

SCALABLE AND ROBUST STREAM PROCESSING

by

VLADISLAV SHKAPENYUK

A Dissertation submitted to the

Graduate School-New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

Graduate Program in Computer Science

written under the direction of

Shanmugavelayutham Muthukrishnan

and approved by

---

---

---

---

New Brunswick, New Jersey

May, 2007

## ABSTRACT OF THE DISSERTATION

Scalable and Robust Stream Processing

By VLADISLAV SHKAPENYUK

Dissertation Director:

Shanmugavelayutham Muthukrishnan

Distributed Data Stream Management Systems (DSMS) are increasingly used for the processing of high-rate data streams in real-time. An effective query optimization mechanism is a critical component that allows DSMS to deal with extreme data rates and large numbers of long-running concurrent queries. This dissertation investigates how to utilize semantic query analysis to perform query optimizations that enable scalable and robust data stream processing.

We address three technical challenges faced by streaming system: (1) monitoring and correlating large number of diverse data streams with significant variations in data rates; (2) the ability to remain stable and produce correct answers even under overload conditions, and (3) supporting efficient distributed query processing to easily scale with increases in the number of processing nodes and stream data rates.

First, we propose a heartbeat mechanism to prevent the DSMS from blocking when some of the monitored streams temporarily stall or slow down. By generating special punctuation messages at low-level query nodes and propagating them throughout the entire query execution plan, our heartbeat mechanism effectively unblocks all stalled query nodes.

The second contribution of this dissertation addresses the problem of DSMS robustness when a load on a system increases by orders of magnitude. We introduce a query-aware sampling mechanism for guaranteeing the system’s stability and the correctness of its query output under overload conditions. The mechanism is generic and supports arbitrary complex query sets.

Finally, we address the problem of scalable distributed evaluation of streaming queries. The key contribution of the dissertation is a query-aware partitioning mechanism that allows us to scale the performance of the streaming queries in a close to linear fashion. We propose a query analysis framework for determining the optimal partitioning and a partition-aware distributed query optimizer that takes advantage of existing partitions.

In summary, the contributions made by this dissertation in the area of streaming query optimization enable Data Stream Management Systems to scale to extreme data rates, gracefully handle overload conditions and support a large number of diverse input streams, enabling industrial-scale applications of DSMS technology.

## **Acknowledgements**

I would like to thank my research advisor and mentor Ted Johnson for providing continual guidance and support throughout my Ph.D. studies. Thanks for teaching me how to do systems research and for infecting me with a healthy dose of cynicism towards academia that greatly helped me to survive graduate school.

I would also like to thank my academic advisor Muthu S. Muthukrishnan and the members of my thesis committee, Amélie Marian, Liviu Iftode, and Divesh Srivastava, for helping me to successfully complete my thesis research.

Finally, I would like to thank my parents and my brother Igor for their love and support. I dedicate this thesis to them.

## Table of Contents

<b>Abstract.....</b>	<b>ii</b>
<b>Acknowledgements .....</b>	<b>iv</b>
<b>Table of Contents .....</b>	<b>v</b>
<b>List of Figures.....</b>	<b>viii</b>
<b>1. Introduction .....</b>	<b>1</b>
1.1 Thesis.....	1
1.2 Data Stream Management Systems .....	1
1.3 Problem Statement.....	6
1.4 Summary of Contributions .....	8
1.5 Thesis Organization.....	9
<b>2. Survey of Data Stream Management.....</b>	<b>10</b>
2.1 Query Languages for Data Streams.....	11
2.1.1. Stream Windows .....	11
2.1.2. Declarative Query Languages .....	12
2.1.3. Procedural Query Languages .....	16
2.2 Streaming Query Processing .....	18
2.2.1. Memory Requirements for Query Processing.....	18
2.2.2. Streaming Join Operators .....	20
2.2.3. Approximate Stream Processing.....	21
2.3 Streaming Query Optimization.....	22
2.3.1. Optimization Objectives for Stream Processing .....	23
2.3.2. Adaptive Query Processing.....	24
2.3.3. Multi-Query Optimization .....	26
2.3.4. Load-shedding.....	28
2.4 Distributed Stream Processing Systems .....	29
2.5 Summary .....	31
<b>3. Architecture of High-Performance Data Stream Management System .....</b>	<b>32</b>
3.1 Stream Query Language .....	33
3.1.1. Aggregation Queries .....	35
3.1.2. Running Aggregates.....	36
3.1.3. Stream Merge and Join Queries .....	37
3.2 Two-Level Query Architecture .....	38
3.3 System Architecture .....	39
3.4 Query Optimization .....	42
3.4.1. Splitting Selection and Aggregation Queries.....	42
3.4.2. Prefilters .....	45
3.4.3. Machine Code Generation.....	46
3.5 Summary .....	47
<b>4. Unblocking Unpredictable Streams .....</b>	<b>48</b>
4.1 Introduction .....	49
4.2 Previous Work .....	51
4.3 Using Heartbeats to Unblock Streaming Operators.....	52
4.4 Heartbeat Generation at LFTA Level.....	53
4.4.1 Low-level Streaming Operators in Gigascope .....	54
4.4.2 Effects of Prefilters .....	55

4.4.3	Heartbeats in Selection LFTAs .....	55
4.4.4	Heartbeats in Aggregation LFTAs .....	56
4.4.5	Inferring Values of Temporal Attributes Based on System Time .....	57
4.5	Heartbeat Propagation at HFTA Level .....	58
4.5.1.	High-level Query Nodes in Gigascope.....	58
4.5.2.	Heartbeats in Selection Operator.....	59
4.5.3.	Heartbeats in Aggregation and Sampling.....	60
4.5.4.	Heartbeats in Stream Merge Operator.....	61
4.5.5.	Heartbeats in Join Operator.....	61
4.6	Other Heartbeat Applications .....	62
4.6.1.	Fault Tolerance.....	62
4.6.2.	System Performance Analysis.....	62
4.6.3.	Distributed Query Optimization.....	63
4.7	Limitations of the Heartbeat Mechanism .....	63
4.8	Performance Evaluation .....	64
4.8.1.	Unblocking Stream Merge Using Heartbeats.....	64
4.8.2.	Unblocking join operators using heartbeats .....	67
4.8.3.	CPU overhead of heartbeat generation.....	69
4.9	Summary .....	70
<b>5.</b>	<b>Query-Aware Data Stream Sampling.....</b>	<b>71</b>
5.1	Introduction .....	72
5.2	Previous Work .....	74
5.3	Semantic Sampling Overview .....	76
5.3.1	Illustrative example .....	77
5.3.2	Compatibility of Sampling Methods.....	79
5.3.3	Suite of Sampling Algorithms.....	80
5.3.3.1	Per-tuple Sampling.....	80
5.3.3.2	Per-group Sampling .....	81
5.4	Semantic Sampling for Individual Queries .....	82
5.4.1	Grouping Set .....	83
5.4.1.1	Dealing with Temporal Attributes.....	84
5.4.1.2	Grouping Sets for Aggregation Queries.....	85
5.4.1.3	Grouping Sets for Join Queries .....	86
5.4.2	Selecting the Sampling Method .....	86
5.4.2.1	Sampling in Selection/Projection Queries.....	87
5.4.2.2	Sampling in Aggregation Queries .....	87
5.4.2.3	Sampling in Stream Merge Queries .....	89
5.4.2.4	Sampling in Stream Join Queries.....	89
5.5	Semantic Sampling for Query Sets.....	90
5.5.1	Placement of sampling in a query DAG.....	91
5.5.2	Reconciling query grouping sets .....	91
5.5.3	Algorithm for assigning sampling methods to leaf nodes in query set .....	92
5.6	Limitations of the Semantic Sampling Framework .....	95
5.7	Experimental Evaluation .....	96
5.6.1	Effective Sampling Rate for Join Queries.....	96
5.6.2	Sampling Sensitive Aggregations .....	98
5.6.3	Semantic Sampling for Query Sets .....	100
5.6.4	Efficiency of Load Shedding .....	102
5.8	Summary .....	103
<b>6.</b>	<b>Query-aware Partitioning for Data Streams.....</b>	<b>105</b>

6.1	Introduction .....	106
6.2	Previous Work .....	110
6.3	Query-Aware Stream Partitioning Overview .....	111
6.3.1	Illustrative Example .....	112
6.3.2	Hash-Based Stream Partitioning .....	115
6.3.3	Partition Compatibility .....	116
6.3.4	Inference of Partitioning Sets for Streaming Queries .....	117
6.3.4.1	Dealing with temporal attributes .....	117
6.3.4.2	Partitioning sets for aggregation queries .....	118
6.3.4.3	Partitioning sets for join queries .....	119
6.4	Partitioning for Query Sets .....	120
6.4.1	Reconciling Partitioning Sets .....	122
6.4.2	Algorithm for Computing a Compatible Partitioning Set .....	123
6.4.2.1	Cost model for streaming query nodes .....	124
6.4.2.2	Computing an optimal compatible partitioning set .....	125
6.5	Query Plan Transformation For a Given Partitioning .....	126
6.5.1	Algorithm for Performing Partition-related Query Plan Transformations .....	126
6.5.2	Transformation for Aggregation Queries .....	128
6.5.2.1	Transformation for compatible aggregation queries nodes .....	128
6.5.2.2	Transformation for incompatible aggregation queries .....	129
6.5.3	Transformation for Join Queries .....	131
6.5.4	Transformations for Selection/Projection Queries .....	133
6.6	Comparison of Semantic Sampling and Query-aware Stream Partitioning .....	133
6.7	Experimental Evaluation .....	134
6.6.1	Partitioning for Simple Aggregation Queries .....	135
6.6.2	Partitioning for Join Queries .....	137
6.6.3	Partitioning for Query Sets .....	140
6.6.4	Partitioning for Complex Queries .....	143
6.8	Summary .....	147
<b>7.</b>	<b>Conclusions and Future Work .....</b>	<b>149</b>
7.1	Dissertation Conclusions .....	149
7.2	Directions for Future Research .....	152
	<b>Curriculum Vita .....</b>	<b>166</b>

## List of Figures

Figure 3-1	Gigascop architecture .....	41
Figure 3-2	Aggregate Query Decomposition .....	45
Figure 4-1	Merge query execution plan .....	65
Figure 4-2	Memory usage of stream merge query .....	66
Figure 4-3	Join query execution plan .....	68
Figure 4-4	Memory usage of join query .....	69
Figure 5-1	Semantic sampling example .....	78
Figure 5-2	Labeling of nodes in the query set .....	93
Figure 5-3	Effective sampling rate .....	96
Figure 5-4	Accuracy for aggregation queries .....	98
Figure 5-5	Accuracy for complex query set .....	101
Figure 5-6	CPU load for different sampling methods .....	102
Figure 6-1	Sample query execution plan .....	110
Figure 6-2	Optimized query execution plan .....	113
Figure 6-3	Partition-agnostic query execution plan .....	125
Figure 6-4	Aggregation transformation for compatible nodes .....	127
Figure 6-5	Aggregation transformation for incompatible nodes .....	129
Figure 6-6	Original query execution plan .....	130
Figure 6-7	Join transformation for compatible nodes .....	130
Figure 6-8	CPU load on aggregator node .....	133
Figure 6-9	Network load on aggregator node .....	134
Figure 6-10	CPU load on aggregator node .....	136
Figure 6-11	Network load on aggregator node .....	137
Figure 6-12	CPU load on aggregator node .....	139
Figure 6-13	Network load on aggregator node .....	140
Figure 6-14	Plan for partially compatible partitioning set .....	141
Figure 6-15	CPU load on aggregator node .....	142
Figure 6-16	Network load on aggregator node .....	143



# Chapter 1

## 1. Introduction

### 1.1 Thesis

This dissertation proposes new query optimizations techniques based on semantic query analysis enabling scalable and robust data stream processing in the presence of high-rate data streams. The proposed techniques enable stream management systems to support large numbers of diverse data streams, handle arbitrary complex query sets even under overload conditions, and support efficient distributed evaluation of streaming queries.

### 1.2 Data Stream Management Systems

A large number of measurement and monitoring applications produce their output in the form of a highly detailed data stream – a potentially unbounded sequence of records describing individual observations. Computer networks generate a variety of different data streams – traffic captured at router interfaces, web server accesses, email requests, p2p streams, and many others. Internet traffic itself can be thought as a ‘high-rate data stream’ composed of IP packets. Many network applications, such as traffic analysis, intrusion detection, router configuration analysis, and performance monitoring and debugging, work by processing these streams. Other examples of

applications that generate high-volume data stream are scientific applications such as astronomical survey projects [43][85][107], and meteorological [116] and geodetics measurements [94]. Large-scale sensor networks [3] being deployed both for military uses (e.g. battlefield monitoring) and commercial applications (manufacturing [46], product tracking [54], etc.) have also emerged as an important source of data streams.

Historically, the main process for handling streaming data has been to load it into a data warehouse and then run analysis programs in an offline fashion. However, the explosion in data volumes and an increased need for real-time data analysis has made this model of processing ill-suited for many applications. Consider, for example, a networking application that needs to monitor flows of data between different hosts on the internet. The number of flows in the AT&T IP backbone alone reaches 2.5TB of data per day [66] (fifty billion fifty byte records) making it very challenging to store them entirely. Furthermore, applications that use flow information to detect network attacks intrusions and attacks, need to report them immediately without the delay introduced by offline processing. These trends create a need for systems capable of monitoring massive volumes of data in real-time.

A commonly used solution for real-time stream processing is to employ specialized hand-written applications designed to deal with large volumes of data. For example, in the area of network variety of different tools were developed for specialized tasks of traffic analysis, performance monitoring and debugging, protocol analysis and development, router configuration, and various ad-hoc analyses. Even though these tools are typically acceptable from a performance perspective, they suffer from the number of problems [66][96][97]:

1. **Specialized hand-written tools are typically hard to extend and maintain.** This issue is especially important on the Internet where new applications are popping up very frequently

- requiring modifications to existing monitoring tools. Security related tools (e.g. intrusion and network attack detection) also suffer from the difficulty of adapting to new types of attacks.
2. **No support for post-collection data management.** Hand-built tools usually lack a metadata management facility, which frequently results in the loss of important information describing the semantics of captured data. Unclear data semantics significantly complicates further data analysis and increases the likelihood of errors. Metadata related problems are further exacerbated in the environment where multiple versions of streaming tools are maintained. Such version might have subtle but important differences, resulting in difficulty of interpreting of collected data.
  3. **Poor scalability with respect to the number of monitoring queries.** Many real-world networking applications require dozens to hundreds of monitoring queries to be running at the same time. Hand-optimizing such a large collection of independently developed tools is an intractable problem further complicated by the lack of clear inter-tool interfaces.
  4. **Lack of support for distributed stream processing.** Massive volumes and a natural physical distribution of streaming data make distributed monitoring a necessity for most applications. Hand-built tools are typically designed for single point monitoring and are notoriously difficult to parallelize.

Recent studies of the applicability of using Database Management Systems (DBMS) for network [97] and financial monitoring [82] demonstrate that most typical queries can be expressed using a small number of basic query types – filtering queries, computing aggregate values and correlating multiple streams. Another popular type of query is a trigger – an alert raised when certain conditions are met. Traditional DBMSs benefit from decades of research and industrial development dealing specifically with executing these basic query types on massive data sets. They also feature extensive query optimization (both for single machine and distributed case) and

metadata management facilities. However, these systems are typically too complex and heavyweight for processing data streams. Many integral components of a traditional DBMS such as transaction support, concurrency control, backup and recovery, indices and materialized views are not needed for data streaming applications. Furthermore, relational databases are not designed for processing rapid data feeds and exhibit poor performance on streaming workloads [108].

The emerging class of Data Stream Management Systems (DSMS) is specifically designed to bring the benefits of traditional data processing and management to a streaming world. Most modern DSMSs are based on a variant of SQL language modified to support streaming operations (alternative languages have also been proposed [80][123]). Use of well-defined structured query language brings the benefits of easy metadata management, query optimizations and the query composition for complex query processing. The query optimizer in a Data Stream Management System can take full advantage of precise language semantics to implement a variety of different optimizations, such as:

1. **Transform query executions plans into semantically equivalent but more efficient plans**

DSMSs based on a variant of the SQL language can take advantage of apparatus of underlying relational algebra to perform many standard optimizations such as pushing down projections and selection and join reordering. Similar optimizations are also possible in systems with alternative languages, such as ones based on a state machines [123] and flow networks [2].

2. **Optimize memory utilization of running queries by using temporal properties of the input streams.**

Many real-world data streams contain some form of timestamp attribute, which is generally monotonically increasing. For example, network data streams have timestamps assigned by interface cards, and sometimes also protocol-specific sequence numbers. It is also the case that most streaming queries make references to these monotonic

attributes [66]. Most modern DSMSs take advantage of the temporal properties of the input streams [38][109] to limit the state required to execute streaming operators.

3. **Automatically parallelize query execution based on the number of available stream processing nodes.** The massive scale of many real-world data streams makes distributing processing a necessity. All distributed DSMSs implement a mechanism [1][71] for automatic query plan parallelization and distribution among a number of streaming engines.
4. **Utilize machine code generation to achieve performance exceeding that of hand-built tools.** Many commercial DBMS (such as IBM DB2, AT&T Daytona) incorporate compiler technology [57] to translate submitted query into machine for efficient execution. High-performance stream processing systems also started incorporating similar mechanisms [38] to deal avoid the overhead of query interpretation.
5. **Perform multi-query optimization by detecting and factoring out common components of multiple queries submitted for execution.** A Data Stream Management System is expected to handle a very large number of queries running on the same sets of input streams, which greatly increases the likelihood of significant overlap between the computations performed by different queries. Modern DSMSs implement a variety of multi-query optimizations techniques for sharing filters [78][88], aggregations [51][124] and joins [64].

Most of these optimization techniques potentially can be manually applied to hand-built streaming applications, but it is generally not feasible and potentially dangerous to apply them to large query sets consisting of hundreds of queries. In order to be usable for real-world applications, all of the optimization must be implemented in completely automated fashion based on semantic analysis of the streaming queries.

The area of streaming processing using a DSMS has received a lot of attention in recent years, with a number of general purpose streaming systems proposed by both academia and industry.

List of currently active academic projects includes Aurora/Borealis [1][2], STREAM [6], TelegraphCQ [23], Nile [63], NiagaraCQ [28], and many others. Commercial examples include Gigascope for network monitoring [38][39][40], and Aleri Streaming Analytics [5], Gemfire Real-time Events [49], and Streambase [111] for financial monitoring. A large body of work by database community addressed a number of question related to data stream management: the design of a general purpose DSMS [21], the semantics of the query languages [8], using synopsis data structures for approximate query processing [89], scheduling execution of streaming operators [12], memory management for different types of queries [110], load shedding mechanisms [14], distributed stream processing [28] and many others.

Despite the existence of a large body of work on data stream management, the focus of most of streaming work has always been on stream processing in the abstract. As a result, many real-world problems related to processing high-rate streams, not entirely obvious from these abstract considerations, have not been addressed. In particular, the issues of dealing with very extreme data rates, complex query sets and large number of diverse data streams have received little attention. One of the main goals of this dissertation is to shift focus back to these important scalability issues.

### 1.3 Problem Statement

Three main problems addressed by the dissertation are related to different aspects of scalable and robust stream processing.

1. **A DSMS should be able to simultaneously monitor and correlate large numbers of diverse data streams.** Many real-world data streams are inherently bursty and unpredictable, in particular IP traffic streams processed by network monitoring applications. Even small scale centralized stream processors correlating a small number of streams can easily block on

bursty or stalled input and exceed the available memory. The ability to robustly handle large numbers of unpredictable streams is even more critical for clustered/distributed systems that are expected to deal with order of magnitude differences in data rates of different input streams.

2. **A DSMS must remain stable and produce correct answers even under overload conditions.** It is widely known phenomena that the load on the streaming systems can increase by an order of magnitude (e.g. during network attacks for network monitoring applications). Widely accepted solutions sacrifice the quality of the answers produced by the query processing system in order to remain stable under such adversarial conditions. The challenge that we address in this dissertation is the design of the system that can continue producing semantically correct answers under all load conditions for arbitrary large and complex query sets.
3. **A DSMS should support efficient distributed query processing to easily scale with the increases in the number of processing nodes and stream data rates.** Given the extreme rates of data streams typical for many real-world applications, distributed processing becomes a necessity. The challenge that we address in this dissertation is the design of a system that can automatically take advantage of available distributed resource and effectively spread the computation costs regardless of the complexity of the query set being executed.

All the problems and challenges mentioned above have been previously addressed in the context of specific individual streaming queries. For example, specific sampling algorithms have been suggested for certain aggregation queries to keep the system stable during overload situations. Similarly, a number of parallel execution strategies have been suggested for certain types of aggregation and join queries. So for small application, one can custom build a query that will not suffer from blocking on bursty input stream, be well parallelized and properly handle overload conditions. However, addressing these challenges in the presence of large and complex query sets

with hundreds of queries is intractable for an application development team, and requires completely automated analysis tools that understand the semantics of individual queries.

## 1.4 Summary of Contributions

The main contribution of this dissertation is to utilize semantic query analysis to perform query optimizations that enable scalable and robust data processing in the presence of high-rate streams and arbitrarily large query sets. In particular, the dissertation proposes the following three techniques:

1. **A punctuation-carrying heartbeat mechanism [70] for unblocking bursty and unpredictable streams.** The proposed heartbeat mechanism allows data stream management system to monitor a large number of diverse data streams without blocking even when some of the streams temporarily stall. By generating special punctuation messages at low-level query nodes and propagating them throughout the entire query execution plan, our heartbeat mechanism effectively unblocks all stalled query nodes. Our experiments with multi-source streaming queries running over high-rate data streams demonstrate the effectiveness of the proposed mechanism and its very low computational overhead. This work was published in the *Proceedings of 31<sup>st</sup> International Conference on Very Large Data Bases* [70].
2. **A query-aware sampling mechanism [72] for guaranteeing the system’s stability and the correctness of its query output under overload conditions.** This dissertation defines a notion of correctness of queries in the presence of sampling and introduces a query analysis framework that suggests a semantic-preserving sampling strategy for arbitrarily complex query sets. The effectiveness of our sampling mechanism was validated on high-rate networking data streams. This work was published in the *Proceedings of the First International Workshop on Scalable Stream Processing Systems* [72].



3. **Query-aware data stream partitioning [71] for efficient work distribution across cooperative distributed stream processors.** The proposed stream partitioning mechanism consists of two main components. The first component is a query analysis framework for determining the optimal partitioning for a given set of queries. The second component is a partition-aware distributed query optimizer that transforms an unoptimized query plan into a semantically equivalent query plan that takes advantage of existing partitions.

## 1.5 Thesis Organization

The remainder of this dissertation is organized as follows. Chapter 2 presents a general survey of the research in data stream management. Chapter 3 provides an overview of architecture of Gigascope stream manager for network monitoring used throughout the dissertation as a vehicle for implementation and evaluation for all proposed techniques.

Chapter 4 introduces the problem of unblocking a DSMS in the presence of multiple bursty and unpredictable streams, and introduces the punctuation-carrying heartbeat mechanism to unblock all stalled streams.

Chapter 5 of the dissertation presents a query-aware semantic sampling framework for selecting the appropriate sampling methods during overload conditions

Chapter 6 introduces the idea of query-aware stream partitioning and develops a query analysis framework for selecting an optimal partitioning scheme for arbitrarily large query sets.

Finally, Chapter 7 presents the dissertation conclusions and includes a discussion of future work directions.

## Chapter 2

### 2. Survey of Data Stream Management

The area of data stream management has received a lot of attention in recent years, with a number of active research projects initiated both in the academic community and in industrial research labs. This research has addressed a number of issues, ranging from largely theoretical ones such as modeling infinite streams to very technical issues related to designing execution environments for efficient stream processing. A number of general purpose and application-specific DSMS we created, including Aurora/Borealis [1][2], STREAM [6], TelegraphCQ [23], Nile [63], NiagaraCQ [28], Gigascope [39], and many others.

In this chapter, we survey the relevant work in data stream management with the emphasis placed on query processing and streaming query optimizations. To keep the dissertation focused, we omit the discussion of application-specific topics related such as stream processing in sensor networks [42][68][87][88][93] and financial monitoring [82][128].

The rest of the chapter is organized follows. In Section 2.1 we survey the query languages for stream processing. Section 2.2 focuses on the implementation aspects of streaming query execution. In section 2.3 we provide the overview of the work in streaming query optimization. Finally, we survey the research in distributed data stream processing in Section 2.4.

## 2.1 Query Languages for Data Streams

### 2.1.1. Stream Windows

Traditional languages for querying Relational Database Management Systems (RDBMS) are designed to express one-time queries over relatively static datasets. Streaming queries, on the other hand, operate on unbounded data streams and continuously (or periodically) produce the query answers. In order to be able to execute such queries using fixed amount of memory, it becomes necessary to limit the scope of output tuples that an input tuple can affect. A standard mechanism employed by all DSMS involves defining a window on the data stream on which the query evaluation will occur at any moment in time. An alternative mechanism for a limiting the scope of the input tuples is by time-decaying the tuple values [31]; however this mechanism is restricted to certain types of aggregations queries and will not be discussed further in this survey.

A variety of different window types were suggested in the research literature. Chandrasekaran et al [26] propose the following classification of stream windows based on three defining characteristics:

- **Movement of the window endpoints:** Window endpoints can be either fixed or sliding in one (generally forward) direction. The most widely used type of stream window is a *sliding window* in which both endpoints are moving as the new tuples are added or old ones expire from the window. Windows with one endpoint fixed and the other one sliding are referred to as *landmark windows*; this query type is useful for expressing running aggregates. Queries using windows with both endpoints fixed are essentially equivalent to one-time queries executing over the subset of the data stream.

- **Window size or extent:** Similar to SQL-99 standard for relational databases [60], query languages for data streams distinguish between physical and logical windows. The size of a physical window is defined in terms of number of tuples that belong to the window. Logical windows are defined in terms of a time interval. Logical windows based on the *ordered* or *temporal* attributes [39] define the size of the window as a range of the values of the ordered stream attribute. Finally, the most general type of the window is called a *predicate window* which allows arbitrary predicates to be used to specify the extent of the window. One example of a streaming query that uses predicate windows is the GSQL running aggregate described in more detail in Section 3.1.2.
- **Window update interval:** The frequency with which the window is advanced is defined either in terms of the number of received tuples (for physical windows) or time (for logical windows). A special case of the sliding window in which the update interval is equal to windows extent is referred to as *tumbling window* [39][112], the end result is a partitioning of the input stream into sequence disjoint windows.

Out of many possible window types, physical sliding windows are the ones the most frequently used in the streaming literature, primarily because they are easier to analyze than logical windows. However, query composition is greatly complicated by the need to synchronize multiple tuple-based windows. We further address the issue of query composition in Section 3.1.

### 2.1.2. Declarative Query Languages

A variety of query languages have been proposed for querying unbounded data streams. Most of them are declarative-style languages, based on SQL, with the necessary extensions to support windows and stream-specific operators. In principle, declarative languages give the DSMS a significant freedom to choose the optimal query execution strategy by employing a query

optimizer. In this section we will cover the following declarative streaming languages: CQL, StreaQuel, AQuery, as well as SQL extensions for defining user defined stream aggregates (UDA-SQL). The GSQL language used by Gigascope DSMS is described in detail in section 3.1.

The Continuous Query Language (CQL) [8] developed as a part of Stanford STREAM project [6] attempts to leverage well-understood SQL semantics for relational queries and apply it to continuous queries. The main idea behind CQL is to convert unbounded streams into *instantaneous relations* using relation-to-stream operators and then executing the query over the resulting relation using standard operations of relational algebra. Instantaneous relations can be thought of as a snapshot of a standard relation taken at certain time instant. In order to support query composition for continuous queries that can reference both streams and relations, CQL supports three types of conversion operators: stream-to-relations (using sliding window constructs), relation-to-stream and relation-to-relation (using standard SQL operators).

Relation-to-stream conversion is accomplished by defining a sliding window using SQL-99 syntax. Three window types are supported: time-based (using RANGE keyword), tuple-based (using ROWS keyword) and partitioned windows. Partitioned windows are defined by splitting a sliding window into groups based on a value of partitioning attributes. No tumbling, fixed or predicate windows are supported by the language, although conceptually they can be added with no effect on the semantics.

CQL uses three different relation-to-stream operators (Istream, Dstream and Rstream) which differ by the nature of the output stream they produce. The Istream operator returns all the new tuples in the relation (ones that did not exist in the previous time instant). Dstream returns all the tuples that were present in previous time instant but are no longer present in current relation. Finally Rstream returns the entire content of the relation at the current time instant. An example query that uses the Rstream conversion is shown below:

```

SELECT Istream( S.a, count(*) )
FROM S [Rows 100], R [Rows 100]
WHERE S.srcIP = R.destIP
GROUP BY S.srcIP

```

The query joins two streams of network packets S and R on IP addresses using 100 tuple sliding windows and computes the number of packets observed for each source IP address. The Istream clause guarantees that the results are updated whenever a new IP address is encountered or the number of packets for the existing IP addresses changes.

The StreaQuel language [23][26] proposed by TelegraphCQ project is a pure-stream query language. Continuous queries can only reference streams and produce streams as an output. The output of a StreaQuel query consists of a time sequence of sets, each set corresponding to the answer set of the query at that time (making it equivalent to CQL queries that Rstream construct). A distinctive characteristic of the language is a rich syntax for defining windows over the input stream. StreQuel queries use a for-loop iterating over time to declare a sequence of windows which query uses to compute the answer. Each data stream listed in for-loop has a corresponding WindowIs statement describing a type of window desired. The general syntax for the for-loop is shown below:

```

for(t=initial_value; continue_condition(t); change(t)){
    WindowIs(Stream A, left_end(t), right_end(t));
    WindowIs(Stream B, left_end(t), right_end(t));
    ...
}

```

Using the StreaQuel syntax it is possible to define both fixed, landmark and sliding windows. An example of a query over stream S running for 60 time unit using a size 10 sliding window is shown below:

```
for(t=start_time; t < start_time + 60; t++)
    WindowIs(S, t-9, t)
```

The time increment part of the loop control the frequency with which the query is executed, omitting it completely allows the user to define one-time snapshot queries.

The AQuery [81] language proposed by Lerner and Shasha is focuses on exploiting natural ordered properties of the input streams. In AQuery, the data model treats the tables and streams as ordered entities called *arrables* (standing for array-tables). The query optimizer can take advantage of stream ordering properties for a more efficient execution of the operators. This optimization is accomplished by utilizing order-dependent versions of standard relational operators such as aggregations and joins. Additionally, AQuery can introduce new orderings by using the ORDERED BY clause (implemented by resorting the stream on a new set of ordering attributes). Another interesting feature of the language is a *column-oriented semantics* which treats the table columns as arrays on which order-dependent operators such as next, prev, first, and last are defined. A query comparing two consecutive tuples in the stream that would normally require a self-join can be trivially expressed using a selection query. An example query that compares stock prices for two consecutive days is shown below:

```
SELECT price - prev(price)
FROM Trades
```

A variant of SQL language suggested by Law et al [80] restricts the query language by only allowing *non-blocking queries*. Non-blocking queries do not need to wait for the end of the input

to produce the output results and can be efficiently implemented using pipelined query operators. A main contribution of this work is the support for non-blocking user-defined aggregates (NB-UDA) using a syntax first introduced in ATLaS [121]. The authors prove that every non-blocking (monotonic) streaming function can be implemented using a combination of NB-UDAs and stream union operations.

### **2.1.3. Procedural Query Languages**

Several Data Stream Management Systems employ procedural languages to support querying of the data streams. Queries in such systems are typically composed out of basic building blocks (streaming operators); however some systems employ general purpose programming languages with additional support for streaming operations. In this section we cover the following systems/languages that rely on procedural approach: Hancock, Tribeca and Aurora/SQuAl.

Hancock [37] language was developed at AT&T Labs for extracting data signatures from transactional streams, such as call detail records, credit card transactions, etc. The language is based on C with additional domain-specific features to facilitate stream processing. Hancock programs look at stream as a sequence of values in a fixed format. A variety of constructs are added to the language to support stream filtering, sorting of the elements, detecting user-defined events in streams and executing user-specified code in response to those events. Hancock supports an efficient block processing of tuples with the ability to have multiple processing passes over each data block.

Tribeca [112] system was developed at BellCore for managing network traffic databases. An interesting feature of the system is the support for an extensible type system (similar to object-oriented databases). Different types of network packets are modeled as instances of abstract data types encapsulating particular network protocols (e.g. TCP, UDP, etc). Data type definitions can



use the inheritance to naturally model the layering of the packet headers typical for network data streams.

Tribeca’s query language uses a stream-in stream-out paradigm and allows the composition of complex queries from a set of basic operators – selection, projection and restricted form of sliding windows aggregation and join. Two unique language constructs supported by the language are stream *mux* (multiplexing) and *demux* (demultiplexing). Stream demultiplexing effectively separates the grouping of the tuples from the computation of the aggregate values and allows different aggregates to be applied to each output of the demultiplex operator.

SQuAl [2] language used by Aurora system developed at MIT/Brandeis/Brown, allows users to construct streaming queries by graphically connecting basic operators or *boxes* with directed *arrows* corresponding to data flows. The language views data stream as an append-only sequence of tuples with uniform schema. Aurora makes no assumption about the arrival ordering of the input tuples and allows windows to be defined using arbitrary stream attributes. SQuAl operators are divided into two major categories – *order-agnostic* and *order-sensitive*. Order-agnostic operators (*Filter*, *Map*, *Union*) do not require any assumption about the input stream to execute using bounded amount of memory. They are conceptually equivalent to standard selection, projection and union operations in other streaming languages. Order-sensitive operators (*BSort*, *Aggregate*, *Join*, *Resample*) on the other hand assume a bounded amount of the disorder present in the input stream to execute with bounded amount of memory. The stream order specification includes both an ordering attribute and a *slack* parameter describing the maximum amount of disorder tolerated (maximum distance between the values of the ordering attribute). Aurora does not support automatic imputation of the slack parameters, so they need to be explicitly specified by the query writer.

## 2.2 Streaming Query Processing

Processing unbounded data streams introduces a number of unique requirements on the design of streaming operators. Perhaps the most important requirement is the ability to execute streaming operators with a bounded amount of memory regardless of the size of the input. The evaluation algorithms used by the join operators also needs to be modified to be more suitable for real-time processing of multiple streams with different input rates. Finally, new approximation algorithms need to be developed to handle scenarios when exact query evaluation is not feasible. In this section we will review the recent work done in these areas of stream processing.

### 2.2.1. Memory Requirements for Query Processing

Processing a streaming query operator with a bounded amount of memory is the most fundamental challenge in stream processing. Tuple-based windows discussed in Section 1.1.1 do provide the guarantee that any streaming operator will only use a fixed amount of memory; however, the more widely used logical windows do not. Research efforts addressing this challenge fall into one of the two categories – characterizing the memory requirements of different streaming operators and designing mechanisms that help to reduce the operators' state.

Formal characterization of the memory requirements of continuous queries was given by Arasu et al. in [7]. The main contributions lie in the analysis framework for Select-Project-Join (SPJ) operators with both duplicate-preserving and duplicate-eliminating projection. The resulting bounded-memory conditions for SPJ operators require all attributes in the projection lists and equijoins to be bounded and the set of unbounded attributes participating in join inequality predicates to be empty (for simple SELECT queries) or to be at most size 1 (for SELECT DISTINCT queries). The analysis techniques were extended to cover aggregation queries with the bounding-memory conditions requiring that every grouping attribute needs to be bounded and

that no aggregated function executed on an unbounded attribute be holistic. In addition to characterizing the query memory requirements, the authors propose an algorithm that produces an execution strategy, which uses bounded size data synopses.

Babu et al. [15] attack the problem of reducing memory requirement for streaming query execution by exploiting certain stream constraints. Four types of useful constraints are identified: many-one joins, ordering, clustering and referential integrity. In order to handle a data stream that can occasionally violate these constraints, the paper introduces the notion of *k-constraints* with adherence parameter  $k$  capturing the degree to which a data stream or joining pair of streams adheres to the strict interpretation of the constraint. The authors further describe the design of a low-overhead runtime monitoring system that can dynamically discover which constraints are holding on an input stream, and estimate their adherence parameters. The proposed query execution subsystem takes advantage of the adherence estimates provided by the monitoring system to do early eviction of tuples from their windows.

The stream punctuation mechanism proposed by Tucker et al. [115] works by embedding special markers into the data stream indicating the end of certain subset of the data. Blocking streaming operators such as aggregations and joins can use the information encoded in punctuations to purge the memory state. Some examples of useful punctuations are ones specifying the ordering properties of the input streams, notifying that all tuples with a range of attribute values has been observed and notifying that all the tuples from a certain list have been seen. The mechanism is very generic and allows users to express arbitrary stream constraints, including the ones described in [15]. The punctuation mechanism has also been successfully applied to efficient evaluation of sliding window aggregate [83][84] and joins queries [44].

### 2.2.2. Streaming Join Operators

Traditional blocking join algorithms used by relational databases are poorly suited for data stream processing. The problem of designing a fully pipelined join operator attracted a lot of attention with a number of algorithms proposed in recent years.

Kang et al. [74] investigate the performance of binary window join algorithms in the presence of input streams with asymmetric data rates. A window join operator must maintain a window's worth of data for each of its input streams. A general procedure for processing a new tuple arriving from the first stream involves scanning the second stream's window for matching tuples, inserting the tuple into first stream's window, and invalidating all expired tuples. The data structure used to maintain a tuple window depends on the join algorithm used (e.g. hash-join will use a hash table, while nested-loop join will maintain a simple list). The paper proposes a cost model for operation (inserting, probing for matches and invalidation) as a function of the input stream rates and evaluates several existing join algorithms using this cost model. The important characteristic of the cost model is that divides the join cost into two independent terms, each corresponding to one of the two join directions. The main observation of this work is that mixing different join methods in one operator is preferable for data streams with asymmetric data rates.

Viglas et al [119] investigate the problem of joining more than two streams and advocate using a single multi-way join operator (called *MJoin*) instead of the commonly-used chain of binary joins. Conceptually, the operator is a straightforward generalization of a symmetric binary hash join. Each arriving tuple is inserted into corresponding stream's hash-table and matched sequentially against tuples stored in other stream's hash-tables. In order to reduce the total probing costs, the probing sequence is constructed in such a way that the most selective predicate is evaluated first. The main benefit of the *MJoin* operator is a better output rate as compared to binary join operators. The authors observe that the per-tuple processing cost of

single MJoin operator increases with the number of streams and suggest breaking it into a bushy tree of smaller MJoin operators. Choosing an optimal tree of smaller MJoins remains an open optimization problem.

The sliding window multi-join operator [53] proposed by Golab and Özsu executes multiple joins as a sequence of the nested for-loops. Two join evaluation strategies are suggested and evaluated by the authors. The eager strategy involves reiterating through outer for-loop each time a new tuple arrives in any of the input streams. On the other hand, the lazy strategy processes the outermost for-loop only for tuples which have arrived since the last operator re-execution. If the lazy evaluation strategy is employed, tuples maintained in the sliding windows need to be expired at most as frequently as the query re-evaluation frequency. Authors also suggest ordering the joins in descending order of binary join selectivities to minimize the execution costs per unit of time. The proposed strategies have also been extended to work with hash-based join evaluation.

### **2.2.3. Approximate Stream Processing**

As discussed previously, the exact computation of many types of streaming queries such as holistic aggregates and joins can potentially require an unbounded amount of memory. A variety of approximate algorithms have been developed in the recent years for evaluating such queries with limited amount of storage while providing accuracy guarantees. The majority of approximation algorithms fall into one of the two main categories: *sampling-based* and *sketch-based*.

Sampling-based algorithms work by selecting a small subset of the elements of the stream and maintaining appropriate statistics about those elements sufficient to produce answers with the desired accuracy. Example stream sampling algorithms include fixed-size reservoir sampling [120], concise sampling [59], geometric sampling [16][65], importance-based sampling [45], and

many others. A sampling-based approach was successfully applied to approximate computation of quantiles [56], heavy hitters [89], distinct counts [58], subset sums [45], set resemblance and rarity [41] and geometric problems [16]. However, for many problems sampling primitives are not powerful enough and require large number of samples for complex aggregate functions [27][92].

Sketch-based algorithms work by maintaining *sketches* of source data with a vector of pseudo-random values chosen from an appropriate distribution. A large number of different sketches have been proposed in the literature for solving specific approximation problems including:

- Flajolet-Martin sketch [47] for counting number of distinct elements
- Count-Min sketch [35] for point estimation, range sums, inner products and heavy hitters
- Variants of  $L_2$  sketches [36] for tracking stream histograms
- Group-Count Sketch (GCS) [32] for tracking wavelet representation of data streams

More detailed description of many sampling- sketch-based approximation techniques is given in the following surveys [48][92].

## 2.3 Streaming Query Optimization

Optimizing the execution of the streaming queries poses large number of unique challenges not arising in traditional relational query optimization. In this section, we will describe research efforts addressing several such challenges: defining new cost models and optimization objectives, adaptive query execution, exploiting commonalities between streaming queries, and techniques for shedding the system load.

### 2.3.1. Optimization Objectives for Stream Processing

The job of the query optimizer for relational databases involves enumerating a number of alternative query execution plans and choosing the one that maximizes certain optimization objective functions (e.g. minimizing the amount of disk I/O). Data stream processing creates a need for new optimization objectives relevant for the needs of the streaming applications. Several optimization algorithms have been suggested in the literature differing by their objectives and algorithm for finding optimal execution strategies.

The rate-based query optimization suggested Viglas and Naughton in [118] maximizes output rate of a query execution plan. New time-dependent cost models are proposed for selection, projection, nested-loop and symmetric hash-join operators replace traditionally used relation cardinalities with stream rates. Streaming operator output rates are computed using selectivity estimates combined with estimated computational and communication costs associated with individual operators. This work does not consider the effects of the operator scheduling on the output rates. Two different optimization objective functions are considered by this work: choosing a plan that will produce the most results by certain time, and choosing a plan that will reach the target number of output tuples the soonest. Since choosing an optimal plan is NP-hard, the paper proposes two heuristics that rely on local optimization to reach globally optimal plan.

The *chain scheduling* algorithm [11] used in the STREAM DSMS is aimed at optimizing the runtime memory usage of the streaming operators. The algorithm treats a tuple's processing path as a chain of filtering operators with known selectivities and tuple processing costs. The order in which the operators belonging to a chain are scheduled has a significant impact on tuple backlog between the operators, particularly when the stream is bursty. The algorithm works by graphically representing an operator chain as a *progress chart* encoding both operator processing cost (an x axis) and selectivity (y axis). The problem of finding an operator that maximizes the reduction in

queue sizes per unit of time is equivalent to the problem of finding a highest slope of the lower envelope segment in a progress chart. The algorithm was shown to produce near-optimal memory usage for single query scenarios, however a negative effect on the query response time was also observed.

The scheduling-based query optimization algorithm in Aurora/Borealis [22] is driven by Quality of Service (QoS) specifications provided by the applications. The scheduling is done using a two-level approach – first the decision is made about which continuous query to process, followed by the decision about which order to process the operators within a query. In order to reduce overhead, Aurora groups the streaming operators (boxes) in atomically scheduled and executed *superboxes*. Superboxes can be selected either statically (covering the entire application) or dynamically (covering only operators with highest priority). The order of execution of operators within a superbox is selected according to one of three performance metrics – minimizing per-tuple processing cost, minimizing output latency or maximizing data consumption per unit of time. Additionally, the scheduler makes a decision how many tuples need to be processed as a single batch to increase the overall memory utilization.

### **2.3.2. Adaptive Query Processing**

The optimize-first, execute-next approach typical for most streaming query optimizers critically depends on the accuracy of assumptions made during optimization phase. For example, the optimizer needs to estimate the stream data rates, selectivity of the different predicates, and processing costs and memory requirements for individual streaming operators. The inherent unpredictability of data streams and the long running nature of stream queries increase the likelihood that these estimates will grow increasingly inaccurate during the lifetime of a query. Research in adaptive stream processing address this problem by dynamically reoptimizing the query execution plans as the system conditions change.



The Eddy operator [10], first introduced as a part Telegraph project, implements query adaptivity by dynamically routing the tuples between different query operators. For each tuple arriving into the system, Eddie selects which operator needs to be invoked next based on observed runtime statistics. After the operator finishes its processing, the tuple is returned back to Eddies which makes the next routing decision. In order to keep track of which operations have already been performed on particular tuple, each tuples must maintain a *tuple lineage* encoding the processing history.

*Continuously Adaptive Continuous Query (CACQ)* [25][88] architecture further extends the Eddies to handle continuous queries. The main idea behind the architecture is to refine the granularity of streaming join operators by breaking them into separate *State Modules (SteMs)*. Conceptually, the SteM is one half of the traditional join operator. It stores tuples in a dictionary data structure and supports insert (*build*), search (*probe*) and optionally delete (*eviction*) operations. All joins operating on the same base stream can reuse the same SteM for builds and probes involving that stream. Eddies in CACQ system actively monitor the state of the all input streams and query operators and dynamically change the order of builds and probes for efficient join processing. An additional challenge that needs to be addressed stems from the fact that not all routing policies correspond to valid execution plans. A set of routing constraints were developed to guarantee that queries do not produce duplicate or missing query results.

An alternative adaptative approach employed by the CAPE system [127] uses a dynamic plan migration to switch the execution from suboptimal query plans to a better plans based on runtime statistics. Two alternative migration strategies are suggested: *moving state strategy* and *parallel track strategy*. The moving state strategy involves pausing of plan execution to drain all the tuple queues, followed by the migration of the relevant state to a new execution plan. A similar approach is also employed by the Aurora stream manager [2]. The parallel track strategy is

unique to CAPE system; it uses a more gradual migration by executing both query plans in parallel while slowly migrating the state from one plan to another.

### **2.3.3. Multi-Query Optimization**

Work in traditional multi-query optimization for relational databases focuses on exploiting the commonalities between several queries to improve the overall performance. The likelihood of such commonalities is greatly increased in stream processing systems that are expected to run a number of concurrent queries on the same data stream. The main areas of research interest are sharing filters in selection/projection queries, state sharing for similar aggregation, and join queries working on the same window over the data stream.

The grouped filter [88] technique introduced as a part of the CACQ architecture for adaptive stream processing combines all selection predicates over the same stream attribute into one common operator. The system creates a predicate index that allows the grouped filter to apply many range predicates over an ordered domain to a single tuple more efficiently than applying each predicate independently. An index consists of four different data structures: two balanced binary trees for answering “less-than” and “greater-than” predicates respectively, and two hash tables for answering “equal” and “non-equal” predicates. Additionally the system maintains the bit-masks indicating which predicates are covered by which queries and uses them to decide whether to pass tuple for further processing to respective queries.

Arasu and Widom [9] address the problem of shared execution of sliding and partitioned window aggregation queries that use the same aggregation function but differ in windows sizes. The proposed approach specifically targets queries that do not actively stream their answers, but instead produce answers only when explicitly polled. Two algorithms (B-INT and L-INT) are suggested for sharing sliding window aggregates. B-INT (standing for Base-Intervals) maintains a

data structure storing subaggregates for set of base intervals. The base intervals are computed in such a way that any interval can be expressed as a disjoint union of a small number of base-intervals. L-INT (standing for Landmark-Intervals) on other hand maintains the subaggregates for landmark windows. Multiple queries can share the data structure maintained by B-INT and L-INT to reduce query computation costs.

Zhang et al [124] consider a problem of optimal execution of multiple aggregation queries that feature overlapping but not identical grouping attributes. The proposed solution uses *phantoms*, special aggregation queries that use a combination of the grouping attributes referenced in submitted queries. Aggregated values maintained by the phantom can then be used to inexpensively compute the results for the queries whose set of grouping attributes are covered by the phantom. The paper evaluates several greedy heuristics for choosing the set of phantoms and for allocation of the space required by hash-tables maintained by both phantoms and individual aggregation queries.

Hammad et al [64] considers a problem of shared execution of multiple join queries operating on the same stream but different in the sizes of their respective sliding windows. Three scheduling algorithms are proposed for shared execution of window joins: Largest Window Only (LWO), Shortest Window First (SWF) and Maximum Query Throughput (MQT). The LWO algorithm works by evaluating all the joins at once using one large window containing all the windows referenced by individual joins. The SWF algorithm optimizes the response time of the joins that use smaller windows by scheduling them first. Finally, the MQT algorithm schedules the processing by first choosing tuples that serve the maximum number of the queries per unit of time (to maximize query throughput).

### 2.3.4. Load-shedding

Load shedding is an important query optimization technique designed to guarantee the stability of the stream processing system under overload conditions. The load is reduced by intelligent discarding of a fraction of the unprocessed tuples. The main research challenges lie in trading the potential load reduction against the decreased accuracy of the query answers.

The load shedding mechanism used in STREAM data manager [14] places random sampling operators at various points in the query execution plans to reduce the overall system load. Only simple aggregation queries are considered for which relative query errors can be estimated using Hoeffding bounds. In order to compensate for the effects of random sampling, aggregate results are appropriately scaled. Load shedding is formally defined as an optimization problem whose objective function is minimizing inaccuracy in query answers, subject to the constraint that system throughput must match or exceed the data arrival rate. This work proposes an algorithm for finding the optimal placement of sampling operators in a query execution plan. It also suggests how runtime stream window statistics can be maintained by aggregation operators to dynamically recalculate relative errors and adjust the placement of sampling operators.

The load shedder proposed as a part of Aurora/Borealis project [113] introduces the idea of dropping tuples based on their *utility*. A tuple's utility is computed based on Quality-of-Service (QoS) graphs that need to be specified for every application. Three types of QoS graphs can be used by the system: a *latency graph* specifies the utility of the tuple as a function of time to propagate through query plan, a *loss-tolerance graph* captures the sensitivity of the application to tuple loss, and a *value-graph* shows which attribute values are more important than others. In this system the load shedding becomes an optimization problem where the objective function is minimizing loss of utility resulting from the tuple loss. This work proposes a number of heuristics for reducing a search space of possible placement of load shedding operators. An interesting

property of the system is use of a precomputed *Load Shedding Road Map (LSRM)* structure, encoding the expected load reductions of different query plans to reduce the cost of finding an optimal placement of load shedders to a simple table lookup.

The Data Triage [100] architecture for load shedding developed in the context of the TelegraphCQ project attempts to combine the benefits of approximate query processing and sampling-based load shedding. The main idea is to store the tuple being dropped in a compressed synopsis data structure instead of just silently dropping them. The operators in the query tree need to be modified to use the synopsis of the dropped tuples in addition to sampled input stream and thereby increase the accuracy of the results. Since the synopsis data structure are fairly specialized to the particular join or aggregate being computed, it is not feasible to construct the synopsis that would suit all operators in the query plan. The system includes a Triage Scheduler component that dynamically decides when to route excess tuples for synopsis computation and when to send summarized data to the query nodes responsible for final processing.

## 2.4 Distributed Stream Processing Systems

The massive rates and natural distribution of many real-world data streams have generated a lot of interest in distributed query processing. A number of research projects have been initiated to explore the architectural issues facing the design of distributed stream processing engines. In this section we describe the main published contributions in distributed DSMS.

The Medusa system built at MIT [30][91] is a distributed DSMS building using Aurora [2] as a single-node stream processing engine. The system can function both as a tightly-coupled collection of nodes under the same administrative domain or as a federation of the autonomous participants. Individual participants use an *overlay network* layered on top of the Internet for inter-node commutation. A set of communication primitives used by the system includes scalable

message passing and routing interface for interface and global directory for all nodes and queries in the system.

The main mechanism employed by Medusa for distributed query processing is the partitioning of Aurora query networks across the participating nodes. Two techniques employed for network partitioning are *box sliding* and *box splitting*. Box sliding involves moving the streaming operators (boxes) from the edge of the node's query network to the network on the neighbour node. Box splitting involves creating a copy of the box on the neighbour node and partitioning the input stream to go to both boxes. In order to preserve the correctness of the results, the partial results produced by partitioned boxes need to be combined before being sent for further processing. Medusa uses economic principles to govern the load migration for participating processing nodes. Nodes use a market mechanism with an underlying currency to negotiate the load migration contracts.

The Borealis project jointly developed at MIT/Brandeis/Brown [18] further extends Aurora and Medusa adding support for dynamic revision of the query results and dynamic query modification. The system drops the fundamental assumption that a data stream is an append-only sequence of tuples and introduces a support for post-factum modifications of the previously processed stream elements. Borealis accomplishes it by using special *revision messages* and a history replay mechanism to update the query results based on these messages. Dynamic query adaptation is accomplished by changing various properties of the processing boxes in a query network (e.g. selection predicates, grouping attributes etc).

The distributed version of TelegraphCQ DSMS [105][106] introduces a new *flux* mechanism to enable parallel stream processing on a shared-nothing cluster of machines. Flux partitions streaming query operators across multiple machines in the cluster and coordinates multiple replicas of individual operator partitions within a larger parallel dataflow. The mechanism

supports automatic on-the-fly recovery for dealing with node failures using live replicas of failed operators. An interesting feature of the Flux is automatic load balancing mechanism that adjusts stream partitioning at runtime depending on observed data skew.

## **2.5 Summary**

In this chapter, we present the relevant work in data stream management with the emphasis placed on query processing and streaming query optimizations. We survey the structured and procedural query languages for stream processing, comparing their expressive power and differentiating features. We also give an overview of related research addressing implementation aspects of streaming query operators. Finally we survey the related work in streaming query optimization and distributed stream processing.

## Chapter 3

### 3. Architecture of High-Performance Data Stream Management System

Designing a high-performance Data Stream Management System (DSMS) requires solving a large number of unique technical challenges stemming from extreme data rates and real-time processing requirements. In this chapter, we present the architecture of the Gigascope stream database specifically designed to address these challenges. Gigascope was developed at AT&T Labs-Research for monitoring high-rate network data streams and is currently widely used as a vehicle for networking and streaming research by both AT&T and collaborating universities [19][20][34][61][62][90][99][102][103][117]. The author of this dissertation is one of the principal members of Gigascope development team, along with Charles Cranor, Theodore Johnson and Oliver Spatscheck.

All the contributions of this dissertation were implemented and evaluated in the context of Gigascope system. Even though the system is specialized for network monitoring applications, most of the Gigascope design principles apply to any streaming application required to process high-rate data streams in real-time. This chapter focuses on these more general aspects of the system and sets the content for the rest of the contribution of the dissertations. For the discussion of issues specific to network monitoring see [38][39][40].



The rest of the chapter is organized follows. In Section 3.1 we provide an overview the stream query language used by Gigascope and describe the main types of streaming queries. In Section 3.2, we provide an overview of the two-level (low- and high-) architecture that Gigascope uses for early data reduction. Section 3.3 describes the main software components of the system. .Section 3.4 discusses a number of query optimizations performed by Gigascope to handle the demands of real-time stream processing. We summarize the chapter in Section 3.5.

### 3.1 Stream Query Language

The Gigascope query language, GSQL, is a pure stream query language with SQL-like syntax (being mostly a restriction of SQL). That is, all inputs to a GSQL are streams, and the output is a data stream. This restriction enables easy query composition and greatly simplifies and streamlines the implementation of efficient streaming operators. The query model used by most of the recently proposed stream database systems is that of a continuous query over a sliding window of the data stream [8]. While this model has some advantages (e.g., presentation of results to the end user) and some areas of best application (e.g. sensor networks), it is poorly suited for processing high-rate feeds such as network data streams, as it suffers from poor performance and is cumbersome for expressing typical network analysis queries [77][97][100]. One of the main problems is the complexity of continuous query model, makes it very difficult to implement efficient streaming operators capable of processing the data at line speeds. Query composition is also complicated by complex stream-to-relation and relation-to-stream transformations happening behind the scene. The input to a query is one or more data streams, but the output is a (continuously changing) relation. Queries can still be composed (i.e., can use the output of as its input), but the differences in the output of must often be reverse interpreted as a data stream.

A second problem is the difficulty of precisely expressing a query - or conversely, understanding what a query means. Let us consider example query from [13] that uses CQL (continuous query language) syntax:

```
(Select Count(*) From C, B
  Where C.src=B.src and C.dest=B.dest and C.id=B.id)
/ (Select Count(*) from B)
```

This query is intended to identify fraction of traffic in the backbone B which can be attributed to a customer network C. However the semantics of the result are not clear. Since the output is used for monitoring, the intended result is not likely to be the evaluation of the query over the entire stream, rather over some recent window. However, the window is not specified, and there are in fact three windows to specify (two in the first subquery, one in the second). The snapshots taken by these three subqueries must be precisely synchronized (but on what is not specified), else the result is erratic and meaningless. If the respective windows are defined by a number of tuples rather than by time, the three windows will certainly be unsynchronized. Although example query appears to be simple, an examination of the evaluation details shows that the semantics are complex.

A primary requirement of a DSMS is to provide a way to unblock otherwise blocking operators such as aggregation and join. As we discussed in Section 2.1 unblocking is generally accomplished by defining a window on the data stream on which the query evaluation will occur at any moment in time. Gigascope uses tumbling windows, which are more suitable for network analysis applications [97]. Unblocking is accomplished by limiting the scope of output tuples that an input tuple can affect using a timestamp mechanism. To implement this mechanism, Gigascope requires that some fields of the input data streams be identified as behaving like timestamps. The locality of input tuples is determined by analyzing how the query references the

timestamp fields. In the following sections we will describe all the basic types of streaming queries in GSQL, paying particular attention to how the timestamp analysis is used to unblock normally blocking queries. All the example queries examples assume the following schema.

PKT(time *increasing*, srcIP, destIP, len)

The time attribute is marked as being ordered, specifically increasing.

### 3.1.1. Aggregation Queries

In an aggregation query, at least one of the group-by attributes must have a timestampness, say monotone increasing. When this attribute changes in value, all existing groups and their aggregates are flushed to the operator's output (similar to the tumble operator [21]). The values of the group-by attributes with timestampness thus define *epochs* in which aggregation occurs, with a flush at the end of each epoch. Consider the following GSQL query:

```
SELECT tb, srcIP, destIP, count(*)
FROM TCP
GROUP BY time/60 as tb, srcIP, destIP
```

Since time is monotone increasing, the tb group-by variable is inferred to be monotone increasing also. This query counts the packets between each source and destination IP address during 60 second epochs.

In addition to supporting all standard SQL aggregate functions such as SUM, COUNT, MIN, etc, Gigascope supports User Defined Aggregate Functions (UDAFs) [33] . In order to incorporate a new UDAF into Gigascope, the user needs to provide the following four functions: an INITIALIZE function, which initializes the state of a scratchpad space, an ITERATE function, which adds a value to the state of the UDAF, and a OUTPUT function, which returns the value of

the aggregate, and DESTROY function, which releases UDAF resources. Gigascope handles all the details of managing the scratchpad space for maintaining the state of aggregates and automatically inserts the calls to corresponding functions.

### 3.1.2. Running Aggregates

Standard aggregate queries described in the previous section suffer from one drawback – the state of the aggregates lives only for one epoch. This restriction makes it difficult to compute the aggregates that require variable size windows such as moving averages. Consider for example a streaming query that for every minute and every TCP connection reports the number of duplicate sequence numbers. A GSQL statement for the query is shown below:

```
SELECT tb, srcIP, dstIP, sum_of_dups(seq)
FROM TCP
GROUP BY time/60 as tb, srcIP, dstIP
```

Since some of the TCP connection spans multiple one minute epochs, the query undercounts all such connections. Gigascope solves this problem by introducing special type of aggregation query, *running aggregation*, which allows a running aggregate to retain its state between the epochs. This is accomplished by introducing a new GSQL keyword – Closing\_When. Whenever a predicate given in Closing\_When clause evaluates to TRUE, the state of the aggregate is discarded from the query's memory. In the example above, the proper condition for closing the aggregate is whenever the termination of the TCP connection is detected (e.g. a FYN packet received). A fixed GSQL statement that properly counts the number of duplicate sequence numbers, using the running aggregate sum\_of\_dups, is shown below:

```
SELECT tb, srcIP, dstIP, sum_of_dups(seq)
FROM TCP
```

```
GROUP BY time/60 as tb, srcIP, dstIP

CLOSING_WHEN Or_aggr(FYN) = TRUE or count(*) = 0
```

Using running aggregate functionality, it becomes possible to express sliding windows queries such as moving averages in GSQL. Similar approach of expressing sliding windows queries using more efficient tumbling windows is also used by Li et al. [84] in their work on pane-based aggregate evaluation.

### 3.1.3. Stream Merge and Join Queries

The merge operator allows us to combine streams from multiple sources into a single stream. The operator is particularly important for applications that need to monitor a number of streams that share the same schema. For example, network monitoring applications frequently need to monitor a number of network links as one logical link.

A merge operator performs a union of two streams R and S in a way that preserves timestamps. R and S must have the same schema, and both must have a timestamp field, say t, on which to merge. If tuples on one stream, say R, have a larger value of t than those in S, then the tuples from R are buffered until the S tuples catch up. For example, the query below merges two TCP streams coming from two separate network interfaces into one logical stream:

```
MERGE R.timestamp : S.timestamp

FROM interface1.TCP R, interface2.TCP S
```

Similarly a join query on streams R and S must contain a join predicate such as  $R.tr=S.ts$  or  $R.tr/2=S.ts+1$ : that is, one that relates a timestamp field from R to one in S. The input streams are buffered (in a manner similar to that done for merge) to ensure that the streams match up on the

timestamp predicate. An example of join query that combines the length of packets with matching IP addresses is shown below:

```
SELECT time, PKT1.srcIP, PKT1.destIP, PKT1.len + PKT2.len
FROM PKT1 JOIN PKT2
WHERE PKT1.time = PKT2.time and PKT1.srcIP = PKT2.srcIP
      and PKT1.destIP = PKT2.destIP
```

Even though Gigascope currently does not support sliding windows joins, it is fairly straightforward to extend GSQL and query translator to support this type of streaming queries.

### 3.2 Two-Level Query Architecture

Gigascope has a *two-level query architecture*, where the *low* level is used for data reduction and the *high* level performs more complex processing [38][39][40]. This approach is employed for keeping up with high streaming rates in a controlled way. High speed data streams from, e.g. a Network Interface Card (NIC), are placed in a large ring buffer. These streams are called source streams to distinguish them from data streams created by queries. The data volumes of these source streams are far too large to provide a copy to each query on the stream. Instead, the queries are shipped to the streams. If a query  $Q$  is to be executed over source stream  $S$ , then Gigascope creates a subquery  $q$  which directly accesses  $S$ , and transforms  $Q$  into  $Q_0$  which is executed over the output from  $q$ . In general, one subquery is created for every table variable which aliases a source stream, for every query in the current query set. The subqueries read directly from the ring buffer. Since their output streams are much smaller than the source stream, the two-level architecture greatly reduces the amount of copying (simple queries can be evaluated directly on a source stream).

The subqueries (which are called *LFTAs*, or low-level queries, in Gigascope) are intended to be fast, lightweight data reduction queries. By deferring expensive processing (expensive functions and predicates, joins, large scale aggregation), the high volume source stream is quickly processed, minimizing buffer requirements. The expensive processing is performed on the output of the low level queries, but this data volume is smaller and easily buffered. Depending on the capabilities of the network interface card (NIC), we can push some or all of the subquery processing into the NIC itself. In general, the most appropriate strategy depends on the streaming rate as well as the available processing resources. Choosing the best strategy is a complex query optimization problem that attempts to maximize the amount of data reduction without overburdening the low level processor and thus causing packet drops. We will give more detailed description of the query splitting optimizations in Section 3.4.1.

The Gigascope DSMS has many aspects of a real-time system: for example, if the system cannot keep up with the offered load, it will drop tuples. To spread out the processing load over time and thus improve schedulability, Gigascope implements traffic shaping policies in some of its operators. In particular, the aggregation operator uses a *slow flush* to emit tuples when the aggregation epoch changes. One output tuple is emitted for every input tuple which arrives, until all finished groups have been output (or the epoch changes again, in which case all old groups are flushed immediately).

### 3.3 System Architecture

The Gigascope database system consists of a four main software components: query translator, runtime system, cluster manager, and applications:

- *Query translator* translates GSQL queries submitted to the system into multiple executable query modules called *FTAs* (stands for Filtering, Transformation, and Aggregation). First all

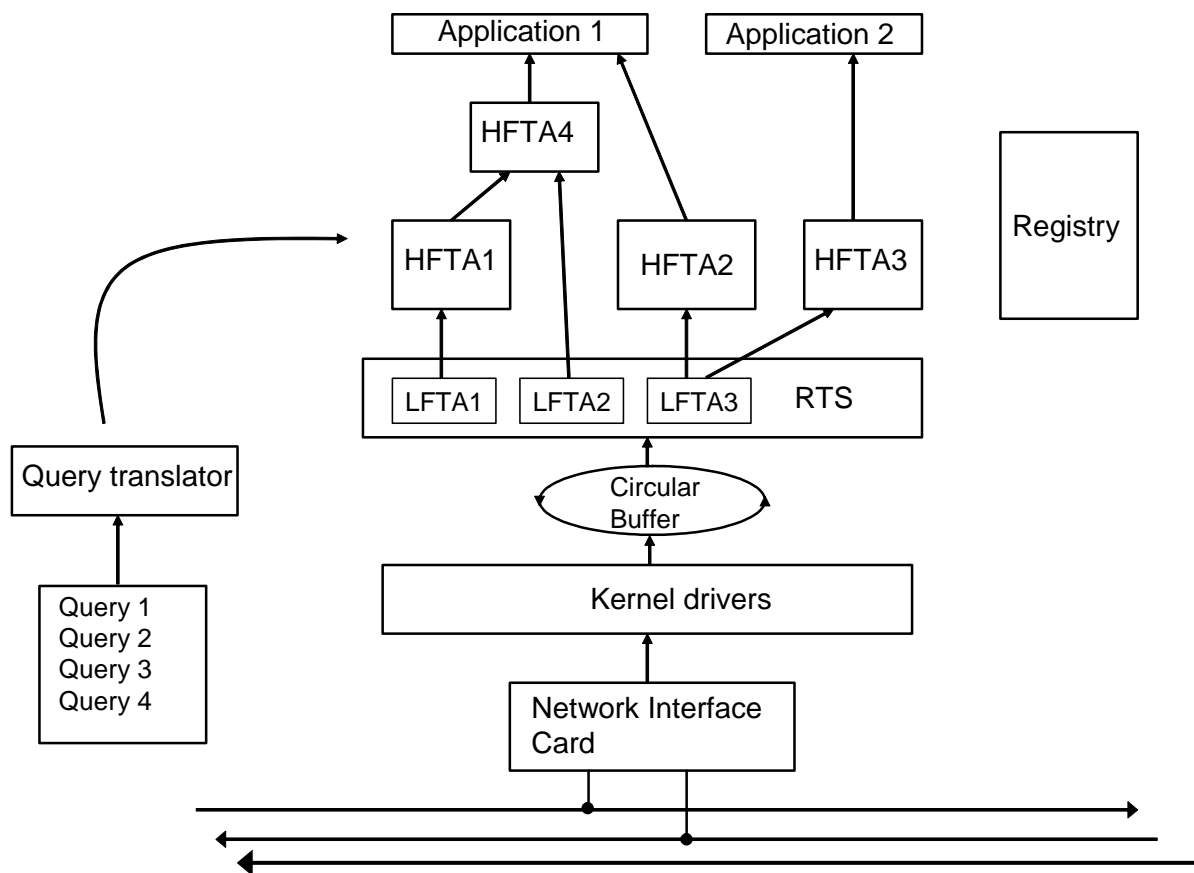
submitted queries are automatically split into lightweight low-level queries performing simple selection and aggregations (LFTAs) and complex high-level queries performing more complex aggregations, merges and joins (HFTAs). After performing the split, the queries are translated into C/C++ code which is then translated into native machine code. All the HFTAs run as separate processes using a standard *stream library* to communicate with other FTAs and applications. All the LFTA modules are linked directly into the runtime system for efficient access to the source streams. A query translator is capable of generating both centralized and distributed query plans depending on particular Gigascope configuration. If a streaming query spans multiple network interfaces or several distributed data streams, the generated code is automatically parallelized to use the available resources.

- *Runtime system* provides the entire infrastructure necessary for running the FTA on the network streams coming from one of the managed interfaces. It provides such services as management and tracking of the data sources (network interface cards, remote source, and file sources), maintaining the registry of all active FTAs, and handling Inter-Process Communications (IPC). Additionally, the runtime system is responsible for the scheduling and execution of all the low-level queries linked directly into it. Each Gigascope node in distributed configurations runs its own runtime system responsible for the local FTAs.
- *Cluster manager* component is responsible for managing a network of cooperating Gigascope nodes. This component is responsible for all aspects of distributed stream processing: placement of the FTAs on participating hosts, failure detection for applications and streaming queries, restart-based recovery, load shedding during overload conditions, and performance monitoring. In addition, a cluster manager is responsible for providing a distributed FTA registry service for remote nodes.



- *Applications* are the main consumers of the output produced by the streaming queries. From the system's perspective there is a little difference between the applications and HFTA modules. Both run as separate processes and can subscribe to and consume the output streams produced by other FTAs using standard stream library. The only difference lies in the fact that application does not produce the output stream of its own and essentially acts as a data sink. Many Gigascope applications dump the processed streaming data into data warehouse for further offline analysis.

A simplified architecture of a single-node Gigascope system is shown in Figure 3-1.



**Figure 3-1: Gigascope architecture.**

### 3.4 Query Optimization

Effective query optimization mechanism is critical for a Data Stream Management System that needs to perform sophisticated query processing at line speeds. Gigascope uses a large number of optimizations to lower the processing cost for both HFTA and LFTA queries. The range of techniques employed includes conventional optimizations based on relational algebra (pushing selection and projection as low as possible, join reordering) and a number of unique streaming query optimizations. In the following subsections we give an overview of streaming-specific Gigascope.

#### 3.4.1. Splitting Selection and Aggregation Queries

In section 3.2, we discussed that optimally splitting streaming queries is a complex optimization problem. Intuitively we would like to maximize the amount of data reduction performed by low-level queries (by pushing more processing to LFTAs), while keeping per-tuple processing costs very low to avoid overburdening the runtime systems and causing an uncontrollable packet drop. The solution used in Gigascope relies on a simple cost model to compare the respective costs of different selection predicates and scalar expressions involving the attributes of the data stream. Only the predicates and functions deemed inexpensive enough to run on low-level (called *LFTA-safe* predicates and functions) are pushed down for execution in an LFTA.

LFTA-safeness largely depends on the restrictions or additional capabilities of the runtime system used in particular Gigascope configuration. For example, if the runtime system is running on a network interface card, complex operations such as regular expression matching will be considered too expensive to be pushed to the LFTA level. However, on specialized hardware using FPGAs to perform fast regular expression matching [104], this operation is natively supported and will therefore be pushed down by the query translator.

We will illustrate how query splitting works using a network monitoring query that extracts the names of the hosts from HTTP requests. The GSQL statement for this selection query is given below:

```
SELECT tb*60, destIP, dest_port,
       str_extract_regex(TCP_data, '[Hh][Oo][Ss][Tt]:[0-9A-Z\\.\.: ]*')
       as hostheader
FROM TCP
WHERE ipversion=4 and offset=0 and protocol=6 and
       str_match_start[TCP_data, 'GET']
```

The query selects only TCP packets that starts with “GET” (using `str_match_start()` function) and extracts the name of HTTP hostnames using `str_extract_regex()`. For the runtime system running on a NIC, `str_extract_regex()` is prohibitively expensive and thus it is move into a high-level subquery. The results of automatic query decomposition for the query are shown below:

**Query `hostnames_low`:**

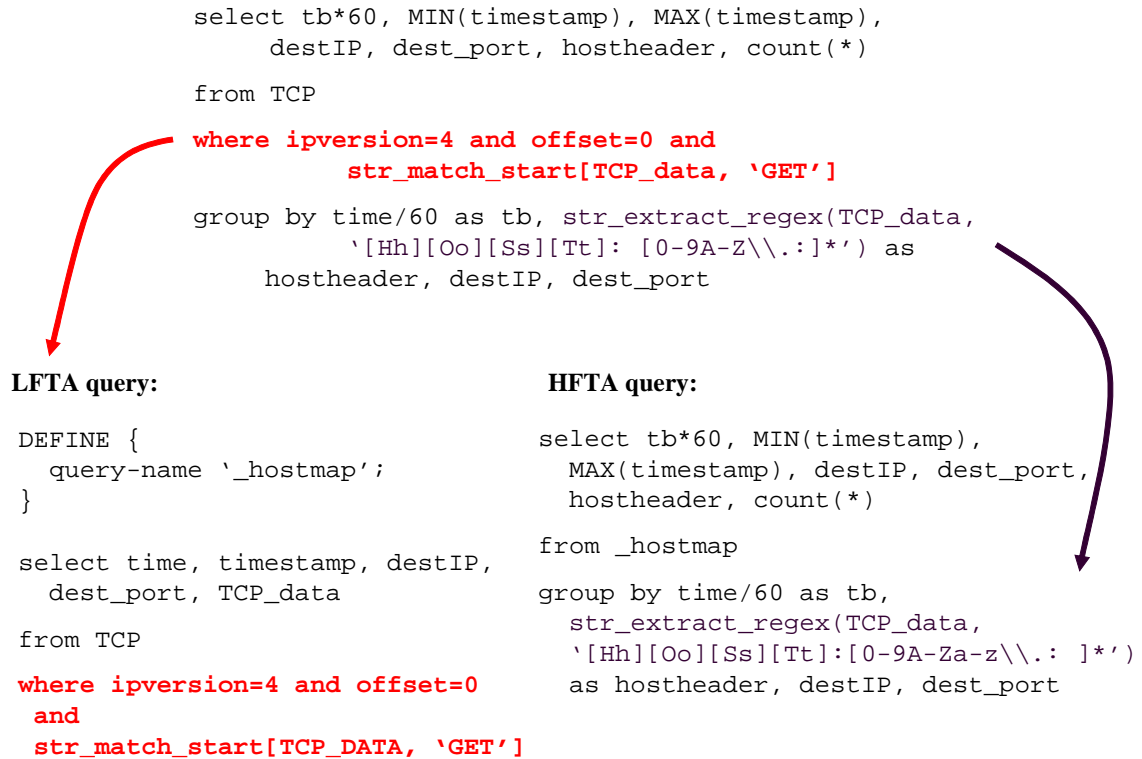
```
SELECT tb*60 as t, destIP, dest_port, TCP_data
FROM TCP
WHERE ipversion=4 and offset=0 and protocol=6
       str_match_start[TCP_data, 'GET']
```

**Query `hostnames_high`:**

```
SELECT t, destIP, dest_port,
       str_extract_regex(TCP_data, '[Hh][Oo][Ss][Tt]:[0-9A-Z\\.\.: ]*')
       as hostheader
FROM hostnames_low
WHERE
```

Splitting aggregation queries is done similarly; however there are additional considerations related to the way that aggregation is implemented at LFTA level. To ensure that aggregation is fast, the low-level aggregation operator uses a fixed-size hash table for maintaining the different groups of a GROUP BY. If a hash table collision occurs, the existing group and its aggregate are ejected (as a tuple), and the new group uses the old group's slot. That is, Gigascope computes a partial aggregate at the low level which is completed at a higher level. The query decomposition of an aggregate query  $Q$  is similar to that of subaggregates and superaggregates in data cube computations [55]. If the definition of one or more grouping variables uses LFTA-unsafe function, it is not possible to decompose the aggregation query into sub- and superaggregates. Instead the low-level query will be restricted to performing selection using LFTA-safe predicates from the original query.

We demonstrate the aggregate query decomposition on a networking query that tracks how many requests people send to different HTTP hosts. The query is similar to previously shown selection query, except now the data is aggregated using extracted hostname as a key. Since the aggregation key (hostname) is computed using an LFTA-unsafe function, the query will be split into a low-level selection and a high-level aggregation query. The final query decomposition is shown in Figure 3-2.



**Figure 3-2: Aggregate Query Decomposition**

### 3.4.2 Prefilters

A Data Stream Management System is expected to handle a very large number of queries running on the same sets of input streams, which greatly increases the likelihood of significant overlap between the computations performed by different queries. In order to avoid performing redundant computations Gigascope utilizes a *prefilter* mechanism which extracts the shared predicates out of streaming queries and executes them only once per input tuple. In order to keep the prefilter very lightweight and to avoid pushing expensive predicates that may not be invoked by LFTAs, only cheap predicates are selected for the inclusion in the prefilter. Non-shared predicates are also considered since pushing them into prefilter allows the Gigascope to avoid relatively expensive LFTA invocations.

The query translator selects the candidate predicates based on the query analysis and generates a special prefilter bit-vector with one bit assigned to each selected predicate. All the predicates selected for inclusion in the prefilter are removed from their corresponding queries. Additionally for every LFTA query, a signature bit-vector is computed denoting which of the prefilter predicates it contains. Whenever an input tuples enters the system, it is passed to the prefilter, which evaluates the selected predicates and sets the corresponding bits of the prefilter bit-vector. The resulting bit-vector is then compared with the signature of each LFTA to figure out whether the tuples should be passed for further processing by that LFTA.

### 3.4.3 Machine Code Generation

Interpreting a streaming query at runtime incurs a significant CPU overhead that should be avoided if real-time tuples processing is required. To avoid this overhead, Gigascope instead uses a generated code system. All the input queries are translated into C (for LFTAs) and C++ (for HFTAs) code which is then translated into native machine code. The object modules corresponding to the low-level queries are linked directly into runtime system. Having multiple LFTA in the same address space removes a lot of synchronization overhead of accessing the ring buffer and leads to good cache locality critical for low-level queries. The obvious drawback of this implementation is the loss of flexibility – it is not possible to add new LFTAs on a fly. However, the performance benefits combined with ability to adapt existing LFTAs using a parameter mechanism make it good choice for real-time stream processing.

An interesting aspect of the query translator is template-based generation of HFTA code. All the high-level streaming operators are implemented as general C++ template classes encapsulating the general functionality of an operator. The query translator specializes the templates by generating a special *functor* class specific to particular query. For example, the template for an aggregation operator implements all the generic functionality required by this type of query:

maintaining a group table, updating the values of the aggregates, flushing the aggregate values of the epoch change, etc. The generated aggregate functor only needs to implement query-specific functionality such as extracting all referenced tuple attributes and generating output tuples based on grouping variables and computed aggregates. The Gigascope approach to template-based code generation combines the performance of generated query system with the ease of extensibility and modification to existing operators.

### **3.5 Summary**

In this chapter, we presented the design of Gigascope – a high-performance database for network applications used throughout the dissertation as a vehicle for implementation and evaluation for all proposed techniques. We presented an overview the stream query language used by Gigascope and described the semantics of the basic types of streaming queries. We also surveyed the two-level architecture for early data reduction and describe several of the streaming query optimizations that Gigascope uses for efficient processing of high-rate streams.

## Chapter 4

### 4. Unblocking Unpredictable Streams

A Data Stream Management System should be able to simultaneously monitor and correlate large numbers of diverse data streams. Many real-world data streams are inherently bursty and unpredictable, in particular IP traffic streams processed by network monitoring applications. Even small scale centralized stream processors correlating a small number of streams can easily block on bursty or stalled input and exceed the available memory. The ability to robustly handle large numbers of unpredictable streams is even more critical for clustered/distributed systems that are expected to deal with order of magnitude differences in data rates of different input streams.

In this chapter, we introduce a punctuation-carrying heartbeat mechanism designed to prevent the DSMS from blocking when some of the monitored streams temporarily stall or slow down. We show how heartbeats can be regularly generated by low-level nodes in query execution plans and propagated upward unblocking all streaming operators on its way. Additionally, we show that the heartbeat mechanism can be used for other applications in distributed settings such as detecting node failures, performance monitoring, and query optimization.

We implemented the heartbeats in the context of Gigascope data, however, proposed techniques should easily apply to any DSMS that uses two-level architecture for early data reduction and relies on timestamp mechanism to unblock streaming operators. We demonstrate the effectiveness



of our approach by running experiments with multi-source streaming queries running over high-rate network data streams. Our experimental evaluation shows that mechanism significantly decreases the query memory utilization while incurring low computational overhead.

## 4.1 Introduction

A Data Stream Management System (DSMS) evaluates queries over potentially infinite streams of tuples. In order for a DSMS to produce useful output, it must be able to *unblock* operators such as aggregation, join, and union. In general, this unblocking is done by limiting the scope of output tuples that an input tuple can affect. One unblocking mechanism is to define queries over windows of the input stream; this technique is particularly applicable to continuous query systems for monitoring applications [6][13][21][23]. Another technique for localizing input tuple scope is to a timestamp mechanism; this technique is particularly applicable to data reduction applications [39][112].

Gigaspice's technique for localizing input tuple scope is to require that some fields of the input data streams be identified as behaving like timestamps, e.g., be monotone increasing. The locality of input tuples is determined by analyzing how the query references the timestamp fields. For example, an aggregation query must have a timestamp field as one of its group-by variables, and a join query must relate timestamp fields of both inputs. We have found the timestamp analysis mechanism to be quite effective for unblocking operators as long as all input streams make progress. However, if one of the input streams stalls, operators such as join or merge which combine two streams can stall, possibly leading to a system failure.

**Example.** Let's consider a concrete example. Gigaspice is designed for network monitoring applications. Many of the sites that we monitor have multiple high-speed links (e.g., Gigabit Ethernet) to the Internet. To ensure high reliability, one or more of these links is a *backup* link.

If a primary link fails, traffic is automatically diverted to a backup link. In order to monitor traffic at these installations, we need to monitor all links simultaneously. At a minimum, we need to monitor the merged traffic of a link and its backup. Since the primary link has gigabit traffic and the backup link has almost no traffic, the merge operator will quickly overflow (i.e., even after optimizations which minimize traffic flow to the merge operator), either running out of buffer space or dropping packets.

The problem we face is that while the presence of tuples carries temporal information, their *absence* does not. A technique that has been proposed in the literature is to use *heartbeats* or *punctuation* [109][115] to unblock operators. However, detailed implementation discussions are lacking.

In this chapter, we present our implementation of punctuation-carrying heartbeats in the Gigascope DSMS. We first implemented these heartbeats to collect load and liveness information about the operators. Our heartbeats originate at source query operators and propagate throughout the query DAG. We show how timestamp punctuations can be generated at the source query nodes and inferred at every other operator in the query DAG. Finally we show how the punctuated heartbeats can unblock otherwise blocked operators.

In this chapter, our focus is on unblocking multi-stream operators such as joins and merges (previous heartbeat work [109] focuses on providing guarantees that tuples arriving to query processor are properly ordered). We demonstrate the need and effectiveness of our punctuation-carrying heartbeats by running experiments with join and merge queries over very high-speed data streams. We find that our punctuation-carrying heartbeat significantly reduces the memory load for join and merge operators with a CPU cost too small to be measured.

The rest of the chapter is organized as follows. We discuss related work in Section 4.2. In Section 4.3, we show to integrate heartbeats into two-level Gigascope DSMS architecture. In Section 4.4,

we describe how heartbeat generation is implemented in Gigascope’s low-level queries. Section 4.5 discusses the heartbeat generation and propagation in higher-level queries. We demonstrate how heartbeat mechanism goes beyond its use for operator unblocking by giving its other applications in Section 4.6. In Section 4.7, we present our experimental study with Gigascope on live network traffic. Conclusions are in Section 4.8.

## 4.2 Previous Work

A heartbeat is a very widely used mechanism in distributed systems to achieve fault tolerance. The most common implementations have remote nodes send periodic heartbeat messages to inform other nodes that they are alive. If no heartbeat is received for certain amount of time, the node is declared dead. Recent research projects in distributed data stream management systems (Aurora+, Medusa, and Borealis) [1][30] also use heartbeats to detect remote node failures.

Stream punctuation [83][84][115] has been proposed as a technique to unblock operators by embedding special marks in the stream that indicate the end of a subset of the data. This mechanism is very generic and allows punctuation to carry arbitrary information that might be helpful to operators (e.g. all future tuples will have the values of the attribute in certain range). However, this work on punctuated streams does not describe how data sources are going to generate the punctuations. It is also not clear how to integrate such a mechanism into high-performance streaming database that needs to process data at line speeds.

The heartbeat mechanism described in [109] is designed to enforce a guarantee that all the tuples are ordered by a timestamp before they are sent to the query processor. This approach assumes that the DSMS includes a special *input manager* that buffers tuples arriving from multiple streams to provide such a guarantee. They focus on eliminating out-of-orderness in input streams, which is different from our problem of unblocking multi-stream operators. Gigascope’s

punctuation-carrying heartbeats that we present here are not restricted to a single system or application time, and are designed for large number of protocol and application-level timestamps and sequence numbers characteristic of network streams.

### 4.3 Using Heartbeats to Unblock Streaming Operators

The heartbeat mechanism was initially designed to collect the runtime statistics about operator loads and to detect node failures when the system is used in distributed settings. Gigascope heartbeats are special messages that are regularly produced by low-level query operators and propagated throughout the query DAG. Since heartbeat messages are propagated using the regular tuple routing mechanism, they incur the same queuing delays as regular tuples and can give a good indication of the system bottlenecks and overloaded nodes. Collecting traces of heartbeats propagating through the query execution DAG gave us a valuable tool in system performance monitoring. Another benefit of having regular flow of heartbeat messages through active operators is the ease of detecting failed nodes.

We approached the problem of unblocking streaming operators that take multiple inputs by implementing a stream punctuation mechanism that injects special temporal update tuples into operator output streams. The purpose of temporal update tuples is to inform the receiving operators about the end of a subset of a data (typically the end of the time window or epoch on which streaming operators such as aggregations, stream merge and joins operate). Our first implementation of stream punctuations used on-demand generation of temporal updates tuples. In this approach blocked operators explicitly request the temporal update tuple from their input nodes. We found that on-demand generation of stream punctuations led to unnecessary complexity in both Gigascope runtime and code generation system. After taking a closer look at Gigascope heartbeat mechanism we realized that heartbeats regularly propagating through query

execution DAG provide a perfect vehicle for carrying the temporal update tuples. The constant flow of punctuation-carrying heartbeats ensures that stalled merge and join operators will be unblocked in timely manner.

Temporal update tuples generated by streaming operator have schema identical to that of regular tuples, but they also have a few important distinctions. All the tuple attributes that are marked as temporal in operator's output schema are initialized with values that are guaranteed not to violate the ordering properties. For example, if attribute *Timebucket* is marked as *temporal increasing*, and operator receives a temporal update tuple with value *Timebucket* =  $t$ , all future tuple are guaranteed to have *Timebucket*  $\geq t$ . All the non-temporal attributes in stream schema are left uninitialized and are ignored by receiving operators. One simple and very conservative scheme for generating such temporal tuples is to always emit the previously produced tuple (cast as a temporal update tuple). However, such mechanism would be useless, as heartbeats will not provide any new information to streaming operators. Our goal is to build a system that will be very aggressive in generating values of temporal attributes and try to set them to highest possible value it can safely guarantee (or lowest value in case of *temporal decreasing* attributes). We will describe our algorithms for generating the values of temporal attributes in sections 4.4 and 4.5.

#### 4.4 Heartbeat Generation at LFTA Level

Heartbeat generation in Gigascope is initiated by low-level operators (LFTAs) regularly injecting the heartbeat messages carrying temporal update tuples into their output streams. In this section, we give a brief overview of low-level streaming operators used in Gigascope and describe the algorithms generation of temporal update tuples.

#### 4.4.1 Low-level Streaming Operators in Gigascope

Gigascope's low-level streaming operators (LFTAs) read data directly from *source data streams* (e.g., packets sniffed from a network interface). Their main purpose is to maximally reduce the amount of data in a stream using filtering, projection and aggregation before it is passed to higher-level execution nodes (requiring a memory copy). Input tuples, typically in the form of networks packets, are read directly from NIC's ring buffer. To avoid overflowing this high input rate buffer, it is essential that the processing of input tuples be as fast as possible. The only two types of streaming operators used in LFTA nodes are selection and aggregation.

The normal mode of operations of the LFTA node in Gigascope is to block, waiting for new tuples to be posted to a NIC's ring buffer. Once a tuple is posted in the buffer, the runtime system invokes the operator's *Accept\_Tuple()* function to process it. In order to make sure that operators regularly produce the heartbeats even in the absence of incoming packets, the runtime system periodically interrupts the LFTA's wait and requests for them to emit a punctuation-carrying heartbeat.

Every low-level operator maintains the necessary state required to correctly generate temporal update tuples. This state always includes the last seen values of all the temporal attributes referenced in operator's select clause, in addition to other operator-specific states. These values are used by the operator to infer the values of the temporal attributes for temporal update tuples. The example of such an inference is given in the following aggregation query:

```
SELECT tb, srcIP, count(*) from TCP
GROUP BY time/60 as tb, srcIP
```

If according to LFTA's internal state the last seen value of 'time' attribute was X, it will use the inference rules to generate a 'tb' value for temporal update tuple to be equal to X/60.

#### 4.4.2 Effects of Prefilters

Preliminary filtering is a form of multiple query optimization employed by Gigascope to avoid the cost of invoking operators on tuples which are certain to fail selection predicates. Even though this technique frequently leads to significant performance gains, it presents a problem for our heartbeat generation system. Consider a scenario in which an arriving tuple has a value of the temporal attributes that would advance the time window used by higher-level aggregation, merge or join operator. If the tuple failed the prefilter test, it will never be delivered to LFTA operators and they would not be aware that the time window in fact advanced.

In order to avoid losing valuable temporal information, we augment the prefilter to save the values of all the temporary attributes used by the queries that share the prefilter. These saved values are made available to all LFTA nodes for use in heartbeat generation.

#### 4.4.3 Heartbeats in Selection LFTAs

Low-level selection operators in Gigascope perform selection, projection, and transformation on packets arriving from a source data stream. The normal tuple processing flow for this operator is to unpack the values of the fields referenced in the query predicate and check if the predicate is satisfied. If so, the output tuple is generated according to the projection list in query select clause. There are a small number of changes that need to be made to the normal tuple processing flow in order to enable heartbeat generation:

- 1) Modify operator's *Accept\_Tuple()* function to save the values of all temporal attributes referenced in query Select clause.

2) Whenever operator receives a regular request to generate a temporal update tuple, use the maximum of the saved value of temporal attributes and a value saved by prefilter to infer the value of the temporal update tuple.

It is important to note that the values of the temporal attributes are saved in *Accept\_Tuple()* regardless of whether tuple satisfies the operator's predicate or not. The generation of attribute values for temporal update tuples is done using the value inference scheme outlined earlier in Section 4.1.

#### **4.4.4 Heartbeats in Aggregation LFTAs**

Gigascop's low-level aggregation queries implement group by and aggregation functionality using small direct-mapped hash table. Whenever a collision in a hash-table occurs, the ejected tuple is sent to output stream; as a result, the output stream can have multiple tuples for the same group. To ensure that the aggregation query always generates the correct output, a low-level query is paired with high-level aggregation node that completes the aggregation of partial results produced by LFTA.

Whenever the incoming tuple advances the epoch, the aggregation operator closes all the aggregates maintained in the hash table and flushes them to the output stream. If the number of groups accumulated during an epoch is very large, the flush puts a large load on a stream manager and can potentially lead to overflow of system buffers. To avoid this effect Gigascop uses a traffic-shaping technique known as *slow flush*. Instead of putting tuples directly into output stream, it gradually emits them as new tuples arrive from the input. This property has a significant effect on generating the values of temporal attributes in heartbeat tuples. Using the largest observed values of temporal attributes may violate the stream ordering properties because some tuples with smaller attribute values remain unflushed.



Similar to selection operator, aggregation nodes save the last seen values of temporal attributes in the input stream and use the value inference to generate temporal update tuples. In addition to the state common to all operators, it also maintains the value of temporal attributes of the last tuple it flushed to the output stream. Whenever a request to produce a heartbeat is received, the following formula is used:

if we have unflushed tuples :

use the value of last flushed tuple

else:

use maximum of the saved value of temporal  
attributes and the value saved by the prefilter

This method guarantees that heartbeat tuples injected into operator's output stream do not violate temporal attribute ordering properties.

#### **4.4.5 Inferring Values of Temporal Attributes Based on System Time**

The heartbeat generation scheme that Gigascope uses in LFTAs works well when each of the monitored links has some amount of traffic. However, the situation becomes more complicated when one of the monitored network cards does not observe any tuple in a long time. In the absence of incoming tuples, the streaming operators will not be able to advance the values of temporal attributes and will conservatively produce heartbeats based on previously observed input. Since most of the temporal attributes in typical network queries are time-based and can be easily correlated with system clock, naturally, Gigascope has the ability to advance the values of temporal attributes based on a system clock.

When advancing temporal attributes using this method, one must however be careful about the skew between the system clock and the timestamps assigned by network interfaces. One source of

the skew is the buffering in packet capture library (pcap) library that can keep already timestamped tuples from being delivered to LFTA nodes. In the presence of low-rate stream, buffering can lead to scenarios where the timestamps of the tuples received by LFTA fall significantly behind a system clock. As part of setting up Gigascope, the administrator needs to specify the maximum skew between host system clock and each of the monitored network interfaces. The heartbeat generation system uses the skew information to automatically advance the time-based temporal attributes. Future tuples that violate the skew specification are discarded by receiving LFTAs.

## **4.5 Heartbeat Propagation at HFTA Level**

A streaming operator in high-level query nodes (HFTA) emits temporal update tuples whenever it receives a heartbeat from one of its source stream. In this section, we give an overview of high-level query nodes and the streaming operators they use as well as algorithms for heartbeat generation and propagation by different streaming operators.

### **4.5.1. High-level Query Nodes in Gigascope**

An important characteristic of the Gigascope architecture is a two-level approach to query execution. Low-level subqueries (LFTAs) executing directly within Gigascope runtime are responsible for early data reduction, while more complicated processing involving expensive predicates or complex operators is performed in high-level query nodes (HFTAs). Even though from application perspective LFTAs and HFTAs are indistinguishable, there are significant differences in their capabilities. High-level nodes are not restricted to running single streaming operator (the way LFTAs are) and can implement arbitrarily complex query execution plans. Currently Gigascope supports selection, multiple types of aggregation, stream merge, and inner

and outer join operators. HFTAs can receive data from multiple different streams produced by LFTAs and other running HFTAs.

The normal mode of execution of an HFTA node in Gigascope is to block, waiting for new tuples to arrive from one of its input streams. After determining which operators in the query execution tree are subscribed to that input stream, the runtime system invokes operator's *Accept\_Tuple()* function to process the incoming tuples. If the processing of the tuple forces the operator to produce some output tuples, they are routed to the appropriate parent operator in query execution plan. In addition to the regular tuples arriving from one of its input stream, an HFTA regularly receives temporal update tuples produced by LFTAs or other HFTAs. We augmented the implementation of all streaming operators to correctly interpret temporal update tuples and use them to unblock themselves. We will describe the changes that we made in subsequent sections dedicated to different types of operators.

Similar to low-level operators described earlier, high-level operators residing in an HFTA maintain the necessary state required to generate temporal update tuples. Normally the state includes the last seen values of all relevant temporal attributes for each of the operator's input streams. High-level operators use these values in addition to operator-specific state to infer the values of the attributes of temporal update tuples.

#### **4.5.2. Heartbeats in Selection Operator**

Heartbeat generation in selection operator is largely identical to the scheme used selection LFTAs discussed earlier. The difference lies in the fact that operator can receive temporal update tuples in addition to regular data tuples. Whenever a temporal update tuples is received, operator updates the saved values of all temporal attributes referenced in query Select clause and generates a new temporal update tuple based on a saved state. The rest of the normal tuple processing is

bypassed. The generation of attribute values for temporal update tuples is done using the value inference scheme outlined earlier in Section 4.1.

### **4.5.3. Heartbeats in Aggregation and Sampling**

The high-level aggregation operator in Gigascope is a non-blocking operator that aggregates the data within a time window (epoch) defined by values of temporal groupby attributes. In contrast with low-level aggregation queries that use direct-mapped hash-table and can emit multiple partial aggregates for the same group, high-level aggregates are required to keep all the groups and corresponding aggregates till the end of epoch before flushing them to output stream. To deal with the increased danger of overflowing system buffers by flushing huge amounts of data at the end of the epoch, aggregation operators rely on the slow flush mechanism that we described earlier.

We made a small number of modifications to Gigascope aggregation operator to enable the generation of the punctuation-carrying heartbeats. These modifications mostly mimic the changes required to implement heartbeats in low-level aggregation queries. The operator maintains the last seen values of all relevant temporal attributes, updating them whenever a new tuple (regular or temporal) arrives. In addition to this state, the operator also maintains the values of temporal attributes of the last flushed tuple (for correctness in the presence of slow flush). These values are combined to infer the attributes of temporal update tuple using the formula from Section 4.4.

In addition to traditional stream aggregation operators, Gigascope also supports more complex aggregation operators – such as the stream sampling operator [69]. However, in all respects related to processing of temporal tuples and heartbeat generation, these operators behave identically to plain aggregation operator and share all the heartbeat-related code.

#### 4.5.4. Heartbeats in Stream Merge Operator

A merge operator in Gigascope performs a union of two streams  $R$  and  $S$  in a way that preserves the ordering properties of the temporal attributes.  $R$  and  $S$  must have the same schema, and both must have a temporal field, say  $t$ , on which to merge. Note that  $t$  is the only attribute that preserves the temporal properties in the merge output schema. The operator maintains the smallest values  $R_{\text{MIN}}$  and  $S_{\text{MIN}}$  of the timestamp observed on each of the input streams. If tuples on one stream, say  $R$ , have a values of  $t$  larger then  $S_{\text{MIN}}$ , then the tuples from  $R$  are buffered until the  $S$  tuples catch up. Note that the values of  $R_{\text{MIN}}$  and  $S_{\text{MIN}}$  are updated whenever a new tuple (regular or temporal) arrives from a stream that has no buffered tuples. Whenever the operator is asked to generate the temporal update tuple, it can trivially generate it by setting the value of  $t$  to  $\text{MIN}(R_{\text{MIN}}, S_{\text{MIN}})$ .

#### 4.5.5. Heartbeats in Join Operator

GSQL queries that join two data streams  $R$  and  $S$  must contain a predicate that relates a timestamp from  $R$  to one in  $S$  (e.g.  $R.\text{tr} = 2 * S.\text{ts}$ ). This requirement is critical for implementing the join using bounded amount of memory without relying on sliding windows. Gigascope implementation of join operator supports inner as well as left, right and full outer equi-joins. Similar to merge operator, join maintains a minimum timestamps  $R_{\text{MIN}}$  and  $S_{\text{MIN}}$  and buffers input streams to ensure they match up on the timestamp predicate. Note that timestamp in GSQL may include a number of temporal attributes, so  $R_{\text{MIN}}$  and  $S_{\text{MIN}}$  could be a composite structure storing minimum values of all attributes that constitute a timestamp. Again the value of the attributes in temporal updates tuples are generated using the  $\text{MIN}(R_{\text{MIN}}, S_{\text{MIN}})$  formula.

## 4.6 Other Heartbeat Applications

The initial goal in implementing heartbeat mechanism for Gigascope was to collect the statistics about the load on query nodes when system is used in distributed settings. Once the mechanism was implemented we discovered that heartbeat infrastructure can be used for variety of other tasks. In addition to carrying stream punctuations (which is the main focus of this chapter) and statistics collection, we are currently applying heartbeats to fault tolerance, query performance analysis, distributed query optimization. In this section we give brief overview of different applications that rely on heartbeat mechanism.

### 4.6.1. Fault Tolerance

A heartbeat is a widely used mechanism in distributed systems to detect node failures. Traditional implementations require every remote node to periodically send heartbeat messages to a resource manager to indicate that the node is still alive. In our Gigascope implementation, we use a slightly different scheme in which heartbeats are periodically generated by low-level queries and propagated upward through the query execution DAG. A constant flow of heartbeat tuples through the system provides an easy way for a running query to identify that a node running one of its subqueries no longer responding. If a subquery does not produce a heartbeat for some specified amount of time, it is declared to be failed and a recovery procedure is initiated. Usually the recovery involves moving an instance of the failed query to another machine.

### 4.6.2. System Performance Analysis

Gigascope relies on the regular tuple routing mechanism to propagate the heartbeat messages from the low-level queries up to top-level nodes that applications subscribe to. As a result, heartbeats are subject to the same queuing delays that regular tuples incur and can be used to

identify backlogged nodes and system bottlenecks. Every heartbeat message emitted by an LFTA is timestamped and contains an identifier of the producing query. In addition to this information, every heartbeat is assigned a special trace identifier (*trace\_id*). As the messages propagate upwards to higher level nodes, they attach their own identifiers along with a timestamps corresponding to the time they received a heartbeat. When a heartbeat message finally reaches a top-level query node, it has a full trace of all the operators it visited on its way along with the delays it incurred in each of the operator's queues. Systems administrators and developers can use these heartbeat traces to identify system performance problems that are otherwise very difficult to detect.

#### **4.6.3. Distributed Query Optimization**

Streaming query optimizers critically depend on accurate statistics describing input stream to drive many optimization decisions, such as order of join evaluation, operator placement and stream partitioning. Heartbeat mechanism is a perfect vehicle to collect all the runtime statistics required by distributed query optimizer. In addition to detailed traces described in previous section, our implementation of heartbeats carries other operator statistics such as predicate selectivities, data arrival rates, query memory utilization and tuple processing costs.

### **4.7 Limitations of the Heartbeat Mechanism**

The heartbeat mechanism proposed in this dissertation is designed for throughput-oriented streaming applications, such as network monitoring, in which the latency of the propagation of the individual tuples through the query graph is not critical. For such applications it is sufficient to generate periodic heartbeats with the frequency as low as the size of the smallest time window used by the application. However, for other types of streaming application, such as battlefield

monitoring or fire sensor monitoring, the latency of the event propagation through the system is critical. To provide the necessary latency guarantees, the heartbeat generation system proposed in this dissertation would be forced to generate heartbeats almost as frequently as rate of the input streams, which will incur significant overhead. Recent research in heartbeat mechanisms for such applications [157] addresses the problem by using on-demand heartbeat generation with special optimizations enabling fast heartbeat forwarding through the query graph.

## 4.8 Performance Evaluation

In this section, we present our experiments with the Gigascope heartbeat mechanism. These experiments were conducted on a live network feed from a data center tap. All our queries monitor the set of 3 network interfaces, two high-speed DAG4.3GE Gigabit Ethernet interfaces (*main1* and *main2*) which see the main bulk of the traffic and one control 100Mbit interface (*control*). Both Gigabit interfaces receive approximately 100,000 packets per second (about 400 Mbits/sec). Our primary focus is to be able to unblock streaming operators that combine the streams from both high-rate main links and low-rate backup links. Since the control interface has very small amount of traffic, its behaviour is a representative of the behaviour of backup interfaces. All experiments were conducted on dual processor 2.8 GHz P4 server with 4 GB of RAM running FreeBSD 4.10.

### 4.8.1 Unblocking Stream Merge Using Heartbeats

We evaluated the effect that punctuation-carrying heartbeats have on memory usage of running queries that use the stream merge operator. For this experiment we used the following GSQL query:

```
SELECT tb, protocol, srcIP, destIP,
```



```

srcPort, destPort, count(*)

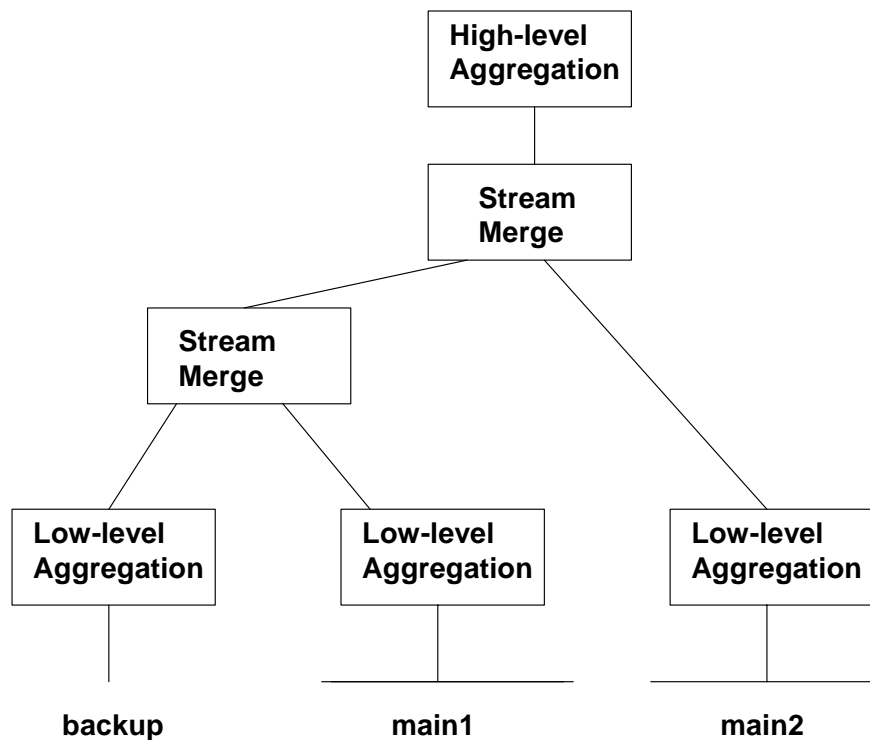
FROM DataProtocol

GROUP BYtime/10 as tb, protocol, srcIP, destIP,

srcPort, destPort

```

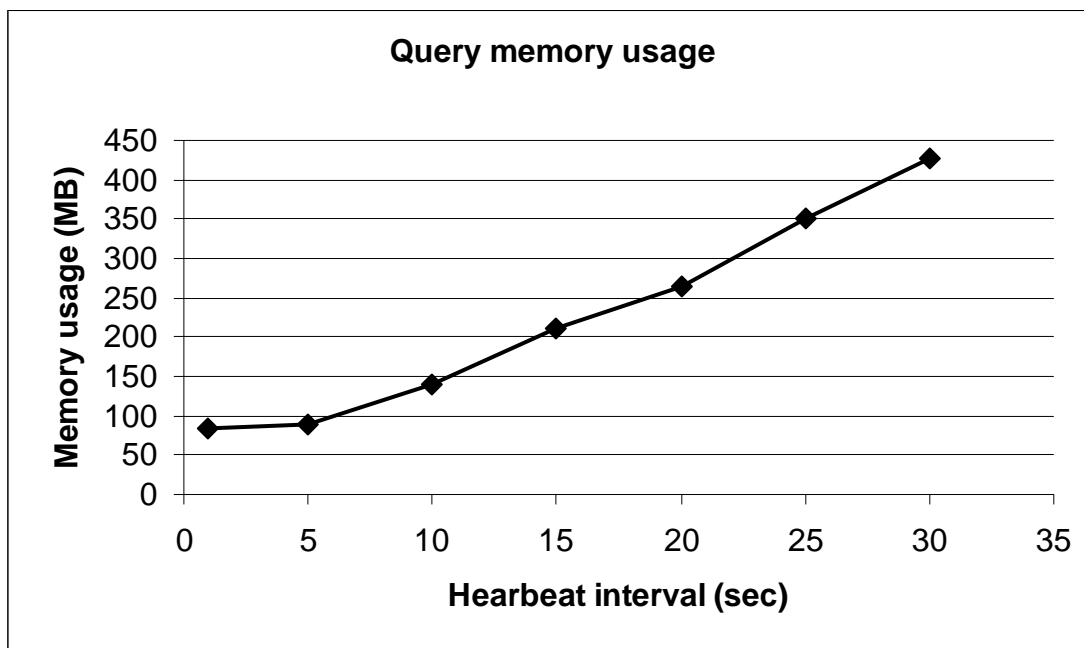
The query computes the number of packets observed in different flows in 10 second time buckets. Since the query is executed on a machine with 3 network interfaces, the Gigascope query planner automatically inserts stream merge operators into query plans to combine the stream from different interfaces. The resulting query plan is shown in Figure 4-1.



**Figure 4-1: Merge query execution plan**

Data is partially aggregated using low-level aggregation queries and then combined using stream merge operators before finally being aggregated by high-level aggregation query. When the *control* link has no traffic, both stream merge operators must buffer a large number of tuples

received from high-rate main links. In this experiment, we varied the interval with which heartbeats are generated and recorded maximum memory that a running query consumes. We varied a heartbeat interval from 1 sec (the default value used in Gigascope) to 30 seconds in 5 second increments. The results of the experiments are presented in Figure 4-2.



**Figure 4-2: Memory usage of stream merge query**

The result of the experiment illustrate that heartbeats successfully unblock the stream merge operators. As the heartbeat interval increases, the amount of state that the merge operators need to maintain before they can advance the epoch is growing linearly. Eventually memory footprint of the query would exceed the available RAM and will cause a system crash.

It is important to notice that increasing the heartbeat intervals not only leads to increased memory footprint, but also significantly increases the amount of data that needs to be flushed by the operator once the epoch advances. Since our stream merge implementation does not currently use traffic-shaping techniques (such as slow flush), the system can cause a query failure even before the memory consumption exceeds the available RAM. In the experiment in which we used 30

second heartbeat intervals, merge operators were instantly flushing 420MB worth of tuples which exceeded the capabilities of tuple transfer mechanism and led to query failure.

#### 4.8.2. Unblocking join operators using heartbeats

In this experiment, we observed how effectively heartbeats unblock join queries and reduce overall query memory requirements. We used the following GSQL query:

Query **flow1**:

```
SELECT tb, protocol, srcIP, destIP,
       srcPort, destPort, count(*) as cnt
FROM [main0_and_control].DataProtocol
GROUP BY time/10 as tb, protocol, srcIP, destIP,
       srcPort, destPort
```

Query **flow2**:

```
SELECT tb, protocol, srcIP, destIP,
       srcPort, destPort, count(*) as cnt
FROM main1.DataProtocol
GROUP BY time/10 as tb, protocol, srcIP, destIP,
       srcPort, destPort
```

Query **full\_flow**:

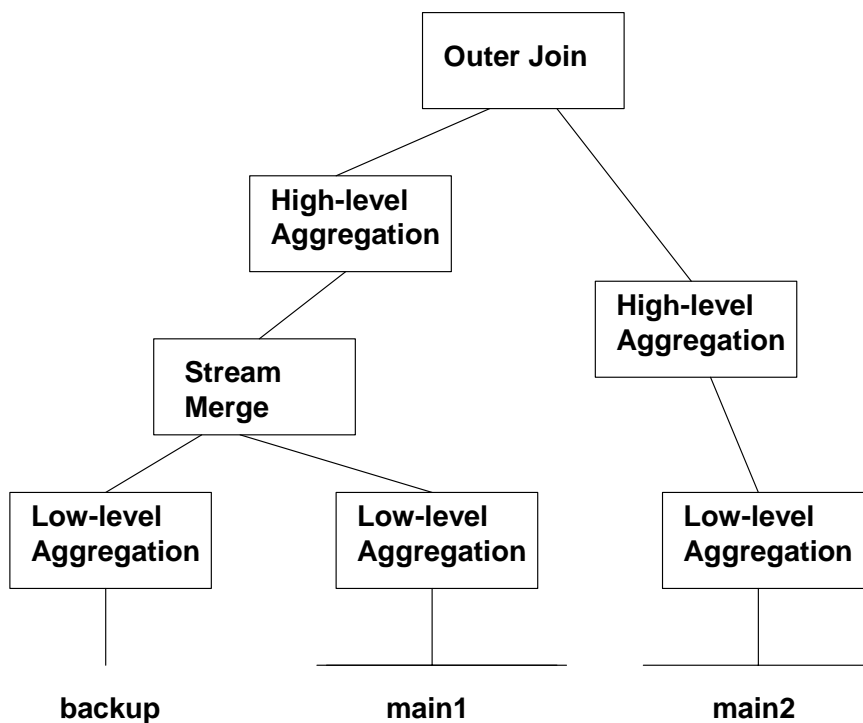
```
SELECT flow1.tb, flow1.protocol, flow1.srcIP,
       flow1.destIP, flow1.srcPort, flow1.destPort,
       flow1.cnt, flow2.cnt
OUTER_JOIN FROM flow1, flow2
WHERE flow1.srcIP=flow2.srcIP and
```

```

flow1.destIP=flow2.destIP and
flow1.srcPort=flow2.srcPort and
flow1.destPort=flow2.destPort and
flow1.protocol=flow2.protocol and
flow1.tb = flow2.tb

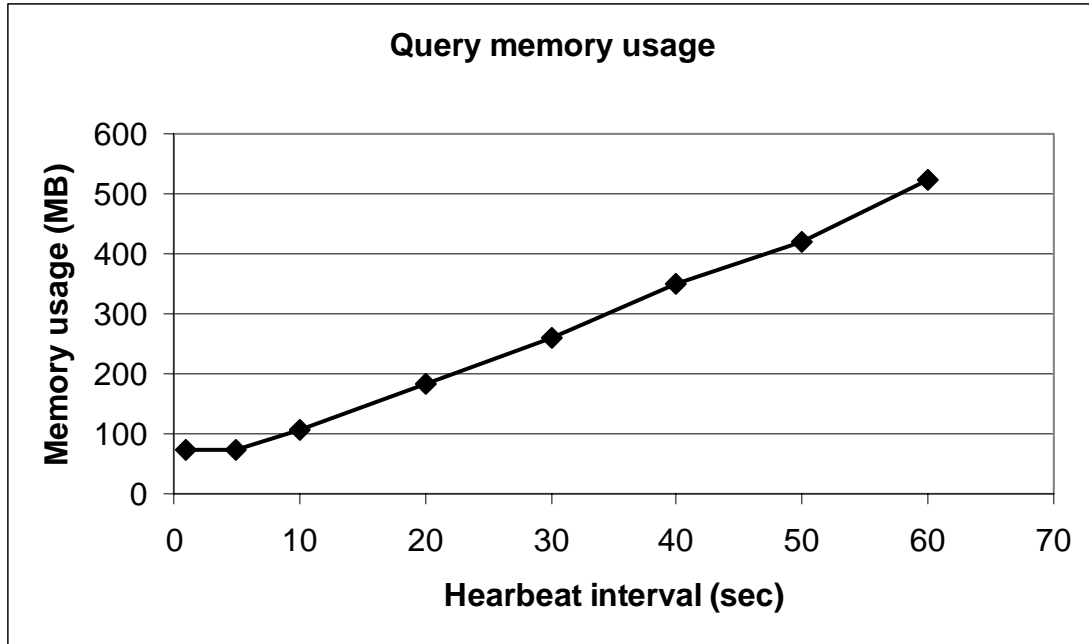
```

Two subqueries (flow1 and flow2) compute the flows aggregated in 10 second timebuckets and observed on interfaces *main1+control* and *main2* respectively. The query results are combined using full outer join to generate a final output. The resulting query plan is shown in Figure 4-3.



**Figure 4-3: Join query execution plan**

In this experiment we varied an interval with which heartbeats are generated from 1 sec to 60 seconds in 10 second increments. The results of the experiments are presented in Figure 4-4.



**Figure 4-4: Memory usage of join query**

The results of the experiments show a similar pattern to the query that just uses stream merge operators. Again punctuation-carrying heartbeats are able to unblock both merge and join operators. The state maintained by query merge, aggregation and join operators linearly grows with the heartbeat interval and reaches 520MB for 60 second interval. At this point, our outer join implementation, which does not use traffic-shaping, instantaneously dumps 520MB of data to receiving application and causes the overflow of system buffers. When we set a heartbeat interval to default value of 1 sec, we not only avoid accumulating large state of blocking operators, but also decreasing the burstiness of their output.

#### **4.8.3. CPU overhead of heartbeat generation**

We measured the CPU overhead that Gigascope's implementation of heartbeats incurs on running streaming queries. We measured the average CPU load of a merge query used in Section 4.7.1 running on two high-rate interfaces (*main1* and *main2*). We compared the CPU load of a system with 1 second heartbeat interval to an identical system which has heartbeats completely disabled.

Since both of the monitored links have moderately high load, the merge operators are naturally unblocked even with heartbeat disabled. Therefore both systems behave identically and allow us to measure overhead of heartbeat generation without significantly changing runtime behavior of the operators. We observed that a version of Gigascope with heartbeats disabled has average CPU load of 37.3%, while enabling heartbeat generation every second raises the load to 37.5%. This difference is so small that it can be explained by variations in traffic load. Hence we conclude that the overhead of the heartbeat mechanism is immeasurably small.

## 4.9 Summary

In this chapter, we introduced a mechanism for punctuation-carrying heartbeat generation that allows a Data Stream Management System to monitor a large number of diverse data streams without blocking, even when some of the streams temporarily stall. We showed how heartbeats can be regularly generated by low-level nodes in query execution plans and propagated upwards. By attaching temporal update tuples as punctuation, the heartbeats unblock any blocked operators. Our heartbeat mechanism can be also be used for other applications in distributed settings, such as detecting node failures, performance monitoring, and query optimization. The mechanism was incorporated into Gigascope, a high-performance streaming database for network monitoring that is operationally used within AT&T's IP backbone. A performance evaluation using live data feeds showed that our system is capable of working at multiple Gigabit line speeds in industrial deployment and can significantly decrease the query memory utilization.

## Chapter 5

### 5. Query-Aware Data Stream Sampling

The robust handling of the overload conditions is an important requirement for Data Stream Management Systems monitoring high-rate streams. It is a widely known phenomena that the load on streaming systems can increase by an order of magnitude (e.g. during network attacks for network monitoring applications). Therefore, load shedding is necessary to preserve the stability of the system, gracefully degrade its performance and extract answers.

Existing methods for load shedding in a general-purpose data stream query system use random sampling of tuples, essentially independent of the query. While this technique is acceptable for some queries, the results may be meaningless or even incorrect for other queries. In principle, a number of different query-dependent sampling methods exist, but they work only for particular queries. In this chapter, we show how to perform query-aware sampling, termed semantic sampling, which works in general. We present methods for analyzing any given query, choosing sampling methods judiciously, and guaranteeing the correct semantic execution of the query. Our experimental evaluation on a high-speed data stream demonstrates with different query sets that our method guarantees semantically correct and accurate results while being efficient in decreasing the load significantly.

## 5.1 Introduction

High-rate data streams from many application domains (network monitoring, financial monitoring, scientific measurements) are inherently busy. For example, there are *flash events* [73] on the network when legitimate traffic spikes sharply. In [69], the authors report that during a Distributed Denial of Service (DDoS) attack, the load on a link can increase from 100,000 packets/sec to 500,000 packets/sec. Trading volumes bursts on individual securities are common, and even occur in entire markets during financial panics (Two examples from the New York Stock Exchange are 10/19/1987 and 10/28/1997 [95]). Even if the DSMS is configured to handle a high volume data stream during normal circumstances, during a burst period the DSMS might exhaust available resources such as CPU cycles, memory, and link capacities.

Perversely, it is precisely during such highly loaded instants such as a DDoS attack that the DSMS is most useful and analysts rely on it crucially to identify the attackers and protect the network. Similarly it is precisely during a financial spike or market volatility that analysts rely on a DSMS in order to identify price trends and protect market positions. Therefore, it is critical to build DSMSs that can gracefully perform and provide useful results even in highly loaded instants. That is, DSMSs often have to target instantaneous – not average – data rates.

The widely accepted solution proposed for use by DSMSs to handle overloaded conditions is *load shedding*. In particular, all published systems employ *per-tuple sampling*: uniform random sampling of tuples at different levels of query hierarchy to reduce the load on processing nodes. However, for a large class of queries, uniform random sampling violates the intended query semantics and leads to meaningless or even incorrect output.

**Example.** Consider the query for computing flows from the packet data - summaries of packets between a source and a destination during a period of time. The group-by attributes are the



source and destination IP address, the source and destination port, and the protocol, while the aggregates include the number of packets, the number of bytes transferred, and so on. Our example is one particular aggregate, i.e., the OR of the TCP flags in the packets that comprise the flow. This information is vital for distinguishing between regular flows and attack flows (attack flows do not follow proper TCP protocols).

If one randomly drops packets, one cannot compute the aggregate on the flags properly, and therefore cannot distinguish between valid traffic and attack traffic. Thus a natural stream query written by an analyst to detect attack traffic will result in incorrect output in existing data stream systems that drop tuples randomly. The problem is that the OR aggregate does not have a good approximation on sampled data. Many other aggregates (e.g. MIN and MAX, among others) share this property.

In principle, there is a different sampling strategy that will work in the example above, namely, to drop all packets that belong to randomly chosen flows. For all flows that are not dropped, the query will correctly compute the OR aggregate of the TCP flags and the output will be correct, albeit a subset of the correct output.

We refer to this type of sampling as *per-group sampling*, where the random choice is over the groups (in this case, the group is defined by the attributes that comprise the flow, but in general, it may be any subset of attributes). Per-group sampling is well known in the folklore as being necessary for computing loss-sensitive aggregates such as OR, Min, Max, count of duplicates, and so on. Join queries are also sensitive to random sampling, so variants of group sampling have been proposed for approximate query systems based on samples of large data sets [4][3][50].

Given any particular query, an advanced user can determine the best sampling strategy. However, a DSMS will support dozens to hundreds of simultaneous queries. Requiring a user to analyze and reconcile the sampling strategies for all queries simultaneously is simply not

scalable. Therefore, we need a principled mechanism to determine a suitable sampling strategy for any query set. We call this query-aware method *semantic sampling*. In this Chapter we present semantic sampling methods, show how to implement them effectively and present experimental results validating the approach. More precisely, the contributions we make are:

1. Introducing the concept of query-aware semantic sampling with a suite of tuple and per-group sampling and suitable notion of correctness in presence of sampling for any query.
2. Analyzing query sets to determine a semantics-preserving sampling strategy. For this, we introduce the concept of grouping sets being compatible with given query and show how to reconcile different grouping sets in a query set.
3. Validating our approach experimentally on real network traffic data streams.

By using the methods described in the chapter, we are able to provide high quality results (which very likely reflect the user’s intended semantics) even under adverse operating conditions, and avoid random sampling without guarantees.

The rest of the chapter is organized as follows. We discuss related work in Section 5.2. In Section 5.3 we give an overview of semantic sampling framework and discuss the suite of sampling algorithms that it uses. We describe how to deduce the compatible sampling methods for individual streaming queries in Section 5.4. In Section 5.5 we extend our semantic sampling framework to cover arbitrary query sets. In Section 5.6, we present our experimental study. Conclusions are presented in Section 5.7.

## 5.2 Previous Work

Two main approaches to gracefully handle high-load conditions have been explored in recent literature: load shedding through per-tuple sampling and approximate query processing.

The load shedding mechanism described in [14] relies on random tuple sampling to discard unprocessed tuples and reduce the system load. Sampling operators are placed at various points in query plans based on statistics accumulated during plan execution. The main goal is to minimize the inaccuracy of the results while keeping up with data arrival rates. In order to compensate for the effects of random sampling, aggregate results are appropriately scaled. This approach is suitable for estimating certain aggregates on sliding windows, but is not suitable for a large class of aggregation queries that generate semantically incorrect results when presented with randomly sampled input (e.g., the OR of TCP flags).

The load shedding mechanism used in Aurora/Borealis [113] is also based on random tuple sampling. The system additionally has a mechanism for dropping tuples based on their *utility*. A tuple's utility is computed based on Quality-of-Service (QoS) graphs that need to be specified for every application. Three types of QoS graphs can be used by the system: a *latency graph* specifies the utility of the tuple as a function of time to propagate through query plan, a *loss-tolerance graph* captures the sensitivity of the application to tuple loss, and a *value-graph* shows which attribute values are more important than others. Their mechanism is restricted to queries that do not change the values of the input tuples (such as filter and sort); thus it is not directly applicable to a wide variety of queries including aggregations. Even though one can approximate the benefits of per-group sampling by carefully constructing value-based QoS graphs, the burden of generating the appropriate QoS graph lies with the application writer, which tends to make it unwieldy.

Recent work on *window-aware load shedding* [114] addresses the problem of preserving the correctness of the answer returned by aggregation queries over data streams. Proposed “window drop” operator drops entire windows instead of individual tuples and thus guarantees that output of the query is the subset of the correct output. The approach is restricted to sliding window

aggregation queries and not easily applicable to arbitrary streaming queries containing combination of selection, aggregation merge and join operators.

A large number of algorithms have been suggested in the literature for approximate query processing. They can be divided into two main categories: sampling-based and sketch-based. Query-independent methods of uniform random and fixed-size reservoir sampling work only for certain queries. A variety of query-specific sampling methods have been suggested in literature. Examples include specialized techniques for computing quantiles [50], heavy hitters [89], distinct counts [58], subset-sums [45], set resemblance and rarity [41], etc. Unfortunately, these techniques do not apply beyond the computation of their intended aggregate functions and their interaction in a set of interrelated queries is not well understood. Similarly, there are sketch-based algorithms that are optimized for specific queries, such as particular types of joins or aggregations being computed, and cannot be easily combined in the same query. These limitations make it very hard to use these techniques in automated fashion, i.e. automatically inferring which approximation is safe to use just based on the query text.

### 5.3 Semantic Sampling Overview

The *semantic sampling* framework proposed here automatically infers the sampling methods for every query in any given query set and guarantees the results remain semantically correct. In order to simplify our task, we consider a suite of only two sampling algorithms: uniform random per-tuple sampling and per-group sampling. These are the most common sampling algorithms and together, they suffice for a large class of streaming queries involving aggregations, merges (stream union) and joins. We will soon provide more details.

### 5.3.1 Illustrative example

We illustrate the semantic sampling framework by working through an example query set. The first query (`dup_and_all_count`, denoted  $\gamma_1$ ), a simplified version of TCP performance analysis, computes the number of duplicate packets and the total number of packets in each TCP flow in each one-minute window (making use of the User Defined Aggregate Function or UDAF, `count_dups`, equivalent to an aggregation subquery with a HAVING clause). The higher-level aggregation query (`dup_ratio`, denoted  $\gamma_2$ ) computes the ratio of the duplicate packets to the total number of packet for each time window. The corresponding SQL statements for both queries are shown below:

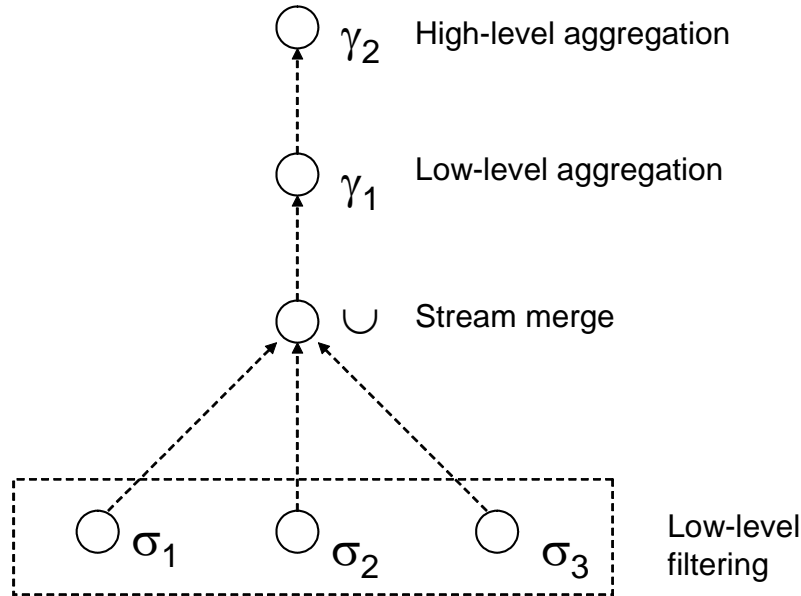
Query **dup\_all\_count**:

```
SELECT tb, count_dups(sequence_number) as dup_cnt,
       count(*) as full_cnt
FROM TCP
GROUP BY time/60 as tb, srcIP, destIP,
       srcPort, destPort;
```

Query **dup\_ratio**:

```
SELECT tb, sum(dup_cnt) / sum(full_cnt)
FROM dup_and_all_count
GROUP BY tb;
```

We would like to run the queries over 3 data streams, so the aggregations need to be performed on their union. A query plan for execution of the queries is shown in Fig 5-1. In order to perform semantic sampling, we need to address the following issues:



**Figure 5-1: Semantic sampling example**

1. *At which level in query hierarchy should sampling be performed?*

The goal is to achieve maximum load reduction without sacrificing the output quality. Intuitively, we should be sampling at the input streams  $\sigma_1$ ,  $\sigma_2$ , and  $\sigma_3$  to drop tuples before investing any time in partially processing them.

2. *What sampling method is needed for each input stream to guarantee that the output of top-level query is semantically correct?*

By analyzing the aggregate functions used in the query `dup_all_count`, we can infer that one of them cannot be approximated using random uniform sampling (in particular, the `count_dup` UDAF.) and therefore per-group sampling needs to be used. Since the `count_dups` UDAF counts the number of duplicate sequence numbers, it will return an accurate answer only if it aggregates over *all* tuples in the group. Instead of uniform random sampling of tuples, it is better to collect all tuples from a uniform random sample of the groups (defined by `srcIP`, `destIP`, `srcPort`, `destPort`), on which the `count_dups` aggregate will be computed exactly. Therefore, the output of query `dup_all_count` is a sample of the exact query output. An analysis of the query `dup_ratio`

reveals that the aggregates it references (SUM) are easily approximated if the input is sampled; therefore per-group sampling of input streams guarantees the semantically correct output for the full query tree, while per-tuple sampling does not have this property.

3. *How do we guarantee the consistency of sampling for all input streams?*

Since the multiple streams are sampled, we need to guarantee consistency of output of the merge. We achieve that by using the same sampling method (for example, per-group sampling using the identical group) for each of the streams.

4. *How do we preserve the metadata describing which sampling methods and sampling rates were used to compute the result?*

Load shedding subsystems are expected to dynamically adjust the sampling levels based on current load conditions. It is therefore critical for application to know exactly what sampling method was used and what were the rates to be able interpret the results and to reason about confidence bounds. In our implementation we preserve this metadata by regularly embedding special punctuations [115] into query output stream describing sampling strategy used to compute the results.

In the rest of the chapter, we will formally define these problems and present the framework for semantic sampling analysis that addresses them.

### 5.3.2 Compatibility of Sampling Methods

In order to reason about whether a particular sampling method guarantees semantically correct results for a given query, we need to have a formal definition.

**Def.** *Sampling method  $M$  is **strongly compatible** with a query  $Q$  if for every time window, the output of the query is a subset of the exact output  $Q$  would produce if no sampling was used.*

While strongly compatible is useful, it does not allow the use of approximations in aggregation queries. Hence:

**Def.** *Sampling method  $M$  is **weakly compatible** with a query  $Q$  if it is strongly compatible, except that fields which are the result of aggregate functions are “good approximations” to the exact values.*

We will use weak compatibility as our test as to whether a sampling method can be used with a query. The measure of “good approximation” depends on the aggregate function. For example, aggregate functions such as sums, counts and quantiles, can be adjusted to give good approximations when used with sampled input. Other aggregate functions, such as MIN, MAX, (bitwise) OR, or count\_dups, cannot provide good approximations on sampled input, and therefore are not even weakly compatible with tuple sampled input.

### 5.3.3 Suite of Sampling Algorithms

We consider two classes of sampling methods in this chapter: per-tuple and per-group sampling.

#### 5.3.3.1 Per-tuple Sampling

Per-tuple sampling is done by uniformly randomly dropping a fraction of tuples from the input stream. This method is independent of the queries that are running in the system and does not need to examine the tuple content to make a decision whether to drop tuple or not. It is inexpensive and works well for selection queries and for aggregation queries that have “good approximations” based on uniform sample of the input. E.g., SUM and COUNT can be approximated by dividing the value of the aggregate by sampling rate – and therefore are weakly compatible. However, per-tuple sampling is not compatible with queries involving more



sophisticated aggregates. Further, per-tuple sampling also leads to poor results for join queries. Acharya et al [3] list two main reasons why uniform sampling is inappropriate for join queries:

- The join of two streams that were uniformly sampled is not a uniform sample of the output of the join. As a result the confidence bounds for the output are significantly degraded.
- The cardinality of the output of k-way join whose input were per-tuple sampled with rate  $\alpha$  ( $\alpha \in (0, 1]$ ) is  $\alpha^k$  of the cardinality of the exact answer. As a consequence, result is less accurate and has low confidence bounds as k increases.

Intuitively we would like to restrict the use of per-tuple sampling to the following scenarios:

1. When we are looking for tuples with very specific content. For example, an application in network monitoring is to capture packets with questionable (worm, attack, P2P) payloads.
2. When we are more interested in the analysis of the group themselves rather than the particular aggregate values for the groups. Examples of such queries are finding ranges of IP addresses, or estimating fraction of one type of traffic to another (e.g. kazaa vs bittorrent).
3. When it is possible to infer the missing values or the aggregates can be easily estimated based on the random sample of the data. Examples of such aggregates are SUM, CNT, quantiles.

We give exact rules for deciding whether per-tuple sampling is compatible with a given query in Section 5.4.2.

### 5.3.3.2 Per-group Sampling

Per-group sampling works by partitioning the tuples of the input stream into disjoint groups, and either sampling all the tuples from the group or dropping the group entirely.

*Def.* Let  $A$  be a set of the tuple attributes  $(attr_1, attr_2, \dots, attr_n)$  and  $H$  be a hash function with large integer domain  $[0, R]$ . Per-group sampling with rate  $\alpha \in (0, 1]$  selects a tuple iff  $H(attr_1, attr_2, \dots, attr_n) \leq \alpha * R$ . The attribute set  $A$  is called a grouping set.

The main property of per-group sampling is exact computation of all aggregate functions for all tuples in the output. Consider our motivational aggregation query:

```
SELECT tb, srcIP, destIP,
       srcPort, destPort, count_dup(sequence_number)
FROM TCP
GROUP BY time/60 as tb, srcIP, destIP, srcPort, destPort
```

Per-group sampling with grouping set  $(srcIP, destIP, srcPort, destPort)$  will guarantee that for every sampled group the value of `count_dup()` aggregate will be computed correctly. In general per-group sampling is preferable for queries that are interested in complex properties of groups which cannot be easily estimated based on a random sample. Examples include `count_dups`, `MIN`, `MAX` and (bitwise) `OR`.

Another class of queries for which per-group sampling is preferable is computing a join between two streams or any type of correlating data by group. Work on join synopses [3] and hierarchical group-based sampling [50] uses the variants of the per-group sampling approach to achieve high accuracy for join results.

## 5.4 Semantic Sampling for Individual Queries

The main component of our semantic sampling framework is the analysis of the query structure to infer compatible sampling methods. We start with the analysis of individual queries consisting of single streaming operators (selection, aggregation, or join), or ensembles consisting of

aggregation or join plus selection and projection. We show how to infer which sampling method is compatible with a given query. Whenever per-group sampling is the strategy of choice, we show how to choose the grouping set to be used for sampling. Individual query analysis allows us to reason about the semantics of the query output and will be used as a building block for analyzing complex query sets in Section 5.5. For simplicity of the discussion we will assume tumbling window semantics for streaming queries (except where otherwise noted). For additional issues, see Section 5.4.1.1.

### 5.4.1 Grouping Set

Recall that per-group sampling hashes the set of tuple attributes called the *grouping set* to a large domain and selects only those tuples that hash into a subset of the domain. Essentially the grouping set defines a partitioning of the domain of tuple values and per-group sampling only selects a random sample of partitions. An important question that needs to be addressed is which attributes should be chosen for a grouping set, such that per-group sampling using the set will result in semantically correct query results. We will formalize this requirement for grouping sets in a definition below:

**Def.** *Grouping set  $GS$  for the stream  $S$  is **compatible** with a query  $Q$  on  $S$  if per-group sampling using  $GS$  is strongly compatible with query  $Q$ .*

Consider the following query  $Q$ :

```
SELECT time/60, srcIP, destIP, max(len)
FROM S
GROUP BY time/60, srcIP, destIP
```

Intuitively, a compatible grouping set partitions the domain of tuple values such that any pair of tuples that have identical values of the grouping attributes will fall in the same partitions. For the query above the trivial example of compatible grouping set is the set of its group-by attributes  $\{\text{time}/60, \text{srcIP}, \text{destIP}\}$ . It is easy to observe that grouping set consisting of any non-empty subset of  $\{\text{time}/60, \text{srcIP}, \text{destIP}\}$  is also compatible with a query. More formally, we can state the requirements for compatible grouping sets in the following way:

**Lemma.** *Let  $G$  be a set of group-by attributes referenced by the query  $Q$  and  $H$  be a hash function used for per-group sampling. Grouping set  $GS$  is compatible with a query  $Q$  iff for any pair of tuples  $tup1$  and  $tup2$   $G(tup1) = G(tup2) \Rightarrow H(GS(tup1)) = H(GS(tup2))$ .*

In addition to using the subsets of the group-by attributes, we can form new compatible grouping sets by using scalar expressions defined on group-by attributes. An example of such compatible grouping set for the query above is  $\{(\text{time}/60)/2, \text{srcIP} \& 0xFFFF0, \text{destIP} \& 0xFF00\}$ . An example of an incompatible grouping set for the query above is  $\{\text{time}, \text{srcIP}, \text{destIP}\}$  (since the fact that  $\text{time1}/60 = \text{time2}/60$  does not imply that  $H(\text{time1}) = H(\text{time2})$ ).

In the following sections, we will list the rules for choosing the grouping sets for two query types that use per-group sampling: aggregations and joins.

#### 5.4.1.1 Dealing with Temporal Attributes

One issue that needs to be considered when selecting a grouping set compatible with a given query is whether to include the temporal attributes. Selecting the temporal attribute in a grouping set will effectively change the hash function used by a sampling method whenever the time epoch changes. This property could be desirable if we want to ensure good coverage of all groups. We can control the periodicity of the sampling change by changing the value of the scalar expression involving the temporal attribute. For example an aggregation query that uses  $\text{time}/60$  to aggregate

in one-minute time buckets can use `time/60/10` as a member of a grouping set to change the hash function every 10 minutes.

For most of the aggregation and join queries, it is impossible to guess whether periodically changing the set of sampled groups is desirable based just on the query text. We make this choice a user option. For sliding window queries that use pane-based evaluation, changing the hashing function in the middle of a window will lead to incorrect query results. Therefore we always remove the temporal attributes from the grouping sets of such queries.

#### 5.4.1.2 Grouping Sets for Aggregation Queries.

In its general form an aggregation query has the following format:

```
SELECT  expr1, ... ,exprn

FROM  STREAM_NAME

WHERE  tup_predicate

GROUP BY  temp_var, gb_var1, ...,gb_varm

HAVING  group_predicate
```

Compatible grouping sets for an aggregation query will have the following form:

$$\{sc\_exp(gb\_var_1), \dots, sc\_exp(gb\_var_n)\}$$

where `sc_exp(x)` is any scalar expression involving `x`. Given that there is infinite number of possible scalar expression, every aggregation query has an infinite number of compatible grouping sets. Furthermore any subset of compatible grouping sets is also compatible.

### 5.4.1.3 Grouping Sets for Join Queries

We will consider a restricted class of join queries, namely two-way equi-join queries that use the semantics of tumbling windows. The general form of such query has the following format:

```
SELECT expr1, ... , exprn

FROM STREAM1 {LEFT|RIGHT|FULL} [OUTER] JOIN STREAM2

WHERE STREAM1.ts = STREAM1.ts and STREAM1.var11 =

    STREAM2.var21 and ... STREAM1.var1k = STREAM2.var2k and

    other_predicates
```

Since a join query has 2 input streams that are independently sampled, we must define two compatible grouping sets – LEFT and RIGHT. The LEFT compatible grouping set will have the following form: {sc\_exp(STREAM1.var<sub>11</sub>), ... ,sc\_exp(STREAM2.var<sub>1k</sub>)} while RIGHT compatible set will be in a form of {sc\_exp(STREAM2.var<sub>21</sub>), ...,sc\_exp(STREAM2.var<sub>2k</sub>)}. As before, any subset of a compatible set is also compatible with additional restriction that LEFT and RIGHT compatible sets must use the same subset.

## 5.4.2 Selecting the Sampling Method

In general, it is difficult to determine the best sampling strategy for a query since the query text does not necessarily reveal the importance of different attributes to the output. Even though a system could require query writer to explicitly specify the sampling method to be used to shed the load in overload situations, it is desirable to automatically infer the compatible strategy to the extent possible just based on the query. Furthermore, automatic selection of the sampling strategy is critical for complex query sets with multiple interconnected queries that have different tolerance to sampling.

In this section, we present the rules that we use for automatically choosing per-tuple or per-group sampling for major classes of streaming operators: selection, aggregation and join. For some operators, both sampling methods will be acceptable in which case we suggest how to break the ties.

#### **5.4.2.1 Sampling in Selection/Projection Queries**

Selection/projection queries perform filter operation on the input stream only allowing tuples that pass the selection predicates. Both per-tuple and per-group sampling methods are strongly compatible with this type of queries according to our definition of compatibility. For standalone queries, per-tuple sampling is clearly preferable due to lower processing overhead – there is no need to read tuple attributes. However, other queries that consume the query’s output stream might affect the choice of compatible sampling. In Section 5 we will present the algorithm that selects the compatible sampling method for the query taking into account all the queries that consume its output stream.

#### **5.4.2.2 Sampling in Aggregation Queries**

The appropriate sampling method for aggregation queries largely depends on the intent of the query writer, which is not always evident just based on a query text. Consider the following aggregation query that computes the statistics for TCP flows:

```
SELECT tb, srcIP, destIP,
       srcPort, destPort, sum(len), count(*)
FROM TCP
GROUP BY time/60 as tb, srcIP, destIP,
       srcPort, destPort
```

If we are interested in getting maximum number of flows (e.g. to use it to compute the ratios of different types of flows), then we will be willing to tolerate the inaccuracy of `sum()` and `cnt()` aggregates. In that scenario uniform random sampling is the most appropriate and cheapest method. If, on other hand, we do not care about capturing all the flows, but are very sensitive to errors in aggregate values, per-group sampling is preferable. Ideally, we would want query writers to explicitly state their intent and tolerance to different sampling methods in the query language. However, it is not practical to expect the users to take the burden of explicitly labeling all the query nodes with acceptable sampling strategies. Explicit labeling is made more complicated by the complex interrelations between the queries in the query sets. Therefore, we would like to automatically infer the safe sampling strategy just based on query text that would guarantee that output remain semantically correct, even if it potentially could be inferior to an explicit sampling specification.

We propose the following rules for selecting sampling methods for aggregation queries:

1. If all the aggregate function computed in the query can be easily estimated based on random uniform sample (e.g. SUM, CNT, quantiles, etc), both per-group and per-tuple sampling are compatible with a query.
2. If at least one of the aggregate functions referenced in a query is incompatible with random uniform sampling (e.g. count duplicate, OR aggregation, etc), per-group sampling must be used.
3. If aggregation query has a HAVING clause referencing the aggregate values, per-group sampling must be used. The intuition behind this rule is that by providing HAVING clause for aggregate values query writer signifies the importance of exact computation of aggregate values.
4. All user-defined aggregate functions (UDAFs) must be explicitly labeled by the authors to specify whether they are sensitive to uniform random sampling or not. This information allows us to treat UDAFs as any other aggregate function when deciding which sampling strategy is compatible with the query.



### 5.4.2.3 Sampling in Stream Merge Queries

A merge query performs a union of two streams  $R$  and  $S$  in a way that preserves the ordering properties of the temporal attributes.  $R$  and  $S$  must have the same schema, and both must have a temporal field, say  $t$ , on which to merge. In order to preserve the abstraction of having one large stream, the sampling of merged streams must be coordinated. Similar to selection/projection queries, both sampling methods are compatible; in addition both streams must be sampled using the same method with the same sampling rates. Additional restrictions on load shedding strategy might be placed by other queries that consume the merged stream; we will discuss it in more details in Section 5.

### 5.4.2.4 Sampling in Stream Join Queries

In a query language with tumbling window semantics, a join between two data streams  $R$  and  $S$  must contain an equality predicate that relates a timestamp from  $R$  to one in  $S$ . In addition to this special equality predicate, join queries might contain any number of other predicates relating the attributes from two streams. Consider the join query below that correlates two streams of TCP packets with matching source and destination IP address.

```
SELECT TCP1.tb, TCP1.srcIP, TCP1.len + TCP2.len
FROM TCP1 JOIN TCP2
WHERE TCP1.srcIP=TCP2.destIP and TCP1.tb = TCP2.tb
```

Both per-tuple and per-group sampling using join attributes guarantee that the output of the query for every time bucket  $tb$  will be a subset of the exact output and therefore satisfy our definition of compatible sampling method. However, using per-tuple random uniform sampling with sampling rate  $R$  reduces the effective sampling rate to  $R^2$ . Per-group sampling with rate  $R$  using  $srcIP$  for stream  $TCP1$  and  $destIP$  for stream  $TCP2$   $k$ , keeps the query effective sampling rate at  $R$  and is

therefore preferable. In general we will always use per-group sampling for join queries except in the special case where the only attribute in the join equality predicates is temporal attribute. In that special case, both per-tuple and per-group sampling are acceptable.

## 5.5 Semantic Sampling for Query Sets

Data stream management systems are expected to run large number of queries simultaneously; queries in turn may contain a number of different query nodes (selections, aggregations, merges, and joins). Each of the nodes might place different requirements for range of acceptable sampling methods.

**Example:** Consider the following query set:

Query **flow\_dup\_count**:

```
SELECT tb, srcIP, destIP, srcPort, destPort,
       count_dups(seq_nbr) as dup_cnt
FROM TCP
GROUP BY time/60 as tb, srcIP, destIP, srcPort, destPort
```

Query **max\_dups**:

```
SELECT tb, srcIP, destIP, MAX(dup_cnt)
FROM flow_dup_count
GROUP BY tb, srcIP, destIP
```

Query `flow_dup_counts` computes the number of duplicate packets in each TCP flow; query `max_dups` computes the maximum number of duplicates for each pair of communication hosts. Query `flow_dup_count` requires per-group sampling to be used with a compatible grouping set of the form of `{sc_exp(srcIP), sc_exp(destIP), sc_exp(srcPort), sc_exp(destPort)}` or any of its non-

empty subsets. Query `max_dups`, on other hand, requires the input stream to be per-group sampled using  $\{sc\_exp(srcIP), sc\_exp(destIP)\}$ . Considering both grouping sets we can infer that per-group sampling of TCP stream using  $\{sc\_exp(srcIP), sc\_exp(destIP)\}$  will satisfy *both* queries. A similar inference is required for join queries whose child queries have different grouping sets.

In what follows, we will present our analysis framework that infers the set of compatible sampling methods for arbitrary Directed Acyclic Graph (DAG) of streaming query nodes.

### 5.5.1 Placement of sampling in a query DAG

The placement of the sampling operators in a query DAG critically affects the effectiveness of load shedding mechanism. One obvious choice is to perform sampling directly on the stream source before processing tuples by low-level operators. Shedding tuples as early as possible avoids investing processing time into computation of aggregate tuples that may eventually be discarded. Dropping tuples at higher-level nodes in query tree is generally less efficient and makes reasoning about the semantics of answers more difficult. One scenario in which sampling on non-leaf query node is justifiable is when output of a query is shared by multiple consumers with different tolerance to the rate with which input stream is sampled. We do not concern ourselves with this scenario and will only consider leaf-level sampling.

### 5.5.2 Reconciling query grouping sets

Previously, we discussed the need to reconcile the different requirements two queries might have for compatible grouping set to generate a new grouping set compatible with both queries. We abstract this issue using `Reconcile_Group_Sets()`, defined as follows:

**Def.** Given two grouping set definitions *GS1* for query *Q1* and *GS2* for query *Q2*, *Reconcile\_Group\_Sets()* is defined to return the largest grouping set *Reconciled\_GS* such that per-group sampling using *Reconciled\_GS* is strongly compatible with both *Q1* and *Q2*.

Considering a simple case of grouping sets consisting of just the stream attributes (no scalar expressions involved), *Reconcile\_Group\_Sets()* computes the intersection of two grouping sets. For example *Reconcile\_Group\_Sets*({srcIP, destIP}, {srcIP, destIP, srcPort, destPort},) is a set {srcIP, destIP }. For more general case of grouping sets involving arbitrary scalar expressions *Reconcile\_Group\_Sets* uses scalar expression analysis to find “least common denominator”. For example *Reconcile\_Group\_Sets* ({sc\_exp(time/60), sc\_exp(srcIP), sc\_exp(destIP)}, {sc\_exp(time/90), sc\_exp(srcIP & 0xFFFF0)} ) is equal to a set {sc\_exp(time/180, sc\_exp(srcIP & 0xFFFF0))}. The *Reconcile\_Group\_Sets* function can make use of either simple or complex analysis based on implementation time that is available. A full discussion is beyond the scope of this dissertation, but we expect that the simple analyses used in the example will suffice for most cases.

### 5.5.3 Algorithm for assigning sampling methods to leaf nodes in query set

We now describe an algorithm for assigning the sampling methods to each of the input stream for arbitrary query sets. The algorithm takes a query DAG as an input and produces labelling of the leaf-nodes with the compatible sampling method. The algorithm is comprised of the following stages:

#### **Transform the query DAG into query forest**

We transform the query DAG by splitting all the query nodes that have multiple parent nodes into set of independent nodes that have a single parent. Since it is possible that multiple copies of the



We label these types of nodes as *sampling-unsafe*, since their output results cannot be approximated.

### Assign and reconcile node grouping sets

In this phase of the algorithm, we reconcile sampling requirements of all dependent query nodes in the query set. Formal description of reconciliation algorithm is given below.

**Input:** Topologically sorted list of nodes in the query tree  $V_1, V_2, \dots, V_n$

**Output:** Labelling of the leaf-nodes with the compatible sampling method.

### Algorithm:

1. For every  $i \in [1 \text{ to } n]$ , compute  $GS(V_i)$ . For binary operators compute  $GS_{left}(V_i)$  and  $GS_{right}(V_i)$ .

If the node is compatible with per-tuple sampling, set  $GS(V_i)$  to the union of all attributes of input schema. If no compatible sampling strategy exists, set  $GS(V_i) = \emptyset$ .

2. For every  $i \in [1 \text{ to } n]$

If  $V_i$  is unary operator with child node  $V_j$ .

set  $GS(V_i) = \text{Reconcile\_Group\_Sets}(GS(V_i), GS(V_j))$

If  $V_i$  is binary operator with child node  $V_{left}$  and  $V_{right}$ .

set  $GS(V_{left}) = \text{Reconcile\_Group\_Sets}(GS(V_i),$

$GS(V_{left}))$  and  $GS(V_{right}) = \text{Reconcile\_Group\_Sets}(GS(V_i), GS(V_{right}))$

3. For every pair of nodes  $V_i$  and  $V_j$  s.t.  $V_i$  and  $V_j$  share common ancestor, set  $GS(V_i) = GS(V_j) =$

$\text{Reconcile\_Group\_Sets}(GS(V_i), GS(V_j))$

### Transform query forest back into query DAG

Remember that in the first phase of the algorithm the nodes that have multiple parents are split to form a forest. In order to guarantee that multiple copies of the same node were not assigned a conflicting grouping sets we perform a final reconciliation of previously split nodes. All the split nodes that cannot be reconciled (result of the `Reconcile_Group_Sets()` is an empty set) are kept separate as independent instances of a query that use two different sampling methods.

## 5.6 Limitations of the Semantic Sampling Framework

Perhaps the biggest limitation of the semantic sampling framework lies in the inability to identify the true intent of the query writer just based on the query text. Consider the previously discussed query which computes TCP flows, maintaining only simple aggregates (`sum()` and `cnt()`). If the query writer is interested in getting maximum number of flows (e.g. to use it to compute the ratios of different types of flows), then we will be willing to tolerate the inaccuracy of `sum()` and `cnt()` aggregates. In that scenario uniform random sampling is the most appropriate and cheapest method. Based on the rules given in section 5.4, our automatic analysis would select this sampling strategy. If, on other hand, the query writer does not care about capturing all the flows, but is very sensitive to errors in aggregate values, per-group sampling is preferable. It is therefore easy to imagine the scenarios where automatic inference of compatible sampling method will fail to generate the preferred solution.

Another limitation of the proposed framework is the rather small suite of base sampling algorithms. For many widely used streaming queries, importance-based sampling algorithms which aim to reduce the variance of the approximation are more appropriate. Incorporating such algorithms into general semantic sampling framework is a promising future work direction that we discuss in Chapter 7.

## 5.7 Experimental Evaluation

In this section, we present our experiments with semantic sampling in the context of the Gigascope streaming database. We implemented both per-group and per-tuple sampling by augmenting query plans with additional selection predicates implementing corresponding sampling method. All sampling predicates were pushed to leaf nodes in query execution plan by query optimizer.

All the experiments were conducted on a live network feed from a data center tap. All our queries monitor the set of two high-speed DAG4.3GE Gigabit Ethernet interfaces. Both Gigabit interfaces receive approximately 100,000 packets per second (about 400 Mbits/sec). Our main goal is to compare the accuracy of the query results for a system that uses random uniform sampling as a load shedding mechanism to accuracy achieved using semantic sampling. We also evaluated the effectiveness with which both approaches can reduce the overall system load. All experiments were conducted on dual processor 2.8 GHz P4 server with 4 GB of RAM running FreeBSD 4.10.

### 5.6.1 Effective Sampling Rate for Join Queries

In this experiment, we evaluate how the results of the join queries are affected by the choice of the sampling strategy. We use a query from the networking domain that computes the round-trip delays for active TCP connections. Round-trip delay is defined as a time difference between SYN packet that initiates the TCP connection and corresponding SYN ACK packet. Computing round trip times requires the join between the streams of SYN and SYN ACK packets. The `syn_ack_delay` query computes the join between two streams as shown below:

Query `syn_ack_delay`:

```
SELECT S.tb, S.srcIP, S.destIP, S.srcPort, S.destPort,
```



```

(A.timestamp - S.timestamp) as rtt

FROM tcp_syn S JOIN tcp_syn_ack A

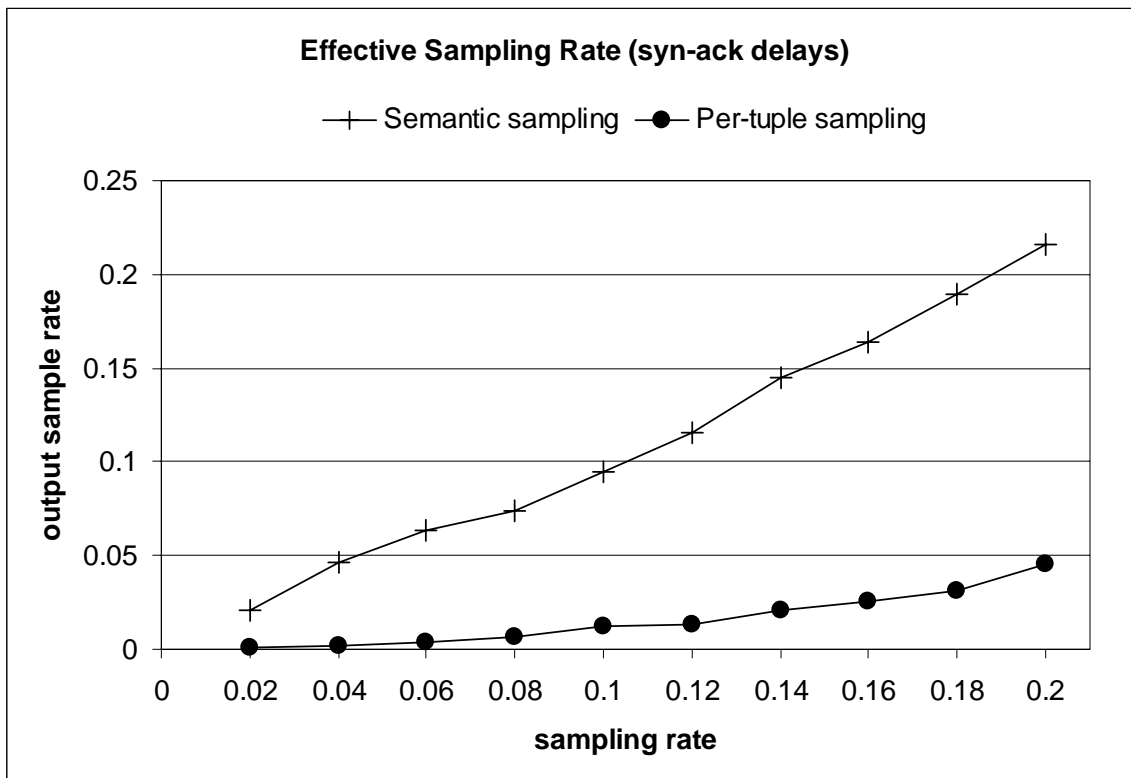
WHERE S.srcIP = A.destIP and S.destIP = A.srcIP and

S.srcPort = A.destPort and S.destPort = A.srcPort and

S.tb = A.tb and S.timestamp <= A.timestamp and

(S.sequence_number + 1) = A.ack_number

```



**Figure 5-3: Effective sampling rate**

To compare the performance of semantic sampling to traditional per-tuple sampling, we compare the ratio of different round trip delays computed by each method with exact value of the round trip delays computed when no sampling is used. Essentially the query computes the effective sampling rate that is achieved by the join using different sampling methods. We varied sampling rates from 0.02 to 0.2, which is typical range for network monitoring applications. The results of the experiments are presented in Figure 5-3.

The results of the experiments demonstrate that effective sampling rate when using semantic sampling with sampling rate  $R$  is very close to  $R$ . On the other hand, uniform random sampling performs poorly for join queries and leads to quadratic effective sampling rate for the output results. In order to achieve the accuracy of the semantic load shedding, traditional sampling method will have to use significantly higher sampling rate  $\sqrt{R}$ , which will correspondingly increase the overall system load.

### 5.6.2 Sampling Sensitive Aggregations

In this experiment, we observe how the accuracy of the results produced by sampling-sensitive aggregation queries is affected by the choice of sampling strategy. The queries used in experiment analyze the network performance by measuring the number of TCP packets that needed to be retransmitted due to packet loss. Monitoring retransmission rates is widely used by network analyst for analyzing the quality of end-to-end communications in managed network. The query `dup_all_count` computes the number of duplicate sequence numbers for each network flow identified by (srcIP, destIP, srcPort, destPort) as well as total number of packets in a flow. The results from query `dup_and_all_count` are further aggregated in query `dup_ratio` that for every 60 second time bucket computes the ratio of duplicate TCP packets to total number of packets sent. Corresponding GSQL queries `dup_all_count` and `dup_ratio` are shown below.

Query **dup\_all\_count**:

```
SELECT tb, count_dups(sequence_number) as dup_cnt,
       count(*) as full_cnt
FROM TCP

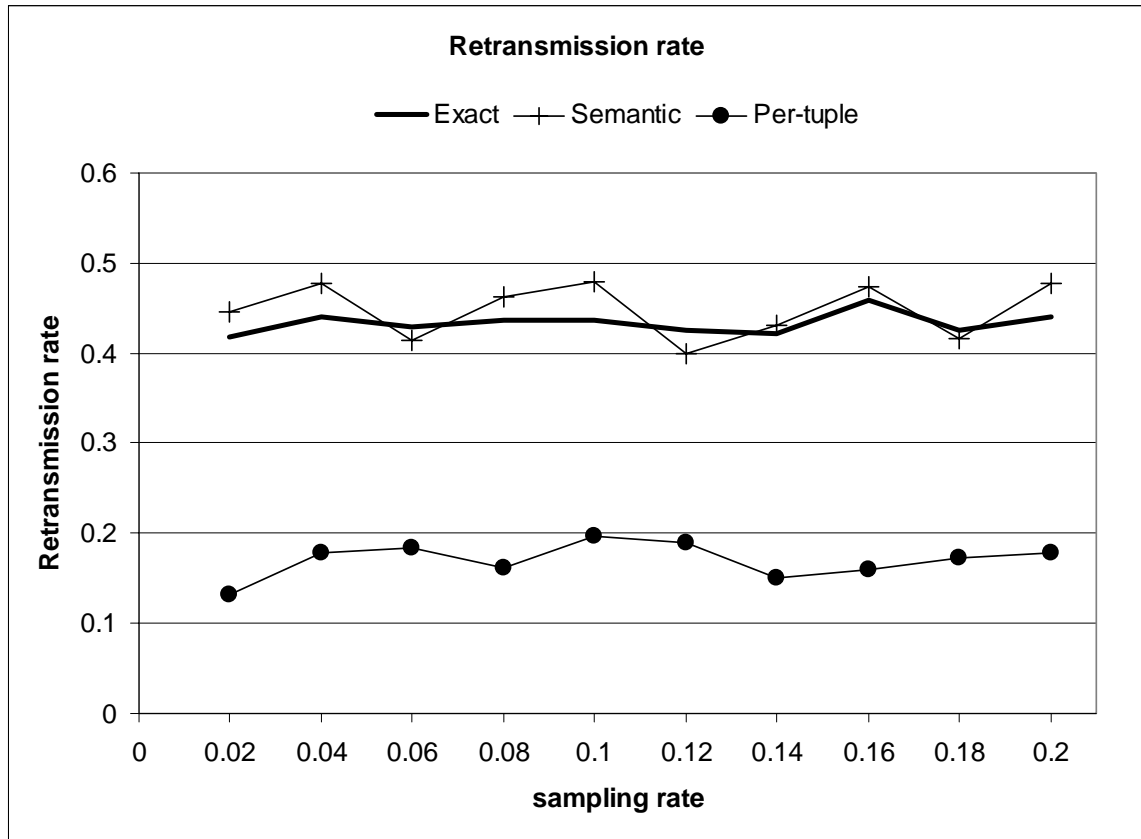
GROUP BY time/60 as tb, srcIP, destIP, srcPort, destPort
```

Query **dup\_ratio**:

```

SELECT tb, sum(dup_cnt) / sum(full_cnt)
FROM dup_and_all_count
GROUP BY tb

```



**Figure 5-4: Accuracy for aggregation queries**

As in previous experiment, we vary sampling rate from 0.02 to 0.2 in 0.02 increments. We compare a baseline configuration computing exact TCP retransmission rates (no sampling) with semantic sampling and per-tuple sampling. For a given sampling rate, we ran all three queries at the same time. The results of the experiments are presented in Figure 5-4. They demonstrate that semantic sampling achieves accuracy from 91 to 98%, while uniform random sampling prevents `count_dups()` aggregate from detecting large number of duplicate sequence numbers and leads to misleading results.

### 5.6.3 Semantic Sampling for Query Sets

In this experiment we observe how the choice of sampling strategy affects the accuracy of query sets involving multiple aggregation and join queries. The query set used in this experiment is designed to detect a particular type of Distributed Denial of Service (DDoS) attack known as a SYN-flood. During a SYN-flood, the attacking hosts send a large number of SYN packets with spoofed random IP addresses, which forces the victim host to wait forever for matching SYN ACK packets. To detect a SYN-flood attack, we compute the ratio of TCP SYN packets that have corresponding SYN ACK packets. The query `matched_syn_count` computes the join between the stream of SYN and SYN ACK packets and aggregates the results by computing the total number of matched SYN packets for every 60 second time bucket. The query `all_syn_count` computes the total number of SYN packets observed in the same timebucket, while `matched_syn_ratio` computes the ration of matched SYN packets to total number of SYN packets:

Query **`matched_syn_count`**:

```
SELECT tb, count(*)
FROM tcp_syn S JOIN tcp_syn_ack A
WHERE S.srcIP = A.destIP and S.destIP = A.srcIP
      and S.srcPort = A.destPort
      and S.destPort = A.srcPort and S.tb = A.tb
      and (S.timestamp <= A.timestamp)
      and (S.sequence_number + 1) = A.ack_number
GROUP BY time/60 as tb
```

Query **`all_syn_count`**:

```
SELECT tb, count(*) as cnt
FROM tcp_syn S
```

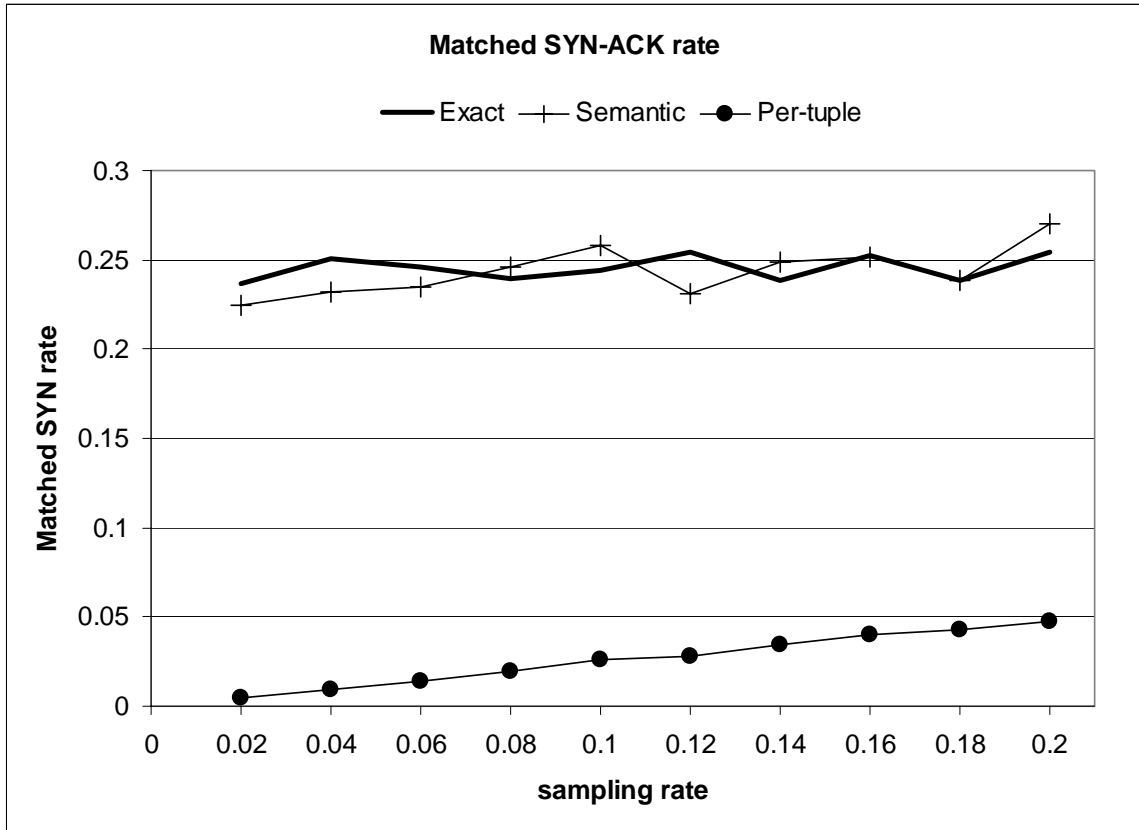
```
GROUP BY time/60 as tb
```

Query **matched\_syn\_ratio**:

```
SELECT A.tb, M.cnt / A.cnt as ratio  
FROM all_syn_count A OUTER_JOIN matched_syn_count M  
WHERE A.tb = M.tb
```

As in previous experiments, we vary sampling rate from 0.02 to 0.2 in 0.02 increments. We compare a baseline configuration computing the exact ratio of matched SYN packets (no sampling) with semantic sampling and per-tuple sampling. The results of the experiments are presented in Figure 5-5.

The results of the experiment confirm that semantic sampling maintains the correct semantics of the output results with observed accuracy in 91-99% range. Uniform random sampling on other hand again leads to misleading results and suggests that there is a SYN flood attack in progress while in fact the ratio of matched SYN packets is within norm.

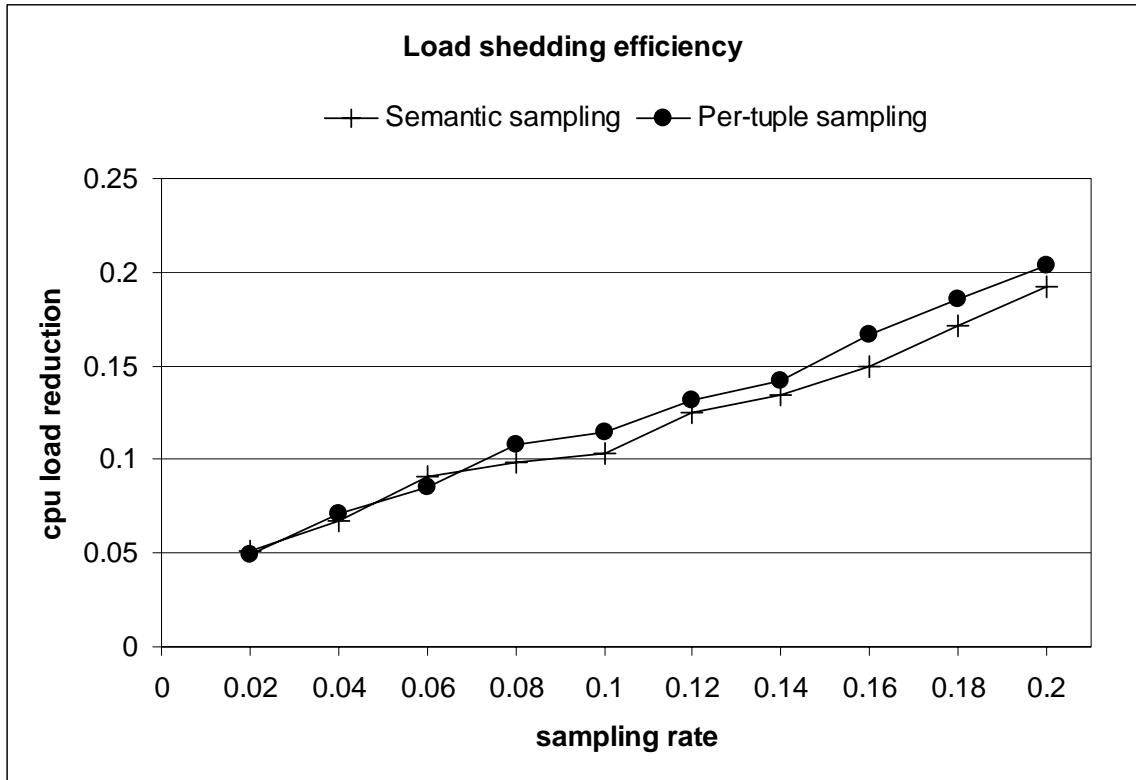


**Figure 5-5: Accuracy for complex query set**

#### 5.6.4 Efficiency of Load Shedding

In our final experiments, we evaluate the effectiveness of different sampling techniques at reducing the overall system load. We used the query set from Section 6.2 and observed the effect of sampling the input stream on average CPU load. Even though the experiments were conducted on live data stream and the load conditions changed slightly from one experiment to another, overall the stream load was stable and our comparison results are still valid.

We varied the sampling rate from 0.02 to 0.2 in 0.02 increments and observed the overall reduction in average CPU load for both semantic and per-tuple sampling. The results of the experiments are presented in Figure 5-6.



**Figure 5-6: CPU load for different sampling methods**

The results of the experiment confirm that sampling is an efficient load shedding strategy; varying the sampling rate from 0.02 to 0.2 we were able to reduce the overall system load from 0.05 to 0.2. It is interesting to note that semantic sampling achieves slightly better load reduction than per-tuple sampling despite the fact that it uses a more expensive sampling predicate. The main reason behind phenomena is that per-group sampling for aggregation queries discards larger percentage of groups compared to per-tuple sampling with the same sampling rate. Since the cost of the execution of aggregation operator is largely determined by the number of groups in the output, semantic sampling is more efficient at reducing the load for aggregation queries.

## 5.8 Summary

Data Stream Management Systems (DSMS) processing high rate data streams are often subject to bursts of high activity, which might overload the system, and have to be robust. In this chapter,

we show how to perform general-purpose query-aware sampling, which we call semantic sampling. We propose methods for analyzing a large class of streaming operators and judiciously choosing sampling methods that guarantee semantically correct results. We extend our single-operator techniques to a general framework for analyzing any set of queries to determine a semantics-preserving sampling strategy. Since it is important for applications to know which sampling methods and sampling rates were used to compute the query results, we propose to embed special punctuations into query output stream that would contain this information. The methods described in this chapter can be extended to handle a larger suite of sampling algorithms, such as various types of “importance” sampling which aim to reduce the variance in approximations [45].

We evaluate our semantic sampling approach by running various sets of streaming queries on high-rate data streams. The results of our experiments confirm that our method provides semantically correct and highly accurate results for scenarios where traditional per-tuple sampling fails to provide semantically meaningful results. We also demonstrate that semantic sampling is effective at reducing the overall system load, which makes it a very valuable technique the load shedder can employ to guarantee the robustness and the correctness of the results under overload conditions.



## Chapter 6

# 6. Query-aware Partitioning for Data Streams

The load generated by many streaming applications frequently exceeds by far the computation capabilities of single centralized server, making distributed query processing a necessity. Query-independent stream partitioning mechanisms widely used by modern distributed DSMS fail to reduce the load of a large class of streaming queries as compared to centralized systems, and can even increase the load.

In this chapter, we present an alternative approach - query-aware data stream partitioning that allows us to scale the performance of the streaming queries in a close to linear fashion. We present methods for analyzing any given query and choosing the optimal partitioning scheme, and show how to reconcile potentially conflicting requirements that different queries might place on partitioning. We propose a query analysis framework for determining the optimal partitioning and a partition-aware distributed query optimizer that takes advantage of existing partitions.

Experiments on a small cluster of processing nodes on high-rate network traffic feeds with different query sets demonstrate that our methods effectively distribute the load across all processing nodes and facilitate efficient scaling whenever more processing nodes become available.

## 6.1 Introduction

The volume of data that needs to be processed in real time for streaming applications can easily exceed the resources available on a centralized server. For example, dual OC768 network links used in Internet backbone generate up to 2x40 Gbit/sec of traffic, which corresponds to roughly 112 million packets/sec. Even a fast 4GHz server can spend at most 26 cycles processing each tuple, which does not allow it to perform any meaningful processing short of incrementing few counters. Furthermore, this data load exceeds by order of magnitude the throughput of fastest computer buses such as PCI-X and PCI-Express. The rate of scientific data feeds is typically lower than network traffic feeds, but the tuple sizes and per-tuple processing costs are significantly larger and can include more complex computation such as Fast Fourier Transforms (FFT). As a result scientific data processing can also result in a load that by order of magnitude exceeds the capacity of a single centralized server.

Distributed DSMSs attack the performance problem by spreading the load across a number of cooperating machines running independent DSMSs. Two commonly used techniques used to distribute the load across the participating machines are partitioning query plans into subplans to be executed in parallel (*query plan partitioning*) and splitting resource-intensive query nodes into multiple nodes working on subset of data feed (*data stream partitioning*). However, query plan partitioning fails to generate feasible execution plans if the original query plan contains one or more operator that is too “heavy” for a single machine. Such query plans are very common in network monitoring applications in which resource consumption of different query nodes is highly non-uniform.

Data stream partitioning as commonly implemented in DSMSs is done in query-independent fashion (e.g. partitioning tuples in random or round robin fashion). However, for a large class of

queries, such data stream partitioning fails to significantly reduce the load as compared to a centralized system and can even lead to increase in the load.

**Example.** Let us consider an example of a network monitoring query computing traffic flows – summaries of packets between a source and a destination during a period of time. The group-by attributes are the source and destination IP address, the source and destination port, and the protocol, while the aggregates include the number of packets, the number of bytes transferred, start and stop times, and so on. These types of queries are popular in various network monitoring applications – from performance monitoring to detecting network attacks [75]. The SQL version of the query is shown below.

```
SELECT time,srcIP,destIP,srcPort,destPort,
        COUNT(*), SUM(len), MIN(timestamp),MAX(timestamp), ...
FROM TCP
GROUP BY time,srcIP,destIP,srcPort,destPort
```

Suppose that data stream partitioning is applied in round robin fashion to evenly distribute input tuples among  $n$  machines that compute partially aggregated flows and send them to a central node that merges the partials flows and computes a final aggregation. It is easy to see that in the worst case, a single flow will result in  $n$  partial flows being computed and transmitted over the network to central aggregating node. In a more typical network monitoring scenario, the query is only interested in a subset of all flows. For example, suppose we want to monitor *attack* flows that do not follow TCP protocols and can frequently be differentiated by OR of the flags of the packets in the flow. In SQL we can write this query by adding a corresponding HAVING clause to the flow query (e.g. `HAVING OR_AGGR(flags)= ATTACK_PATTERN`). It is easy to see that none of the nodes performing local aggregation will be able to apply the HAVING clause to filter out regular flows. Depending on the number of participating hosts, the CPU and network link load on

final aggregation node can exceed the load on single node in centralized case, rendering the execution strategy infeasible.

For this example, a more reasonable approach for distributing the load among the participating machines is to partition the input data stream based on flows (e.g. evenly distributing entire flows). If such partitioning is utilized, all flows can be computed locally and filtered using the HAVING clause before being transmitted over the network. The problem of determining a good partitioning scheme for certain classes of individual relational queries (aggregations and equijoins) has been studied in the context of parallel relational databases; however in the streaming environment existing approaches do not scale to complex query sets and massive data rates. Data stream management systems are generally expected to run a large number of queries simultaneously; queries in turn may contain a number of different subqueries. Each of the subqueries might place different requirements for the way partitioning has to be done. These requirements can easily be in conflict with each other and it would not be always possible to satisfy all of them. It is also not feasible to dynamically repartition the inputs to suit individual queries as is commonly done in parallel relational databases, since each such repartitioning puts the entire stream back into inter-node network without any data reduction, which greatly increases communication costs. In general, we need a partitioning mechanism that can automatically analyze an arbitrary complex query set and determine the optimal initial stream partitioning scheme.

In order to incorporate the results of the analysis into distributed query optimization, we need to make the optimizer fully aware of the partitioning scheme used. However, we cannot make an assumption that the actual partitioning scheme used by the system is identical to the optimal one recommended by the analysis. For many applications, the implementation details of particular systems can place additional constraints on what kind of partitioning can be used. Consider for example an application that wants to monitor high-speed OC768 network. Monitoring the

80Gbit/sec link requires specialized network interface cards that can partition the data at line speeds. Even though such interface cards are typically programmable using FPGAs, the limited number of available gates place restrictions on a type of partitioning can be performed in hardware. For example it is possible to implement partitioning based on TCP fields such as source or destination IP addresses, but accessing fields from higher-level protocols such as HTTP requires regular expression processing that is not feasible to do at OC768 speeds. Therefore, we need a distributed query optimizer that is flexible enough to take advantage of any available partitioning.

The query-aware data stream partitioning mechanism proposed in this chapter includes both an analysis framework for determining the optimal partitioning and a partition-aware distributed query optimizer that transforms the unoptimized query plan into a semantically equivalent query plan that takes advantage of existing partitions.

The contributions we make in this chapter are as follows. We

1. Develop the concept of query-aware data stream partitioning for distributed stream processing.
2. Present techniques for determining the appropriate partitioning scheme for a given query.
3. Design and develop a framework a framework for analyzing a set of queries to determine a partitioning strategy that would satisfy all the queries in a set.
4. Determine a set of query transformation rules to be used by query optimizer to take advantage of existing data stream partitioning.
5. Perform detailed experiments with a live cluster of stream processing nodes and show that our partitioning methods lead to highly efficient distributed query execution plans that scale linearly with the number of nodes.

The rest of the chapter is organized as follows. We discuss related work in Section 6.2. We give an overview of query-aware data stream partitioning in Section 6.3. In section 6.4 we present the partition analysis framework for arbitrary query sets. Section 6.5 covers partition-aware query transformation rules for distributed query optimization. In Section 6.6 we present the results of experimental evaluation of our techniques. Conclusions and promising directions for future work are in Section 6.7.

## 6.2 Previous Work

A number of currently active research data streaming projects focus on extending DSMS to enable scalable distributed stream processing [18][106]. Two main approaches used to distribute the load across the cooperating machines are query plan partitioning and data stream partitioning.

The load distribution mechanism used in Borealis [18] relies on query plan partitioning to balance the load on cooperating DSMSs. As we discussed earlier, this approach is not feasible if a query plan contains one or more operators that are too “heavy” for a single machine. In addition to query plan partitioning, Borealis also employ fairly simple data stream partitioning mechanism called *box splitting*. However, partitioning is done in a query-independent manner and requires expensive processing of partial results generated by split query nodes.

The FLUX load partitioning operator used in TelegraphCQ DSMS [106] supports a variety of data stream partitioning schemes including the hash-based strategy used in our mechanism. The primary goal of FLUX is to avoid imbalance in the load caused by the data scheme. To address the imbalance problem it uses an adaptive partitioning adjusted at runtime depending on observed data skew. The partitioning itself is still however operator-independent and suffers from excessive load on the node combining partial results.

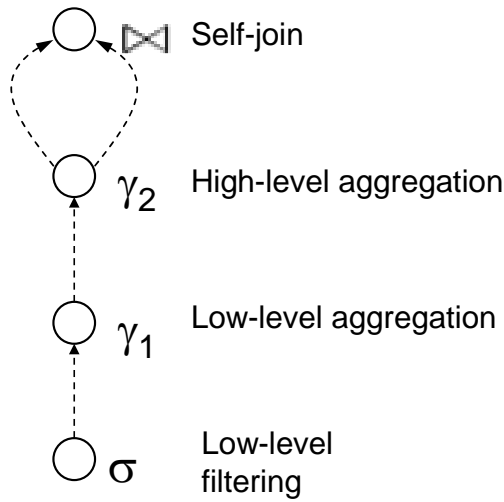
The Grid Stream Data Manager (GSDM) described in [67] proposes *operator-dependent windows split* strategy which partitions the input data stream in such a way that partial results can be inexpensively combined. The query writer is expected to manually provide specific *stream distribute/stream merge* routines for all query nodes eligible for optimization. The authors do not address the issue of automatic inference of an optimal splitting strategy for arbitrary query sets.

Recent work on automating physical database design for relational databases [98] addresses a problem of choosing a database partitioning scheme that is optimal or close to optimal for a given query workload. The proposed system relies on the IBM DB2 query optimizer for both recommending candidate partitions and estimating potential query costs. Even though some of the technique suggested by the authors can potentially be applied to stream processing, the main focus of the work is on processing of large-scale stored data sets.

### 6.3 Query-Aware Stream Partitioning Overview

The goal of the query-aware data stream partitioning mechanism is to distribute input tuples across multiple machines in such a way that maximizes the amount of data reduction that can be performed locally before shipping the intermediate results to a node that produces final results. We would call such partitioning *compatible* with a given query. In this section we will give a formal definition of partition compatibility and show how to infer compatible partitioning scheme for two major classes of streaming queries – aggregations and joins.

### 6.3.1 Illustrative Example



**Figure 6-1: Sample query execution plan**

We illustrate the query-aware partitioning mechanism by working through an example query set. The first query (**flows**, denoted  $\gamma_1$ ) computes simplified TCP traffic flows for every 60 second time epoch (for each communicating source and destination host it produces a number of packets sent between them). The higher-level aggregation query (**heavy\_flows**, denoted  $\gamma_2$ ) computes “heaviest” flows for each source (heaviest flows have the largest number of packets). Finally a self-join query (**flow\_pairs**, denoted  $\Join$ ) correlates heavy flows that span consequent time epochs. The corresponding SQL statements for both queries are shown below:

Query **flows**:

```
SELECT tb,srcIP,destIP,COUNT(*) as cnt
FROM TCP
GROUP BY time/60 as tb,srcIP,destIP
```

Query **heavy\_flows**:

```
SELECT tb,srcIP,max(cnt) as max_cnt
```



```
FROM flows


GROUP BY tb, srcIP
```

Query **flow\_pairs**:

```
SELECT S1.tb, S1.srcIP, S1.max_cnt, S2.max_cnt
FROM heavy_flows S1, heavy_flows S2
WHERE S1.srcIP = S2.srcIP and S1.tb = S2.tb+1
```

A query plan for execution of the queries is shown in Figure 6-1.

1. *Which partitioning scheme is optimal for each of the queries in an execution plan?*

Intuitively, a lower-level aggregation query (node  $\gamma_1$ ) will benefit the most from a partitioning which guarantees that no tuples with identical pair of attributes (srcIP, destIP) will end up in different partitions. Any partitioning that satisfies this properly would allow  $\gamma_1$  to be evaluated in parallel on all participating hosts with linear scalability. Following a similar intuition, the query nodes  $\gamma_2$  and self-join node  will benefit the most if the input stream was partitioning using (srcIP). Later in the section, we will formally define what requirements a partitioning scheme must satisfy and give inference rules to compute an appropriate partitioning for major classes of streaming queries.

2. *How to reconcile potentially conflicting partitioning requirement from different queries in a query set?*

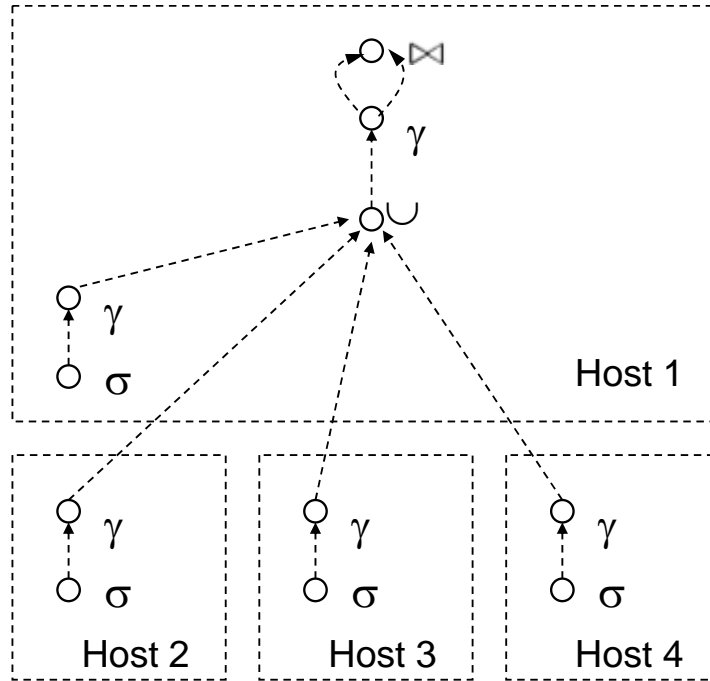
As we have seen previously, query  $\gamma_1$  will benefit mostly from partitioning based on attributes (srcIP, destIP), while the rest of the queries would prefer partitioning on (srcIP). Since it is (usually) not feasible to partition the input stream simultaneously in multiple ways, we need to reconcile partitioning requirements of different query nodes. It is easy to see that partitioning on

(srcIP) can satisfy all queries in our sample query set. More generally, we will need an algorithm for inferring an optimal set of attributes to be used for partitioning for arbitrary complex query set. We will present such an algorithm in Section 6.4.

3. *How can we use the information about the scheme used for partitioning in distributed query optimizer?*

Assuming the input stream is partitioned as recommended by the query analysis, we can use this information to drive the distributed query optimizer. In our prototype implementation the optimizer works by invoking a set of partition-aware transformation rules on nodes of original query plan in bottom-up fashion.

In many real life applications, the query writer does not have complete control over how the partitioning is done. As we mentioned in the introduction, processing capabilities of the hardware used for partitioning can place restrictions on the partitioning scheme. For example we could have hardware that can only split the input stream based on (destIP). The query optimization framework needs to be flexible enough to maximally take advantage of any partitioning, even if it is different from the optimal one. An example distributed query plan produced by the optimizer under the assumption that partitioning is done based on (destIP) is shown in Figure 6-2.



**Figure 6-2: Optimized query execution plan**

### 6.3.2 Hash-Based Stream Partitioning

The main goal of any stream partitioning scheme is to distribute tuples evenly across multiple distributed nodes in such a way that load is evenly spread across all nodes. There are multiple ways in which such scheme could be implemented, but one of the simplest can be done by hashing selected set of tuple attributes. More formally we will define hash-based tuple partitioning in the following way:

**Def.** Let  $A$  be a set of the tuple attributes ( $attr_1, attr_2, \dots, attr_n$ ) and  $H$  be a hash function with large integer domain  $[0, R]$ . A hash-based tuple partitioning for stream  $S$  is defined as partitioning  $S = \cup Partn\_i$ , where  $Partn\_i$  is defined as a set of tuples with  $i * R/M \leq H(attr_1, attr_2, \dots, attr_n) < (i + 1) * R/M$ , where  $M$  is the number of partitions. The set of attributes  $A$  will be further called **partitioning set**.

It is easy to see that hash-based partitioning has a property that  $\forall i \neq j \text{ Partn\_}i_i \cap \text{Partn\_}j = \emptyset$ . It is often beneficial not to restrict ourselves to using singleton tuple attributes and instead allow grouping sets to include arbitrary scalar expression involving tuple attributes. For example one choice of partitioning set for could be (srcIP & 0xFFFF0, destIP) which will effectively partition tuples based on subnet that srcIP belongs to. Let  $\text{sc\_exp}_i(\text{attr}_i)$  represent a scalar expression. For the rest of the chapter we will only assume more general definition of partitioning set:

$$(\text{sc\_exp}_1(\text{attr}_1), \text{sc\_exp}_2(\text{attr}_2), \dots, \text{sc\_exp}_n(\text{attr}_n))$$

### 6.3.3 Partition Compatibility

The choice of the partition set critically impacts the ability of the query optimizer to reorganize the query plans for distributed evaluation. Consider the following aggregation query that computes simple network flows:

```
SELECT tb, srcIP, destIP, sum(len)
FROM PKT
GROUP BY time/60 as tb, srcIP, destIP
```

It is easy to see that partitioning using partitioning set (time/60, srcIP, destIP) allows each host to execute the aggregation query locally on corresponding partition with no further aggregation necessary. A partition-aware query optimizer can replace the aggregation query by stream union of the identical queries running on individual partitions. However, if (srcIP, destIP, srcPort, destPort) is used as partitioning set, this optimization would not be possible. We will capture the notation of “optimizer-friendly” partitioning set in the following definition:

**Def.** Partitioning set  $P$  is **compatible** with a query  $Q$  if for every time window, the output of the query is equal to a stream union of the output of the  $Q$  running on all partitions produced by  $P$ ,

An example of such compatible partitioning set for the query above is  $\{(time/60)/2, srcIP \& 0xFFFF0, destIP \& 0xFF00\}$ . An example of an incompatible grouping set for the query above is  $\{time, srcIP, destIP\}$  (since tuples belonging to the same 60 second epoch will end up in different partitions).

In the following sections, we will give the rules for inferring the compatible partitioning sets for two major classes of streaming queries - aggregations and joins. Other types of streaming queries (selection, projection, union) are always compatible with any partitioning sets and therefore we will omit the discussion of these query types.

### 6.3.4 Inference of Partitioning Sets for Streaming Queries

The definition of partition compatibility given in the previous section is very generic and does not directly tell us how to infer the partitioning set for a given query. In this section, we give equivalent definitions of query compatibility for both aggregation and join queries that can be directly applied to compute the partitions. For simplicity of the discussion we will assume tumbling window semantics for streaming queries (except where otherwise noted).

#### 6.3.4.1 Dealing with temporal attributes

One issue that needs to be considered when selecting a partitioning set compatible with a given query is whether to include the temporal attributes. Selecting the temporal attribute in a partitioning set will effectively change the hash function used by a partitioning method whenever the time epoch changes. This property could be desirable if we want to avoid bad hash functions that fail to uniformly spread the load across the participating machines. We can control the periodicity of the partitioning change by changing the value of the scalar expression involving the temporal attribute. For example an aggregation query that uses  $time/60$  to aggregate in one-minute time buckets can use  $time/60/10$  as a member of a partitioning set to change the hash

function every 10 minutes. Note that for sliding window queries that use pane-based [84], changing the hashing function in the middle of a window will lead to incorrect query results. Therefore, we always remove the temporal attributes from the partitioning sets of such queries.

For most of the aggregation and join queries, it is impossible to guess whether periodically changing a hash-based partitioning is desirable based just on the query text. It would require a priori knowledge of the distribution of the values of tuple attributes. An alternative solution is to monitor how well the hash function distributes the load at the runtime and adjust the hash function on epoch boundaries if necessary. A full discussion of the runtime load monitoring system is beyond the scope of the dissertation.

#### 6.3.4.2 Partitioning sets for aggregation queries

In its general form an aggregation query has the following format:

```
SELECT expr1, expr2, ... , exprn
FROM STREAM_NAME
WHERE tup_predicate
GROUP BY temp_var, gb_var1, ... , gb_varm
HAVING group_predicate
```

We only consider a subset  $G$  of these groupby variables ( $gb\_var_1, \dots, gb\_var_m$ ) that can be expressed as a scalar expression involving an attribute of one of the source input streams (ignoring grouping variables that are, e.g., results of aggregations computed in lower-level queries).

**Lemma 1.** *Let  $G$  be a set of group-by attributes referenced by the query  $Q$  and let  $P$  be partitioning set,  $P = (sc\_expr(attr_1), sc\_exp(attr_2), \dots, sc\_exp(attr_n))$ . Query  $Q$  is compatible*

with partitioning set  $P$  iff for any pair of tuples  $\text{tup1}$  and  $\text{tup2}$   $G(\text{tup1}) = G(\text{tup2}) \Rightarrow P(\text{tup1}) = P(\text{tup2})$ .

Following Lemma 1, any compatible partitioning set for aggregation query  $Q$  will have the following form:

$$\{\text{sc\_exp}(\text{gb\_var}_1), \dots, \text{sc\_exp}(\text{gb\_var}_n)\}$$

where  $\text{sc\_exp}(x)$  is any scalar expression involving  $x$ . Given that there is an infinite number of possible scalar expressions, every aggregation query has an infinite number of compatible partitioning sets. Furthermore, any subset of a compatible partitioning set is also compatible.

#### 6.3.4.3 Partitioning sets for join queries

We will consider a restricted class of join queries, namely two-way equi-join queries that use the semantics of tumbling windows. The general form of such query has the following format:

```
SELECT expr1, expr2, ... ,exprn
FROM STREAM1 AS S {LEFT|RIGHT|FULL}
      [OUTER] JOIN STREAM2 as R
WHERE STREAM1.ts = STREAM2.ts and
      STREAM1.var11 = STREAM2.var21 and ...
      STREAM1.var1k = STREAM2.var2k and
      other_predicates
```

For ease of the analysis, we will only consider join queries whose WHERE clause is in Conjunctive Normal Form (CNF) in which at least one of the CNF terms is equality predicate between the scalar expressions involving attributes of the source streams. Let  $J$  be a set of all such equality predicates  $\{\text{sc\_exp}(R.\text{rattr}_1) = \text{sc\_exp}(S.\text{sattr}_1), \dots, \text{sc\_exp}(R.\text{rattr}_n) = \text{sc\_exp}(S.\text{sattr}_n)\}$ . As with aggregation queries, we will only consider scalar expressions involving attributes of the

source input streams. Join queries that do not satisfy these requirements will be considered incompatible with any partitioning set.

**Lemma 2.** *Let  $J$  be a set of equality join predicates of the query  $Q$  and let  $P$  be a partitioning set,  $P = (sc\_expr(attr_1), sc\_exp(attr_2), \dots, sc\_exp(attr_n))$ . Query  $Q$  is compatible with partitioning set  $P$  iff there exists a non-empty subset  $J'$  of  $J$  s.t. for any pair of tuples  $tup1$  from  $R$  and  $tup2$  from  $S$  s.t.  $J'$  is satisfied  $\Rightarrow P(tup1) = P(tup2)$ .*

Following Lemma 2, we can compute the partitioning sets for both streams  $S$  and  $R$  using  $Partn\_R = \{ sc\_exp(R.attr_1), \dots, sc\_exp(R.attr_n) \}$  and  $Partn\_S = \{ sc\_exp(S.attr_1), \dots, sc\_exp(S.attr_n) \}$  respectively. It also follows that join query is compatible with any non-empty subset of its partitioning set. Since it is not feasible to partition the input stream simultaneously in multiple ways,  $Partn\_R$  and  $Partn\_S$  will need to be reconciled to compute a single partitioning scheme. More details on reconciliation procedure are given in Section 6.4.

## 6.4 Partitioning for Query Sets

Data stream management systems are expected to run a large number of queries simultaneously; queries in turn may contain a number of different subqueries (selections, aggregations, unions, and joins). Each of the subqueries might place different requirements on partitioning set to be compatible with it.

**Example:** Consider the following query set:

Query **tcp\_flows**:

```
SELECT tb, srcIP, destIP, srcPort, destPort,
       COUNT(*), SUM(len)
FROM TCP
```



```
GROUP BY time/60 as tb, srcIP, destIP, srcPort, destPort
```

Query **flow\_cnt**:

```
SELECT tb, srcIP, destIP, count(*)
FROM flow_dup_count
GROUP BY tb, srcIP, destIP
```

Query `tcp_flows` computes the number of packets and total number of bytes sent in each flow; query `flow_cnt` computes a number of distinct flows active during the time epoch for each pair of communication hosts.

Based on our analysis for individual queries, `tcp_flows` is compatible with partitioning set of the form of  $\{sc\_exp(srcIP), sc\_exp(destIP), sc\_exp(srcPort), sc\_exp(destPort)\}$  or any of its non-empty subsets. Query `flow_cnt`, on other hand, requires the input stream to be partitioned using  $\{sc\_exp(srcIP), sc\_exp(destIP)\}$  to be compatible with distributed optimization. Considering both partitioning sets we can infer that partitioning based on  $\{sc\_exp(srcIP), sc\_exp(destIP)\}$  will be compatible with *both* queries. A similar inference is required for join queries whose child queries have different compatible partitioning sets.

In what follows, we will present our analysis framework that infers the compatible partitioning set for arbitrary set of streaming queries. Our framework makes a simplifying assumption that all of the source input streams processed by a query set are partitioned using the same partitioning set. Expanding the analysis algorithms to handle different partitioning schemes for different input stream is part of planned future work.

### 6.4.1 Reconciling Partitioning Sets

Previously, we discussed the need to reconcile the different requirements two queries might have for a compatible grouping set to generate a new grouping set compatible with both queries. We abstract this issue using `Reconcile_Partn_Sets()`, defined as follows:

**Def.** *Given two partitioning set definitions  $PS1$  for query  $Q1$  and  $PS2$  for query  $Q2$ , `Reconcile_Partn_Sets()` is defined to return the largest partitioning set  $Reconciled\_PS$  such that both  $Q1$  and  $Q2$  are compatible with partitioning using a set  $Reconciled\_PS$ . The empty set is returned if no such  $Reconciled\_PS$  exists.*

Considering a simple case of partitioning sets consisting of just the stream attributes (no scalar expressions involved), `ReconcilePartn_Sets()` returns the intersection of the two partitioning sets. For example `Reconcile_Partn_Sets({srcIP, destIP}, {srcIP, destIP, srcPort, destPort})` is the set { srcIP, destIP }. For a more general case of partitioning sets involving arbitrary scalar expressions, `Reconcile_Partn_Sets` uses scalar expression analysis to find “least common denominator”. For example

```
Reconcile_Partn_Sets (
    {sc_exp(time/60), sc_exp(srcIP), sc_exp(destIP)},
    {sc_exp(time/90), sc_exp(srcIP & 0xFFFF0)} )
```

is equal to a set

```
{sc_exp(time/180, sc_exp(srcIP & 0xFFFF0))}.
```

The `Reconcile_Partn_Sets` function can make use of either simple or complex analysis based on the implementation time that is available. A full discussion is beyond the scope of this

dissertation, but we expect that the simple analyses used in the example will suffice for most cases.

### 6.4.2 Algorithm for Computing a Compatible Partitioning Set

We represent a set of streaming queries as a Directed Acyclic Graph (DAG) of streaming *query nodes*, where each query node is a basic streaming query (selection/projection, union, aggregation, and join). Even though most real systems also use more complicated streaming operators, we can always express them using a combination of basic query nodes. Note that based on the analysis in Section 6.3, we know how to compute compatible partitioning sets for all individual query nodes.

Computing a compatible partitioning for an arbitrary query set essentially requires reconciling all the requirements that all nodes in the query graph place on compatible partitioning sets. A simplified implementation of the procedure of computing compatible set  $PS$  for a DAG with  $n$  nodes would look the following way:

1. For every query node  $Q_i$  in a query DAG, compute the compatible partitioning set  $PS(Q_i)$ .
2. Set  $PS = PS(Q_1)$ .
3. For every  $i \in [1 \text{ to } n]$ , set  $PS = \text{Reconcile\_Partn\_Sets}(PS, PS(Q_i))$ .

Unfortunately, for many realistic query sets we would expect the resulting partitioning set  $PS$  to be empty due to conflicting requirements of different queries. A more reasonable approach would be to try to satisfy a subset of nodes in a query DAG in order to minimize the total cost of the query execution plan. There are a variety of different cost models that can be used to drive the optimization; in this chapter we will use a simple model that approximates a maximum network load on single node.

### 6.4.2.1 Cost model for streaming query nodes

The cost model that we are going to use in this chapter defines a cost of query execution plan to be the maximum amount of data a single node in query execution plan is expected to receive over the network during one time epoch. The intuition behind this model is trying to avoid query plans that overload a single host with excessive amounts of data sent from query nodes residing on different hosts.

Let  $R$  be the rate of the input stream on which the query set is operating, and  $PS$  be a partitioning set. For each query node  $Qi$  in a potential query execution plan we define the following variables:

- $\text{selectivity\_factor}(Qi)$ . The selectivity factor estimates the expected ratio of the number of output tuples to the number of input tuples  $Qi$  receives during one epoch.
- $\text{out\_tuple\_size}(Qi)$ . Expected size of the output tuple produced by  $Qi$ .
- We recursively define  $\text{input\_rate}(Qi)$  to be  $R$  if  $Qi$  is a leaf node and to be the sum of all  $\text{output\_rate}(Qj)$  s.t.  $Qj$  is a child of  $Qi$ .
- $\text{output\_rate}(Qi) = \text{input\_rate}(Qi) * \text{selectivity\_factor}(Qi) * \text{out\_tuple\_size}(Qi)$ .

We define the  $\text{cost}(Qi)$  in the following way:

- 0 if it processes only local data
- $\text{input\_rate}(Qi)$  if  $Qi$  is incompatible with  $PS$
- $\text{output\_rate}(Qi)$  if  $Qi$  is compatible with  $PS$

The intuition behind this cost formula is that an operator partitioned using a compatible partitioning set only needs to compute the union of the results produced by remote nodes, and therefore the rate of the remote data it is expected to receive is equal to its output rate.

Finally, we define the cost of the query plan  $Qplan$  given partitioning  $PS$   $cost(Qplan, PS)$  to be the max  $cost(Q_i)$  for all  $i$ .

#### 6.4.2.2 Computing an optimal compatible partitioning set

We now describe an algorithm for computing an optimal partitioning set for arbitrary query sets. The algorithm takes a query DAG as an input and produces a partitioning set that minimizes the cost of the query execution plan. The basic idea is to enumerate all possible compatible partitioning sets using dynamic programming to reduce the search space. The outline of the algorithm is given below:

1. For every query node  $Q_i$  in a query DAG, compute its compatible partitioning set  $PS(i)$  and  $cost(Qplan, PS(i))$ . Add non-empty  $PS(i)$  to a set of partitioning candidates.
2. Set  $PS$  to be  $PS(i)$  with minimum  $cost(Qplan, PS(i))$ .
3. For every candidate pair of partitioning sets  $PS(i)$  and  $PS(j)$  compute compatible partitioning set  $PS(i, j) = Reconcile\_Partn\_Sets(PS(i), PS(j))$  and  $cost(Qplan, PS(i, j))$ . Add non-empty  $PS(i, j)$  to a set of candidate pairs.
4. Set  $PS$  to be  $PS(i, j)$  with minimum  $cost(Qplan, PS(i, j))$ .
5. Similarly to previous step, expand candidate pairs of partitioning sets to candidate triples and compute corresponding reconciled partitioning sets and minimum cost.
6. Continue the iterative process until we exhaust the search space or end up with an empty list of candidates for the next iteration.

Since it is impossible for a partitioning set to be compatible with a node and not to be compatible with one of the node predecessors, we can use the following heuristics to further reduce the search space:

- Only consider leaf nodes for a set of initial candidates

- When expanding candidate sets only consider adding a node that is either an immediate parent of a node already in the set or is a leaf node.

## 6.5 Query Plan Transformation For a Given Partitioning

The query analysis framework presented in Section 6.4 provides a way to automatically infer the optimal partitioning scheme for a given set of streaming queries. In order to incorporate the results of the analysis into distributed query optimization, we need to make the optimizer fully aware of the partitioning scheme used. We implemented all partition-related optimizations as a set of transformation rules invoked by the query optimizer on compatible query nodes. All query transformation rules that we use work by replacing a qualifying subtree in query execution plan by equivalent optimized version (under the assumption that the input stream was partitioned using a compatible partitioning method).

As discussed earlier, we cannot assume that the partitioning scheme used by the actual system is identical to the optimal one recommended by the query analyzer. Therefore, the distributed query optimizer needs to take advantage of any partitioning that used by the system, even if it differs from the optimal one.

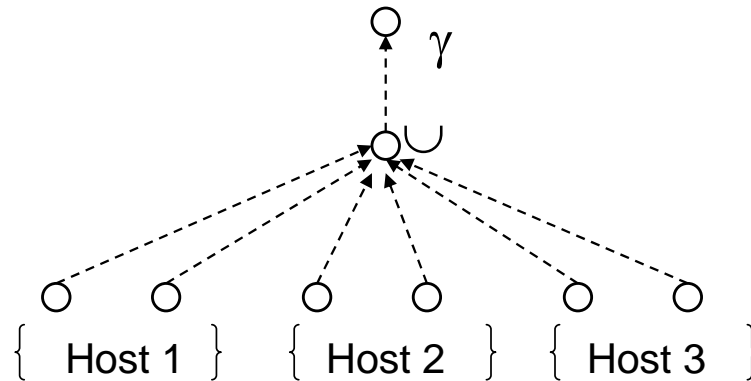
### 6.5.1 Algorithm for Performing Partition-related Query Plan Transformations

Our algorithm for transforming query execution plans based on available partitioning information consists of the following two phases:

#### **Build partition-agnostic query execution plan**

Let  $S$  be the partitioned source input stream consumed by a query set,  $S = \cup Partn_i$ . We construct a partition-agnostic query plan by creating an additional merge query node that computes a

stream union of all the partitions and making all query nodes that consume  $S$  read from the merge node. Since each host might have multiple CPUs/Cores, we can allocate multiple partitions to each participating host depending on the host capabilities. An example of a partition-agnostic plan for an aggregation query is shown in Figure 6-3. In this example an input stream  $S$  is split into 6 different partitions, with 2 partitions assigned to each host.



**Figure 6-3: Partition-agnostic query execution plan**

Even though such a query execution plan is clearly inefficient since it forces all the partitioned streams to be shipped to a single host before performing any processing, in the absence of any information about partitioning scheme used it is often the only feasible plan.

### **Perform query plan transformation in bottom-up fashion**

All transformation rules that we use for partition-related query optimization consist of two procedures: *Opt\_Eligible()* and *Transform()*. *Opt\_Eligible()* is a Boolean test that takes a query node and returns true if it is eligible for partition-related optimization. *Transform()* replaces the node that passed *Opt\_Eligible()* test by equivalent optimized plan. The pseudo code for query optimizer is given below:

1. Compute a topologically sorted list of nodes in the query DAG  $Q_1, Q_2, \dots, Q_n$  starting with the leaf nodes.

2. For every  $i \in [1 \text{ to } n]$

*If Opt\_Eligible( $Q_i$ )*

*Transform( $Q_i$ , Partitiong\_Info)*

Performing the transformation in a bottom-up fashion allows us to easily propagate the transformation compatible leaf nodes through the chain of compatible parent nodes. In the following section we will give a detailed description of the implementation of *Opt\_Eligible()* and *Transform()* for all major classes of query nodes – aggregations, joins and selection/projection.

### 6.5.2 Transformation for Aggregation Queries

The *Opt\_Eligible()* procedure for an aggregation query  $Q$  and partitioning set  $PS$  returns true if the following conditions are met:

- query  $Q$  has a single child node  $M$  of type merge (stream union)
- each child node of  $M$  is operating on single partition consistent with  $PS$
- $Q$  is compatible with  $PS$
- $Q$  is the only parent of  $M$

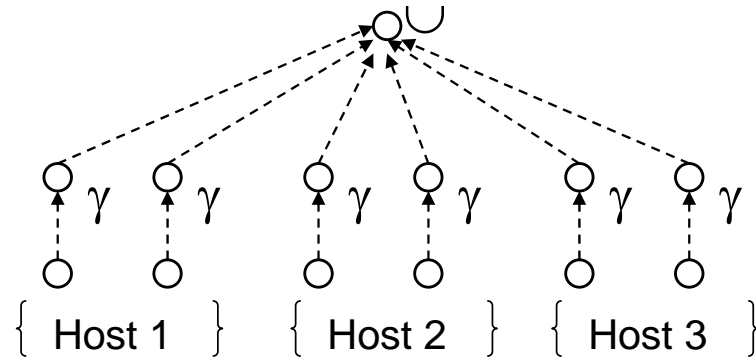
The last requirement is important to prevent the optimizer from removing the merge nodes that are used by multiple consumers. An example of a query node that stultifies all of the conditions required by *Opt\_Eligible()* is shown Figure 6-3.

#### 6.5.2.1 Transformation for compatible aggregation queries nodes

The main idea behind the *Transform()* procedure for eligible aggregation query  $Q$  is to push the aggregation operator below the merge  $M$  and allow it to execute independently on each of the



partitions. For each of the inputs of  $M$  we create a copy of  $Q$  and push it below the merge operator. The resulting optimized query execution plan is shown in Figure 6-4.



**Figure 6-4: Aggregation transformation for compatible nodes**

The correctness of the transformation follows directly from our definition of partition compatibility. Note, that data is fully aggregated before being sent to central node and does not require any additional processing.

### 6.5.2.2 Transformation for incompatible aggregation queries

For many aggregation queries that fail the *Opt\_Eligible()* test, we can still do better than use the default partition-agnostic query execution plan. The main idea behind the proposed optimization is the concept of partial aggregates. This idea is widely used in a number of streaming database engines [30][33], sensor networks [1][28] and traditional relational databases [79]. We illustrate this idea on a query that computes a count of number of packets sent between pairs of hosts:

Query **tcp\_count**:

```
SELECT time, srcIP, destIP, srcPort, COUNT(*)
FROM TCP
GROUP BY time, srcIP, destIP, srcPort
```

We can split **tcp\_count** into two queries called sub- and super-aggregate:

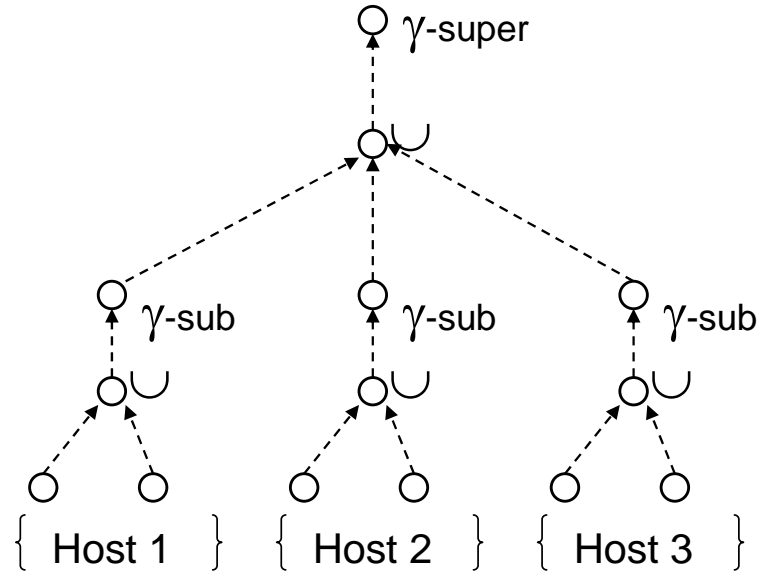
Query **super\_tcp\_count**:

```
SELECT time, srcIP, destIP, srcPort, SUM(cnt)
FROM sub_tcp_count
GROUP BY time, srcIP, destIP, srcPort
```

Query **sub\_tcp\_count**:

```
SELECT time, srcIP, destIP, srcPort, COUNT(*) as cnt
FROM TCP
GROUP BY time, srcIP, destIP, srcPort
```

All the SQL's built-in aggregates can be trivially split in a similar fashion. Many commonly used User Defined Aggregate Functions (UDAFs) can also be easily split into two components as was suggested in [33]. Note that we can push all the predicates in the query's WHERE clause to sub-aggregates, but all predicates in HAVING clause need complete aggregate values and therefore must be evaluated in super-aggregate. The query execution plan produced by this optimization is shown in Figure 6-5.



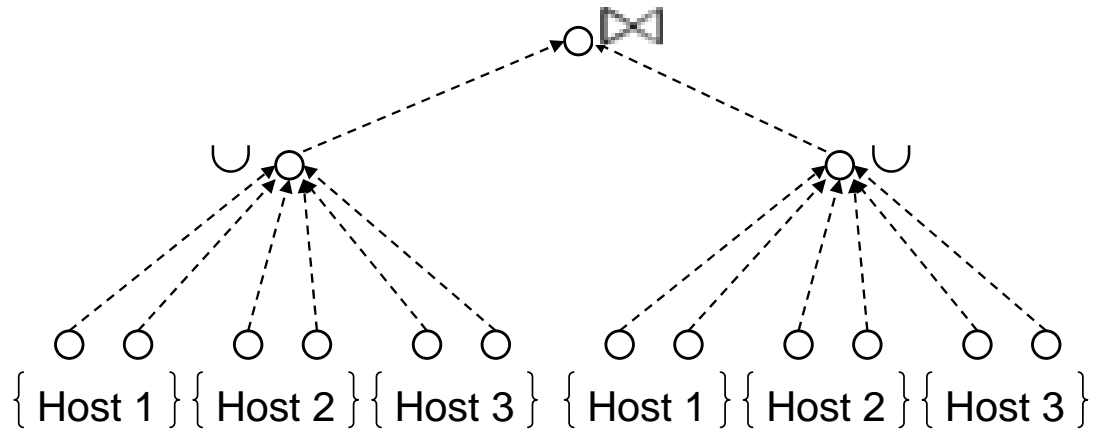
**Figure 6-5: Aggregation transformation for incompatible nodes**

### 6.5.3 Transformation for Join Queries

In this section, we will only consider two-way join queries, since all multi-way joins can be easily expressed by combination of two-way joins. The *Opt\_Eligible()* procedure for a join query  $Q$  and partitioning set  $PS$  returns true if the following conditions are met:

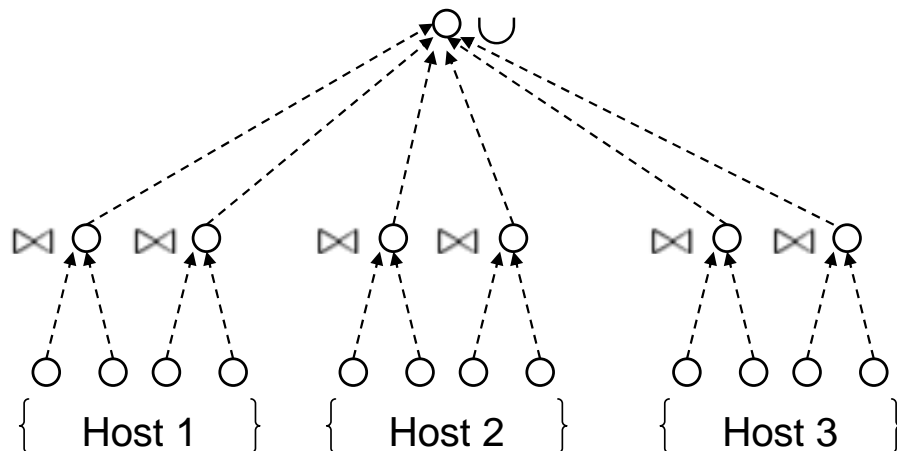
- query  $Q$  has a two children nodes  $M1$  and  $M2$  of type merge (stream union)
- each child node of  $M1$  and  $M2$  is operating on single partition consistent with  $PS$
- $Q$  is compatible with  $PS$
- $Q$  is the only parent of  $M1$  and  $M2$

An example query execution plan that satisfies *Opt\_Eligible()* test is shown in Figure 6-6.



**Figure 6-6: Original query execution plan**

The main idea behind the *Transform()* procedure for an eligible join query  $Q$  is to perform pair-wise joins for each of partition of input stream. This is accomplished by creating a copy of join operator and pushing it below the child merges. The left side partitions that do not have matching right side partitions and similarly unmatched right side partitions are ignored for inner join computations. For outer join computations, unmatched partitions are passed through special projection operator that adds appropriate NULL values needed by outer join. The output tuples produced by the projection operator are then merged with the rest of the final results. The resulting optimized query execution plan for inner-join query is shown in Figure 6-7.



**Figure 6-7: Join transformation for compatible nodes**

### 6.5.4 Transformations for Selection/Projection Queries

Selection/projection queries are always compatible with partition optimization and can be trivially pushed below child merge operators. Even though this transformation does not necessarily provides significant performance improvements, it is critical to ensure that partition-related optimization propagate further up the query tree.

## 6.6 Comparison of Semantic Sampling and Query-aware Stream Partitioning

Query-aware stream partitioning for distributed processing and semantic sampling framework presented in Chapter 5 are both based on the same analytical foundation. Both frameworks rely on partitioning of the input stream into disjoint groups of tuples when then are distributed across all participating nodes or selectively sampled to produce the approximate query answers. The definitions of grouping and partitioning compatibility are defined in a very similar fashion. In fact it easy to show that an individual query  $Q$  compatible with the sampling method that uses grouping set  $GS$  if and only if it is compatible with the partitioning using partitioning set  $GS$ . Similarly the reconciliation procedures defined by both frameworks can be shown to be equivalent to each other. This commonality in the analysis tools greatly simplifies that implementation of both subsystems within a data stream manager and allows for significant code reuse.

The main difference between the two frameworks lies in multi-query optimization aspects. Partial compatibility is not acceptable in semantic sampling framework since it would mean that some of the running queries could produce semantically incorrect results. The algorithm for computing compatible grouping sets for an arbitrary query DAG presented in section 5.5 has the freedom to choose different sampling strategy for each leaf node and even have multiple sampling methods

executed by the same node to guarantee the correctness of all the queries in a set. Furthermore the choice of the sampling method is not restricted by the limitations imposed by the partitioning hardware since it is implemented fully in the software by generating corresponding filtering predicates for the leaf nodes. Stream partitioning for distributed query processing on the other hand is restricted by both the partitioning hardware and the requirement the input stream can be partitioned only once. This property leads to different set of constraints and objectives for the query optimization, encapsulated in the algorithm given in section 6.5.

## 6.7 Experimental Evaluation

In this section, we present the result of experimental evaluation of query-aware partitioning in the context of the Gigascope streaming database. Gigascope fully supports distributed query evaluation using TCP as a protocol for transmitting tuples between cooperating hosts. We augmented Gigascope's query analysis framework to add support for stream partitions. We also modified the query optimizer to fully implement all query transformation rules and thus support partitioned evaluation of distributed queries.

All the experiments were conducted by replaying a one-hour trace of network packets and feeding it to a cluster of Gigascope nodes. The trace was obtained by combining four different one-hour traces captured concurrently using four data center taps. Each network tap captured two separate streams of packets for each traffic direction, each direction receiving approximately 100,000 packets/sec (about 400 Mbits/sec). We used a cluster of four dual core 3.0GHz Intel Xeon servers (2 cores per/CPU) with 4 GB of RAM running Linux 2.4.21. Servers were equipped with dual Intel(R) PRO/1000 network interface cards and were connected via Gigabit Ethernet LAN.

The goal of the experiments was to compare the performance of partition-agnostic query evaluation strategy with alternative strategies that take advantage of stream partitioning.

### 6.6.1 Partitioning for Simple Aggregation Queries

In this experiment, we observe how the performance of an aggregation query is affected by the choice of partitioning strategy. The query used in the experiment computes network traffic flows returning only suspicious flows that do not follow the TCP protocol (i.e. have an abnormal value of OR aggregate of TCP flags). In our packet trace, suspicious flows accounted for about 5% of the total number of flows. The corresponding GSQL statement for the query is shown below.

```
SELECT tb, srcIP, destIP, srcPort, destPort,
       OR_AGGR(flags) as orflag, COUNT(*), SUM(len)
FROM TCP
GROUP BY time as tb, srcIP, destIP, srcPort, destPort
HAVING OR_AGGR(flags) = #PATTERN#
```

We compared three different system configurations:

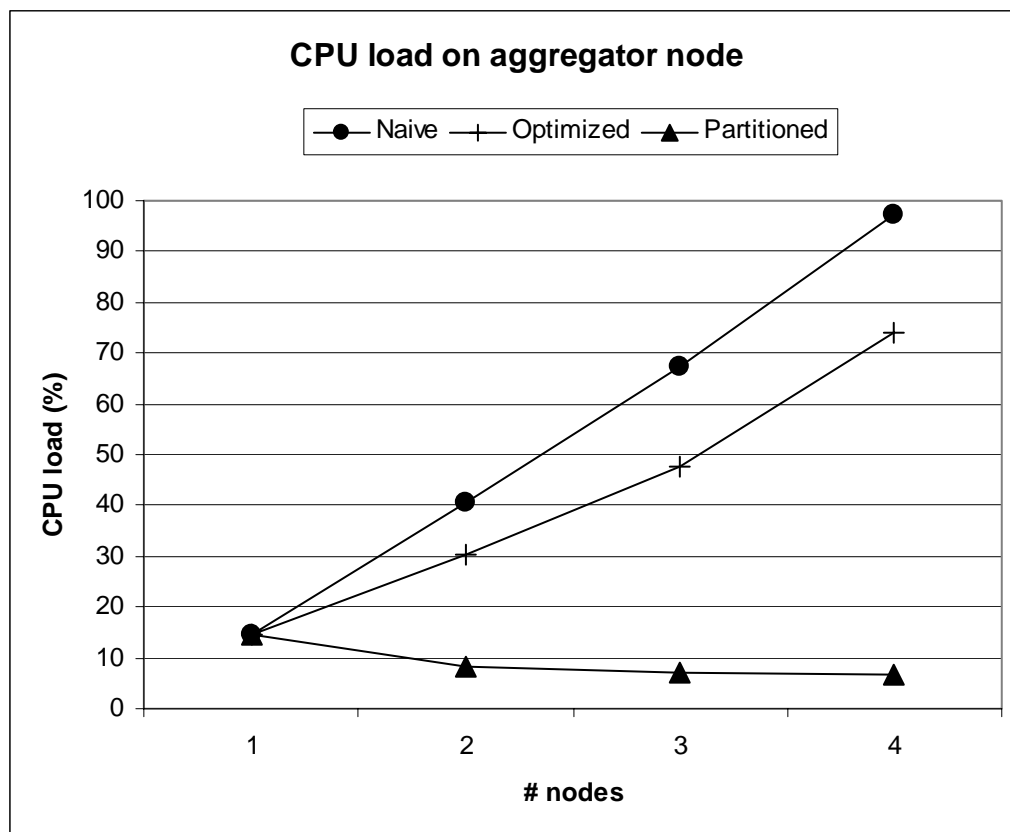
- a) Naïve – data stream is partitioned in a round robin fashion
- b) Optimized – data stream is partitioned round robin, but all the host's data is partially aggregated before being sent for final aggregation
- c) Partitioned – data stream is partitioned using optimal compatible partitioning set (srcIP, destIP)

We varied the number of machines in the cluster from 1 to 4 while varying the number of stream partitions from 2 to 8 respectively. In each experiment, we assign two partitions to each host to make better use of multiple processing cores. We will denote the host assigned to execute a root of the query tree as the *aggregator* node and to the rest of the nodes as leaf nodes.

In a course of the experiments, we observed that all three configurations are very effective at reducing the CPU load on leaf nodes. The load on each host drops from 80.4% to 23.9%

(combined CPU utilizations of the leaf nodes) as the number of hosts grows from 1 to 4. However, the load on the aggregator node shows completely opposite behavior. The results of the measuring the load on aggregator node are shown in Figure 6-8.

As we can observe from the graphs for Naïve configuration, the load grows linearly with a number of hosts and reaches almost 100% CPU utilization for 4 machines. At this point the system is clearly overloaded and starts dropping input tuples. Enabling partial aggregation helps reduce the load by 20-22% but overall trend of linear growth continues. The configuration using partitioning set recommended by the query analyser, on other hand, reduces the load on both aggregator and leaf nodes and enables true linear scaling.



**Figure 6-8: CPU load on aggregator node**



In addition to the CPU load on aggregator nodes, we also measured network load that query evaluation places on aggregator node. The results of the experiments are shown in Figure 6-9. As we can see from the graph, both partition-agnostic configurations suffer from transmitting the same partial flows to aggregator multiple times and exhibit linear grows in the network load. The slope of the Partitioned configuration is nearly flat with maximum network load limited by the cardinality of the query output.

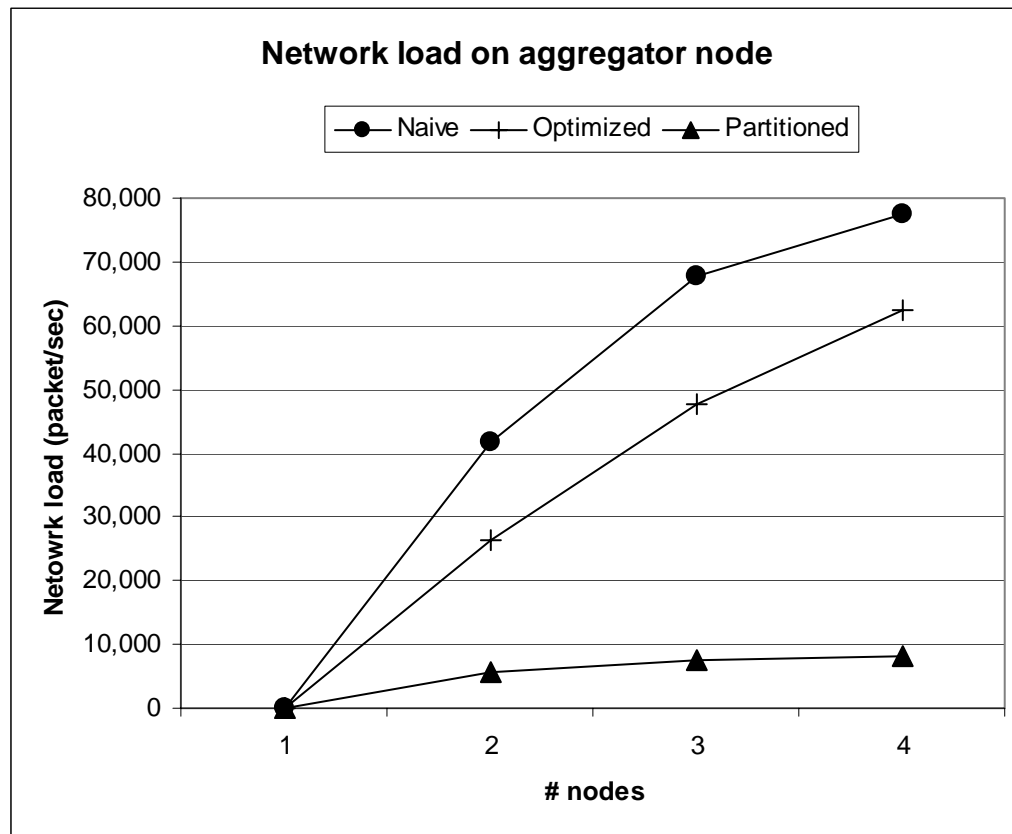


Figure 6-9: Network load on aggregator node

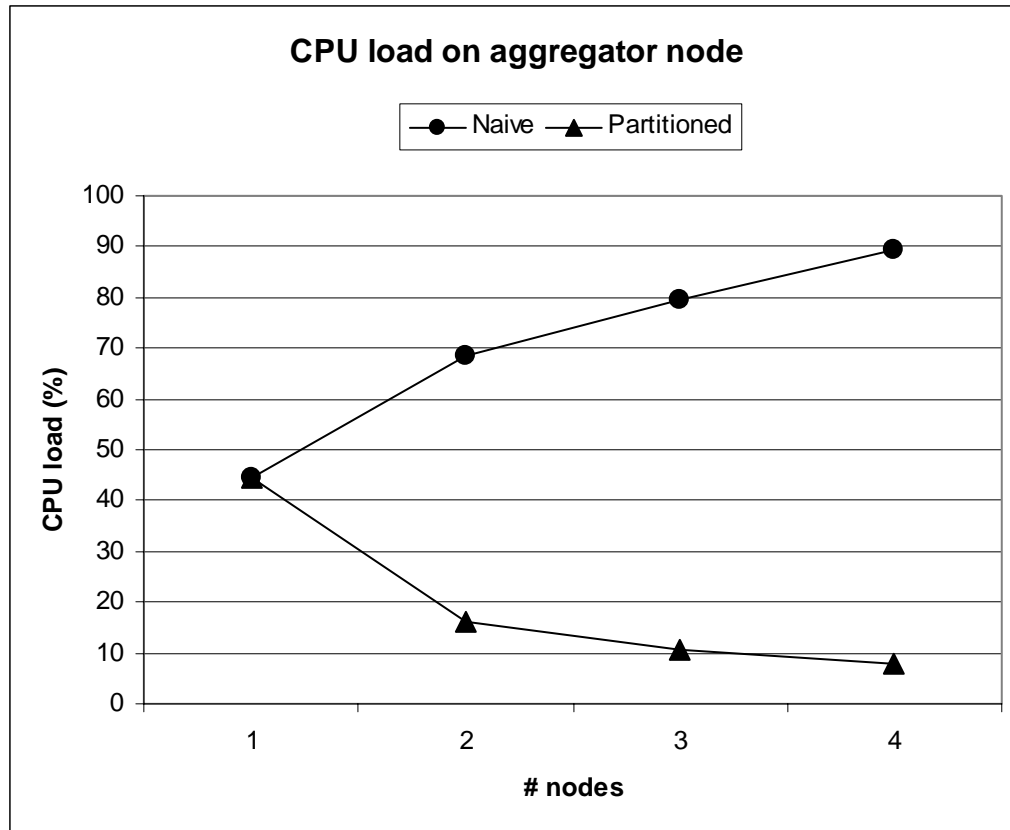
### 6.6.2 Partitioning for Join Queries

In our second set of experiments, we study the performance of a self-join query on both the partition-agnostic and partition-optimized configurations. The query used in the experiment

computes delays between consecutive TCP packets within the same traffic flow. This particular query is often used by network analysis for monitoring TCP session jitter. We only select a subset of the results with large TCP delays ( $> 1000\text{ms}$ ). A GSQL statement for the query is shown below.

```
SELECT S1.tb, S1.srcIP, S1.destIP, S1.srcPort,
      S1.destPort, (S2.timestamp-S1.timestamp) as delay
FROM TCP as S1, TCP as S2
WHERE S1.tb=S2.tb and S1.srcIP=S2.srcIP
      and S1.destIP=S2.destIP and
      S1.srcPort=S2.srcPort and
      S1.destPort=S2.destPort and
      (S1.seq_number+1)=S2.seq_number and
      (S2.timestamp - S1.timestamp) > 1000
```

We varied the number of machines in the cluster in the cluster from 1 to 4 with 2 partitions assigned to each host. Similar to the previous experiment, we observe that both configurations are very effective at reducing the CPU load on leaf nodes (load on drops from 77.4% to 19.7% as the number of hosts grows from 1 to 4). The results of the measuring the load on aggregator (root of the query tree) node are shown in Figure 6-10.



**Figure 6-10: CPU load on aggregator node**

As we can see from the graph, the load rises rapidly for the partitioning-agnostic scheme and reaches 90% CPU utilization for 4 participating hosts. These results are not surprising if we consider that leaf nodes cannot perform any filtering of the data and need to send the entire incoming tuple stream to an aggregator node. As the number of hosts increases, the number of remote tuples that need to be processed by the aggregator increases correspondingly. Since the cost of processing remote tuples is significantly higher than that of local ones (due to TCP overhead and extra memory copying), the load on the aggregator node rises with the number of participating hosts. The partition-optimized execution plan, on the other hand, effectively enables a linear scaling.

Figure 6-11 shows the results of the experiments measuring the network load on the aggregator node. Unable to do local join processing, the partition-agnostic configuration exhibits an almost linear increase in the network load. The partition-compatible configuration, on other hand, not only evaluates all the joins locally but can also apply an additional selection predicate ( $\text{delay} > 1000$ ) to further reduce the network load on aggregator. The resulting network load curve is almost flat, and in the worst case is limited by the cardinality of the output of the query.

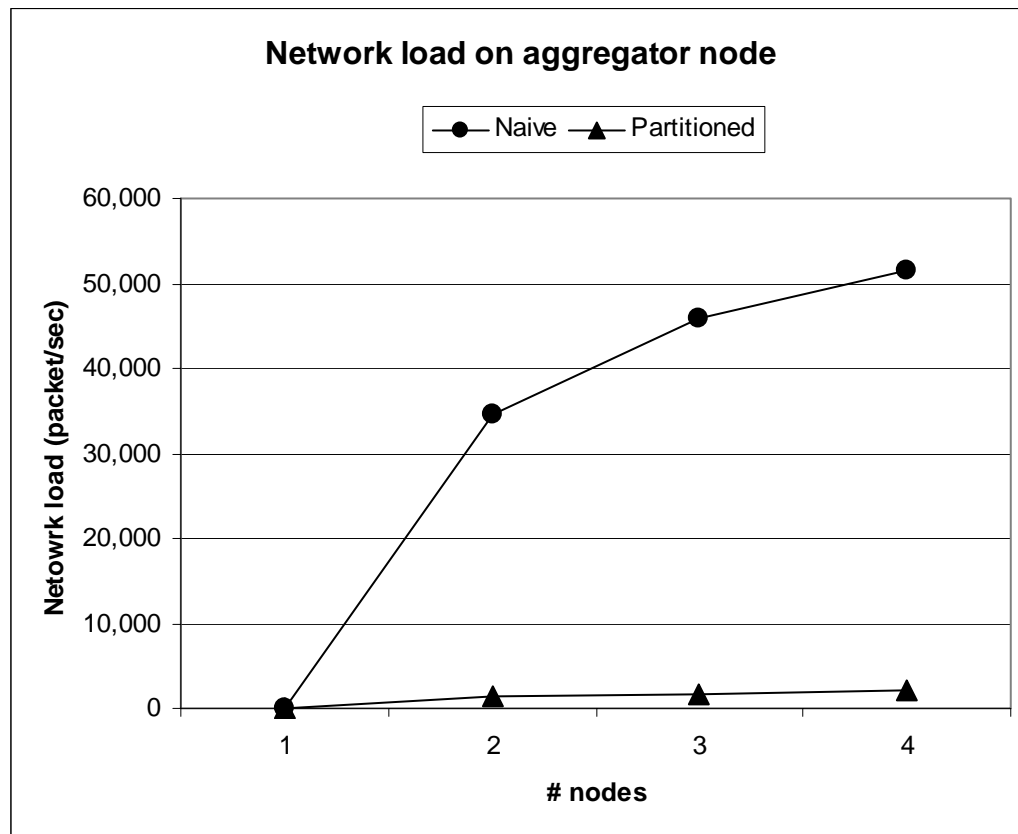


Figure 6-11: Network load on aggregator node

### 6.6.3 Partitioning for Query Sets

In this set of experiments, we study the performance of a query set consisting of independent aggregation and self-join queries. The aggregation query computes the statistics for packets sent

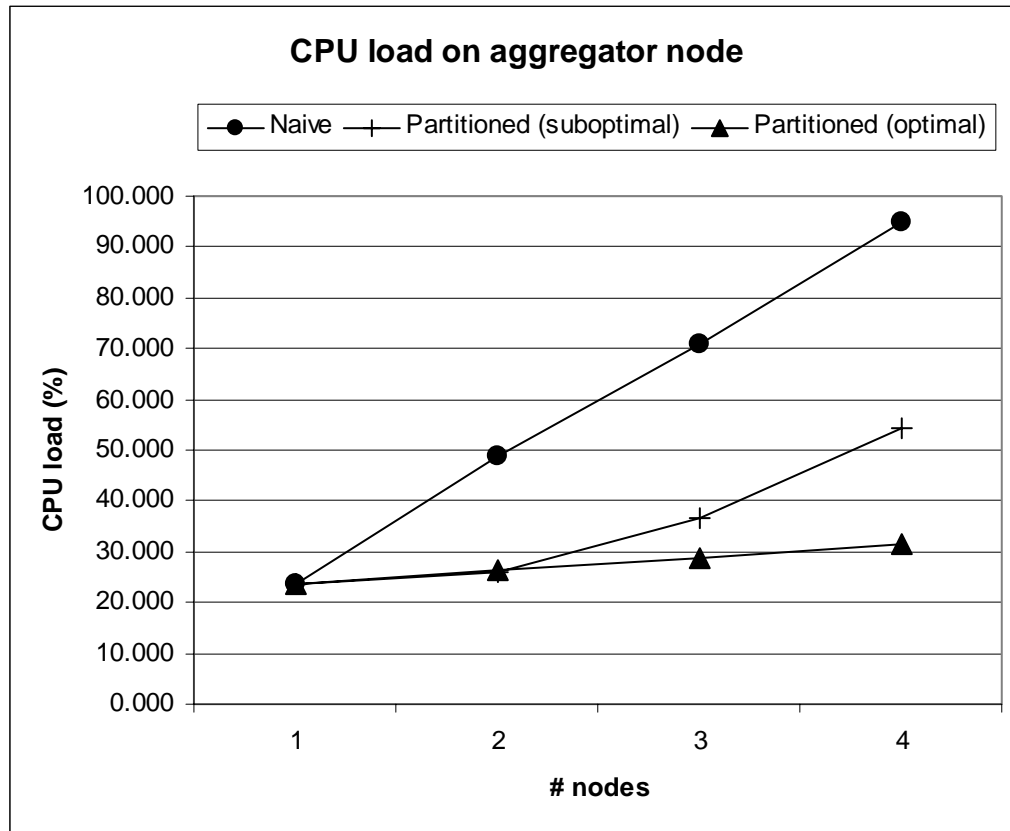
between the source subnets and destination hosts (grouping attributes are (srcIP & 0xFFFF0, destIP)). The self-join query computes delays between consecutive TCP packets within the same traffic flow. The optimal partitioning set for aggregation query is (srcIP & 0xFFFF0, destIP), while for the join query it is (srcIP, destIP, srcPort, destPort). We model a scenario where the restrictions of the partitioning hardware do not allow us to partition the data in a way that is compatible with both queries. According to the cost model presented in Section 4, the optimal partitioning set is (srcIP & 0xFFFF0, destIP), which is compatible only with the aggregation query.

We compared three different system configurations:

- a) Naïve – data stream is partitioned in a round robin fashion.
- b) Partitioned (suboptimal) – the data stream is partitioned using the suboptimal partitioning set (srcIP, destIP, srcPort, destPort) compatible with the join query
- c) Partitioned (optimal) – the data stream is partitioned using the optimal compatible partitioning set (srcIP & 0xFFFF0, destIP).

We varied the number of machines in the cluster in the cluster from 1 to 4 with 2 partitions assigned to each host. The results of the measuring the load on aggregator (root of the query tree) node are shown in Figure 6-12.

As we can see from the graph, the load rises rapidly for the partitioning-agnostic scheme and reaches 95% CPU utilization for 4 participating hosts. Suboptimally partitioned configuration compatible with the join query reduces the load by 43-47% reaching 54% utilization for a 4 host configuration. However, the linear load growth trend is still present due the fact since the workload is dominated by incompatible aggregation query. The load growth curve for the optimal partitioning scheme is much flatter, reducing the load to 31% for 4 host configuration.



**Figure 6-12: CPU load on aggregator node**

Figure 6-13 shows the results of the experiments measuring the network load on the aggregator node. Unable to perform any significant load reduction, the partition-agnostic configuration exhibits an almost linear increase in the network load. Suboptimal configuration, on other hand, evaluates all the joins locally and reduces the network load on aggregator node by 36-52% as the number of participating nodes increases to 4. The optimal configuration has an almost flat growth and effectively reduces the network load by 64-70% depending on number of hosts. These experiments demonstrate that our cost model correctly identifies the dominant queries in a query set and computes the globally optimal partitioning.

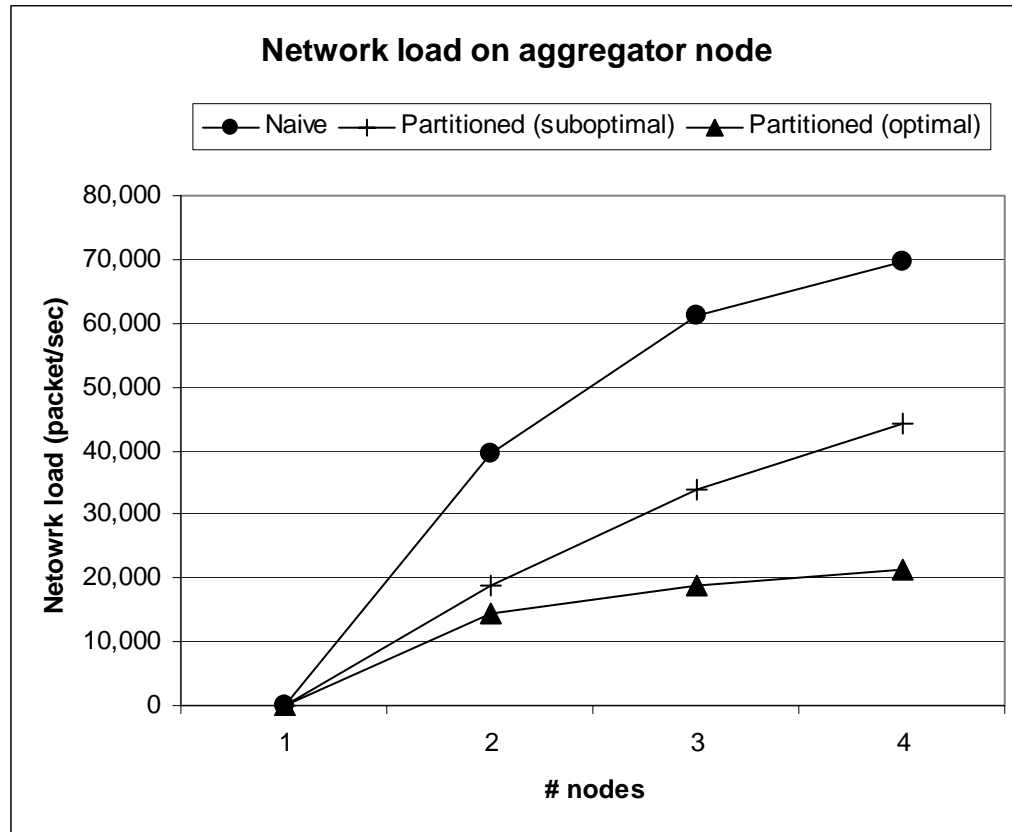


Figure 6-13: Network load on aggregator node

#### 6.6.4 Partitioning for Complex Queries

In the final set of experiments, we use a more complex query set involving multiple aggregation and join queries. This query set is identical to the one we used in Section 6.3 to illustrate query-aware partitioning framework. The corresponding GSQL statements for the queries are shown below.

Query flows:

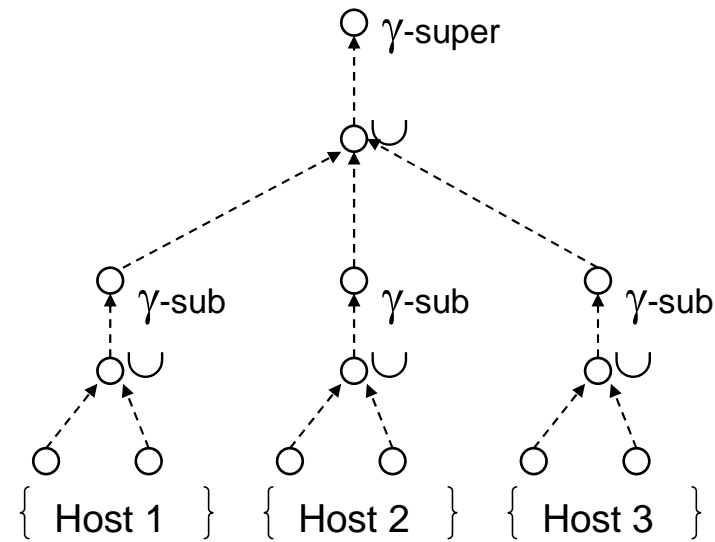
```
SELECT tb, srcIP, destIP, COUNT(*) as cnt
FROM TCP
GROUP BY time/60 as tb, srcIP, destIP
```

Query **heavy\_flows**:

```
SELECT  tb, srcIP, max(cnt) as max_cnt
FROM    flows
GROUP BY tb, srcIP
```

Query **flow\_pairs**:

```
SELECT S1.tb, S1.srcIP, S1.max_cnt, S2.max_cnt
FROM heavy_flows S1, heavy_flows S2
WHERE S1.srcIP = S2.srcIP and S1.tb = S2.tb+1
```



**Figure 6-14: Plan for partially compatible partitioning set**

We compared four different system configurations:

- a) Naïve – data stream is partitioned in a round robin fashion
- b) Optimized – data stream is partitioned round robin, all the host's data is partially aggregated before being sent for final aggregation



- c) Partitioned (partial) – the data stream is partitioned using the suboptimal partitioning set (srcIP, destIP)
- d) Partitioned (full) –the data stream is partitioned using the optimal compatible partitioning set (srcIP)

Note that in the Partitioned (partial) configuration, only query **flow** is compatible with partitioning set while the rest of the queries are incompatible. The query plan generated by the optimizer for suboptimal partitioning is shown in Figure 6-12. As in previous experiments we varied the number of machines in the cluster from 1 to 4 with 2 partitions assigned to each host.

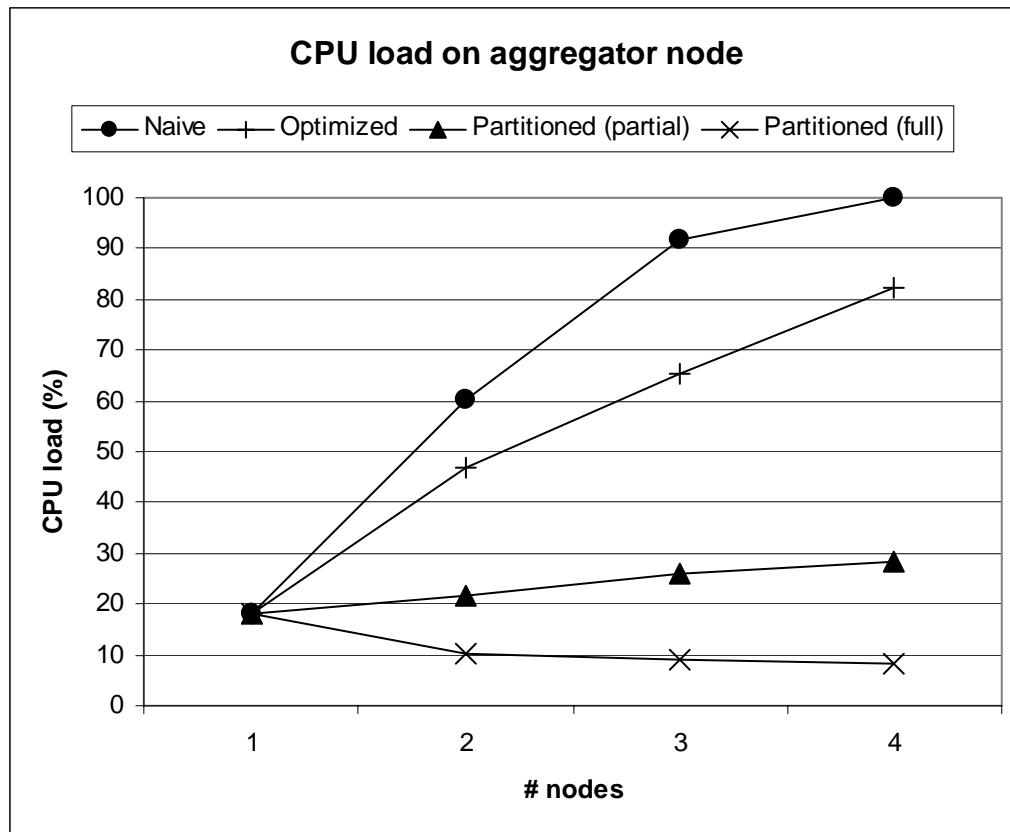


Figure 6-15: CPU load on aggregator node

Since the CPU load on leaf nodes followed the same patterns as in previously shown experiments, we concentrate on discussing the load on aggregator node. The results of the measuring the CPU load on aggregator (root of the query tree) node are shown in Figure 6-15.

As we can observe from the graphs for the Naïve configuration, the load on aggregator node grows linearly with a number of hosts. For a four machine configuration, the system overloaded and is forced to drop tuples from the input stream. The optimized configuration with partial aggregation enabled reduces the load by 23-24% reaching 82% utilization for a 4 host configuration. However, the linear load growth trend is still present and adding one more machine to the cluster will lead to the aggregator overload.

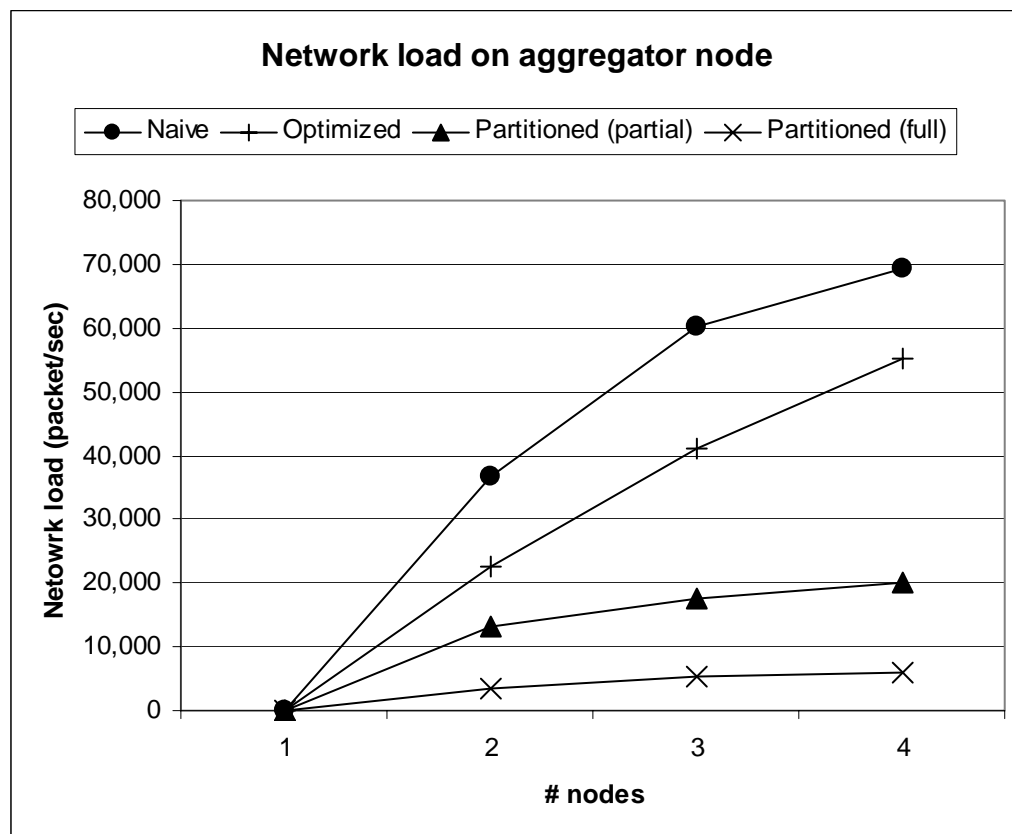


Figure 6-16: Network load on aggregator node

The load for the partially compatible configuration exhibits a nearly flat growth curve, primarily due to the fact that the most expensive query in a query set **flows** fully takes advantage of the compatible partitioning set. The load on aggregator node reaches only 18.4% which leaves a lot of room for further increase in the number of hosts. Finally, the fully compatible configuration exhibits true linear scaling, with the load on an aggregator node reaching 8.4% for a 4 machine setup.

Figure 6-16 shows the results of the experiments measuring the network load on the aggregator node. Here we observe the trends similar to previous experiments. Both Naïve and Optimized configuration with partial aggregates suffer from transmitting duplicate partial flows to the aggregator node and exhibit linear load growth. The partially and fully compatible configurations, on other hand, have flat growth curve with the maximum load approaching the cardinalities of **flows** and **flow\_pairs** respectively.

## 6.8 Summary

Distributed Data Stream Management Systems (DSMS) are increasingly used for processing of high-rate data streams. Two main approaches used to distribute the load across the cooperating machines are query plan partitioning and query-independent data stream partitioning. However, for a large class of queries both approaches fail to reduce the load compared to centralized system, and can even lead to increase in the load.

In this chapter, we introduce the idea of query-aware data stream partitioning that allows us to scale the performance of streaming queries in close to linear fashion. Our stream partitioning mechanism consists of two main components. The first component is a query analysis framework for determining the optimal partitioning for a given set of queries. The second component is a

partition-aware distributed query optimizer that transforms an unoptimized query plan into a semantically equivalent query plan that takes advantage of existing partitions.

We evaluate our query-aware partitioning approach by running sets of streaming queries of various complexities on a small cluster of processing nodes using high-rate network data streams. The results of our experiments confirm that the partitioning mechanism leads to highly efficient distributed query execution plans that scale linearly with the number of cooperating processing hosts. We also demonstrate that even suboptimal query-aware partitions offer significantly better performance than conventionally used query-independent partitioning.

## Chapter 7

# 7. Conclusions and Future Work

### 7.1 Dissertation Conclusions

Data Stream Management Systems (DSMS) are gaining acceptance for applications that require sophisticated processing of large volumes of data in real time. Monitoring potentially unbounded data streams presents a large number of unique problems, such as designing a programming language for querying infinite streams, developing query operators with bounded memory requirements, approximate query processing, and many others. One of the most important issues to be addressed by a streaming system is that of query optimizations. Having an effective query optimization mechanism is critical for dealing with extreme data rates, query sets consisting of hundreds of queries, and a large number of diverse data streams. The main contribution of this dissertation is to utilize semantic query analysis to perform query optimizations that enable scalable and robust data processing in the presence of high-rate streams and arbitrarily large query sets.

The first problem addressed by this dissertation is enabling DSMSs to simultaneously monitor and correlate large numbers of diverse data streams. The inherent burstiness and unpredictability of many real-world streams cause streaming operators to buffer input streams indefinitely and exceed the available system memory. The punctuation-carrying heartbeat mechanism proposed in

this dissertation guarantees that all streaming operators use a bounded amount of memory even in the presence of temporarily stalled data streams. The main idea behind the heartbeat mechanism is to generate special punctuation messages at low-level query nodes and propagate them throughout the entire query execution plan. Temporal information contained in the heartbeat messages allow streaming operators to unblock and purge all the state that would otherwise be indefinitely buffered. The proposed mechanism relies on a sophisticated timestamp analysis for aggressive but safe heartbeat generation.

Our experiments in Chapter 4 with multi-source streaming queries running over high-rate data streams demonstrate the effectiveness of the proposed mechanism even when some of the streams stall completely. We further demonstrated the validity of our approach by incorporating punctuating-carrying heartbeats into a production high-performance DSMS used for monitoring high-rate network streams.

In Chapter 5, we address the problem of handling bursts of high activity, during which the load on a streaming system can increase by an order of magnitude. We demonstrate that the widely used solution of uniform random sampling sacrifices the quality of the query answers to bring the system load to acceptable level. This dissertation proposes a general-purpose query-aware sampling framework for effective load reduction while still guaranteeing that the query answers remain semantically correct. We formally define the notion of query correctness in the presence of sampling, and give an algorithm for choosing the suitable sampling strategy for individual streaming queries. Further, this dissertation extends the single-query techniques to a general framework for analyzing any set of queries to determine a semantics-preserving sampling strategy. Since it is important for applications to know which sampling methods and sampling rates were used to compute the query results, we propose to use the heartbeat mechanism to embed special punctuations into query output stream that would contain this information.

We experimentally evaluated the semantic sampling approach by running various sets of streaming queries on high-rate network data streams. The results of the experiments presented in Chapter 5 demonstrate that proposed methods provides semantically correct and highly accurate results for scenarios where traditional approaches fails to provide semantically meaningful results.

Finally, Chapter 6 addresses the problem of scalable distributed processing in the presence of massive data streams. We first demonstrate that the query-independent stream partitioning mechanisms widely used by modern distributed DSMS fail to reduce the load of a large class of streaming queries as compared to centralized systems, and can even increase the load. Instead, this dissertation introduces the idea of query-aware data stream partitioning, which allows us to scale the performance of streaming queries in a close to linear fashion. We propose a query analysis framework for determining the optimal partitioning for an arbitrary large and complex query set. We also develop a partition-aware distributed query optimizer that transforms an unoptimized query plan into a semantically equivalent query plan that takes advantage of existing partitions.

This dissertation includes a comprehensive experimental evaluation of query-aware partitioning on a cluster of processing nodes using high-rate network data streams. The results of our experiments confirm that the partitioning mechanism leads to highly efficient distributed query execution plans that scale linearly with the number of cooperating processing hosts. We also demonstrate that even suboptimal query-aware partitions offer significantly better performance than conventionally used query-independent partitioning.

In summary, the contributions made by this dissertation in the area of streaming query optimization enable Data Stream Management Systems to scale to extreme data rates (up to 80Gbit/sec for network monitoring applications), gracefully handle overload conditions with no

resulting quality degradation, and support a large number of diverse input streams, enabling industrial-scale applications of DSMS technology. All of the presented techniques are generic and support arbitrary large and complex sets of streaming queries.

## 7.2 Directions for Future Research

Work presented in this dissertation can be further extended along the following lines:

1. **Handling partially disordered streams.** Most Data Stream Management Systems rely on temporal properties of the input streams for consistent query semantics. Specifically, a DSMS assumes the existence of a monotonically increasing timestamp or other application-defined time associated with every incoming tuple. However, when a streaming system is monitoring a large number of distributed data streams, there is significant possibility that input tuples will become partially disordered due to effects of routing and retransmissions over wide-area network. It is therefore important for a distributed DSMS to be capable of handling a small amount of disorder and still be capable of executing streaming operators with a bounded amount of memory. One possible solution would be to extend the heartbeat mechanism to embed appropriate punctuations specifying the level of out-of-orderness in the stream. All the streaming operators will also need to be extended to properly interpret such punctuations and use them to purge the runtime state.
2. **Scalable trigger processing.** There is a significant interest in a special class of stream monitoring applications that are interested in executing triggers – queries designed to notify subscribers that certain events occurred in the input streams. For such applications, a critical performance consideration is the latency of event reporting to the end user. One example of such an application is network intrusion detection software that needs to raise an alarm as soon as an attack intrusion is detected. Designing a scalable trigger processing system for stream applications poses a number of new challenges. The design of the streaming operators



- needs to be changed to shift the focus from optimal throughput to the latency of event detection and reporting. One possible solution would be to utilize heartbeat mechanism to propagate the event notifications throughout the query execution plans.
3. **Adaptive distributed query processing.** Streaming query optimizers critically depend on accurate statistics describing input stream to drive many optimization decisions, such as order of join evaluation, operator placement and stream partitioning. The inherent unpredictability of data streams and the long running nature of stream queries increase the possibility that once optimal query plans will become highly inefficient during the lifetime of the query. Fully dynamic and adaptive query execution environments that do not suffer from this problem are too inefficient to handle high-rate data streams [97]. In Chapter 4, we suggest that the heartbeat mechanism could be used to collect the comprehensive statistics required by distributed query optimizer. An interesting area of potential future research is in periodic query plan reoptimization driven by these runtime statistics.
  4. **Extending the suite of sampling algorithms used by semantic sampling.** Query-aware sampling framework suggested in Chapter 5 relies on small suite of algorithms: per-tuple and per-group sampling. A large number of alternative sampling algorithms has been suggested in the literature, including reservoir sampling [120], geometric sampling [16][65], importance-based sampling [45], and many others. It would be interesting to extend the semantic sampling framework to incorporate large number of sampling methods and being able to reason about interaction of various sampling methods when used within the same query set.
  5. **Dynamic stream repartitioning.** Our query-aware stream partitioning mechanism relies on a good hash function to evenly distribute the input stream among all participating streaming engines. Significant fluctuations in the values of the partitioning attributes can easily lead to a load imbalance and even to the overload of individual streaming engines. One extension of our partitioning mechanism would be to incorporate dynamic stream repartitioning based on

- the runtime statistics collected from distributed nodes. The challenges here are efficient distributed statistics collection in the presence of high stream data rates, the design of the feedback mechanism to drive repartitioning, and changing the partitioning scheme in the middle of operator execution.
6. **Using partitioning-based optimizations for processing non-streaming data.** A large number of one-pass algorithms were developed in the context of Data Stream Management Systems for efficient query evaluation over unbounded data stream. Recently, there has been a significant interest in applying some of the techniques developed for stream processing for one-pass processing on non-streaming data (for example in active data warehousing applications). Similar to streaming systems, these applications have the properties of soft real-time systems and can benefit from a load shedding mechanism to deal with overload conditions. An interesting research direction would be to investigate using semantic sampling to help queries meet the execution deadlines while still maintaining the semantic correctness of the output. Furthermore, the partitioning analysis framework used for distributed evaluation of streaming queries can be applied to the problem of horizontally partitioning the stored datasets based on the structure of submitted queries and their expected frequencies in the workload.
  7. **Combining semantic sampling with query-aware stream partitioning.** Semantic sampling and query-aware streaming partitioning frameworks are designed to solve different problems in data stream management and can in principle be applied independently of each other. However, combining both mechanisms in a high-performance distributed DSMS raises an issue of how these mechanisms should interact under overload conditions. One possible alternative is to always apply the partitioning first and make individual host independently apply semantic sampling to react to overload conditions. Alternatively, one can try to react to overload conditions by sampling the data before it is distributed to the participating query

processing nodes, and avoid sending the tuple only to have it immediately discarded by the remote host. Exploring the performance tradeoffs for the two design options and potential for combining both mechanisms in the single partitioning hardware will be an interesting research direction.

## Bibliography

- [1] D. Abadi et al. The design of the Borealis stream processing engine. In *Proc. Biennial Conference on Innovative Data Systems Research (CIDR)*, pages 277-289, 2005.
- [2] D. Abadi et al. Aurora: A new model and architecture for data stream management. *The VLDB Journal*, 12(2):120-139, August 2003.
- [3] D. J. Abadi, W. Lindner, S. Madden, and J. Schuler. An Integration Framework for Sensor Networks and Data Stream Management Systems. Demonstration. In *Proc. International Conference on Very Large Data Bases (VLDB)*, 2004.
- [4] S. Acharya, P.B. Gibbons, V. Poosala, S. Ramaswamy. Join synopses for Approximate Query Answering. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 275-286, 1999.
- [5] Aleri. <http://www.alerilabs.com>
- [6] A. Arasu et al. STREAM: The Stanford stream data manager. *IEEE Quarterly Bulletin on Data Engineering*, 26(1):19–26, 2003.
- [7] A. Arasu, B. Babcock, S. Babu, J. McAlister, J. Widom. Characterizing Memory Requirements for Queries over Continuous Data Streams. In *Proc. ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, pages 221-232, 2002.
- [8] A. Arasu, S. Babu and J. Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *The VLDB Journal*, 2005.
- [9] A. Arasu and J. Widom. Resource sharing in continuous sliding-window aggregates. In *Proc. International Conference on Very Large Data Bases (VLDB)*, pages 336–347, 2004.
- [10] R. Avnur and J. Hellerstein. Eddies: Continuously adaptive query processing. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 261-272, 2000.
- [11] B. Babcock, S. Babu, M. Datar, R. Motwani: Chain: Operator Scheduling for Memory Minimization in Data Stream Systems. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 253-264, 2003.

- [12] B. Babcock, S. Babu, M. Datar, R. Motwani, and D. Thomas. Operator Scheduling in Data Stream Systems. *The VLDB Journal*, 2005.
- [13] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proc ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, pages 1–16, 2002.
- [14] B. Babcock, M. Datar, R. Motwani: Load Shedding for Aggregation Queries over Data Streams. In *Proc. International Conference on Data Engineering (ICDE)*, pages 350-361, 2004.
- [15] S. Babu, U. Srivastava, J. Widom. Exploiting k-constraints to reduce memory overhead in continuous queries over data streams. *ACM Transactions Database Systems*, 29(3):545-580, Sep 2004.
- [16] A. Bagchi, A. Chaudhary, D. Eppstein, and M. Goodrich. Deterministic sampling and range counting in geometric streams. *ACM Symp Computational Geometry*, 2004.
- [17] Y. Bai, H. Thakkar, H. Wang, C. Zaniolo. A Flexible Query Graph Based Model for the Efficient Execution of Continuous Queries. SSPS 2007.
- [18] M. Balazinska, H. Balakrishnan, M. Stonebraker. -Based Load Management in Federated Distributed Systems. In *Proc. of the First Symposium on Networked Systems Design and Implementation (NSDI)*, 2004.
- [19] L. Bent, M. Rabinovich, G. M. Voelker, Z.Xiao. Characterization of a large web site population with implications for content delivery, WWW 2004: 522-533.
- [20] L. Bent, M. Rabinovich, G. M. Voelker, Z.Xiao. Towards Informed Web Content Delivery. WCW 2004: 232-248.
- [21] D. Carney et al. Monitoring streams - a new class of data management applications. In *Proc. International Conference on Very Large Data Bases (VLDB)*, pages 215–226, 2002.
- [22] D. Carney, U. Cetintemel, A. Rasin, S. B. Zdonik, M. Cherniack, M. Stonebraker. Operator Scheduling in a Data Stream Manager, In *Proc. International Conference on Very Large Data Bases (VLDB)*, 2003.
- [23] S. Chandrasekaran et al. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proc. Biennial Conference on Innovative Data Systems Research (CIDR)*, 2003.
- [24] S. Chandrasekaran and M. J. Franklin. PSoup: a system for streaming data. *The VLDB Journal*, 12(2):140-156, Aug 2003.
- [25] S. Chandrasekaran, M. J. Franklin. Streaming Queries over Streaming Data. In *Proc. International Conference on Very Large Data Bases (VLDB)*, pages 203-214, 2002.

- [26] S. Chandrasekaran, S. Krishnamurthy, S. Madden, A. Deshpande, M. J. Franklin, J. M. Hellerstein, M. Shah. Windows Explained, Windows Expressed, 2003, Available at <http://www.cs.berkeley.edu/~sirish/research/streaquel.pdf>
- [27] S. Chaudhuri, R. Motwani and V. Narasayya. Random sampling for histogram construction: How much is enough? In *Proc. ACM SIGMOD International Conference on Management of Data*, 1998, 436-447.
- [28] J. Chen, D.J. DeWitt, F. Tian and Y. Wang, NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 379-390, 2000.
- [29] J. Chen, D.J. DeWitt, J. F. Naughton. Design and Evaluation of Alternative Selection Placement Strategies in Optimizing Continuous Queries. In *Proc. International Conference on Data Engineering (ICDE)*, pages 345-356, 2002.
- [30] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. Zdonik. Scalable Distributed Stream Processing. In *Proc. Biennial Conference on Innovative Data Systems Research (CIDR)*, 2003.
- [31] E. Cohen and M. Strauss. Maintaining time-decaying stream aggregates. In *Proc. ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, pages 223-233, 2003.
- [32] G. Cormode, M. N. Garofalakis, D. Sacharidis. Fast Approximate Wavelet Tracking on Streams. In *Proc. International Conference on Extending Database Technology (EDBT)*, pages 4-22, 2006.
- [33] G. Cormode, T. Johnson, F. Korn, S. Muthukrishnan, O. Spatscheck, and D. Srivastava. Holistic UDAFs at streaming speeds. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 35-46. 2004.
- [34] G. Cormode, T. Johnson, F. Korn, S. Muthukrishnan, and D. Srivastava. Effective Computation of Biased Quantiles over Data Streams. In *Proc. International Conference on Data Engineering (ICDE)*, pages 20-31, 2005.
- [35] G. Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. In *Proc. Latin American Theoretical Informatics (LATIN)*, pages 29-38, 2004.
- [36] G. Cormode and S. Muthukrishnan. What is hot and what is not: Tracking most frequent items dynamically. In *Proc. ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, 2003.
- [37] C. Cortes, K. Fisher, D. Pregibon, A. Rogers, and F. Smith. Hancock: A language for extracting signatures from data streams. In *Proc. International Conference on Knowledge Discovery and Data Mining*, pages 9-17, 2000.

- [38] C. Cranor, T. Johnson, O. Spatscheck, and V. Shkapenyuk. Gigascope: high performance network monitoring with an SQL interface. In *Proc. ACM SIGMOD International Conference on Management of Data*, page 262, 2002.
- [39] C. Cranor, T. Johnson, O. Spatscheck, and V. Shkapenyuk. Gigascope: A stream database for network applications. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 647–651, 2003.
- [40] C. Cranor, T. Johnson, O. Spatscheck, and V. Shkapenyuk. The Gigascope stream database. *IEEE Data Engineering Bulletin*, 26(1): pages 27–32, 2003.
- [41] M. Datar and S. Muthukrishnan. Estimating rarity and similarity on data stream windows. *Proc. ESA*, 2002. 323-334.
- [42] A. Deshpande, C. Guestrin, and S. Madden. Using probabilistic models for data management in acquisitional environments. In *Proc. Biennial Conference on Innovative Data Systems Research (CIDR)*, pages 317-328, 2005.
- [43] Digital Sky, [http://www.cacr.caltech.edu/SDA/digital\\_sky.html](http://www.cacr.caltech.edu/SDA/digital_sky.html)
- [44] L. Ding, N. Mehta, E. Rundersteiner, G. Heineman. Joining Punctuated Streams. In *Proc. International Conference on Extending Database Technology (EDBT)*, 2004.
- [45] N. Duffield, C. Lund, M. Thorup. Learn more, sample less: control of volume and variance in network measurement. *SIGCOMM 2001 Measurement workshop*.
- [46] K. Finkenzeller. RFID Handbook: Fundamentals and Applications in Contactless Smart Cards and Identification. John Wiley and Sons, 2003.
- [47] P. Flajolet and G. N. Martin. Probabilistic counting. In *Proc. IEEE Conference on Foundations of Computer Science*, pages 76–82, 1983.
- [48] M. N. Garofalakis, J. Gehrke, R. Rastogi. Querying and Mining Data Streams: You Only Get One Look: A Tutorial. In *Proc. ACM SIGMOD International Conference on Management of Data*, page 635.
- [49] Gemfire. <http://www.gemstone.com>
- [50] R. Gemulla, H. Berthold, and W. Lehner. Hierarchical group-based sampling. In *BNCOD*, pages 120-132, 2005.
- [51] L. Golab, K. G. Bijay, M. T. Özsu. Multi-Query Optimization of Sliding Window Aggregates by Schedule Synchronization, In *Proc. Int. Conf. on Information and Knowledge Management (CIKM)*, November 2006.
- [52] L. Golab, M. T. Özsu. Issues in Data Stream Management. *ACM SIGMOD Record*, Volume 32, Number 2, June 2003, pages 5-14.

- [53] L. Golab, M. T. Özsu. Processing Sliding Window Multi-Joins in Continuous Queries over Data Streams. In *Proc. International Conference on Very Large Data Bases (VLDB)*, pages 500-511, 2003.
- [54] H. Gonzalez, J. Han, X. Li, and D. Klabjan. Warehousing and analyzing massive RFID data sets, In *International Conference on Data Engineering*, 2006.
- [55] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: a relational aggregation operator generalizing group-by, cross-tab, and sub-totals. In *Proc. of the 12th Intl.Conf. on Data Engineering*, pages 152–159, 1996.
- [56] M. Greenwald and S. Khanna. Space-Efficient Online Computation of Quantile Summaries. In *Proc. ACM SIGMOD International Conference on Management of Data*, 2001.
- [57] R. Greer. Daytona and the Fourth Generation Language Cymbal. In *Proc. ACM SIGMOD International Conference on Management of Data*, 1999.
- [58] P. Gibbons. Distinct Sampling for Highly-Accurate Answers to Distinct Values Queries and Event Reports. In *Proc. International Conference on Very Large Data Bases (VLDB)*, pages 541-550, 2001.
- [59] P. B. Gibbons and Y. Matias. New Sampling-Based Summary Statistics for Improving Approximate Query Answers. In *Proc. ACM SIGMOD International Conference on Management of Data*, 1998.
- [60] P. Gultzan and T. Pelzer. SQL-99 Complete, Really. CMP Books, 1999.ISBN: 0-87930-568-1.
- [61] L.Guo, S. Chen, Z. Xiao, and X. Zhang. Analysis of multimedia workloads with implications for internet streaming. WWW 2005: 519-528.
- [62] L.Guo, S. Chen, Z. Xiao, and X. Zhang. DISC: Dynamic Interleaved Segment Caching for Interactive Streaming. ICDCS 2005: 763-772.
- [63] M. Hammad et al. Nile: A Query Processing Engine for Data Streams. Demonstration In *Proc. International Conference on Data Engineering (ICDE)*, page 851, 2004.
- [64] M. Hammad, M. J. Franklin, W. Aref, and A. Elmagarmid. Scheduling for shared window joins over data streams. In *Proc. International Conference on Very Large Data Bases (VLDB)*, pages 297-308, 2003.
- [65] J. Hershberger and S. Suri. Adaptive Sampling for Geometric Problems over Data Streams. In *Proc. ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, 2004.
- [66] Informal discussions with AT&T network analysts.



- [67] M. Ivanova and T. Risch. Customizable Parallel Execution of Scientific Stream Queries. In *Proc. International Conference on Very Large Data Bases (VLDB)*, 2005.
- [68] S. Jeffery, G. Alonso, M. Franklin, W. Hong, and J. Widom. A pipelined framework for on-line cleaning of sensor data streams. In *Proc. International Conference on Data Engineering (ICDE)*, page 140, 2006.
- [69] T. Johnson, S. Muthukrishnan, I. Rozenbaum. Sampling Algorithms in a Stream Operator. In *Proc. ACM SIGMOD International Conference on Management of Data*, 2005.
- [70] T. Johnson, S. Muthukrishnan, V. Shkapenyuk, O. Spatscheck. A Heartbeat Mechanism and its Application in Gigascope. In *Proc. International Conference on Very Large Data Bases (VLDB)*, 2005.
- [71] T. Johnson, S. Muthukrishnan, V. Shkapenyuk, O. Spatscheck. Query-Aware Partitioning for High-Rate Data Streams. Submitted to *International Conference on Very Large Data Bases (VLDB)*, 2007.
- [72] T. Johnson, S. Muthukrishnan, V. Shkapenyuk, O. Spatscheck. Query-Aware Sampling for Data Streams. SSPS 2007.
- [73] J. Jung, B. Krishnamurthy, and M. Rabinovich. Flash Crowds and Denial of Service Attacks: Characterization and Implications for CDNs and Web Sites. WWW 2002.
- [74] J. Kang, J. F. Naughton, S. Viglas: Evaluating window joins over unbounded streams. In *Proc. International Conference on Data Engineering (ICDE)*, pages 37-48, 2003.
- [75] R. R. Kompella, S. Singh, and G. Varghese. On scalable attack detection in the network. In ACM Internet Measurement Conference IMC 2004, pages 187 - 200.
- [76] N. Koudas and D. Srivastava. Data stream query processing: A tutorial. In *Proc. International Conference on Very Large Data Bases (VLDB)*, page 1149, 2003.
- [77] J. Krämer and B. Seeger. A temporal foundation for continuous queries over data streams. COMAD 2005, pages 70-82.
- [78] S. Krishnamurthy, M. Franklin, J. Hellerstein, and G. Jacobson. The case for precision sharing. In *Proc. International Conference on Very Large Data Bases (VLDB)*, pages 972-986, 2004.
- [79] Per-Ake Larson. Data Reduction by Partial Preaggregation. In *Proc. International Conference on Data Engineering (ICDE)*, 2002.
- [80] Y. N. Law, H. Wang, C. Zaniolo: Query Languages and Data Models for Database Sequences and Data Streams. In *Proc. International Conference on Very Large Data Bases (VLDB)*, 2004.

- [81] A. Lerner, D. Shasha. AQuery: Query Language for Ordered Data, Optimization Techniques and Experiments Query Language for Ordered Data, Optimization Techniques, and Experiments. In *Proc. International Conference on Very Large Data Bases (VLDB)*, pages 345-356, 2003.
- [82] A. Lerner and D. Shasha. The virtues and challenges of ad hoc + streams querying in finance. *IEEE Quarterly Bulletin on Data Engineering*, 26(1):49-56, 2003.
- [83] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, Peter A. Tucker. Semantics and Evaluation Techniques for Window Aggregates in Data Streams. In *Proc. ACM SIGMOD International Conference on Management of Data*, 2005.
- [84] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, Peter A. Tucker. No Pane, No Gain: Efficient Evaluation of Sliding-Window Aggregates over Data Streams. *ACM SIGMOD Record*, 2005.
- [85] LOFAR, <http://www.lofar.nl/>
- [86] S. Madden and M. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *Proc. International Conference on Data Engineering (ICDE)*, 2002.
- [87] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. The design of an acquisitional query processor for sensor networks. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 491-502, 2003.
- [88] S. Madden, M. Shah, J. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 49-60, 2002.
- [89] G. Manku and R. Motwani. Approximate frequency counts over data streams. In *Proc. International Conference on Very Large Data Bases (VLDB)*, pages 346-357, 2002.
- [90] P. McDaniel, S. Sen, O. Spatscheck, J. Van der Merwe, W. Aiello, and C. Kalmanek. Enterprise security: A community of interest based approach. In *Proc. NDSS*, Feb.2006.
- [91] The Medusa project, <http://nms.lcs.mit.edu/projects/medusa/>.
- [92] S. Muthukrishnan. Data Streams: Algorithms and Applications. Foundations and Trends in Theoretical Computer Science, Vol 2, 2005.
- [93] S. Nath, P. Gibbons, S. Seshan, and Z. Anderson. Synopsis diffusion for robust aggregation in sensor networks. In *Proc. ACM Conf. on Embedded Networked Sensor Sys. (SENSYS)*, pages 250-262, 2004.
- [94] National Geodetic Survey, <http://www.ngs.noaa.gov/>
- [95] New York Stock Exchange.

<http://www.nyse.com/marketinfo/datalib/1022221393023.html#dlyvolume>

- [96] V. Paxson. Some Not-So-Pretty Admissions About Dealing With Internet Measurements, Workshop on Network-Related Data Management (NRDM 2001), May, 2001
- [97] T. Plagemann, V. Goebel, A. Bergamini, G. Tolu, G. Urvoy-Keller, E.W. Biersack. Using Data stream Management for Traffic Analysis – A Case Study. *LNCS Springer Proceedings Passive and Active Measurement Workshop*, Juans-les-Pins, France, April 2004
- [98] J. Rao, C. Zhang, N. Megiddo, G. M. Lohman: Automating physical database design in a parallel database. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 558-569, 2002.
- [99] A. Reibman, S. Sen and J. van der Merwe. Analyzing the spatial quality of Internet streaming video. Workshop on Video Processing and Quality Metrics for Consumer Electronics (VPQM), Scottsdale, AZ, January 2005.
- [100] F. Reiss, J. M. Hellerstein. Data Triage: An Adaptive Architecture for Load Shedding in TelegraphCQ. In *Proc. International Conference on Data Engineering (ICDE)*, pages 155-156, 2005.
- [101] F. Reiss, J.M. Hellerstein: Declarative Network Monitoring with an Underprovisioned Query Processor. In *Proc. International Conference on Data Engineering (ICDE)*, page 56, 2006.
- [102] M. Roughan, S. Sen, Oliver Spatscheck, N. G. Duffield: Class-of-service mapping for QoS: a statistical signature-based approach to IP traffic classification. Internet Measurement Conference 2004: 135-148.
- [103] S. Sen, O. Spatscheck and D. Wang. Accurate, Scalable In-Network Identification of P2P Traffic Using Application Signatures. Proceedings International WWW Conference, New York, USA, May 2004.
- [104] R. Sidhu and V. K. Prasanna. Fast regular expression matching using FPGAs. In IEEE Symposium on Field- Programmable Custom Computing Machines, Rohnert Park, CA, USA, April 2001.
- [105] M. A. Shah, J. M. Hellerstein, and E. Brewer. Highly available, fault-tolerant, parallel data flows. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 827-838, 2004.
- [106] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, M. J. Franklin. Flux: An Adaptive Partitioning Operator for Continuous Query Systems. In *Proc. International Conference on Data Engineering (ICDE)*, 2003
- [107] Sloan Digital Sky Survey, <http://www.sdss.org/>

- [108] M. Stonebraker and U. Cetintemel. One Size Fits All: An Idea Whose Time has Come and Gone. In *Proc. International Conference on Data Engineering (ICDE)*, 2005.
- [109] U. Srivastava, J. Widom. Flexible Time Management in Data Stream Systems, In *Proc. ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, pages 263-274, 2004.
- [110] U. Srivastava and J. Widom. Memory-Limited Execution of Windowed Stream Joins. In *Proc. International Conference on Very Large Data Bases (VLDB)*, 2004.
- [111] Streambase. <http://www.streambase.com>
- [112] M. Sullivan and A. Heybey. Tribeca: A system for managing large databases of network traffic. In *Proc. USENIX Annual Technical Conference*, 1998
- [113] N. Tatbul, U. Çetintemel, S. B. Zdonik, M. Cherniack, M. Stonebraker: Load Shedding in a Data Stream Manager. In *Proc. International Conference on Very Large Data Bases (VLDB)*, pages 309-320, 2003
- [114] N. Tatbul and S. Zdonik. Window-aware Load Shedding for Aggregation Queries over Data Streams. In *Proc. International Conference on Very Large Data Bases (VLDB)*, 2006.
- [115] P. A. Tucker, D. Maier, T. Sheard , L. Fegaras. Exploiting Punctuation Semantics in Continuous Data Streams, *IEEE Transactions on Knowledge and Data Engineering*, v.15 n.3, p.555-568, March 2003.
- [116] Unidata, <http://www.unidata.ucar.edu/data/data.detail.html>
- [117] P. Verkaik, O. Spatscheck, J. van der Merwe, and A. Snoeren. PRIMED: A Community-of-Interest-Based DDoS Mitigation System, SIGCOMM LSAD 2006.
- [118] S. Viglas, J. F. Naughton: Rate-based query optimization for streaming information sources. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 37-48, 2002.
- [119] S. Viglas, J. F. Naughton, J. Burger: Maximizing the Output Rate of Multi-Way Join Queries over Streaming Information Sources. In *Proc. International Conference on Very Large Data Bases (VLDB)*, 2003.
- [120] J. S. Vitter. Random sampling with reservoir. *ACM Transactions on Mathematical Software*, 11(1):37-57, 1985.
- [121] H. Wang, C. Zaniolo, and R. C. Luo. ATLaS: A Turing-complete extension of SQL for data mining applications and streams. Available at <http://wis.cs.ucla.edu/publications.html>, 2002.
- [122] J. Widom *et al.* Stanford stream data manager. <http://www-db.stanford.edu/stream/sqr>, 2003.

- [123] E. Wu, Y. Diao, S. Rizvi. High-Performance Complex Event Processing over Streams. In *Proc. ACM SIGMOD International Conference on Management of Data*, 2006.
- [124] R. Zhang, N. Koudas, B. C. Ooi, D. Srivastava. Multiple Aggregations Over Data Streams. In *Proc. ACM SIGMOD International Conference on Management of Data*, 2005.
- [125] Q. Zhao, A. Kumar, J. Wang, J. Xu. Data streaming algorithms for accurate and efficient measurement of traffic and flow matrices. In *Proc. ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 350-361, 2005.
- [126] Y. Zhou, Y. Yan, F. Yu, A. Zhou. PMJoin: Optimizing Distributed Multiway Stream Joins by Stream Partitioning. In *Proc. International Conference on Database Systems for Advanced Applications (DASFAA)*, 2006.
- [127] Y. Zhu, E. Rundensteiner, and G. Heineman. Dynamic plan migration for continuous queries over data streams. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 431-442, 2004.
- [128] Y. Zhu and D. Shasha. Fast approaches to simple problems in financial time series streams. In *Proc. of the Workshop on Management and Processing of Data Streams (MPDS)*, pages 181-192, 2003.

## Curriculum Vita

### VLADISLAV SHKAPENYUK

#### **EDUCATION:**

01/2005 – 05/2007	Rutgers University Ph. D program in Computer Science Thesis Supervisor: Muthu S. Muthukrishnan
09/2002 - 12/2004	Carnegie Mellon University MS in Computer Science
09/1998 - 01/2001	Polytechnic University, Brooklyn Campus BS Summa Cum Laude in Computer Science
09/1995 - 07/1997	Polytechnic University, Odessa, Ukraine Major: Computer Science

#### **WORK EXPERIENCE:**

01/2005 – Present	Researcher/Programmer
06/2001 – 08/2002	AT&T Labs Research, Florham Park, NJ
	<ul style="list-style-type: none"> <li>• Worked on internals of Gigascope, a data stream management system for analyzing high speed IP traffic streams             <ul style="list-style-type: none"> <li>- Developed and implemented several key components: code generation for high-level queries, sampling for load shedding during high load, query optimization for running multiple Gigascopes in parallel, as well as performance analysis framework</li> <li>- Contributed code to many other components of the system, from runtime support to query translation and analysis</li> </ul> </li> <li>• Developed IPScope, a system for searching, managing, and analyzing patent text databases             <ul style="list-style-type: none"> <li>- Implemented and adapted various algorithms for bibliographic analysis, patent classification and clustering</li> <li>- Designed and implemented flexible GUI to simplify patent analysis tasks</li> </ul> </li> <li>• Developed graphical management tool for Daytona, a proprietary massively scalable database             <ul style="list-style-type: none"> <li>- Implemented scheme and data partition browsing, data sampling and flexible querying modules</li> </ul> </li> <li>• Contributed code to various internal database tools</li> </ul>

- |                   |   |
|-------------------|---|
| 01/2001 – 06/2001 | <p>Programmer<br/>Polytechnic University, Brooklyn, NY</p> <ul style="list-style-type: none"> <li>• Developed a system for information delivery to mobile clients using broadcast and multicast over the wireless network</li> <li>• Developed a high-performance distributed web crawler in C++ and Python</li> </ul>  |
| 04/1998 – 12/2000 | <p>Web Applications Developer<br/>The LanguageWorks Inc., New York, NY</p> <ul style="list-style-type: none"> <li>• Designed and implemented multilingual database applications for Lotus Domino Server</li> <li>• Developed software to automate software localization process: translation management system, import/export utilities, HTML editing tools using Lotus Domino and Java</li> <li>• Designed and implemented various Netscape Plug-Ins and Internet Explorer ActiveX modules to extend the browser functionality for handling Adobe PDF files</li> </ul> |

## **PUBLICATIONS:**

Query-Aware Sampling for Data Streams. Theodore Johnson, S. Muthukrishnan, Vladislav Shkapenyuk, Oliver Spatscheck. *International Workshop on Scalable Stream Processing Systems (SSPS)*, 2007

Simultaneous Pipelining in QPipe: Exploiting Work Sharing Opportunities Across Queries. K. Gao, S. Harizopoulos, I. Pandis, V. Shkapenyuk, A. Ailamaki. *International Conference on Data Engineering (ICDE)*, 2006

A Heartbeat Mechanism and Its Application in Gigascope. T. Johnson, S. Muthukrishnan, V. Shkapenyuk, O. Spatscheck. *International Conference on Very Large Data Bases (VLDB)*, 2005

QPipe: A Simultaneously Pipelined Relational Query Engine. Stavros Harizopoulos, Vladislav Shkapenyuk, Anastassia Ailamaki. *International Conference on Management of Data (SIGMOD)*, 2005

Finding (Recently) Frequent Items in Distributed Data Streams. Amit Manjhi, Vladislav Shkapenyuk, Kedar Dhamdhere, Christopher Olston. *International Conference on Data Engineering (ICDE)*, 2005

The Gigascope Stream Database. C. Cranor, T. Johnson, O. Spatscheck, V. Shkapenyuk. *Data Engineering Bulletin*, March 2003

Gigascope: A Stream Database for Network Applications. T. Johnson, C. Cranor, O. Spatscheck, V. Shkapenyuk. *International Conference on Management of Data (SIGMOD)*, 2003

Gigascope: high performance network monitoring with an SQL interface. C. Cranor, Y. Gao, T. Johnson, V. Shkapenyuk, Oliver Spatscheck. *International Conference on Management of Data (SIGMOD)*, 2002

Mining database structure; or, how to build a data quality browser. T. Dasu, T. Johnson, S. Muthukrishnan, V. Shkapenyuk. *International Conference on Management of Data (SIGMOD)*, 2002

Design and Implementation of a High-Performance Distributed Web Crawler. V. Shkapenyuk, T. Suel. *International Conference on Data Engineering (ICDE)*, 2002