection algorithms and index tuning approach.

In the fourth part of this dissertation, we extended our vision on constraint-exploiting stream query processing to the area of complex event processing (CEP) where events arrive to CEP system as high speed streams. We proposed to exploiting constraints to optimize CEP by detecting and terminating the unsatisfiable query processing at the earliest possible time. We abstracted our problem into a query unsatisfiability problem. We formally defined runtime query unsatisfiability (RunSAT) problem and its extreme case, static query unsatisfiability (SunSAT). We then studied the incremental properties of the RunSAT checking procedure, which includes two key operations, dynamic constraint derivation and RunSAT reasoning. Based on the incremental properties, we described a solution to pre-compute the query failure conditions by employing abductive reasoning. We also presented a constraint-aware CEP architecture that integrates our proposed techniques with state-of-the-art CEP techniques. We showed an extensive experimental study based on online order processes. Our experimental results on sequence, AND queries demonstrated that significant performance gains can be achieved through our approach, while the optimization cost is small.

Chapter 32

Ideas for Future Work

This chapter discusses several future work topics that are important for constraint-aware continuous query processing. In particular, the topics for future work include: 1) punctuation-aware memory management, 2) handling probabilistic punctuations, 3) punctuation generation, and 4) handling more complex constraints in CEP. In the following, I will discuss issues and possible solutions for each of these topics.

32.1 Punctuation-Aware Memory Management

Given dynamic constraints such as punctuations, optimizations of memory utilization in different aspects are possible. The techniques proposed in this dissertation focus on equipping the operators with the ability to efficiently identify and then purge no-longer-needed data from their states. Punctuations can also be exploited for optimizing the memory allocation and state spilling and prefetching, as detailed below.

32.1.1 Punctuation-Aware Memory Estimation

With punctuations, operators may no longer maintain unboundedly growing state even when input data streams are potentially infinite. [72] proposes the technique for identifying whether a continuous join query can be safely executed under a given set of punctuation schemes. However, an even more important issue, i.e., how to estimate the memory needed for executing a safe query with regard to a given set of punctuation schemes, remains unexplored in the literature. The problem is important because only when we know the memory needed by each stateful operator in the query plan, can we intelligently allocate memory to the operators to best achieve optimization goals on memory usage and query throughput.

The memory estimation needs to take into consideration at least four factors, i.e., data arrival rate, data distribution, punctuation arrival rate and punctuation distribution. Data arrival rate determines how fast the operator state grows while the other three factors determine how much the operator state can be shrunk. Together these parameters can be used to estimate how much memory is needed for each stateful operator.

The problem is difficult problem due to the unknown and changing nature of both data and punctuation arrival patterns. Accordingly, the following issues need to be addressed. First, an estimation method (formula) needs to be designed. Second, a statistics collection technique is needed to gather the runtime statistics used by the estimation method. The technique should be lightweight yet efficient such that high-quality statistics can be gathered in a timely manner with low cost. Third, an adaptive framework is needed to adaptively tune the estimation based on the fluctuations occurred at runtime.

32.1.2 Punctuation-Aware State Spilling and Prefetching

In typical cases, the stream query system will experience large numbers of concurrent queries and fast-arriving data streams. Therefore, the available memory is usually not enough to hold all the operator states. To guarantee the exact query result, some data in the state must be moved to the secondary storage such as the disk. Since I/O operations are much more expensive than the in-memory execution, it is desired that the *hot* data, i.e., the data that will be used next, stay in memory while the *cold* data is flushed to disk and is fetched back later when they become hot.

Punctuations can be used to identify hot data and cold data since it signals what attribute values will or will not be carried by the incoming tuples. Using an equi-join over streams S_1 and S_2 on attribute A as an example, if a punctuation arrives from stream S_2 saying that the next 5000 tuples will only have A$>$2000. Then all tuples with A$\leq$2000 in the state for stream S_1 can be flushed to disk since they will not be useful for evaluating the next 5000 tuples from S_2. Accordingly, tuples from the S_1 stream with A$>$2000, if having been flushed to disk before, should be fetched back to memory since they become hot now.

To realize punctuation-aware state spilling, the following issues need to be addressed. First, a mechanism for efficiently identifying hot and cold data, such as lightweight yet efficient index, is needed. The construction of the index should take punctuation types into consideration, e.g., ranged punctuation or set punctuation, positive punctuation or negative punctuation, etc. Second, the on-disk state organization method is needed to effectively organize data on disk so that they can be efficiently located when they need to be fetched back to memory. In terms of queries with window specification, the data on disk need to be purged appropriately after they expire from the window. Third, the prefetching strategy should be designed to timely and efficiently fetch the hot data from disk without affecting the query execution.

32.2 Handling Probabilistic Punctuations

So far the existing work related to punctuations only considers exact punctuations, i.e., punctuations signaling exact information that is taken to be always correct. Such punctuations can only be obtained from a limited set of applications. In many cases, the query systems has to derive punctuations by themselves, as will be discussed in the next section (Section 32.3). The generation of exact punctuations need to buffer and preprocess the data, which incurs preprocessing costs and may affect query responsiveness. A more lightweight solution for generating punctuations is to use sampling. However, punctuations generated by sampling can no longer be exact, but rather they can only be probabilistic. For example, a punctuation may announce that "the next 5000 tuples will have A>2000 with probability of 85%". Given such probabilistic punctuations, none of the existing techniques are applicable. Therefore, an interesting future topic is to design and develop efficient query optimization strategies that exploit probabilistic punctuations and provide guarantees on the precision bound of the approximate query answer.

32.3 Punctuation Generation

Another important topic to be explored is the generation of punctuations. Punctuations can be generated either by the data source providers, or by intermediate devices in network, or by the query processing system itself. The applications may be configured to produce punctuations based on certain rules. For example, the online auction application can be instructed to append a punctuation to the Bid stream whenever the corresponding auction is closed. However, this imposes all responsibility for generating punctuations to data sources. From the query system perspective, it's less controllable. It can also be inefficient since applications may pay significant expense to generate many punctuations that are not useful for query execution. This not only increases network burden but also introduces more punctuation processing overhead to query operators, which is undesirable.

A more promising method that could be explored is for the query system itself to derive punctuations. The query system can derive punctuations in two ways, by sampling data or by pre-processing data. The punctuations derived by sampling are probabilistic punctuations. To derive exact punctuations that can be exploited by existing techniques, a preprocessing method must be used. For example, a query system can employ a buffer to periodically buffer the input data. Then the buffered data are sorted or partitioned based on the needs of query processing and punctuations are generated with attribute value information. Heartbeat [86] is an example of generating punctuations about timestamp using buffering.

There exists a tradeoff between data buffering and query response time. The more data get buffered, the punctuations with longer lifespan can be generated, which therefore reduces optimization overhead. However, this may affect query responsive time, which is very important for stream applications. The punctuation generation mechanism needs to deal with this tradeoff appropriately. In addition, the overhead for preprocessing the data shouldn't outweigh the benefit that it brings for query processing. That is, the punctuation generation mechanism should intelligently determine which attributes to generate punctuations for. That is, it should only generate punctuations that can be used to significantly reduce memory usage and/or increase query throughput.

32.4 Handling More Complex Constraints in CEP

In this dissertation, we considered a core set of event constraints. That is, the failing conditions derived from these constraints involve only a single event type. While these are the most common event constraints in our targeted applications, other more complex constraints may also be very useful. For example, we may have an event constraint such that if events A and B both occur, event C will not occur. Hence for an event query that requires the occurrence of event C, we can monitor the co-occurrence of events A and B as failing conditions for this query. This is not covered by our existing work though.

For general occurrence and order constraints, some failing conditions may involve more than one event type. The cost of monitoring these failing conditions is no longer constant and could be very expensive. Therefore, unlike our existing work that exploits all possible failing conditions, a cost-based optimizer is needed to identify the most beneficial failing conditions and apply them.

Bibliography

[1] D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, C. Erwin, E. Galvez, M. Hatoun, A. Maskey, A. Rasin, A. Singer, M. Stonebraker, N. Tatbul, Y. Xing, R. Yan, and S. Zonik. Aurora: A data stream management system. In *SIGMOD*, page 666, June 2003.

[2] 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.

[3] A. Adi and O. Etzion. Amit - the situation manager. *VLDB Journal*, 13(2):177–203, 2004.

[4] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated selection of materialized views and indexes in sql databases. In *VLDB*, pages 496–505, 2000.

[5] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. Matching events in a content-based subscription system. In *PODC*, pages 53–61, 1999.

[6] A. V. Aho and J. D. Ullman. Optimal partial-match retrieval when fields are independently specified. *ACM Transactions on Database Systems (TODS)*, 4(2):168–179, 1979.

[7] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, I. Nishizawa, J. Rosenstein, and J. Widom. STREAM: The stanford stream data manager. In *SIGMOD*, page 665, June 2003.

[8] A. Arasu, B. Babcock, S. Babu, J. McAlister, and J. Widom. Characterizing memory requirements for queries over continuous data streams. In *PODS*, pages 221–232, June 2002.

[9] A. Arasu and J. Widom. Resource sharing in continuous sliding-window aggregates. In *VLDB*, pages 336–347, 2004.

[10] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *SIGMOD*, pages 261–272, 2000.

[11] A. Ayad and J. F. Naughton. Static optimization of conjunctive queries with sliding windows over infinite streams. In *SIGMOD*, pages 419–430, June 2004.

[12] B. Babcock, S. Babu, R. Motwani, and J. Widom. Models and issues in data stream systems. In *PODS*, pages 1–16, 2002.

[13] S. Babu, R. Motwani, K. Munagala, I. Nishizawa, and J. Widom. Adaptive ordering of pipelined stream filters. In *SIGMOD*, pages 407–418, 2004.

[14] S. Babu, K. Munagala, J. Widom, and R. Motwani. Adaptive caching for continuous queries. In *ICDE*, pages 118–129, April 2005.

[15] S. Babu and J. Widom. Continuous queries over data streams. *SIGMOD Record*, 30(3):109–120, 2001.

[16] S. Babu and J. Widom. Exploiting k-constraints to reduce memory overhead in continuous queries over data streams. *ACM Transactions on Database Systems*, 39(3):545–580, Sep 2004.

[17] R. Barga, J. Goldstein, M. Ali, and M. Hong. Consistent streaming through time: A vision for event stream processing. In *CIDR*, pages 363–374, 2007.

[18] P. Bizarro, S. Babu, D. J. DeWitt, and J. Widom. Content-based routing: Different plans for different data. In *VLDB*, pages 757–768, 2005.

[19] E. Borger, E. Gradel, and Y. Gurevich. *The Classical Decision Problem*. Springer, 1997.

[20] Business Process Execution Language for Web Services. http://www.ibm.com/developerworks/ library/ws-bpel.

[21] Business Rules Markup Language (BRML). http://xml.coverpages.org/brml.html.

[22] 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, 2002.

[23] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim. Composite events for active databases: Semantics, contexts and detection. In *VLDB*, pages 606–617, 1994.

[24] U. S. Chakravarthy, J. Grant, and J. Minker. Logic-based approach to semantic query optimization. *ACM Transactions on Database Systems*, 15(2):162–207, 1990.

[25] S. Chandrasekaran, O. Cooper, A. Deshpande, M. Franklin, J. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*, pages 269–280, Jan 2003.

[26] S. Chandrasekaran and M. J. Franklin. PSoup: a system for streaming queries over streaming data. *VLDB Journal*, 12(2):140–156, 2003.

[27] D. Chatziantoniou, M. O. Akinde, T. Johnson, and S. Kim. The md-join: An operator for complex olap. In *ICDE*, pages 524–533, 2001.

[28] S. Chaudhuri, M. Datar, and V. R. Narasayya. Index selection for databases: A hardness study and a principled heuristic solution. *IEEE Trans. Knowl. Data Eng.*, 16(11):1313–1323, 2004.

[29] J. Chen, D. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *SIGMOD*, pages 379–390, June 2002.

[30] Q. Cheng, J. Gryz, F. Koo, C. Leung, L. Liu, X. Qian, and B. Schiefer. Implementation of two semantic query optimization techniques in DB2 universal database. In *VLDB*, pages 687–698, 1999.

[31] Q. Cheng, J. Gryz, F. Koo, T. Y. C. Leung, L. Liu, X. Qian, and K. B. Schiefer. Implementation of two semantic query optimization techniques in db2 universal database. In *VLDB*, pages 687–698, 1999.

[32] Complex Event Processing - Applications, products, research, and developments in event processing. http://www.complexevents.com/.

[33] A. Das, J. Gehrke, and M. Riedewald. Approximate join processing over data streams. In *SIGMOD*, pages 40–51, Jun 2003.

[34] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows. In *SODA*, pages 635–644, 2002.

[35] A. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma, and W. White. Cayuga: A general purpose event monitoring system. In *CIDR*, 2007.

[36] A. J. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. M. White. Towards expressive publish/subscribe systems. In *EDBT*, pages 627–644, 2006.

[37] V. den Poel Dirk and W. Buckinx. Predicting online-purchasing behavior. *European Journal of Operational Research*, 166(2):557–575, 2005.

[38] A. Deshpande and J. M. Hellerstein. Lifting the burden of history from adaptive query processing. In *VLDB*, pages 948–959, 2004.

[39] L. Ding, S. Chen, E. A. Rundensteiner, J. Tatemura, W.-P. Hsiung, and K. S. Candan. Runtime semantic query optimization for event stream processing. In *ICDE*, 2008.

[40] L. Ding, N. Mehta, E. A. Rundensteiner, and G. T. Heineman. Joining punctuated streams. In *EDBT*, pages 587–604, 2004.

[41] L. Ding and E. A. Rundensteiner. Evaluating window joins over punctuated streams. In *CIKM*, pages 98–107, 2004.

[42] L. Ding and E. A. Rundensteiner. Herald-driven runtime query optimization over streaming data. submitted for publication, 2008.

[43] L. Ding and E. A. Rundensteiner. Index tuning for parameterized stream groupby queries. In *SSPS*, 2008.

[44] L. Ding, E. A. Rundensteiner, and G. T. Heineman. MJoin: A metadata-aware stream join operator. In *DEBS*, 2003.

[45] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of ICSE*, pages 411–420, 1999.

[46] T. Eiter and K. Makino. Generating all abductive explanations for queries on propositional horn theories. In *Conference for Computer Science Logic*, pages 197–211, 2003.

[47] T. Eiter and K. Makino. On computing all abductive explanations. In *Proceedings of AAAI*, pages 62–67, 2003.

[48] F. Fabret, H.-A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe. In *SIGMOD*, pages 115–126, 2001.

[49] L. Fegaras, D. Levine, S. Bose, and V. Chaluvadi. Query processing of streamed xml data. In *CIKM*, pages 126–133, Nov 2002.

[50] G. R. Finnie and J. Barker. Real-time business intelligence in multi-agent adaptive supply networks. In *EEE*, pages 218–221, 2005.

[51] M. R. Frank, E. Omiecinski, and S. B. Navathe. Adaptive and automated index selection in RDBMS. In *EDBT*, pages 277–292, 1992.

[52] M. J. Franklin, S. R. Jeffery, S. Krishnamurthy, F. Reiss, S. Rizvi, E. Wu, O. Cooper, A. Edakkunni, and W. Hong. Design considerations for high fan-in systems: The hifi approach. In *CIDR*, pages 290–304, 2005.

[53] N. H. Gehani, H. V. Jagadish, and O. Shmueli. Composite event specification in active databases: Model & implementation. In *VLDB*, pages 327–338, 1992.

[54] D. Georgakopoulos, M. F. Hornick, and A. P. Sheth. An overview of workflow management: From process modeling to workflow automation infrastructure. *Distributed and Parallel Databases*, 3(2), 119–153.

[55] A. Gerevini. Incremental qualitative temporal reasoning: Algorithms for the point algebra and the ord-horn class. *Artificial Intelligence*, 166(1–2):37–80, 2005.

[56] L. Golab, S. Garg, and M. T. Ozsu. On indexing sliding windows over on-line data streams. In *EDBT*, pages 712–729, 2004.

[57] L. Golab and M. T. Ozsu. Processing sliding window multi-joins in continuous queries over data streams. In *VLDB*, pages 500–511, Sep 2003.

[58] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–169, June 1993.

[59] J. Grant, J. Gryz, J. Minker, and L. Raschid. Semantic query optimization for object databases. In *ICDE*, pages 444–453, 1997.

[60] S. Guo, W. Sun, and M. A. Weiss. Solving satisfiability and implication problems in database systems. *ACM Trans. Database Syst.*, 21(2):270–293, 1996.

[61] A. Gupta, V. Harinarayan, and D. Quass. Aggregate-query processing in data warehousing environments. In *VLDB*, pages 358–369, 1995.

[62] P. Haas and J. Hellerstein. Ripple joins for online aggregation. In *SIGMOD*, pages 287–298, June 1999.

[63] M. Hammad, W. Aref, M. Franklin, M. Mokbel, and A. Elmagarmid. Efficient execution of sliding-window queries over data streams. Technical Report CSD TR 03-035, Purdue University, Dec 2003.

[64] M. A. Hammad, M. J. Franklin, W. G. Aref, and A. K. Elmagarmid. Scheduling for shared window joins over data streams. In *VLDB*, pages 297–308, Sep 2003.

[65] M. Hammer and A. Chan. Index selection in a self-adaptive data base management system. In *SIGMOD*, pages 1–8, 1976.

[66] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *SIGMOD*, pages 205–216, 1996.

[67] G. Ho, H. Lau, C. Lee, A. Ip, and K. Pun. An intelligent production workflow mining system for continual quality enhancement. *IJAMT*, 28(7–8):792–809, 2006.

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

[69] J. King. Quist: A system for semantic query optimization in relational databases. In *Proceedings of VLDB*, pages 510–517, 1981.

[70] J. J. King. Quist: A system for semantic query optimization in relational databases. In *VLDB*, pages 510–517. IEEE Computer Society, 1981.

[71] S. Krishnamurthy, C. Wu, and M. J. Franklin. On-the-fly sharing for streamed aggregation. In *SIGMOD*, pages 623–634, 2006.

[72] H. Li, S. Chen, J. Tatemura, D. Agrawal, K. S. Candan, and W. Hsiung. Safety Guarantee of Continuous Join Queries over Punctuated Data Streams. In *VLDB*, pages 19–30, 2006.

[73] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker. No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. *SIGMOD Record*, 34(1):39–44, 2005.

[74] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker. Semantics and evaluation techniques for window aggregates in data streams. In *ACM SIGMOD*, pages 311–322, 2005.

[75] B. Liu and E. A. Rundensteiner. Revisiting pipelined parallelism in multi-join query processing. In *VLDB*, pages 829–840, 2005.

[76] B. Liu, Y. Zhu, M. Jbantova, B. Momberger, and E. A. Rundensteiner. A dynamically adaptive distributed system for processing complex continuous queries. In *VLDB*, pages 1338–1341, 2005.

[77] B. Liu, Y. Zhu, and E. A. Rundensteiner. Run-time operator state spilling for memory intensive long-running queries. In *ACM SIGMOD*, pages 347–358, 2006.

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

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

[80] P. Maheshwari and S. S.-L. Tam. Events-based exception handling in supply chain management using web services. In *AICT/ICIW*, pages 151–156, 2006.

[81] M. F. Mokbel, M. Lu, and W. G. Aref. Hash-merge join: A non-blocking join algorithm for producing fast and early join results. In *ICDE*, pages 251–262, Mar/Apr 2004.

[82] 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 *CIDR*, pages 245–256, Jan 2003.

[83] E. A. Rundensteiner, L. Ding, T. Sutherland, Y. Zhu, B. Pielech, and N. Mehta. CAPE: Continuous query engine with heterogeneous-grained adaptivity. In *VLDB*, pages 1353–1356, Aug/Sep 2004.

[84] E. A. Rundensteiner, L. Ding, Y. Zhu, T. Sutherland, and B. Pielech. *Stream Data Management (Advances in Database Systems Series)*, chapter 5, pages 83–111. Springer Verlag, 2005.

[85] T. J. Schaefer. The complexity of satisfiability problems. In *Proceedings of ACM Symposium on Theory of Computing*, pages 216–226, 1978.

[86] U. Srivastava and J. Widom. Flexible time management in data stream systems. In *PODS*, pages 263–274, 2004.

[87] Stanford University. Stream query repository. http://www-db.stanford.edu/stream/sqr/, 2002.

[88] H. Su, E. A. Rundensteiner, and M. Mani. Semantic query optimization for xquery over xml streams. In *VLDB*, pages 277–288, 2005.

[89] H. Su, E. A. Rundensteiner, and M. Mani. Semantic query optimization for xquery over xml streams. In *VLDB*, pages 277–288, 2005.

[90] Y. Tao, M. L. Yiu, D. Papadias, M. Hadjieleftheriou, and N. Mamoulis. Rpj: Producing fast join results on streams through rate-based optimization. In *ACM SIGMOD*, pages 371–382, 2005.

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

[92] A. H. M. ter Hofstede, M. E. Orlowska, and J. Rajapakse. Verification problems in conceptual workflow specifications. *Data Knowl. Eng.*, 24(3):239–256, 1998.

[93] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras. Exploiting punctuation semantics in continuous data streams. *IEEE Transactions on Knowledge and Data Engineering*, 15(3):555–568, May/June 2003.

[94] P. A. Tucker, K. Tufte, V. Papadimos, and D. Maier. Nexmark - a benchmark for querying data streams. Technical report, OGI, 2003.

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

[96] T. Urhan and M. J. Franklin. Dynamic pipeline scheduling for improving interactive query performance. In *VLDB*, pages 501–510, Sep 2001.

[97] W. van der Aalst, A. ter Hofstede, B. Kiepuszewski, and A. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(3):5–51, 2003.

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

[99] M. B. Vilain and H. A. Kautz. Constraint propagation algorithms for temporal reasoning. In *AAAI*, pages 377–382, 1986.

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

[101] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *SIGMOD*, pages 407–418, 2006.

[102] S. C. Yoon, I. Y. Song, and E. K. Park. Semantic query processing in object-oriented database using deductive aproach. In *CIKM*, pages 150–157, 1995.

[103] R. Zhang, N. Koudas, B. C. Ooi, and D. Srivastava. Multiple aggregations over data streams. In *SIGMOD*, pages 299–310, 2005.

[104] Y. Zhu, E. A. Rundensteiner, and G. T. Heineman. Dynamic plan migration for continuous queries over data streams. In *SIGMOD*, pages 431–442, 2004.