Abstract of “Fast and Highly-Available Stream Processing” by Jeong-Hyon Hwang, Ph.D., Brown University, May 2009.

Recently, there has been significant interest in applications where high-volume, continuous data streams need to be processed with low latency. Such applications include financial market monitoring, network monitoring, intrusion detection, call analysis, battlefield monitoring, asset tracking, and ecosystem monitoring. Since these applications monitor real-time events, the value of a result decays rapidly over time. Therefore, low-latency processing is a key requirement.

Stream processing systems enable efficient implementation of the aforementioned applications. Currently, many such systems are geared toward distributed processing because a large number of applications inherently involve geographically dispersed data sources and the processing capability of a system improves as more servers are used. However, the more computation and communication resources, the higher the odds of failure. In stream processing, a failure prevents low-latency processing because it blocks the flow of data streams. To make matters worse, it may also result in losing data essential to producing correct results.

In this dissertation, we propose various techniques that realize both reliable and timely processing of data streams in the face of server and network failures. We first discuss our basic recovery approaches, while comparing them in terms of recovery speed, CPU and network utilization, as well as their relationship to various recovery semantics. Next, we describe a fast recovery technique for commodity server clusters. In this technique, operators on each server are backed up on different servers and thus can be recovered in parallel. This technique assigns backup servers and schedules checkpoints in a manner that maximizes the recovery speed. Finally, we discuss our approach for Internet-scale stream processing. In this approach, multiple operator replicas send outputs to downstream replicas, allowing each replica to use whichever data arrives first. To further reduce latency, replicas run without coordination, possibly processing data in different orders. Despite this relaxation, the approach guarantees that applications always receive the same results as in the non-replicated, failure-free case. It also deploys replicas at locations that effectively improve performance and availability. Our experimental results demonstrate the effectiveness of the approaches above. These results were obtained from a server cluster at Brown University and a worldwide network testbed called PlanetLab.

Fast and Highly-Available Stream Processing

by
Jeong-Hyon Hwang
B. S., Korea University, Seoul, Korea, 1998
M. S., Korea University, Seoul, Korea, 2000
M. S., Brown University, Providence, RI, USA, 2003

A dissertation submitted in partial fulfillment of the
requirements for the Degree of Doctor of Philosophy
in the Department of Computer Science at Brown University

Providence, Rhode Island
May 2009

© Copyright 2009 by Jeong-Hyon Hwang

This dissertation by Jeong-Hyon Hwang is accepted in its present form by the Department of Computer Science as satisfying the dissertation requirement for the degree of Doctor of Philosophy.

Date ________________ ________________________________

Stan Zdonik, Director

Recommended to the Graduate Council

Date ________________ ________________________________

Uğur Çetintemel, Reader

Date ________________ ________________________________

John Jannotti, Reader

Approved by the Graduate Council

Date ________________ ________________________________

Sheila Bonde

Dean of the Graduate School

Acknowledgements

I would never have finished this dissertation without the support and encouragement of my colleagues, friends, and family.

First of all, I would like to express my deep gratitude to my advisor, Stan Zdonik. He guided my Ph.D. study with passion and insightful advice on identifying and tackling valuable research problems, evaluating solutions, and presenting research outcomes. I will always be grateful for the time and effort that he invested in me. All the lessons that I learned from him will be priceless assets for my future academic career.

My Ph.D. work would not have been the same without Uğur Çetintemel. He has constantly inspired me and made significant technical contributions to my research. I would like to thank him for his humor, kindness, support, and invaluable advice drawn upon his own experience as an international student and a junior professor.

I am very grateful to John Jannotti for being on my thesis committee and sharing his expertise in computer networks. He raised interesting questions and gave me helpful advice in the course of my research. All of them have been of great value to this dissertation.

The Aurora/Borealis project has been an exceptional opportunity for me. I would like to thank all the team members, including Professors Michael Stonebraker, Hari Balakrishinan, Sam Madden, and Mitch Cherniack, and colleagues, Daniel Abadi, Yanif Ahmad, Magdalena Balazinksa, Brad Berg, Don Carney, Christian Convey, Eddie Galvez, Wolfgang Linder, Anurag Maskey, Olga Papaemmanouil, Alexander Rasin, Esther Ryvkina, Jon Salz, Nesime Tatbul, Richard Tibetts, Robin Yan, Wenjuan Xing, and Ying Xing. I would also like to thank other colleagues in the Brown Database Group, including Mert Akdere, Tingjian Ge, Alptekin Kupcu, Jennie Rogers, and Sanghoon Cha. I owe special thanks to Michael Stonebraker and Hari Balakrishnan for their guidance to my early research, Magdalena Balazinska for her significant contribution to my work and recent support for my job search, Nesime Tabul for her advice that helped me throughout my graduate years, Don Carney for his humor, Robin Yan and Yanif Ahmad for their friendship, Ying Xing, Olga Papaemmanouil, and Jennie Rogers for being wonderful officemates, and Sanghoo Cha for his contribution to my SIGMOD 2008 demonstration.

I would also like to thank all the members of the Brown Korean community. Too many Korean friends to list here helped my wife and me settle down in Providence and raise two children. Their friendship and support indeed made our life enjoyable.

Finally, I am deeply indebted to my family. I thank my parents, Sang-Ro Hwang and Keum-Soon Kim, and my younger brother, Ji-Hyon, for their consistent love and support. Most importantly, I would like to express my deepest gratitude to my wife, Hyung Hwa, and my children, Yaurie and Andrew. Without their love and patience, this work would never have come into existence.

Credits

This dissertation is based on several papers jointly written with Professors Stan Zdonik, Uğur Çetintemel, Michael Stonebraker, and other members of the Aurora/Borealis project. Chapter 2 is based on our design papers on the Aurora system [32, 2] and the Borealis system [1, 41]. Chapters 3 and 4 are based on our ICDE 2005 paper [68] and ICDE 2007 paper [69], respectively. Finally, Chapter 5 is based on our ICDE 2008 paper [71], SIGMOD 2008 demonstration [72], and a workshop paper [70].

The Aurora/Borealis project has been supported by the National Science Foundation under Grant No. IIS-0086057 and IIS-0325838.

To my wife, Hyung Hwa, and children, Yaurie and Andrew.

Contents

List of Tables xii

List of Figures xiii

1 Introduction 1

1.1 Stream Processing . . . 1

1.2 High-Availability Challenges in Stream Processing . . . 6

1.3 Contributions . . . 9

1.3.1 Basic Approaches for Highly-Available Stream Processing . . . 9

1.3.2 Highly-Available Stream Processing in Server Clusters . . . 10

1.3.3 Fast and Highly-Available Stream Processing over the Internet . . . 10

1.4 Organization . . . 11

2 Background 12

2.1 Stream Data Model . . . 12

2.2 Query Model . . . 12

2.2.1 Stateless Operators . . . 13

2.2.2 Stateful Operators . . . 14

2.3 Quality of Service . . . 15

2.4 The Aurora System . . . 16

2.5 The Borealis System . . . 18

2.5.1 The Overall Architecture . . . 18

2.5.2 The Architecture of a Borealis Node . . . 20

2.6 Failure Models . . . 20

2.6.1 Server Failures . . . 21

2.6.2 Network Failures . . . 21

3 Basic Approaches for Highly-Available Stream Processing 22

3.1 Assumptions . . . 24

3.2 High-Availability Semantics . . . 25

3.2.1 Recovery Types . . . 25

3.2.2 Operator Classification 27
3.3 Gap Recovery 28
3.4 No Loss Recovery Protocols 29
3.4.1 Passive Standby 30
3.4.2 Upstream Backup 31
3.4.3 Active Standby 35
3.5 Extensions for Precise Recovery 36
3.5.1 Passive Standby 36
3.5.2 Active Standby 37
3.5.3 Upstream Backup 37
3.6 Evaluation 38
3.6.1 Runtime Overhead vs Recovery Time 38
3.6.2 Cost of Precise Recovery 40
3.6.3 Effects of Query Network Type on Recovery 41
3.6.4 Size of Query Network State 42
3.6.5 Rate of Query Network State Change 43
3.6.6 Effect of Network Size 43
3.6.7 Discussion 43
3.7 Summary 44

4 Highly-Available Stream Processing in Server Clusters 46
4.1 Our Backup Model 47
4.1.1 Assumptions 48
4.1.2 The Basic Architecture 48
4.1.3 No Loss Guarantee 50
4.1.4 Non-Failure Time Operation for High Availability 50
4.1.5 Failure and Recovery 51
4.2 Formation of Checkpoint Units 52
4.2.1 Impact of Load Management Principles 52
4.2.2 Merging Checkpoint Units 52
4.2.3 Safety during Checkpoint Unit Reformation 53
4.3 Checkpoint Scheduling 53
4.3.1 Choosing the Best Capture Task 53
4.3.2 Capture vs Paste 55
4.3.3 The Complete Scheduling Algorithm 55
4.3.4 Discussion 56
4.4 Dynamic Backup Assignment 57
4.4.1 Determining Backup Load Imbalance 57
4.4.2 The Backup Reassignment Algorithm 57
4.5 Delta Checkpointing 59

4.5.1 Aggregate 60
4.5.2 Join 60
4.6 Experimental Results 61
4.6.1 The Setup 61
4.6.2 Checkpointing Costs 61
4.6.3 Recovery Time and End-to-End Latency 63
4.6.4 Scheduling and Backup Assignment 64
4.6.5 Effects of Different Query Networks 65
4.6.6 Processing Load 66
4.7 Summary 66
4.8 Formal Definitions of Terms used in Chapter 4 67
4.8.1 Formal Definition of Recovery Time 67
4.8.2 Expected Recovery Time after a Checkpoint 69

5 Fast and Highly-Available Stream Processing over the Internet **71**
5.1 Background 72
5.1.1 Motivating Examples 72
5.1.2 Assumptions 74
5.1.3 The Basic Architecture 74
5.1.4 Problem Statements 75
5.2 Replication Transparency 75
5.3 Extension for Replication Transparency 77
5.3.1 Management of Punctuations 77
5.3.2 Duplicate Filtering 78
5.3.3 Sorting Streams 79
5.3.4 Stateless Operators and Replica Consistency 79
5.3.5 Extending Join for Replica Consistency 80
5.3.6 Extending Aggregate for Replica Consistency 80
5.4 Management of Replicas 82
5.4.1 Deployment of Replicas 82
5.4.2 Garbage Collection 83
5.4.3 Adaptation to Changes 85
5.5 Experimental Results 85
5.5.1 The Setup 87
5.5.2 Comparison of Techniques for Reliable Stream Processing 87
5.5.3 Impact of Replication on Latency 89
5.5.4 CPU cost for Replications 90
5.5.5 Impact of Replica Deployment 90
5.5.6 Impact of Garbage Collection 92
5.6 Summary 93

6 Related Work **94**
6.1 High Availability in Database Management Systems . . . 94
6.1.1 Disk-Based Rollback Recovery . . . 94
6.1.2 Process-Pair . . . 95
6.1.3 High-Availability Techniques for Distributed Database Systems . . . 96
6.1.4 Persistent Queues . . . 96
6.1.5 High Availability in Workflow Systems . . . 97
6.2 High-Availability Techniques in Distributed Systems . . . 97
6.2.1 Rollback Recovery . . . 97
6.2.2 Reliable Group Communication . . . 99
6.3 Stream Processing . . . 99
6.3.1 Stream Processing Systems . . . 100
6.3.2 Operator Placement . . . 101
6.3.3 High Availability in Stream Processing . . . 102

7 Conclusion **104**
7.1 Basic Approaches for Highly-Available Stream Processing . . . 104
7.2 Highly-Available Stream Processing in Server Clusters . . . 105
7.3 Fast and Highly-Available Stream Processing over the Internet . . . 106
7.4 Future Directions . . . 107

Bibliography **109**

List of Tables

3.1 Summary of Notation 29
3.2 Types of No Loss Recovery 29
3.3 Recovery Time and Runtime Overhead of each High-Availability Approach 30
3.4 Added Overhead for Precise Recovery 36
3.5 Simulation Parameters and Their Default Values 38
3.6 Effects of Query Network Type 41
3.7 Effects of Query Network State Size 42
3.8 Effects of Rate of Query Network State Change 42

5.1 CPU cost of a Replica (% CPU cycles) 89

List of Figures

1.1 Stream rate variations in network traffic traces . . . 2
1.2 The Paradigm Shift of Data Management . . . 3
1.3 Query Example . . . 3
1.4 Query Distribution Example . . . 4

2.1 Query Network Example . . . 13
2.2 Quality of Service . . . 16
2.3 Aurora System Architecture . . . 17
2.4 Borealis System Architecture . . . 18
2.5 Borealis Node Architecture . . . 19

3.1 Primary/Backup Pairs . . . 24
3.2 Examples of Outputs under Each Recovery Type . . . 26
3.3 Taxonomy of Aurora Operators . . . 27
3.4 Communication between Servers in Upstream Backup . . . 32
3.5 One Iteration of Upstream Backup . . . 33
3.6 CPU Overhead and Recovery Time . . . 39
3.7 Bandwidth Overhead and Recovery . . . 39
3.8 Bandwidth Overhead and Recovery Time (the communication interval varies from 25 ms to 200 ms as indicated by the arrows) . . . 40
3.9 Effects of the Number of Operators (the arrows indicate the directions of the trends) 44

4.1 The Backup Model . . . 49
4.2 Recovery Time (Round-Robin) . . . 54
4.3 Recovery Time (Min-Max) . . . 54
4.4 Backup Reassignment . . . 58
4.5 Checkpoint Costs . . . 62
4.6 Recovery Time . . . 63
4.7 End-to-End Processing Latency . . . 64
4.8 Scheduling & Backup Assignment Effects . . . 65
4.9 Query Network and Recovery Time . . . 66

4.10 Processing Load and Recovery Time 67

5.1 Non-Replicated Stream Processing. 73
5.2 Replicated Stream Processing 73
5.3 Borealis-R Visualizer (Latency) 86
5.4 Borealis-R Visualizer (Network Cost) 86
5.5 Experimental Setup 88
5.6 Comparison of Reliability Techniques 89
5.7 Impact of Replication on Latency 90
5.8 Impact of Replica Deployment 91
5.9 Impact of Garbage Collection 92

Chapter 1

Introduction

This dissertation focuses on the problem of coping with slow or failed servers and networks in the context of stream processing. In this chapter, we first introduce the area of stream processing (Section 1.1). Next, we describe the challenges that motivated this dissertation (Section 1.2) and highlight our contributions made in this dissertation (Section 1.3). Finally, we conclude this chapter with a brief outline of this dissertation (Section 1.4).

1.1 Stream Processing

In recent years, we observed a new suite of applications that must process high-speed data streams with low latency. For example, advances in wireless networking and miniaturization enabled various applications that continuously monitor the physical world through small embedded sensors [137, 79], location-sensing devices [66, 63, 108], and electronic tags [54]. These applications include sensor-based environment monitoring (e.g., highway traffic monitoring [13], habitat monitoring [131], building monitoring [50], seismic activity monitoring [106]), RFID-based asset tracking [55], GPS-based location tracking [92], military applications (e.g., platoon tracking, target detection), and medical applications (e.g., real-time patient monitoring) [117, 133]. There are also other kinds of applications where data that diverse computer systems generate must be analyzed in near real time. These applications include on-line financial data processing (e.g., real-time risk management, automated trading) [147, 150], network monitoring [21, 83], software worm tracking, telephone call analysis [25], and click stream analysis [144]. All of these applications require prompt processing of *data streams* that continuously flow from various remote sources. For this reason, we call them *stream processing applications*.

Stream processing applications usually have the following characteristics:

1. **High and Variable Input Rates.** Stream data rates can be very high. For example, in the financial domain, some applications have to process more than 100,000 messages per second [1]. Furthermore, input rates often fluctuate in an unpredictable fashion as shown in Figure 1.1.

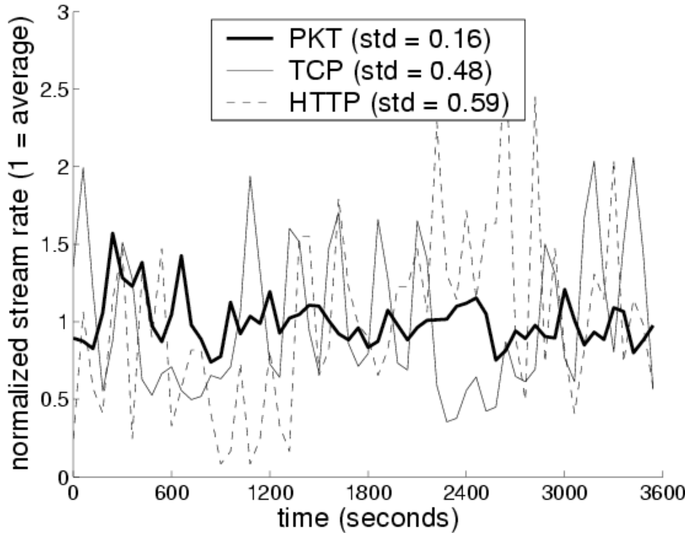


Figure 1.1: Stream rate variations in network traffic traces

The figure plots data rate variations for three real-world traces [15]: a wide area packet traffic trace (PKT), a TCP connection trace (TCP), and an HTTP request trace (HTTP).

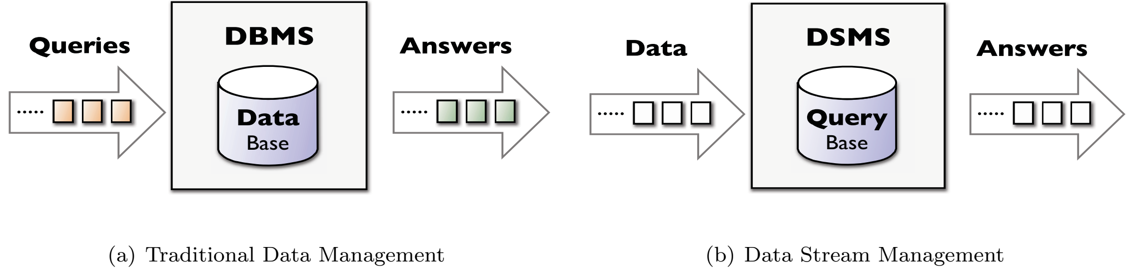
2. **Unbounded and Ordered Data.** A data stream may have no end as the related data sources continuously produce data. The order of a data stream may also be of semantic significance. For example, if the elements of a data stream represent events of moving in certain directions, the trajectory that the data stream represents can change depending on the order of the stream. Another important characteristic is that we do not have control over the order in which data elements arrive, within a data stream as well as across data streams.

3. **Low-Latency Requirement.** Stream data usually represents real-time events. Therefore, its value quickly decays over time. In network monitoring, for instance, current information about ongoing intrusion is more valuable than stale information about earlier attacks. Due to this property of stream data, low latency is a key requirement in stream processing applications.

4. **Imprecise Data.** Stream data is often lost or even intentionally omitted. For example, an object being monitored may move out of range of a sensor system. Furthermore, in high load situations, it might be necessary to drop less important data, thereby trading off accuracy for the timeliness of results. All of them lead to approximate answers.

5. **Push-Based Processing.** In a stream processing application, a number of data sources continuously push data to the system for processing. On the other hand, client applications passively wait for alerts or periodic updates that match their interest. For this reason, the overall processing is done in a push-based fashion.

A question that arises at this point is whether or not traditional database management systems (DBMSs), such as Oracle [46], IBM DB2 [121], and Microsoft SQL Server [45], can adequately



(a) Traditional Data Management

(b) Data Stream Management

Figure 1.2: The Paradigm Shift of Data Management

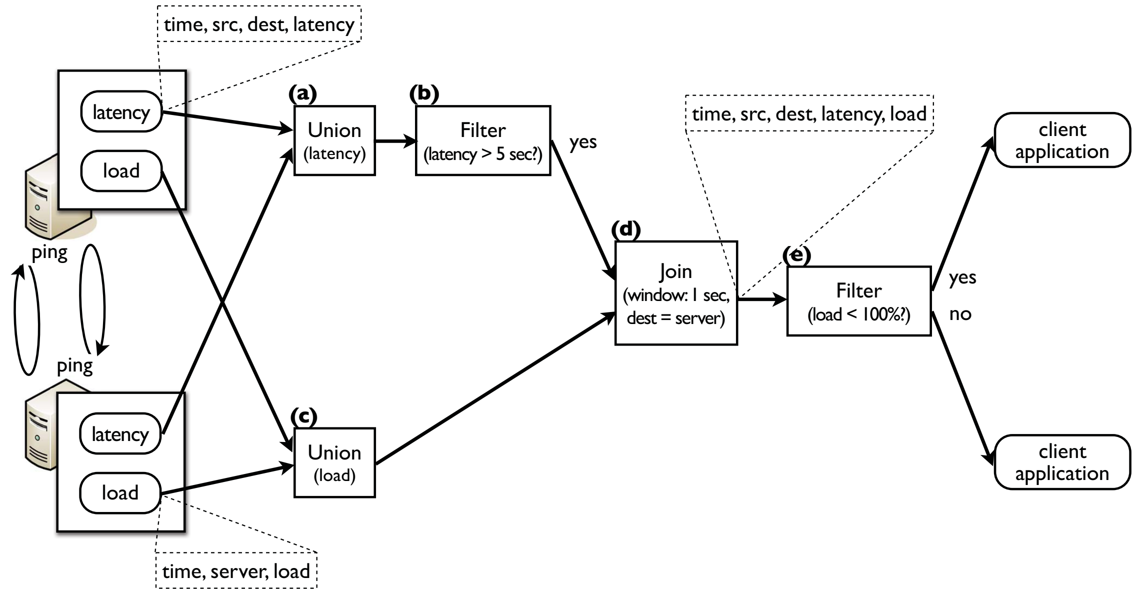


Figure 1.3: Query Example

support stream processing applications. While these systems have been extensively used in diverse commercial, civil, and academic applications, it is generally noted that they have limitations in the stream processing domain. One main reason behind this is that these database management systems are best suited to run one-time queries over finite data sets. In detail, these systems commonly employ the "process-after-store" (equivalently, "store-before-process" or "store-then-process") model. As illustrated in Figure 1.2(a), this model first stores data in a database and then, for each query submitted by a user, pulls the relevant data from the database. In this case, however, the database can easily become the performance bottleneck if high-rate stream data enters the system for processing.

To efficiently facilitate stream processing applications while overcoming the limitations of traditional database management systems, several research prototypes (e.g., Aurora/Borealis [32, 2, 1], STREAM [18, 99], TelegraphCQ [36, 94], Niagara [39]) and commercial products (e.g., StreamBase [128], IBM System S [75, 9], Gigascope [47], Coral8 [43], Amalgamated Insight [8]) have been

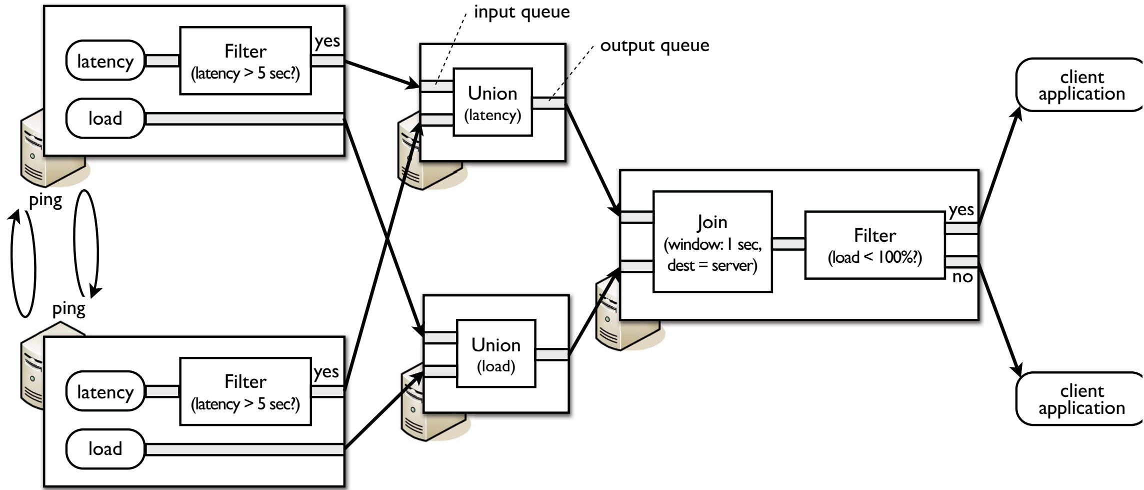


Figure 1.4: Query Distribution Example

developed. These systems commonly adopt a new “push-based, on-the-fly” processing model. In this model, as Figure 1.2(b) shows, a potentially large number of queries are preserved in the system and evaluated continuously over the stream data pushed from the sources. For this reason, we call such queries *Continuous Queries (CQs)*. As illustrated in Figure 1.3, continuous queries are typically expressed as a network of operators that collectively transform input streams into output streams that match the interest of client applications. The described systems are usually called Data Stream Management Systems (DSMSs), Stream Processing Systems (SPSs), Stream Processing Engines (SPEs), and continuous query processors, interchangeably.

Example. Figure 1.3 depicts a stream processing application that analyzes long-latency communication paths among servers in a wide area network. First, servers in the network ping each other and report the latency readings in the form of data streams (see stream sources labeled “latency”). These input streams are then merged at a Union operator (a) and the resulting stream is sent to a Filter that passes only the readings whose latency values exceed 5 seconds (b). To identify latencies resulting from network problems only (i.e., those certainly not caused by busy remote servers), a Join operator correlates the report of slow paths with the report of server loads (d). These load readings are initially sent from the participating servers (see stream sources labeled “load”) and then merged at a Union (c) before being forwarded to the Join operator. The final Filter operator categorizes slow paths according to the load levels and sends the results to client applications (e).

Many of the stream processing systems mentioned above are also geared toward distributed processing [36, 41, 119, 75, 128, 43]. Distribution in this context is beneficial because stream processing applications usually involve data sources spread over many remote locations and performing computation (e.g., filtering, summarization) at such sources can significantly reduce the amount of data to send across the network. In Figure 1.4, for example, Filter operators at stream sources selectively send latency readings, thereby saving network bandwidth. Furthermore, clusters of inexpensive

commodity servers are becoming more available. By making use of such machines, we can enhance the overall processing capability in a cost-efficient manner.

Example. Figure 1.4 illustrates an example where the query in Figure 1.3 is distributed over multiple servers (to reduce network usage, Filter operators are pushed to the stream sources). In this example, servers run stream processing operators while receiving streams from data sources or upstream servers and sending output streams to client applications or other servers for further processing. As shown in Figure 1.4, each operator receives data through its input queues. If an operator sends data to a remote server, it ensures data delivery by preserving its output data in an output queue until the data arrives at the destination server.

To develop a full-fledged stream processing system, we need to address various challenges. These challenges include:

- **Query Model.** Executing queries continuously over data streams, while taking the data arrival order into account, requires a radical change from the relational query model adopted in traditional database management systems. This is because the relational model relies on set-based (i.e., unordered) operations on finite data. If these operations are applied to unbounded data streams, they would either eternally wait for the non-existing end of input (e.g., Aggregate operations) or keep infinitely growing internal states (e.g., Join operations). For this reason, new query models tailored to stream processing have been developed [32, 99, 16]. The query model of the Aurora/Borealis system is summarized in Section 2.2.

- **Scheduling.** As illustrated in Figure 1.4, a server in a stream processing system is usually assigned multiple operators. Therefore, we need smart operator scheduling strategies that can maximize the system performance according to application-specific criteria (e.g., throughput, timeliness of results) [33, 17]. These strategies must consider resource constraints, such as memory size.

- **Load Shedding.** In stream processing, busty inputs may increase the processing load beyond the system capacity. In this case, the quality of service (particularly, the timeless of results) can fall below acceptable levels. One solution to this overload problem is to selectively drop stream data so that the result latencies decrease. Such load shedding techniques must be light-weight and minimize the loss in accuracy [134, 136, 135, 19, 112, 127]

- **Query Distribution.** Distributed and parallel computing environments have been gaining more popularity. To take advantage of these environments, we need to distribute queries over multiple machines. In stream processing, load balancing is difficult since data streams usually exhibit significant variations in their rates. Therefore, we need distribution strategies that can avoid bottlenecks for the largest variations in input rates [148]. We also need ways to find the query fragment (i.e., a subset of operators and data streams that interconnect them) migrating which will have the largest benefit, relative to the migration cost [149]. In wide area networks, query distribution also determines the routes of data streams. These routes, in turn, affect

both performance (in terms of latency) and network cost. Therefore, query distribution should be done in a manner that improves performance and decreases the network cost [3, 106].

- **High Availability.** A stream processing system can involve a large number of data sources, client applications, and operators spread over diverse geographic locations. As the system scale increases, the possibility of experiencing sever and network problems also increase. A failed server blocks data streams and may lose data essential for processing. Furthermore, computer networks are vulnerable to link congestion and may experience outages. Therefore, we need techniques that can prevent the system from being disrupted by slow or failed severs and networks. We continue this discussion in Section 1.2.

1.2 High-Availability Challenges in Stream Processing

Just like other software systems, a stream processing system can experience various problems. For example, servers may become unavailable either due to unexpected failures or planned reboot. Common sources of such unexpected failures are administrative errors, non-deterministic errors in the underlying software, exhaustion of system resources, and hardware malfunctions [60, 51]. Furthermore, communication between servers can be interrupted due to network link congestions and failures. In the case of network partition, a subset of servers can be completely isolated from the rest of the network. Since continuous queries can run for an indefinite period, they are bound to experience the problems above. Failures also tend to occur more frequently as the system scale increases.

In stream processing, a failure can have the following negative impacts:

- *A failure prevents low-latency processing.* If a server fails, it can no longer receive/process data and send the results to the next servers or client applications. Server overload, network link congestion, network failures, and network partitions also block the flow of data streams. In such cases, applications cannot receive timely results.

- *A failure may result in losing data essential to producing correct results.* As discussed in Section 2.2.2, some stream processing operators maintain states. For example, to compute the hourly average of stock prices per stock symbol, an operator usually remembers the sum and count of stock prices for each stock symbol during each hourly window. To minimize the processing overhead, operators typically keep their states in memory rather than slower disks. If a server that runs such operators crashes, it loses the states of the operators and thus cannot produce the same results as in the non-failure case.

The problems descried above are fatal in many stream processing applications. In financial applications, such problems can lead to missed trading opportunities and consequent revenue losses. For this reason, we need high-availability techniques that can allow client applications to always receive the correct results with low latency, despite server overload, server failures, network link congestion, and network failures.

As summarized in Sections 6.1 and 6.2, a wide variety of high-availability techniques have been developed for traditional database systems and distributed systems. These techniques commonly replicate the state of computation (i.e., all of the data required for computation) on either durable storage (e.g., a RAID disk array [40]) or an independently failing system component (e.g., a remote server). These techniques can be categorized into two general approaches:

- **Rollback recovery.** This approach periodically suspends computation and copies the state onto an independent location [52]. Between these periodic tasks, called checkpoints, this approach logs the input to the computation. Upon failure, this approach loads the state of the most recent checkpoint and reprocesses the logged input. In this way, the approach can rebuild the pre-failure state of the computation. This approach has low runtime overhead, but the recovery time gets longer as the checkpoint interval increases.

- **The state machine approach.** This approach replicates the computation on independent machines [86, 115]. Because the replicas run in parallel and any of their outputs can be used by others, a failure can cause little disruption to the processing. The resource usage, however, increases in proportion to the degree of replication.

Similar to the approaches mentioned above, high-availability techniques for stream processing also require some form of replication. To develop successful techniques, however, we need to consider the requirements of stream processing applications and characteristics of the stream processing model. Broadly speaking, the main challenge in realizing highly-available stream processing is to devise solutions that fulfill the following requirements:

1. **Minimal Disruption.** The intrinsic real-time nature of stream processing requires high-availability techniques that can always keep result latencies low. In other words, these techniques must provide fast recovery and, during non-failure periods, must be non-disruptive to regular processing.

2. **Resource Efficiency.** As pointed out in Section 1.1, stream data rates can be very high. For this reason, many efforts in stream processing have centered on efficiently using system resources [33, 17, 134, 136, 135, 19, 112, 127]. In general, high availability is achieved at the expense of using additional resources due to its reliance on replication. Therefore, economic use of resources ia a key requirement.

3. **Adaptivity.** One main characteristic of stream data is that its rate can significantly vary over time. Such variations can affect the processing load of operators as well as idle CPU cycles that can be used for the purpose of high availability. Furthermore, as an operator processes more input data, the state of the operator and, in turn, the overhead of checkpointing the operator can increase. To be successful, a high-availability technique must be able to adapt to such changes.

4. **Scalable Design.** A stream processing system can consist of a large number of components. Therefore, high-availability techniques must be designed in a scalable fashion.

One challenge related with the first and second requirements above is to devise relevant recovery semantics for different types of stream processing applications. Although many financial applications require that any post-failure results be identical to those without failure, other applications (e.g., fire detection, theft prevention) may tolerate slightly different post-failure results as long as no data is lost. For the latter applications, the less rigorous guarantee can be more advantageous than the strict guarantee if achieving it leads to faster recovery, smaller disruption to regular processing, and lower resource usage.

In the case of developing rollback recovery techniques for stream processing, one implication of the first requirement above is that disk-based techniques [97, 84] are not appropriate. Although these approaches have been extensively used in database management systems, they store data on disks and rely on database transaction mechanisms to ensure database consistency. However, disks have high latencies and the concept of a database transaction (i.e., a unit of work formed by several operations on a database [122]) cannot be applied to never-ending data streams. For this reason, a crucial challenge in this case is to checkpoint onto the memory of servers in a manner that satisfies the aforementioned requirements. In particular, for the first requirement, checkpoints must be done frequently and the duration of each checkpoint must be short. The reason is that result latencies during failure and non-failure periods decrease as the checkpoint frequency increases and the checkpoint duration decreases, respectively. One way of reducing checkpoint duration is to perform checkpoints at a fine granularity. For instance, we need not checkpoint disconnected operators together even if they run on the same server. Separately checkpointing them does not cause any inconsistency problem because the operators have no dependency with each other. To expedite recovery, we also need to schedule checkpoints while considering the characteristics of operators. The cost of checkpointing an operator increases as the state size of the operator increases. On the other hand, the higher the processing load, the larger the benefit of checkpoint. Note that a checkpoint removes the recovery load (i.e., the amount of work to do during recovery) and the recovery load is proportional to the processing load.

Applying the state-machine approach to stream processing also raises challenges. For example, replicas of a non-deterministic operator (e.g., an operator that forwards whichever data arrives first from multiple sources) can produce outputs in different orders. Because order has semantic significance in stream processing (as described in Section 1.1), we need solutions that either force non-deterministic replicas to run identically or reconcile the outputs of such replicas. Furthermore, we require ways to switch between replicas without losing data even if replicas run at different speeds.

Finally, there are challenges due to the characteristics of the target environments. In server clusters, for example, we need high-availability techniques that can efficiently use CPU cycles. CPU usage is a primary concern in these environments because the maintenance cost, including the cost for space and electricity, can increase significantly as more servers are used. In wide area networks,

one crucial challenge is to determine the geographic locations of replicas because these locations determine the routes of data flows and, in turn, the network usage. Another challenge is to cope with network problems, such as link congestion and failures. These problems complicate failure detection and can significantly degrade performance (i.e., increase result latencies).

1.3 Contributions

We have been tackling the challenges described in Section 1.2. In this section, we introduce our basic approaches for highly-available stream processing and customized solutions for two popular types of computing environments, namely now ubiquitous commodity server clusters and the Internet. The details of these techniques are presented in Chapters 3, 4, and 5.

1.3.1 Basic Approaches for Highly-Available Stream Processing

In stream processing, latency is a significant issue. Therefore, our early interest was laid in recovery approaches where if a server fails, a different backup server immediately takes over the failed one. The three developed approaches mainly differ in how the primaries (i.e., the servers that take the role of query processing) and backups prepare for failures. In the first approach, called *passive standby*, each primary delta-checkpoints its state onto a backup. Delta-checkpointing refers to copying the difference between the primary's current state and the state at the moment of the previous checkpoint. The state of a primary indicates the data structures that the primary uses for stream processing. In passive standby, a backup remembers the state of its primary as of the last checkpoint. Thus, if the primary fails, the backup must bring its old state up-to-date by re-processing the data that the primary consumed after the last checkpoint. Unlike passive standby, *active standby* allows backups to receive inputs and run in parallel with primaries. Active standby provides fast recovery because backups are always up-to-date. In *upstream backup*, primaries log their output data while backups remain completely idle. If a primary fails, an empty backup rebuilds the latest state of the primary using a subset of the logs at upstream servers.

With these three basic approaches, we also defined four recovery guarantees tailored to different requirements of stream processing applications. In principle, the guarantee of *precise recovery* (i.e., the output produced with failure recovery is the same as the output without failure) incurs a higher runtime overhead than other weaker recovery guarantees (e.g., no data is lost, but the output can be different from the output without failure). We also analyzed the characteristics of stream processing operators as well as their impact on recovery guarantees. Finally, using our Aurora/Borealis prototype and a detailed simulator, we observed the unique advantages of our recovery approaches in terms of CPU and network utilization and recovery speed.

1.3.2 Highly-Available Stream Processing in Server Clusters

Server clusters are a popular form of shared-nothing computing architectures where commodity servers are connected by fast local area networks. For these environments, we designed and implemented a new parallel, checkpoint-based recovery technique where operators at each server are backed-up on different servers. Because the backups are scattered over multiple servers, these servers can perform recovery in parallel, thereby significantly reducing the recovery time. This technique is also highly flexible in that each server can act as a backup for other servers, while still remaining a primary.

For this parallel recovery framework, we also studied the problem of (1) grouping operators into checkpoint units, (2) determining backup servers for these units, and (3) scheduling checkpoints. To fully utilize parallelism during recovery, our solution determines backup servers and schedules checkpoints in a manner that balances the recovery load over multiple servers. To further accelerate recovery, it also groups operators and schedules checkpoints according to the characteristics (e.g., processing load, checkpoint cost) of operators. Using our simulator and Aurora/Borealis prototype, we substantiated the effectiveness of our strategies for forming checkpoint units, assigning backup servers, and scheduling checkpoints. We also experimentally demonstrated the impact of operators and input data on the checkpoint cost and recovery speed.

1.3.3 Fast and Highly-Available Stream Processing over the Internet

The Internet is the worldwide, publicly accessible network of millions of academic, business, and government networks. While this attractive environment can enable prompt analysis of tremendous events occurring around world, it introduces new difficulties. In the Internet, for example, losing access to remote servers is the norm rather than the exception. Furthermore, it is usually very difficult to identify the real cause of a system error because there are many possibilities such as server failures, network link congestion, and network failures. We observed that previous high-availability techniques have limitations in this environment due to their reactive nature – backups do not respond to a failure until they ensure that the failure indeed has occurred. In practice, it may also take a long time for multiple servers to agree upon a failure [87, 114]. Until the moment of agreement, the related processing may completely stop.

In contrast to these previous approaches, our approach replicates stream processing operators at different locations and allow all of them to send data downstream. In this way, any downstream processing can proceed by relying on the fastest input flow without being held back by slow or interrupted ones. For this reason, our approach improves not only *reliability* but also *performance*, sharply contrasting with previous approaches. It also can naturally mask failures and local congestions even without detecting them.

One challenge in this work was to guarantee correctness while executing replicas without coordination. The reason behind this is that synchronizing remote replicas can significantly delay processing. Although replicas can run differently, our solution provides *replication transparency*, a

guarantee that applications always receive the same results as in the non-replicated, failure-free case. For this guarantee, we extended operators such that replicas of them always produce the same data, but possibly in different orders, as long as they consume the same input data in any order.

For this Internet-scale replication-based framework, we also developed a strategy for managing replicas. This strategy deploys replicas at locations that improve *performance* as well as *availability*, while reducing the *network communication cost*. To efficiently cope with changes in system conditions, the strategy also dynamically discards the least useful replicas or adds new replicas. To verify the utility of this work, we conducted experiments over scores of machines on PlanetLab [107].

1.4 Organization

The remainder of this dissertation is organized as follows. In Chapter 2, we set the ground for this dissertation by providing an overview of stream processing and our Aurora/Borealis system. In Chapter 3, we present our three basic approaches for highly-available stream processing and compare them from various perspectives. Next, we present our customized solutions for server clusters and the Internet in Chapters 4 and 5, respectively. We present related work in Chapter 6 and conclude this dissertation in Chapter 7.

Chapter 2

Background

Our research on highly-available stream processing has been conducted in the context of two stream processing systems, Aurora [32, 2] and its follow-on Borealis [1, 4]. These systems have been developed by researchers from Brandeis University, Brown University, and MIT since 2001. In this chapter, we first provide an overview of the data, query, and quality of service models that these systems commonly employed (Sections 2.1, 2.2, and 2.3). Next, we illustrate the architectures of these systems (Section 2.4 and 2.5). Finally, we discuss the types of failures that can be considered in stream processing (Section 2.6).

2.1 Stream Data Model

A *data stream* is defined as an append-only sequence of *data elements*. These data elements, often also called *tuples*, can be generated continuously by various sources, such as sensors that measure a particular kind of environmental condition (e.g., temperature) and computer programs that report information of interest (e.g., bids/asks in the stock market). A data element is again composed of *attribute values*. All of the data elements contained in a data stream have the same set of attributes. We call such a set of attributes the *type* or *schema* of the stream. In this dissertation, the ith data element of data stream S is denoted as $S[i]$.

2.2 Query Model

As Figure 2.1 shows, queries in our Aurora and Borealis systems are defined as a boxes-and-arrows dataflow diagram. Each box represents an operator and each arc represents a dataflow between two operators. An operator can be connected to multiple downstream operators. All such splits send identical tuples downstream and enable sharing of computation across different queries. A number of streams can also be merged by an operator with multiple inputs. We call a collection of such queries a *query network*.

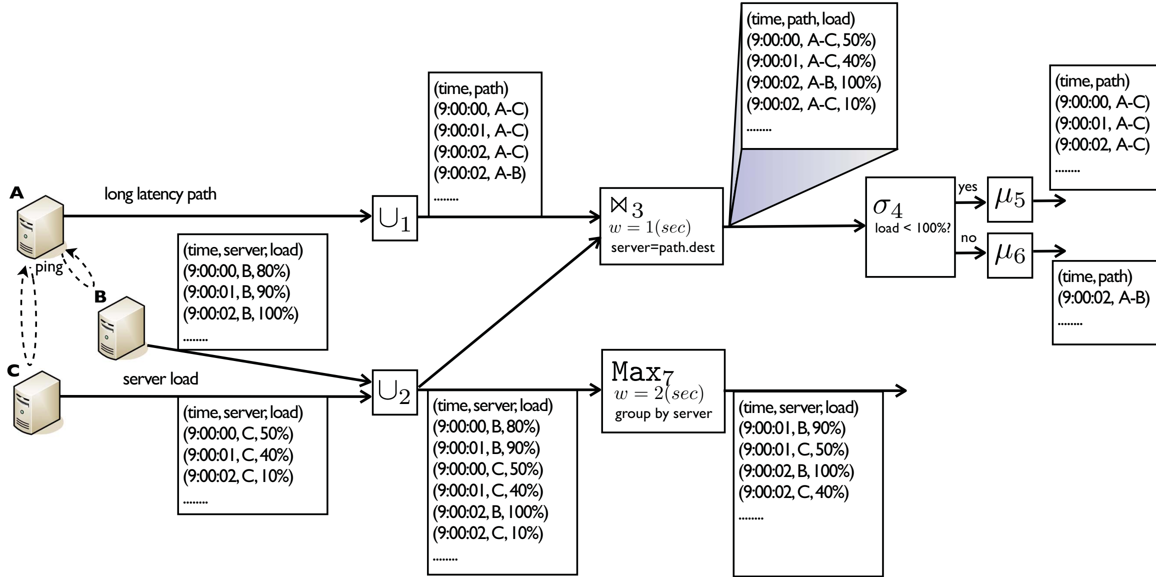


Figure 2.1: Query Network Example

Stream processing operators play the role of processing tuples and producing results of interest. Because queries are expressed as combinations of them, they enable query optimization (if a query can be expressed as multiple equivalent combinations, we choose the combination with the minimum execution cost) as well as sub-query sharing. In this section, we summarize these operators. Further details on the Aurora/Borealis operators are presented in [1].

2.2.1 Stateless Operators

A stateless operator performs its computation based on one input tuple at a time without holding any state. Here, state refers to the data structures that an operator maintains for its computation. Aurora and Borealis provide the following stateless operators:

- *Filter* is the equivalent of the selection operator in the relational algebra. Filter applies a predicate to every input tuple and passes tuples that satisfy the predicate. Tuples that do not satisfy the predicate are either dropped or forwarded on an optional second output stream. Filter can have multiple predicates. In this case, it acts as a case statement. In other words, it propagates each tuple on the output stream that corresponds to the first matched predicate. In Figure 2.1, Filter operator σ_4 evaluates its predicate for tuples (9:00:00, A-C, 50%), (9:00:01, A-C, 40%), (9:00:02, A-B, 100%), (9:00:02, A-C, 10%), $\cdots$. This operator forwards (9:00:00, A-C, 50%), (9:00:01, A-C, 40%), (9:00:02, A-C, 10%) to operator μ_5 because their load attribute values are less than 100%. It sends (9:00:02, A-B, 100%) to operator μ_6.

- *Map* extends the projection operator in the relational algebra. Map transforms input tuples into output tuples by applying a set of functions to the tuple attributes. In Figure 2.1, Map

operators μ_5 and μ_6 remove the load field from each input tuple and send the resulting tuple downstream.

- *Union* merges a set of input streams (all with the same schema) into a single output stream. Because this operator simply passes input tuples as they arrive, the output order depends on the inter-arrival order of the input streams. In Figure 2.1, Union operator $\cup_2$ merges streams (9:00:00, B, 80%), (9:00:01, B, 90%), (9:00:02, B, 100%), $\cdots$ and (9:00:00, C, 50%), (9:00:01, C, 40%), (9:00:02, C, 10%), $\cdots$ into (9:00:00, B, 80%), (9:00:01, B, 90%), (9:00:00, C, 50%), (9:00:01, C, 40%), (9:00:02, B, 100%), (9:00:02, C, 10%), $\cdots$.

2.2.2 Stateful Operators

In contrast to stateless operators, stateful operators produce each output tuple by processing a collection of input tuples. We present the following representative operators:

- *Aggregate* applies an aggregate function such as sum, count, average, maximum, and minimum, to sub-sequences of its input stream. Because data streams are potentially unbounded and operators cannot hold infinitely many data items, we use finite windows over data streams. A window on a data stream is defined over the ordering attributes of the stream or over the arrival order of the stream. A sliding window is defined as a sequence of windows with the same size but different starting values. The difference between the starting points of adjacent windows is called the step size. For example, suppose that stream S has timestamp attribute T and another attribute A. To get the average value of attribute A over the most recent one minute every second, we need an Aggregate operator that computes the average value of A over a sliding window with a size of one minute and a step size of one second.

  The input stream of Aggregate can be partitioned into sub-streams using the "group by" clause. In this case, a sliding window is defined on each sub-stream. A window starts when a first tuple that falls into the window arrives. It is then closed after all the tuples that belong to the window arrive, or when the window times out. The longest time a window can stay open is specified by the "timeout" clause. Aggregate produces an output tuple whenever it closes a window. For this, the operator must maintain a certain amount of information (usually some form of summaries) for each open window. Such information collectively forms the state of Aggregate. The state size of Aggregate is usually proportional to the number of open windows.

  In Figure 2.1, Aggregate operator Max_7 finds the maximum load value using a 2-second time window that advances 1 second at each step. This operator uses the server identifier as the group-by attribute. It produces (9:00:01, B, 90%) from (9:00:00, B, 90%) and (9:00:01, B, 50%). It generates (9:00:01, C, 50%) from (9:00:00, C, 50%) and (9:00:01, C, 40%).

- *Join* extends the join operator in the relational algebra. To deal with the unbounded nature of data streams, it assumes a window on its two input streams. For example, suppose that a Join operator with predicate P has input stream S_1 ordered by attribute A and input stream

S_2 ordered by attribute B. Suppose also that the operator has a window w on the ordering attributes. If tuple u from S_1 and tuple v from S_2 satisfy the condition $|u.A - v.W| < w$ and predicate P, the operator produces the concatenation of u and v as the output tuple. To find such *matching input tuples*, Join must preserve recent input tuples. These tuples can be safely discarded only if no future tuples can match with them. Therefore, the state of Join consists of all the tuples that the operator keeps for its computation. The state size is proportional to the product of the window size and input stream rates.

In Figure 2.1, Join operator $\bowtie_3$ processes two input streams from $\cup_1$ and $\cup_2$ with a 1-second time window. This operator concatenates tuples u from the first input stream and v from the second input stream if both u and v belong to the same time window and the server identifer attribute value of u and the destination identifer attribute value of v match. This operator generates (9:00:00, A-C, 50%) from matching input tuples (9:00:00, A-C) and (9:00:00, C, 50%). It produces (9:00:01, A-C, 40%) from (9:00:01, A-C) and (9:00:01, C, 40%).

- *BSort* is an approximate sort operator. Because a complete sort is not possible over an infinite stream with finite time and space, BSort uses a buffer of size n for sorting. In detail, BSort inserts each input tuple in a buffer as it arrives. Whenever the buffer is full, the operator removes, from the buffer, a tuple with the minimal value of the sort attribute and emits the tuple as output. In this case, the output stream corresponds the sequence resulting from executing $n-1$ bubble sort passes. The state of BSort is proportional to the predefined buffer size.

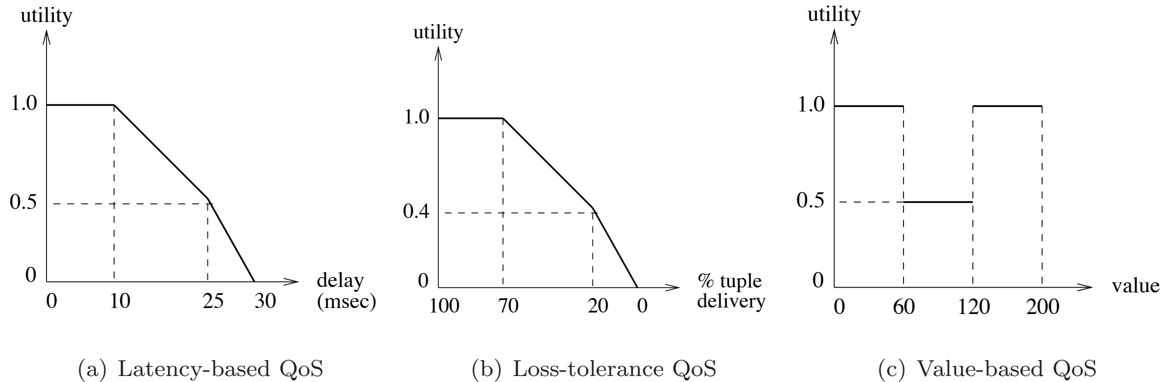
- Resample is used to align pairs of streams by interpolating missing values. Given two input streams S_1 and S_2, this operator generates, for each tuple in S_1, an interpolated value from S_2 by applying an interpolation function F to a window defined on S_2. In this case, the state of this operator comprises all the tuples that are currently kept in the current window on input stream S_2.

- SQL-read and SQL-write operators, respectively, query and update relational database tables for each input tuple. These database tables belong to the states of these operators.

2.3 Quality of Service

Aurora and Borealis continuously send query results to client applications. They also measure the utility of the results based on a number of QoS functions. Because the resulting utility values represent the usefulness of the results from the perspective of applications, these system can conduct various optimizations in application-specific ways. Figure 2.2 illustrates the following representative QoS functions:

- **Latency-based QoS.** This function maps result latencies to utility values such that as output tuples get delayed, their utility degrades. The latency of an output tuple is defined as the



(a) Latency-based QoS (b) Loss-tolerance QoS (c) Value-based QoS

Figure 2.2: Quality of Service

difference between the time when the last among the input tuples that contributed to the output tuple entered the system and the time when the output tuple arrives at the application.

- **Loss-tolerance QoS.** If the overall processing load surpasses the processing capability, a typical solution is to drop a faction of tuples in order to improve the timeliness of results. The accuracy of the results, however, usually decreases as more tuples are dropped. The loss-tolerance function maps the percentage of tuple delivery to a utility value. This value indicates the application-specific usefulness of the approximate answers.
- **Value-based QoS.** This function shows which values in the output value space are most important. For example, in an application that monitors patient heartbeats, extreme heart rates are more important than normal rates and thus should be assigned high utility values.

Aurora and Borealis can schedule operators in a manner that maximizes the latency-based QoS function [33]. In this case, they produce the most timely results. They can also make load shedding decisions based on the loss-tolerance and value-based QoS functions [134].

2.4 The Aurora System

Aurora is a stream processing system that has been developed by researchers from Brandeis University, Brown University, and MIT since 2001 [2, 134, 33]. In 2003, the research prototype was also commercialized into the StreamBase System [128]. Hereafter, we provide an overview of the system.

Figure 2.3 illustrates the architecture of Aurora as well as the flow of data and control between its components. These components play the following roles:

- The *catalog* stores information about the query network, QoS functions, and run-time statistics (e.g., selectivity and processing cost of each operator). Because this information is essential for efficient system management, all the system components can access the catalog.

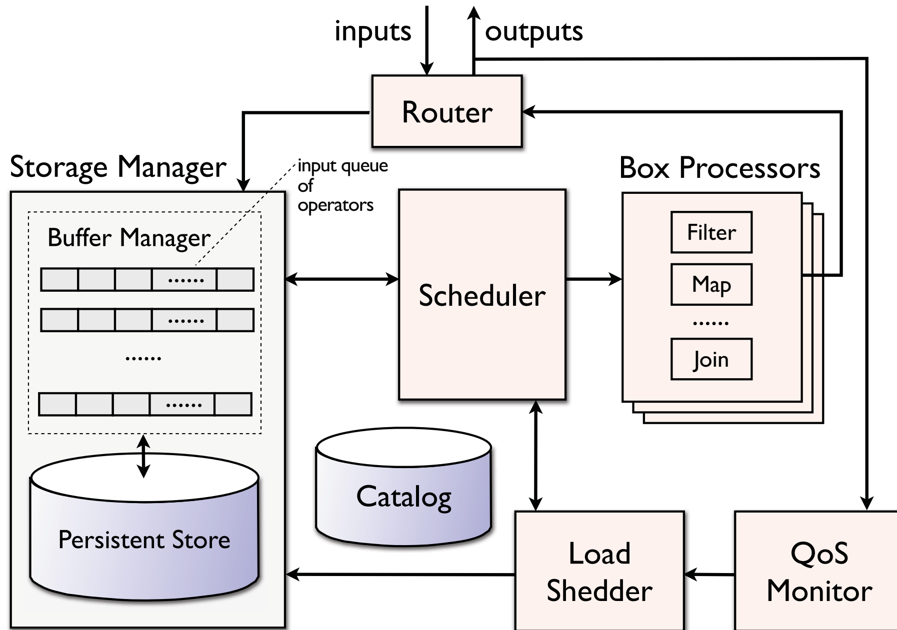


Figure 2.3: Aurora System Architecture

- The *router* is responsible for forwarding tuples between run-time components. It can receive tuples from data sources as well as box processors and then send them to the storage manager.
- The *storage manager* is responsible for efficient storage and retrieval of data. For operators to immediately process tuples stored in their input queues, the storage manager maintains these queues in an in-memory buffer pool. It also manages a persistent store to support ad hoc queries.
- The *scheduler* determines the order of executing operators [32, 33]. It selects an operator with tuples in its queues and feeds that operator one or more of the input tuples.
- The multi-threaded *box processors*, whenever invoked by the scheduler, execute the appropriate operator module and then forwards the output tuples to the router so that the next processing cycle begins.
- The *QoS monitor* continually monitors the system performance and triggers the load shedder if it detects decrease in QoS.
- The *load shedder* is responsible for handling overload due to input bursts [134]. It obtains system statistics and query network description from the catalog. In overload situations, it modifies the running query network to bring the CPU usage down to an appropriate level.

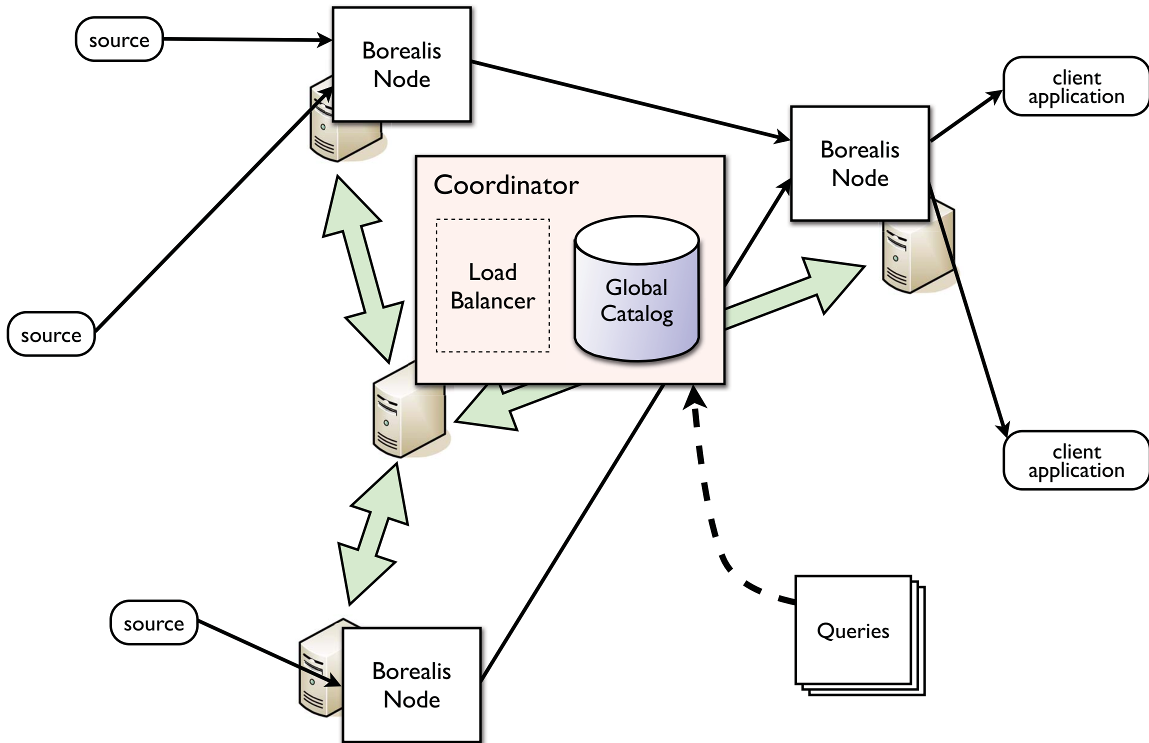


Figure 2.4: Borealis System Architecture

2.5 The Borealis System

Borealis is a distributed system that inherits the core stream processing functionalities from Aurora [1, 4] and communication capabilities from Medusa [23]. Borealis aims to achieve the following objectives:

- **Distributed Operation.** The system can distribute queries over multiple available machines in a manner that avoids performance bottlenecks and efficiently uses system resources.
- **Scalability.** The system can scale up and deal with increasing load and more queries with the addition of new computation and network resources.
- **Adaptivity.** The system can cope with changing load and resource availability without disrupting its operation.
- **High Availability.** The system can continue its operation in spite of server and network failures.

2.5.1 The Overall Architecture

As Figure 2.4 illustrates, Borealis comprises the following components:

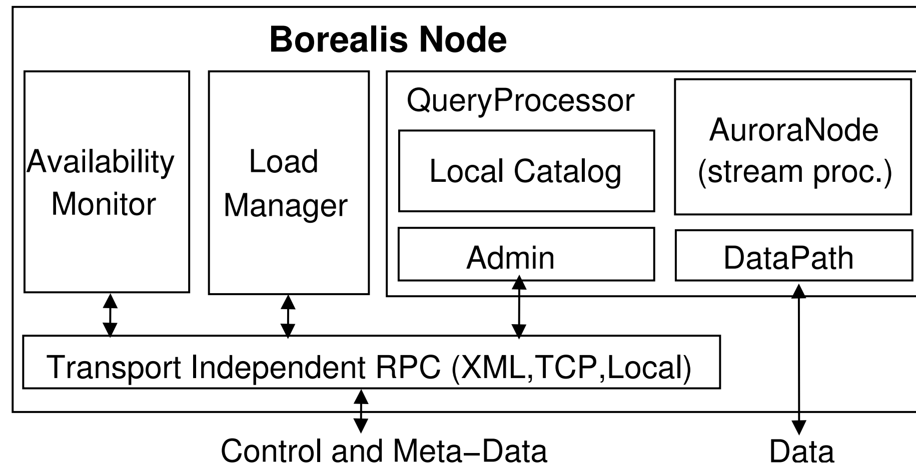


Figure 2.5: Borealis Node Architecture

- *Borealis nodes* perform the actual stream processing. Each node runs a number of stream processing operators and collects various statistics, such as its load level and processing latency. Each node also performs tasks to manage its load and ensure high availability. We present the details of Borealis nodes in Section 2.5.2.

- The *coordinator* supervises Borealis nodes. It stores in the *global catalog* information about Borealis nodes, the complete query diagram, and the current query deployment (i.e., assignment of operators to nodes). The coordinator starts empty. As the administrator or the end-users submit queries, the coordinator converts the query into an acyclic graph of operators and deploys the operators. Currently, the communication between the coordinator and Borealis nodes are done by an XML RPC scheme. This scheme uses XML descriptions to specify the query network as well the query deployment.

- *Stream sources* continuously send data to Borealis nodes. They are usually physical devices with sensing and communication capabilities or software programs.

- *Client applications* are computer programs that receive the results of stream processing and then accordingly interact with humans or perform further tasks (e.g., trade stocks or actuate machines). Borealis provides application programming interfaces and tools to help the development of client applications.

- Borealis *administration tools* ease system maintenance. With visual interfaces, they help users quickly define queries and system administrators monitor the current deployment and various system conditions.

2.5.2 The Architecture of a Borealis Node

Each Borealis node runs an autonomous stream processing engine, called *query processor*. As illustrated in Figure 2.5, a query processor has the following components:

- An *administration interface* handles all incoming requests. These requests are sometimes to modify the local query fragment by creating operators and data streams between them (or by discarding operators and streams). They may also ask the query processor to move some operators to other remote query processors. In this case, the administration interface handles the detailed steps of operator migration.
- A *local catalog* holds the current information about the local query fragment, including local operators, streams, and subscriptions.
- The *DataPath* component routes tuples from sources and remote nodes to the query processor. It also sends tuples to remote nodes as well as client applications.
- *AuroraNode* is the local stream processing engine. It receives tuples from DataPath, processes them, produces outputs, and sends them back to DataPath. To perform the processing, AuroraNode instantiates operators and schedules their execution. Each operator receives tuples from its input queues (one per input) and stores outputs in its output queues. AuroraNode also collects statistics about the runtime performance, data rates of streams, CPU cost of operators. These statistics are made available to remote nodes through the administration interface.

In addition to the query processor, each Borealis node has the following modules:

- The *availability monitor* monitors the state of other nodes and notifies the query processor if any of the states change. The availability monitor is a generic monitoring component for high availability purposes.
- The *load manager* collects information about local and remote loads and balances the processing load over nodes through operator migration.

2.6 Failure Models

In general, developing a high-availability technique begins with assuming failure models for system components. How well these models correspond to real system failures significantly affects the utility of the developed technique in a given setting. A stream processing system makes use of a number of servers and network links connecting them. Therefore, we can consider the following failure models in this context, as commonly done in other kinds of distributed systems.

2.6.1 Server Failures

As discussed in Section 1.2, servers may become unavailable either due to planned reboot or unexpected failures such as administrative errors, software bugs, exhaustion of system resources, and hardware malfunctions. Types of server failures that are generally assumed are as follows:

- **Fail-stop failures**. A failed server stops execution and loses its volatile state (e.g., everything that was in memory). Other servers in the system can easily detect the fail-stop failure of a server [116].

- **Crash failures**. In this model, a failed server forever stops sending messages and responding to any requests (e.g., a runaway computation). Crash failures may not be detectable by other servers [116]. This model is a superset of, and thus more widely applicable than, the fail-stop model.

- **Byzantine failures**. A failed server exhibits arbitrary behavior [88]. For example, erroneous output by a server (e.g., due to a buffer overflow) is considered a Byzantine failure. Although Byzantine failures are not studied in this dissertation, while left for future research, well-known techniques for handling these failures exist [95]. Because this model is the most general of the three, these techniques impose higher overheads than those for other failure models.

2.6.2 Network Failures

In general, communication channels are unreliable and can cause message losses, re-ordering, and delays. The high-availability techniques in this dissertation count on reliable, in-order communication protocols like TCP for transferring data between servers. With TCP, data is always delivered reliably and in order, but the delivery can incur arbitrary delays (in the case of network failures or simply congestion) and connections can also fail permanently. Network failures, which cause messages to be arbitrarily delayed or TCP connections to go down, can occur in a local area network but are more common in wide area networks. Network failures can sometimes cause network partitions, where the entire system is split into two or more groups of components that cannot communicate with those in other groups. In Chapter 5 of this dissertation, we explore solutions to handle network failures in the Internet scale.

Chapter 3

Basic Approaches for Highly-Available Stream Processing

In stream processing, the failure of a single server can significantly disrupt or even halt the overall processing. Such a failure indeed causes the loss of a potentially large amount of transient information and, perhaps more importantly, prevents downstream servers from making progress. A distributed stream processing system therefore must incorporate a high-availability mechanism that allows processing to continue despite server failures. In this chapter, we focus on approaches where if a server fails, a backup server immediately takes over the operation of the failed one. Tightly synchronizing a primary and a backup so that they always have the same state will incur high run-time overhead. Hence, we explore approaches that relax this requirement, allowing the backup to *rebuild* the state of the primary.

Because different stream processing applications have different high-availability requirements, we define three types of recovery guarantees that address different needs:

1. *Precise recovery* hides the effects of a failure perfectly, except some transient increase in processing latency. This guarantee is well-suited for applications that require the post-failure output be identical to the output without failure. Many financial applications require this strict guarantee.

2. *No loss recovery* avoids information loss without guaranteeing precise recovery. The output produced after a failure is "equivalent" to, but not necessarily the same as, the output of an execution without failure. The output may also contain duplicate tuples. To avoid information loss, the system must preserve all the necessary input data for the backup server to rebuild (from its current state) the primary's state at the moment of failure. No loss recovery is thus appropriate for applications that cannot tolerate information loss but may tolerate imprecise

output caused by the backup server reprocessing the input somewhat differently than the primary did. Example applications include those that monitor specific conditions in the world (e.g., fire alarms, asset tracking). We show in Section 3.5 that this recovery guarantee can be provided more efficiently than precise recovery in terms of runtime overhead and recovery speed.

3. *Gap recovery*, our weakest recovery guarantee, addresses the needs of applications that operate solely on the most recent information (e.g., sensor-based environment monitoring), where dropping some old data is tolerable for reduced recovery time and runtime overhead.

We define these recovery semantics more precisely in Section 3.2. Commercial database management systems typically offer precise or gap recovery capabilities [42, 103, 111]. To the best of our knowledge, no existing solution addresses no loss recovery or a similar weak recovery model.

In this chapter, we also investigate four recovery approaches that can provide one or more of the above recovery guarantees. Since each approach employs a different combination of redundant computation, checkpointing, and remote logging, they offer different tradeoffs between runtime overhead and recovery performance.

We first introduce *amnesia*, a lightweight scheme that provides gap recovery without any runtime overhead (Section 3.3). We then present *passive standby* and *active standby*, two process-pairs [24, 60] approaches tailored to stream processing. In passive standby, each primary server periodically reflects its state updates to its backup server. In active standby, backup servers process all tuples in parallel with their primaries. We also propose *upstream backup*, an approach that significantly reduces runtime overhead compared to the standby approaches while trading off recovery speed. In this approach, upstream servers act as backups for their downstream neighbors by preserving tuples in their output queues while the downstream neighbors process them. If a server fails, a recovery server rebuilds the primary's latest state by reprocessing the tuples logged at upstream servers. In Section 3.4, we describe the details of these approaches with an emphasis on the unique design challenges that arise in stream processing. Upstream backup and the standby approaches provide no loss recovery in their simplest forms and can be extended to provide precise recovery at a higher runtime cost, as illustrated in Section 3.5.

Interestingly, for a given high-availability approach, the overhead to achieve precise recovery can noticeably change with the properties of the operators constituting the query network. We thus develop in Section 3.2 a taxonomy of stream processing operators, classifying them according to their impact on recovery semantics. Section 3.5 shows how we can use such knowledge to reduce high-availability costs and choose the most appropriate high-availability technique in a given setting.

Finally, by comparing the runtime overhead and recovery performance for each combination of recovery approach and guarantee (Section 3.6), we characterize the tradeoffs among the approaches and describe the scenarios where each approach is most appropriate.

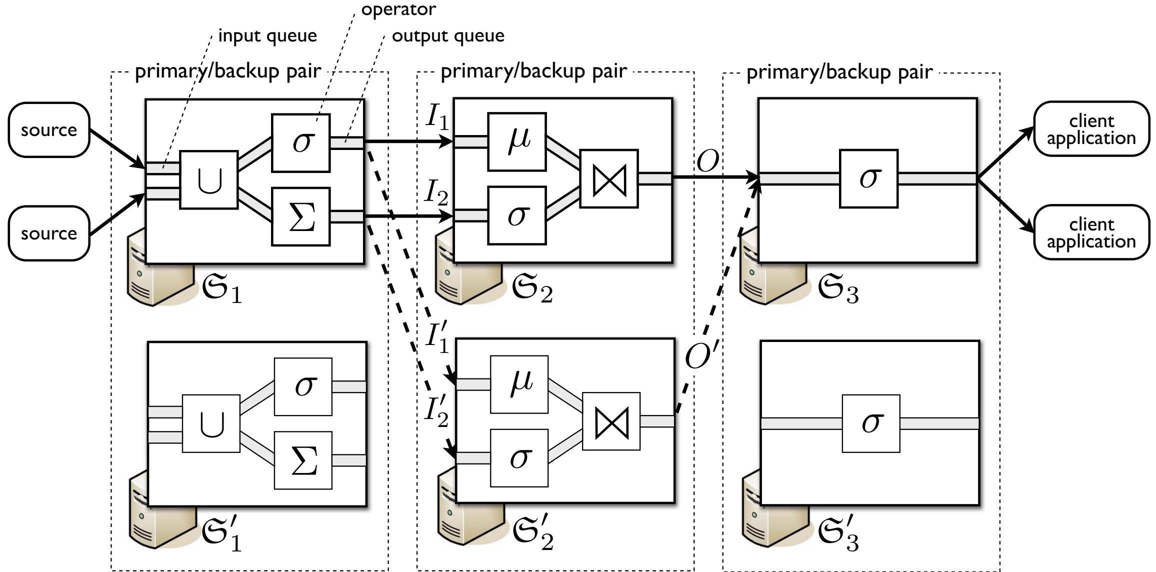


Figure 3.1: Primary/Backup Pairs

3.1 Assumptions

In this chapter, we assume a network of stream processing operators (e.g., Map, Filter, Union, Join, Aggregate) that receive input tuples through their input queues (one for each input stream) and produce output tuples based on their execution semantics. We also assume that the system distributes the query network over multiple servers, each of which runs a stream processing engine. Figure 3.1 illustrates a query network distributed over three servers, $\mathfrak{S}_1$, $\mathfrak{S}_2$, and $\mathfrak{S}_3$. In the figure, streams are represented by solid line arrows whereas operators are represented as boxes labeled with symbols denoting their functions. Since streams I_1 and I_2 send tuples from $\mathfrak{S}_1$ to $\mathfrak{S}_2$, $\mathfrak{S}_1$ is said to be *upstream* of $\mathfrak{S}_2$, and $\mathfrak{S}_2$ is said to be *downstream* of $\mathfrak{S}_1$. We assume that the communication network ensures order-preserving, reliable message transport (e.g., TCP).

To handle *single fail-stop server failures*, we associate each server $\mathfrak{S}_i$ with a backup server $\mathfrak{S}'_i$ that detects as well handles the failure of $\mathfrak{S}_i$. We call $\mathfrak{S}_i$ a *primary server*. For $\mathfrak{S}'_i$, we use terms such as "recovery server", "backup server", and "secondary server", interchangeably. Each backup server has its own stream processing engine and has the same query network fragment as its primary. The state of a backup server, however, is not necessarily the same as that of the primary.

To detect failures, we assume that each backup server periodically sends keep-alive requests to its primary and assumes that the latter failed if a few consecutive responses do not return within a timeout period (for example, our prototype uses three messages with a 100 ms transmission interval, for an average failure detection delay of 250 ms). If a backup server detects the failure of its primary and if it has not received input data, it asks the upstream servers to send the data (in Figure 3.1, I'_1 and I'_2 are used instead of I_1 and I_2, respectively). The backup server also starts forwarding

its outputs to downstream servers (in Figure 3.1, O' is used instead of O). We call this process of responding to a failure *failover*.

If the backup server needs to reprocess some earlier input tuples to bring its state up-to-date, the upstream servers must remember such required tuples. For this purpose, each stream to a remote server has an *output queue* as a temporary storage (in Figure 3.1, I_1 and I_2 have output queues on $\mathfrak{S}_1$).

Finally, if a failed server comes back to life, it can assume the role of the backup. Another server may also volunteer to do so. As we discuss in Section 3.4, each high-availability approach requires a different amount of time for the new backup to coordinate with its primary to prepare for the next failure.

3.2 High-Availability Semantics

In this section, we define three recovery types based on their effects as perceived by the servers downstream from the failure. Since some operator properties facilitate stronger recovery guarantees, we also classify operators based on their effects on recovery semantics.

3.2.1 Recovery Types

We assume that a query network fragment, Q, is given to a primary/backup pair. Q has a set of n input streams $(I_1, I_2, ..., I_n)$ and produces one output stream O. The definitions below can easily be extended to query network fragments with multiple output streams.

Because the processing may be non-deterministic as we discuss in Section 3.2.2, executing Q over the same input streams may each time produce a different sequence of tuples on the output stream. We define an *execution* as a sequence of events (such as the arrival, processing or production of a tuple) that occur while a server runs Q. Given an execution e, we denote with O_e the output stream produced by e. We express the overall output stream after failure and recovery as $O_f \odot O'$, where f is the pre-failure execution of the primary and O' is the output stream produced by the backup after it takes over. Based on the notation, we define three recovery guarantees as follows:

- **Precise Recovery.** The strongest failure recovery guarantee, called *precise recovery*, completely masks a failure and ensures that the output produced by an execution with failure recovery is identical to the output produced by an execution e without failure (i.e., $O_f \odot O' = O_e$).

- **No loss Recovery.** A weaker recovery guarantee, called *no loss recovery*, ensures that failures do not cause information loss. More specifically, it guarantees that the effects of all input tuples are always forwarded to downstream servers despite failures. To achieve this guarantee, we need to meet the following requirements:

  1. *Input preservation* - The upstream servers must store in their output queues all tuples that the backup needs to rebuild the primary's state, from its current state. We refer to such tuples as *duplicate input tuples* because the primary already has consumed them.

Recovery Type	Before Failure			After Failure			
Precise	t_1	t_2	t_3	t_4	t_5	t_6	...
Gap	t_1	t_2	t_3		t_5	t_6	...
No loss							
Repeating	t_1	t_2	t_3	$\mathbf{t_2}$	$\mathbf{t_3}$	t_4	...
Convergent	t_1	t_2	t_3	$\mathbf{t'_2}$	$\mathbf{t'_3}$	t_4	...
Divergent	t_1	t_2	t_3	$\mathbf{t'_2}$	$\mathbf{t'_3}$	$\mathbf{t'_4}$	...

Figure 3.2: Examples of Outputs under Each Recovery Type

2. *Output preservation* - If a backup is running ahead of its primary, the backup must store tuples in its output queues until all the downstream servers receive the corresponding tuples from the primary. The tuples at the backup are then considered *duplicate*.

We use the configuration in Figure 3.1 to illustrate these concepts. We cannot discard tuples in the output queues of I_1 and I_2 if $\mathfrak{S}'_2$ requires them to rebuild the state of $\mathfrak{S}_2$. Similarly, if $\mathfrak{S}'_2$ is running ahead of $\mathfrak{S}_2$, $\mathfrak{S}'_2$ must preserve all tuples in the output queue of O' until the tuples become duplicate (i.e., until $\mathfrak{S}_3$ receives from $\mathfrak{S}_2$ the tuples resulting from processing the same input tuples).

No loss recovery allows the backup to forward *duplicate output tuples* downstream. The characteristics of Q determine the characteristics of such duplicate output tuples as well as the properties of $O_f \odot O'$. We distinguish three types of no loss recovery. In the first type, *repeating recovery*, duplicate output tuples are *identical* to those produced previously by the primary. With the second type, *convergent recovery*, duplicate output tuples are different from those produced by the primary. The details of such situations are discussed in Section 3.2.2. Under both recovery types, the concatenation of O_f and O' *after removing duplicate tuples* is identical to an output without failure, O_e. Finally, the third type of recovery, *divergent recovery*, has the same properties as convergent recovery regarding duplicate output tuples. Eliminating these duplicates, however, does not lead to an output that is achievable without failure. Such divergent recovery is caused by non-determinism in processing.

Because the execution of the backup can be different from that of the primary, duplicate output tuples are not necessarily identical to those that the primary produced. We consider an output tuple t at the backup to be *duplicate* if the primary has already processed *all* input tuples that "affected" the value of t and forwarded the resulting output tuples downstream.

- **Gap Recovery.** Any recovery technique that does not ensure both input and output preservation may result in information loss. This recovery type is called *gap recovery*.

Example. Figure 3.2 shows examples of outputs produced by each recovery type. With precise recovery, the output corresponds to an output without failure: tuples t_1 through t_6 are produced in sequence. With gap recovery, the failure causes the loss of tuple t_4. Repeating recovery produces tuples t_2 and t_3 twice. Convergent recovery generates different tuples t'_2 and t'_3 (compared to t_2 and

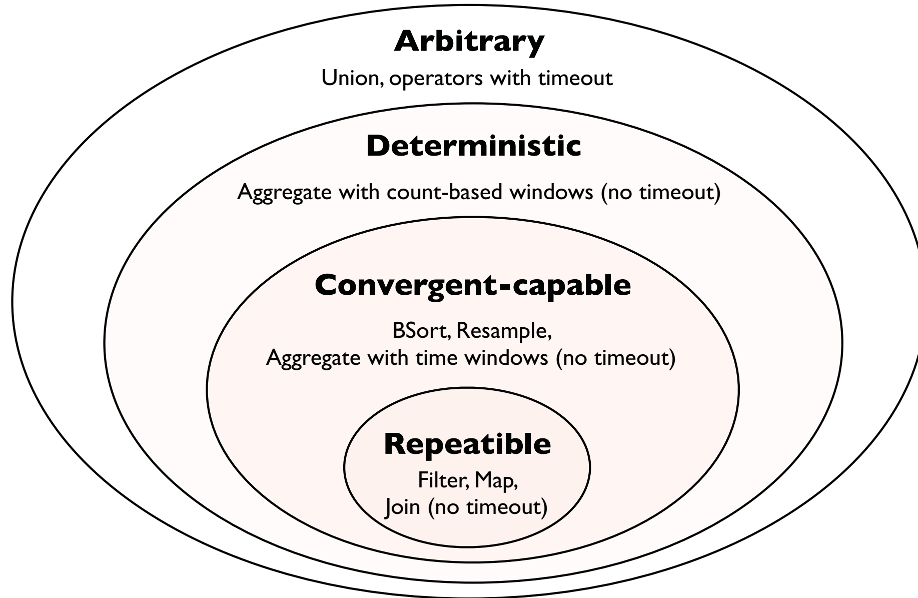


Figure 3.3: Taxonomy of Aurora Operators

t_3) after failure but then produces tuples t_4 and following as in a failure-free execution. Finally, divergent recovery keeps producing equivalent rather than identical tuples after the failure.

Propagation of Recovery Effects. The semantics above define the effects of failure and recovery on the output stream of a query network fragment. These effects then propagate through the rest of the query network until they reach client applications. Because precise recovery masks failures, no side effects propagate. Gap recovery may lose tuples. Therefore, client applications may miss a number of consecutive tuples after a failure. Because the query network may aggregate many tuples into a single output tuple, missing tuples may also result in incorrect output values (e.g., a sum operator may produce a lower sum). No loss recovery does not lose tuples but may generate duplicate tuples. The final output stream may thus contain a burst of either redundant or incorrect tuples (e.g., a sum operator downstream may produce a higher sum value). It is also possible, however, that duplicate-insensitive operators (e.g., max) downstream can always guarantee correct results. Finally, missing or duplicate tuples may have a permanent effect on operators with count-based windows by shifting the window alignment points of those operators. In general, the recovery type for each server must be chosen based on the applications' correctness criteria and the characteristics of the operators on the server and downstream.

3.2.2 Operator Classification

We distinguish four types of operators based on their effects on recovery semantics: *arbitrary* (including non-deterministic), *deterministic*, *convergent-capable*, and *repeatable*. Figure 3.3 depicts the containment relationship among these operator types and the classification of the Aurora/Borealis

operators [2, 11]. The type of a query network is determined by the type of its most general operator.

An operator is *deterministic* if it produces the same output stream every time it starts from the same initial state and receives the same sequence of tuples from each input. There are three possible causes of non-determinism in operators: dependence on time (either execution time or input tuple arrival times), dependence on the arrival order of tuples across different inputs (e.g., Union, which interleaves tuples from multiple streams), and use of non-determinism in processing such as randomization (e.g., random filter, which randomly drops tuples for the purpose of load shedding [2, 134]).

A deterministic operator is called *convergent-capable* if it leads to a convergent recovery when it restarts from an empty internal state and re-processes the same input streams, starting from an arbitrary earlier point in time. To be convergent-capable, an operator must rebuild its internal state from scratch and update it on subsequent inputs in a manner that eventually converges to the execution that would have existed without failure. Window alignment is the only possible cause that prevents a deterministic operator from being convergent-capable. This is because window boundaries define the sequences of tuples that an operator processes to produce an output tuple. Therefore, a deterministic operator is convergent-capable if and only if its window alignments always converge to the same alignment when restarted from an arbitrary point in input.

A convergent-capable operator is *repeatable* if it leads to a repeating recovery whenever it restarts from an empty internal state and re-processes the same input streams, starting from an arbitrary earlier point in time (the operator must produce identical duplicate tuples). A necessary condition for an operator to be repeatable is for the operator to use at most one tuple from each input stream to produce an output tuple. If a sequence of multiple tuples contributes to an output tuple, then restarting the operator from the middle of that sequence may yield at least one different output tuple. Aggregates are thus not repeatable in general, whereas Filter (which simply drops tuples that do not match a given predicate) and Map (which transforms tuples by applying functions to their attributes) are repeatable as they have one input stream and process each tuple independently of others. Join (without timeout) is also repeatable because its windows defined on input streams have alignments relative to the latest input tuple being processed.

In Sections 3.3, 3.4, and 3.5, we present approaches for gap recovery, no loss recovery, and precise recovery, respectively. For each approach, we discuss the impact of the query network type on recovery and analyze the tradeoffs between recovery time and runtime overhead. Table 3.1 summarizes the notation that we use.

3.3 Gap Recovery

The simplest approach to high availability is for each backup server to restart the failed query network from an empty state and continue processing input tuples as they arrive. This approach, called *amnesia*, produces a gap recovery for all types of query networks. In amnesia, the failure detection delay, the rates of input streams, and the state size of the query network determine the

p	per-tuple processing time
d	network transmission delay between servers
λ	input rate
C	size of checkpoint message
c	size of queue trimming message
M	checkpoint or queue trimming interval
D	failure detection delay
r	time to redirect input streams
nb_ops	number of operators in the query network
nb_paths	number of paths from input to output streams
$\boldsymbol{\Delta}$	number of lost or redundant tuples
K	delay before processing first duplicate input tuple
Q	average number of input tuples to process during recovery
rec_time	time spent recreating the failed state (after failure detection)
bw_overhead	$\frac{\text{bandwidth consumed for high availability}}{\text{bandwidth consumed for tuple transmission}}$
proc_overhead	$\frac{\text{CPU cycles consumed for high availability}}{\text{CPU cycles consumed for regular stream processing}}$

Table 3.1: Summary of Notation

	Query Network Type			
Approach	**Repeatable**	**Convergent-Capable**	**Deterministic**	**Arbitrary**
Passive Standby	Repeating	Repeating	Repeating	Divergent
Upstream Backup	Repeating	Convergent	Divergent	Divergent
Active Standby	Repeating	Repeating	Repeating	Divergent

Table 3.2: Types of No Loss Recovery

number, Δ, of lost tuples. This approach imposes no overhead at runtime, as shown in Table 3.3.

We define *recovery time* as the interval between the time when a backup server discovers the failure of its primary and the time it reaches the primary's pre-failure state (or an equivalent state for a non-deterministic query network). Recovery time thus measures the time spent recreating the failed state.

Since amnesia does not recreate the lost state, while dropping tuples until the backup server is ready to accept them, the recovery time is zero. It takes time r to redirect the inputs to the backup, but when processing restarts, the first tuples processed are those that would have been processed at the same time if the failure did not occur. For this reason, there is no extra delay due to failure recovery.

3.4 No Loss Recovery Protocols

We present three approaches for no loss recovery, each one using a different combination of redundant computation, checkpointing, and logging. We first present *passive standby*, an adaptation of the process-pairs model with passive backup. Next, we introduce *upstream backup*, where upstream servers in the processing flow serve as backup for their downstream neighbors by logging their output tuples. Finally, we describe *active standby*, another adaptation of the process-pairs model

	rec_time	bw_overhead	proc_overhead
Amnesia	0	0	0
Passive Standby	$K+Qp$, where $K=r+d$; $Q=\frac{M\lambda}{2}$	$f_1(\frac{1}{M}, C)$	$f_2(\frac{1}{M}, C)$
Upstream Backup	$K+Qp$, where $K=r+d$; $Q=\lvert\text{state}\rvert+M\lambda+2d\lambda$	$f_3(\frac{1}{M}, c)$	$f_4(\frac{1}{M}, \text{nb_ops}, \text{nb_paths})$
Active Standby	ϵ (negligible)	100% + $f_3(\frac{1}{M}, c)$	100% + 2 * $f_4(\frac{1}{M}, \text{nb_ops}, \text{nb_paths})$

Table 3.3: Recovery Time and Runtime Overhead of each High-Availability Approach

where each backup performs processing in parallel with its primary. We discuss active standby last, because it relies on concepts introduced in upstream backup.

For each approach, we examine the recovery guarantees it provides, the average recovery time, and the runtime overhead. We divide the runtime overhead into processing and communication (or bandwidth) overhead. Table 3.2 shows the recovery types achieved by each approach. Table 3.3 summarizes their performance metrics.

3.4.1 Passive Standby

In passive standby, each primary periodically delta-checkpoints onto its backup so that the backup can take over from the latest checkpoint when the primary fails. Here, delta-checkpointing refers to copying the difference between the primary's current state and the state at the time of the previous checkpoint. The state of the primary consists of the states of input queues of operators, operators themselves, and the output queues (one for each output stream to a remote server). Each checkpoint message (a.k.a. state update message) thus captures the changes to the states of the operators and queues on the primary since the last checkpoint message was composed. For each queue, the checkpoint message contains the newly enqueued tuples as well as the last dequeue position. For an operator, however, the content of the message depends on the operator type. For example, the message is empty for stateless operators. On the other hand, the message stores, for an Aggregate operator, some form of summaries (e.g., count, sum) and, for a Join operator, the actual tuples that newly entered the operator's state.

To reduce the suspension of processing, the composition of a checkpoint message is conducted along a virtual "sweep line" that moves from left (upstream) to right (downstream). At every step, an operator closest to the right of the sweep line is chosen and once its state difference is saved in the checkpoint message, the sweep line moves to the right of the operator. The primary is free to execute operators away from the sweep line both upstream and downstream because these concurrent tasks do not violate the consistency of the checkpoint message. Indeed, executing operators to the left of the sweep line is equivalent to executing them after checkpointing. Executing operators to the right of the sweep line corresponds to executing them before the message composition.

Passive standby guarantees no loss recovery for the following reasons:

1. *input preservation* - each upstream primary preserves output tuples in its output queues until

they are safely stored at the downstream backups. In Figure 3.1, for example, whenever backup server $\mathfrak{S}'_2$ receives a checkpoint from $\mathfrak{S}_2$, $\mathfrak{S}'_2$ informs upstream server $\mathfrak{S}_1$ about the new tuples that $\mathfrak{S}_2$ has recently consumed. $\mathfrak{S}_1$ discards only those acknowledged tuples from its output queues.

2. *output preservation* - the backup is always "behind" the primary because its state corresponds to the last checkpointed sate.

If a primary fails, the backup takes over and sends all tuples from its output queues to the downstream servers. The backup also asks upstream primaries to start sending their output streams, including the tuples stored in their output queues. When the failed server joins the system again, it assumes the role of the backup. Because the new backup has an empty state, the primary sends its complete state in the first checkpoint message.

Because the backup restarts from a past state of its primary, passive standby provides repeating recovery for deterministic query networks and divergent recovery for others.

Recovery Time. Passive standby has a short recovery time because the backup holds a complete and recent snapshot of the primary's state. Recovery time is equal to $K + Qp$, where K is the delay before the backup server receives its first input tuple, Q is the number of duplicate input tuples it reprocesses, and p is the average processing time per input tuple. K is the sum of r (the time to redirect input streams) and d (the time for the first tuple to propagate from the upstream primaries). Q is on average half a checkpoint interval worth of input tuples. The average number, Δ, of duplicate tuples is close to $M\lambda_{out}$, where M is the checkpoint interval and λ_{out} is the rate of tuples on output streams.

Overhead. The bandwidth overhead under passive standby is inversely proportional to the checkpoint interval and proportional to the size of checkpoint messages. The processing overhead consists of generating and processing checkpoint messages (proportional to the bandwidth overhead). The checkpoint interval (M) determines the tradeoff between runtime overhead and recovery time. Table 3.3 summarizes these results.

3.4.2 Upstream Backup

In upstream backup, *upstream servers act as backups for their downstream servers* by logging tuples in their output queues until all downstream servers completely process these tuples. In Figure 3.1, for instance, server $\mathfrak{S}_1$ serves as a backup for server $\mathfrak{S}_2$. If $\mathfrak{S}_2$ fails, $\mathfrak{S}'_2$ restores the lost state by re-processing the tuples logged at $\mathfrak{S}_1$. When a failed server rejoins the system, it assumes the role of the backup server starting from an empty state. The system is then able to tolerate a new failure without further delay.

The main difficulty of this approach is to determine the maximum set of logged tuples that can safely be discarded despite the presence of non-deterministic operators and the many-to-many relationship between input and output tuples.

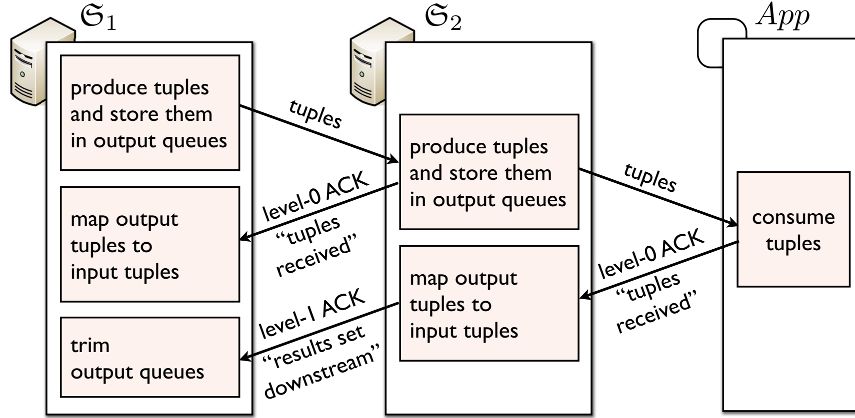


Figure 3.4: Communication between Servers in Upstream Backup

Figure 3.4 shows a typical communication sequence between servers $\mathfrak{S}_1$ and $\mathfrak{S}_2$ and client application *App*. Each server produces and sends tuples downstream while storing them in its output queues. Each server also periodically acknowledges receipt of input tuples by sending level-0 acks to its upstream neighbors. When a server (e.g., $\mathfrak{S}_2$) receives level-0 acks from downstream neighbors (e.g., *App*), it notifies its own upstream neighbors (e.g., $\mathfrak{S}_1$) about the earliest logged tuples (one per $\mathfrak{S}_1$'s output) that contributed to producing the acknowledged tuples (i.e., the oldest logged tuples necessary to re-build its current state). Discarding only earlier tuples allows the system to survive single failures. The notifications are thus called level-1 acks (denoted ACK$(1, S, u)$, where S identifies a stream and u identifies a tuple on that stream). Servers that output to client applications use level-0 acks to trim output queues.

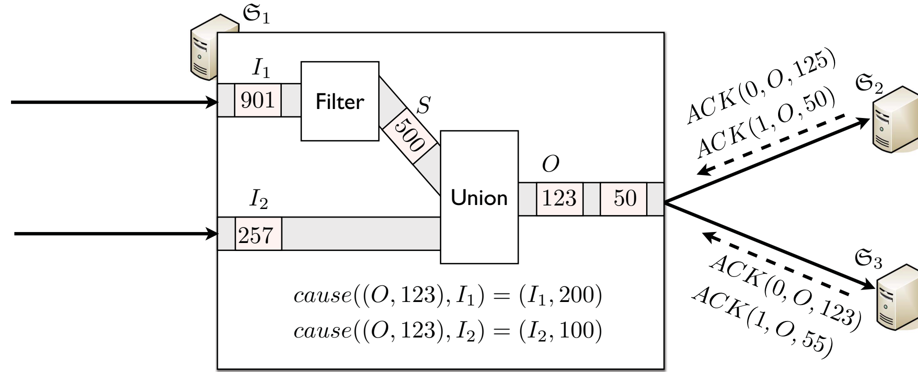
Upstream backup provides no loss recovery as follows:

- *input preservation* - upstream servers log all the tuples necessary for the backup to re-build the primary's state from an empty state.
- *output preservation* - the backup restarts from an empty state.

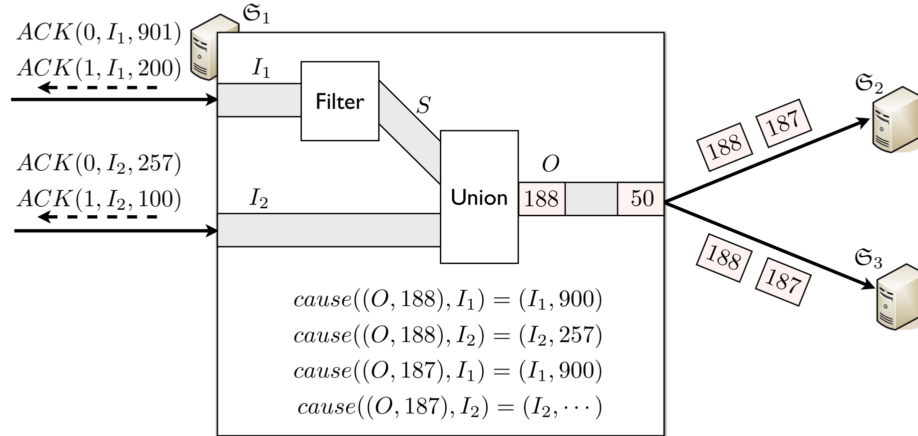
Queue Trimming Protocol

To avoid spurious transmissions, servers produce both level-0 and level-1 acks every M seconds. A lower ack frequency reduces bandwidth utilization, but increases output queue size and recovery time.

To compose level-1 acks, each server finds, for each output stream O, the latest output tuple $O[v]$ acknowledged at level-0 by all downstream neighbors. For each input stream I, the server maps $O[v]$ back onto the earliest input tuple $I[u]$ that caused $O[v]$. This backward mapping is conducted by a function $cause((O, v), I) \rightarrow (I, u)$, where (I, u) denotes the identifier of tuple $I[u]$ and marks the beginning of the sequence of tuples on I necessary to regenerate $O[v]$. We discuss the *cause* function in the next section. The server performs these mappings for each output stream and identifies the



(a) The Filter produces $S[500]$ from $I[900]$. Then, $\mathfrak{S}_1$ receives acks from downstream neighbors and new tuples $I_1[901]$ and $I_2[257]$ from upstream neighbors.



(b) $\mathfrak{S}_1$ trims its output queue at $(O, 50)$ while pushing new tuples $O[187]$ and $O[188]$ downstream. $\mathfrak{S}_1$ also maps the lowest level-0 ack received, $(O, 123)$, onto level-1 acks $ACK(1, I_1, 200)$ and $ACK(1, I_2, 100)$.

Figure 3.5: One Iteration of Upstream Backup

earliest tuple on each input stream that can now be trimmed. The server produces level-1 acks for these tuples. Each upstream neighbor trims its output queues up to the position that corresponds to the oldest tuple acknowledged at level-1 by all downstream neighbors.

Figure 3.5 illustrates one iteration of the upstream-backup algorithm on one server. In the example, server $\mathfrak{S}_1$ receives level-0 and level-1 acks from two downstream neighbors $\mathfrak{S}_2$ and $\mathfrak{S}_3$. First, since both neighbors have sent level-1 acks for tuples up to $O[50]$, $\mathfrak{S}_1$ removes from its output queue all tuples preceding $O[50]$. Second, since both $\mathfrak{S}_2$ and $\mathfrak{S}_3$ have sent level-0 acks for tuples up to $O[123]$, $\mathfrak{S}_1$ maps $O[123]$ back onto the first input tuples that caused it. $\mathfrak{S}_1$ sends level-1 acks for these tuples, identified with $(I_1, 200)$ and $(I_2, 100)$. In the example, $\mathfrak{S}_1$ also receives tuples $I_1[901]$ and $I_2[257]$ from its upstream neighbors and acknowledges the recept of them with level-0 acks.

Mapping Output Tuples onto Input Tuples

We now discuss how servers compute the cause function, $cause((O, v), I) \rightarrow (I, u)$. This function maps an arbitrary output tuple $O[v]$ on stream O onto the earliest input tuple $I[u]$ on input stream I that has contributed to the production of $O[v]$ (i.e., affected the value of $O[v]$). To facilitate this mapping, we propose to keep track of the oldest input tuples that affect any computation, by appending *input-tuple indicators* to tuples as they travel through operators on a server. For a tuple $O[v]$, these indicators, denoted with $indicators(O, v)$, contain the identifiers of the oldest tuples on input streams necessary to generate $O[v]$. We also call these indicators *low watermarks*. On any stream, indicator values are monotonically non-decreasing.

When a tuple enters a server, its indicators are initialized to its identifier: e.g., $indicators(I, u) = \{(I, u)\}$. Each operator uses the indicators of its input tuples to compute the indicators for its output tuples. When it is first set up, each operator o initializes a watermark variable ω for each server-wide input stream I that contributes to each input stream S of o (i.e., $\omega[I, S] \leftarrow 0$). As it processes tuples, the operator updates each $\omega[I, S]$ to hold the corresponding indicator of the oldest tuple currently in the state or, for stateless operators, that of the last tuples processed. When it produces a tuple t, the operator iterates through all ω values and appends (I, ω_{min}) to $indicators(t)$, where ω_{min} is the minimum of all $\omega[I, *]$.

Some operators, such as Union, have multiple input streams but only a few of them actually contribute to any single output tuple. These operators can reduce the number of indicators on output tuples by appending only indicators for input streams that actually affected the output tuple value. Thus, $cause((O, v), I)$ refers to the indicator of $O[v]$ that corresponds to stream I, or to the indicator of the most recent preceding tuple affected by I, if $O[v]$ was not affected by I. Note that indicators are not sent to downstream servers.

Figure 3.5 shows an example of managing input-tuple indicators. In Figure 3.5(a), the Filter produces $S[500]$ from $I_1[900]$. Hence, $indicators(S, 500)=\{(I_1, 900)\}$. In Figure 3.5(b), the Union operator processes tuples $S[500]$ and $I_2[257]$ to produce $O[187]$ and $O[188]$ respectively. Hence, $indicators(O, 187) = \{(I_1, 900)\}$ and $indicators(O, 188) = \{(I_2, 257)\}$. Therefore, $cause((O, 188), I_1)$ $= (I_1, 900)$, $cause((O, 188), I_2) = (I_2, 257)$, and $cause((O, 187), I_1) = (I_1, 900)$. On the other hand, $cause((O, 187), I_2)$ depends on the indicators of the tuples preceding O.

Upstream backup restarts from an empty state producing a repeating recovery for repeatable query networks, a convergent recovery for convergent-capable query networks and a divergent recovery for all others. These guarantees are weaker than those of the standby approaches.

Recovery Time. The time, K, to receive the first tuple is the same as for passive standby but the recovery server may re-process significantly more tuples. It must re-process (1) all tuples that contributed to the lost state, (2) a complete queue-trimming interval worth of tuples on average (due to the periodic transmission of both level-0 and level-1 acks), and (3) some extra tuples that account for the propagation delays of level-0 acks. The number, Δ, of redundant tuples is the product of the number of tuples to reprocess (Q) and the query network selectivity minus the number of tuples that remain as part of the query network state.

Overhead. Upstream backup has the lowest bandwidth overhead because queue-trimming messages, which contain only the tuple identifiers for streams crossing server boundaries, are significantly smaller than checkpoint messages used by the other approaches. The processing overhead is also small since operators keep track of the oldest tuple (and its indicators) on each of their input streams that contributes to their current states. Furthermore, we can reduce the spatial and computational overhead of managing indicators by processing them and appending them to tuples occasionally. In general, the total overhead, as summarized in Table 3.3, is proportional to the number of operators and the number of paths, where a path is a data flow connecting an input stream to an output stream.

3.4.3 Active Standby

Active standby is another variation on the process-pairs model. In contrast to passive standby, each backup server in active standby receives tuples from upstream neighbors and processes them in parallel with the primary. The backup server, however, does not send any output tuples downstream. It logs these tuples in its output queues instead.

The challenge of active standby lies in bounding the output queues on each backup, while ensuring output preservation. Because the primary and backup may have non-deterministic operators, they may have different tuples in their output queues. To identify duplicate output tuples, we add a second set of input-tuple indicators to each tuple. For a tuple, $O[v]$, this second set contains for each input stream I, the identifier (I, u) of the *most recent* tuple that contributed to the production of $O[v]$. We call these identifiers *high watermarks*. A tuple at the backup server is duplicate if it has a lower-valued high watermark than a tuple at the primary. Indeed, this tuple results from processing the same or even older input tuples. Each backup server thus trims all logged output tuples that have a high watermark lower than the high watermarks of the tuples already received by downstream servers. For high watermarks to be correct, we need to distinguish input-tuple indicators that travel on different paths through a server.

Watermarks are never sent between upstream and downstream servers but they are sent between primary and backup servers, as illustrated in the following example. We use Figure 3.5 to illustrate active standby but we assume indicators are high watermarks. When $\text{ACK}(0, O, 125)$ and $\text{ACK}(0, O, 123)$ arrive, server $\mathfrak{S}_1$ determines that $O[123]$ is now acknowledged at level-0 by both downstream neighbors. Since tuple $O[123]$ maps onto input tuples identified with $(I_1, 200)$ and $(I_2, 100)$, the set of identifiers $\{(I_1, 200), (I_2, 100)\}$ is added to the queue-trimming message as the entry value for O. When the backup server $\mathfrak{S}'_1$ receives the queue-trimming message, it discards tuples u (from the output queue corresponding to O) for which $cause((O, u), I_1)$ returns a tuple older than $I_1[200]$ and $cause((O, u), I_2)$ returns a tuple older than $I_2[100]$.

If the primary fails, the backup takes over by sending the logged tuples to all downstream neighbors and continuing its processing. When the failed server rejoins the system as the new backup, it starts with an empty state and becomes up-to-date with the new primary only after processing sufficiently many input tuples. Active standby guarantees no loss recovery since:

	Passive standby		
Q. network	bw_overhead	proc_overhead	rec_time
Deterministic	none	negligible	none
Arbitrary	none	negligible	none
	Active standby		
Q. network	bw_overhead	proc_overhead	rec_time
Deterministic	none	negligible	r
Arbitrary	determinants	determinants	$r + f_5$(log. freq.)
	Upstream backup		
Q. network	bw_overhead	proc_overhead	rec_time
Repeatable	$\frac{f(k)*\text{size(tuple_id)}}{\text{size(tuple)}}$	negligible	none
Convergent	$\frac{f(k)*\text{size(tuple_id)}}{\text{size(tuple)}}$	double	negligible
Arbitrary	determinants	determinants	negligible

Table 3.4: Added Overhead for Precise Recovery

- *input preservation* - each backup always receives what its primary receives.
- *output preservation* - each backup discards logged output tuples only when they become duplicate.

Because backups process tuples in parallel with primaries, active standby provides repeating recovery for all deterministic query networks and divergent recovery for others.

Recovery Time. Because the backup continues processing during failure, it only needs to transmit all duplicate tuples in its output queue to reach a state equivalent to that of the primary. Recovery time is therefore negligible. The number, Δ, of redundant tuples is on average $\frac{M\lambda_{out}}{2} + 2d\lambda_{out}$ for each output stream. M determines the trimming interval for the backup's output queues.

Overhead. Because all processing is replicated by the backup server, both proc_overhead and bw_overhead are approximately 100%. The overheads are actually somewhat higher due to the processing of input-tuple indicators and transmitting queue-trimming messages. Table 3.3 summarizes these results.

3.5 Extensions for Precise Recovery

Our recovery approaches can achieve precise recovery for convergent-capable query networks, by eliminating duplicate tuples during convergence. It is also possible, though much more costly, to provide precise recovery for arbitrary networks. Table 3.4 summarizes the extra runtime overhead and recovery time required for precise recovery.

3.5.1 Passive Standby

Passive standby provides repeating recovery for deterministic query networks. To make recovery precise, before sending any output tuples, the failover server must ask downstream neighbors for the

identifiers of the last tuples they received and then discard all tuples preceding the identified ones. These requests can be made while the recovery server regenerates the failed state, thereby achieving precise recovery without additional delay. Given a non-deterministic query network, passive standby in its original form provides divergent recovery. One solution to enforce precise recovery in this case is to hold the outputs of the primary (while keeping them in output buffers) until the next checkpoint completes. In this solution, the backup always possesses a state that has generated all the output tuples that the primary has sent downstream. Thus, the backup can achieve precise recovery by removing duplicates in its output buffers and sending downstream the tuples remaining in the buffers. This solution, however, causes bursty output and increases output latency.

3.5.2 Active Standby

Given a deterministic query network, active standby can provide precise recovery by asking downstream servers for the identifiers of the latest tuples they received. The delay imposed by this request cannot be masked and thus extends the recovery time by r. For other non-deterministic query networks, precise recovery requires that both the primary and backup produce the same outputs. To do so, whenever a non-deterministic operator executes, the primary collects information that enables the backup to run the operator identically. We call such information determinants [52][1]. Sending determinants from primaries to backups affects both bandwidth and processing overhead. The frequency of sending determinants affects the recovery time because non-deterministic operators on the backup cannot run until they obtain appropriate determinants. The frequency also affects the output latency since the primary cannot send tuples downstream until the backup receives all the determinants related to the tuples.

3.5.3 Upstream Backup

In repeatable query networks, operators produce output tuples by combining at most one tuple from each input stream. Input-tuple indicators therefore uniquely identify tuples and can serve for duplicate elimination, offering precise recovery with negligible extra processing overhead. For a convergent query network, the backup must be able to remove duplicate output tuples during recovery. It achieves this by using the additional high watermarks as discussed in Section 3.4.3. This approach thus doubles the processing overhead. In both cases above, the extra bandwidth overhead is approximately $\frac{f(k) * \text{size(tuple_id)}}{\text{size(tuple)}}$, where $f(k)$ is a function of the average number of input streams (at a server) that contribute to an output stream. As in active standby, upstream backup can provide precise recovery for more complex query networks by logging determinants from primary to backup. Unlike active standby, these determinants are processed only when the backup takes over.

[1]The representation of a determinant depends on the operator type. For example, determinants for Union must include the exact inter-arrival order of input tuples through multiple inputs.

Parameter	Meaning	Default
λ	input tuple arrival rate (tuples/s)	1000
D	delay to detect the failure of a server (ms)	250
M	queue-trimming/checkpoint interval (ms)	50
r	time to redirect input streams (ms)	40
Tuple	size of a tuple and size of a tuple id (bytes)	50, 8
Network	bandwidth(Mbps) and delay(ms)	16, 5
Proc. Cost	avg. processing time per input tuple (μs)	
Filter	—	10
Aggregate	(Proc. Cost of Filter) $*$ Window $* \frac{1}{\text{Advance}}$	100
Selectivity	expected value of $\frac{\text{\# of output tuples emitted}}{\text{\# of input tuples consumed}}$	0.1

Table 3.5: Simulation Parameters and Their Default Values

3.6 Evaluation

We evaluate and compare the performance of our recovery approaches, based on our Aurora/Borealis prototype and a detailed simulator. The simulator was written in C++ using the CSIM [96] library. Table 3.5 summarizes the main simulation parameters. Some of the parameter values were obtained from our Aurora/Borealis prototype. Each point in Figures 3.8 and 3.9 is the average of 25 simulation runs, at least one simulated minute each. Because amnesia has no overhead and a zero recovery time, while providing only gap recovery, we focus our evaluation on the other three approaches.

We first examine the overhead and recovery performance of each approach for no loss recovery (Section 3.6.1). We then evaluate the added overhead of achieving precise recovery (Section 3.6.2) and examine the effect of query network types and other query network properties on the performance of each approach (Sections 3.6.3, 3.6.4, 3.6.5). Finally, we examine how performance changes as a function of query network size (Section 3.6.6).

3.6.1 Runtime Overhead vs Recovery Time

We observe the runtime overhead and recovery time tradeoffs using results from our Aurora/Borealis prototype. The first operator that we examined was an Aggregate that processed a wide-area TCP packet trace [15]. The input rate of the operator was 2.0K tuples/sec on average. The operator groups packets by source IP address while counting the number of packets using windows of 20 seconds that appear every 1 sec. The processing load of the operator was approximately 3% on our AMD Sempron 2000+ CPU with 1GB main memory.

Figures 3.6 and 3.7 show the CPU and bandwidth costs of our recovery techniques for the Aggregate operator. We can see that the CPU and network overhead of upstream backup is near zero because the backup is idle during non-failure periods. The recovery time in this case, however, is approximately 0.6 seconds since the backup during recovery has to reprocess 20 seconds (i.e., the window size) worth of input tuples and processing 1 second worth of these tuples while fully utilizing the CPU would take 0.03 seconds.

In contrast to upstream backup, active standby uses 100% more CPU and network resources

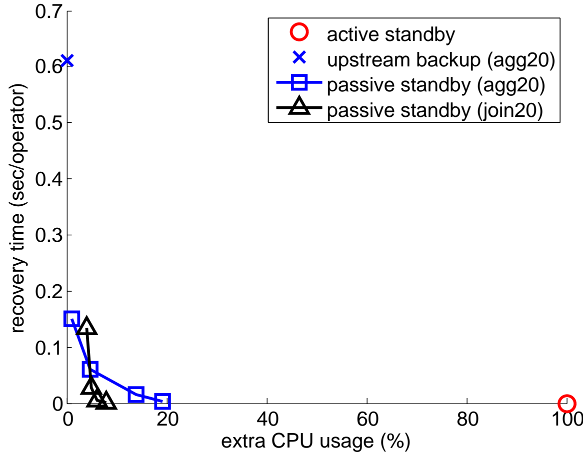


Figure 3.6: CPU Overhead and Recovery Time

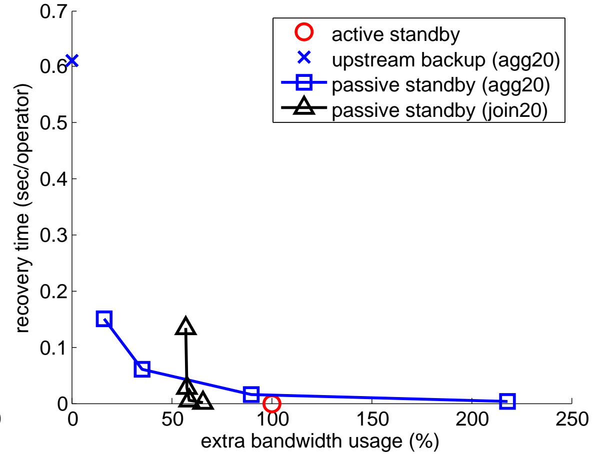


Figure 3.7: Bandwidth Overhead and Recovery

because the backup receives/processes data as the primary. Despite this penalty, active standby provides very fast recovery since the backup is always up-to-date.

Figures 3.6 and 3.7 also illustrate how the runtime overhead and recovery time of passive standby vary as we change the checkpoint interval. Initially, the Aggregate operator was checkpointed every 10 seconds. The average recovery time in this case is 0.15 seconds because the backup is 5 seconds stale on average and using the CPU only to reprocess 1 second worth of input tuples would take 0.03 seconds. As the checkpoint frequency increases (i.e., checkpoint interval decreases), the recovery time decreases because the backup in general becomes more up-to-date. The runtime overhead, however, increases as more changes in the primary's state are sent to the backup.

As Figures 3.6 and 3.7 present, the cost of checkpointing a Join operator is not very sensitive to the checkpoint interval. This is because the operator maintains an append only data structure in its state and thus any element in the state is checkpointed at most once.

Figure 3.8 illustrates supplementary results that we obtained from our simulator. For this evaluation, we assumed that an Aggregate with a window size of 100 ms and a step size of 10 ms consumes 10% of a server's processing capacity. For other parameters, we used the default values.

The only tunable parameter for each approach is the communication interval. In upstream backup and active standby, this interval corresponds to the queue-trimming interval. In passive standby, it represents the checkpoint interval. Figure 3.8 shows the relation between recovery time and bandwidth overhead as the communication interval varies from 25, to 50, 100, 150, and 200 ms.

In terms of the runtime overhead, upstream backup is the clear winner with an overhead close to zero. Even with a 25 ms communication interval, the server transmits only one 8-byte tuple identifier for every 25 tuples it receives, leading to an overhead of 0.64%. Upstream backup, however, has the slowest recovery speed because it must recreate the complete state of the failed query network. Compared to others, upstream backup's recovery time is the most sensitive to the communication interval. Frequent queue trimming reduces recovery time for a negligible added overhead until the size of the query network and the time to redirect the input streams (r is 40 ms in our prototype)

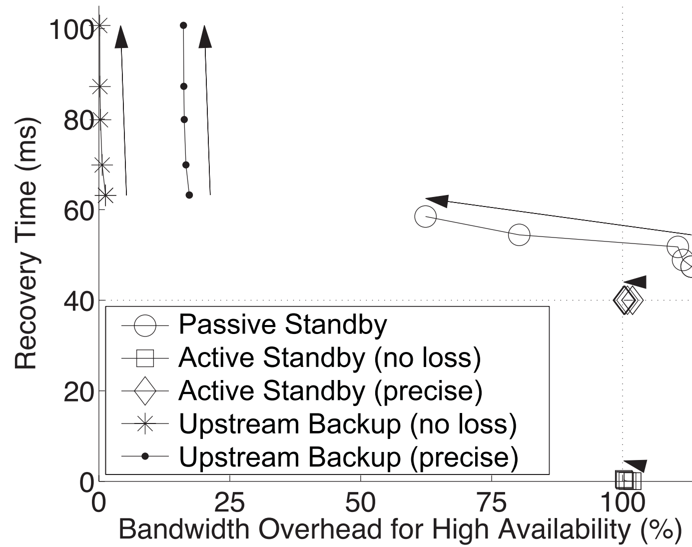


Figure 3.8: Bandwidth Overhead and Recovery Time (the communication interval varies from 25 ms to 200 ms as indicated by the arrows)

eventually limits the recovery speed. In all of the cases, upstream backup's recovery time is relatively short compared to the 250 ms failure detection delay.

Active standby has an overhead of at least 100% because the backup receives all input tuples in parallel with the primary. Queue-trimming messages to discard output tuples from the backup make the overhead slightly exceed 100%. The main advantage of active standby is that it has the fastest recovery speed (i.e., it has a negligible recovery time). During recovery, the backup only needs to resend half the queue-trimming interval worth of duplicate tuples stored in its output queues.

In terms of recovery speed, passive standby ranks between active standby and passive standby. In passive standby, the backup already has a snapshot of the last checkpoint but must re-process, on average, half the checkpoint interval worth of tuples. Passive standby's overhead varies significantly with the checkpoint interval because each checkpoint message contains the change in the query network state since the last checkpoint. In Figure 3.8, the knee at the 100 ms communication interval represents that the size of each checkpoint message does not grow if the interval gets longer than 100 ms (due to the 100 ms window size of the Aggregate). The curve would be smoother for a larger window size.

3.6.2 Cost of Precise Recovery

Figure 3.8 also presents the recovery time and runtime overhead of precise recovery. In passive standby and active standby, precise recovery of convergent-capable query networks adds no runtime overhead compared to no loss recovery. Precise recovery increases the runtime overhead of upstream backup by slightly over 16% (equal to $\frac{k * \text{size(tuple_id)}}{\text{size(tuple)}}$, with $k = 1$ and $\frac{\text{size(tuple_id)}}{\text{size(tuple)}} = \frac{8}{50} = 0.16$) because watermarks are now sent downstream. The overhead thus still remains much lower than

Query Network Type	Result	Upstream Backup	Active Standby	Passive Standby
Repeatable	Bw overhead (%)	0.64	100.96	101.27
	Rec. time (ms)	47.62	1.80	45.88
Convergent-capable	Bw overhead	0.64	100.96	**111.55**
	Rec. time	**69.86**	0.07	48.88
Non-deterministic	Bw overhead	1.28	101.91	101.90
	Rec. time	50.92	1.82	47.24

Table 3.6: Effects of Query Network Type

that of the process-pair based approaches.

In upstream backup and passive standby, the precise recovery time is almost the same as the no loss recovery time. For precise recovery, upstream backup must process additional offset indicators. This, however, adds negligible delay. In all of the three approaches, backup servers must ask downstream neighbors for the latest tuples they received. In upstream backup and passive standby, this communication proceeds in parallel with the re-processing of tuples at the backup server. In contrast, active standby cannot mask this delay, thus increases the recovery time by r (40 ms in our prototype). In summary, all of the approaches can offer precise recovery for convergent-capable query networks at a negligible extra cost.

3.6.3 Effects of Query Network Type on Recovery

We now examine the effects of query network types on the basic performance of no loss recovery. Table 3.6 summarizes the recovery time and bandwidth overhead of each approach for a repeatable Filter with selectivity 1.0, a convergent-capable Aggregate, and a non-deterministic Union that merges two streams (500 tuples/second each) into a stream. Interestingly, the results show that the type of the query network does not affect the overheads as well as the recovery times of the approaches. In contrast, the size of the query network state and the rate and magnitude of the state changes affect the overhead and recovery time of passive standby and the recovery time of upstream backup. The reasons are as follows:

- Upstream backup and active standby use queue-trimming messages. The cost of managing these messages (i.e., the overhead of these approaches) is determined by the relative rates of these messages and tuples, rather than any other property of the query network. In Table 3.6, the Union has a slightly higher overhead than others because it has two input streams at half the rate each.

- The overhead of passive standby depends on the checkpoint frequency and the size of each checkpoint message. The size of each checkpoint message is proportional to the magnitude of changes in the query network state between consecutive checkpoints. In Table 3.6, the Aggregate has the highest overhead because it has the greatest difference in state between checkpoints.

Window size (tuples)	100	200	300	400	500
PS overhead (%)	111.55	111.55	111.54	111.54	111.54
PS rec. time (ms)	48.9	51.7	54.6	60.0	63.9
UB rec. time (ms)	69.9	98.9	138.7	188.5	248.3

Table 3.7: Effects of Query Network State Size

Advance (tuples)	100	50	25	10	5
PS overhead (%)	102.6	103.6	105.6	111.6	121.5
PS rec. time (ms)	47.5	47.5	47.6	48.8	51.6
UB rec. time (ms)	62.6	61.4	61.3	69.9	83.8

Table 3.8: Effects of Rate of Query Network State Change

- The recovery speeds of passive standby and upstream backup are determined by the amount of tuples that the backup must re-processes during recovery. In both approaches, the recovery speed thus depends on processing complexity. In upstream backup, the number of tuples that the backup must re-process is affected by the size of the query network state. For these reasons, the Aggregate has the longest recovery time with these approaches, particulary with upstream backup. In passive standby, the increase in recovery time is negligible compared to the stream redirection delay.

- The recovery time of active standby is determined by the amount of tuples that the backup transmits from its output queues during recovery. The amount of these tuples depends on the queue-trimming interval and the output rate of the query network. In Table 3.6, the Aggregate has a faster recovery speed than other operators because it has a ten times lower output rate due to its 10 ms step size.

3.6.4 Size of Query Network State

We examine the effects of increasing the size of the query network state for an Aggregate operator with increasing window size (100 to 500 tuples) and a constant 10-tuple step size. Table 3.7 shows the resulting passive standby (PS) overhead and both passive standby and upstream backup (UB) recovery times.

Increasing the size of the query network state does not necessarily increase the rate at which that state changes. In this experiment, the overhead of passive standby remains constant at 112%. The recovery time of passive standby due to reprocessing tuples (the part in excess of 40 ms) increases by about a factor of three when the size of the state quintuples. This increase is due to the heavier per-tuple processing cost (i.e., as the aggregate values are computed over larger numbers of tuples). The increase in recovery time is more pronounced for upstream backup. The time spent reprocessing tuples increases roughly linearly with the size of the state.

3.6.5 Rate of Query Network State Change

We examine the impact of increasing the rate at which the state of a query network changes. For this, we decrease the step size of an Aggregate operator from 100 ms to 5 ms (i.e., increase the selectivity from 0.01 to 0.2). Table 3.8 shows the impact of increasing the update rate in the query network state on the overhead of passive standby and the recovery times of passive standby and upstream backup.

As expected, the overhead of passive standby increases with the magnitude of changes in query network state. The step size of the Aggregate determines the number of tuples that the operator produces during a checkpoint interval. This number increases from 1 to 20 as the step size decreases from 100 to 5 ms. The increase in per-tuple processing cost (due to a smaller step size) slightly prolongs recovery (visible for a step size of 10 tuples of less). Although one might expect the same effect to cause a slight increase in the recovery time of upstream backup, we observe a decrease instead. The reason is that upstream backup periodically updates the identifiers of the oldest tuples on each input stream that contribute to the current query network state. When the state changes more rapidly, the older tuples are discarded faster and recovery restarts from a later point. This in turn results in a faster recovery. For a small enough step size, however, the added processing cost dominates the recovery time. As the step size reaches 10 ms, the recovery time starts increasing.

In summary, the size of the query network state increases affects backup's recovery time. On the other hand, the rate and magnitude at which that state changes impacts the runtime overhead of passive standby.

3.6.6 Effect of Network Size

Increasing the size and complexity of the query network translates into increasing the size of the query network state, the rate at which this state changes, and the processing complexity. As an example, Figure 3.9 shows the performance of each approach for a chain of 1 to 5 Aggregate operators (with the parameter values from Table 3.5). Other configurations yield similar results.

As expected, increasing the number of operators increases the overhead of passive standby because the number of tuples that are produced inside or at the output of the query network increases. Larger query networks also slightly increase recovery time for passive standby because the processing complexity of each tuple increases. The recovery time of upstream backup increases rapidly as the state of the query network increases with each extra Aggregate. It reaches 170 ms for 5 operators, which is still relatively short compared to the 250 ms failure detection delay. Interestingly, even with a larger query network, upstream backup still provides precise recovery at a fraction of the cost of the other approaches.

3.6.7 Discussion

The results show that each approach poses a clear tradeoff between recovery time and processing overhead. Active standby, with its high overhead and negligible recovery time, appears particularly

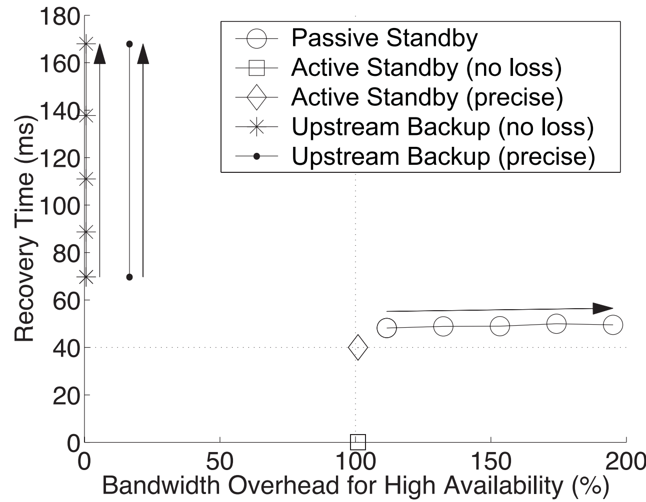


Figure 3.9: Effects of the Number of Operators (the arrows indicate the directions of the trends)

well suited for systems where quick recovery justifies high runtime costs (e.g., financial services, military applications).

Passive standby is the only approach that easily provides precise recovery for arbitrary query networks. It is thus best suited for applications, such as patient monitoring and other medical applications, that impose a somewhat lower load on the system but necessitate precise recovery. Additionally, both in our prototype and simulator, we make the first servers in the system adopt the passive-standby model since other approaches impose extra requirements on stream sources: active standby requires that each source sends the stream to two different locations and upstream backup requires that each source logs the tuples it produces. Passive standby can also withstand high load situations by less frequent checkpoints (i.e., by trading off recovery speed).

Upstream backup provides precise recovery for most query networks with the lowest runtime overhead but at the cost of a longer recovery. The recovery time of this approach, however, can be significantly reduced by distributing the recovery load over multiple servers. In general, upstream backup is appropriate when efficient resource usage is important, query stats are small, and moderate recovery delays are tolerable.

3.7 Summary

In this chapter, we showed that the distributed and dataflow nature of stream processing applications raises novel challenges and opportunities for high availability. We defined three recovery guarantees and categorized operators based on their impact on the cost of providing these recovery guarantees. Within this framework, we introduced three recovery approaches that provide the proposed guarantees with different combinations of redundant processing, checkpointing, and logging.

Using analysis and results obtained from our Aurora/Borealis prototype and a detailed simulator, we quantitatively characterized the runtime overhead and recovery time tradeoffs among the approaches. We also found that each approach covers a complementary portion of the solution space. Active standby provides the fastest recovery but at a high cost. It is thus best suited for environments where fast failure recovery (i.e., minimal disruption) justifies higher runtime costs. Passive standby can easily provide precise recovery for arbitrary query networks. It can also balance the recovery speed and resource usage by adjusting the checkpoint interval. Compared to these approaches, upstream backup has a significantly lower runtime overhead but a longer recovery time that depends mostly on the size of the query network state. This approach is thus appropriate for an environment where failures are infrequent and moderate recovery delays are tolerable.

Chapter 4

Highly-Available Stream Processing in Server Clusters

In this chapter, we devise a parallel recovery solution for server clusters. While the methods in Chapter 3 mask a server failure by failing over to a dedicated server, our approach in this chapter enables multiple servers to collectively take over the failed execution, realizing significantly faster recovery. This approach also has low negative impact on regular processing because it uses idle resources to conduct short-duration tasks for high availability.

To maintain backups spread over multiple machines, the developed technique uses *checkpointing*, which periodically copies any change in state to the backups. The reason for using checkpointing for server clusters is that checkpointing effectively works for a larger set of workload and usage cases than other alternatives that are based on either replay or redundant parallel execution (see Sections 4.1 and 4.6.3 for a detailed discussion). For example, checkpointing can gracefully deal with situations where the processing load nearly reaches the system's capability, whereas redundant execution (i.e., active standby) always requires at least half the resources to be available and devoted to high availability.

In this chapter, we tackle the following subproblems.

- **Query Partitioning.** Each server needs to partition its query graph into several subgraphs so that each subgraph can be assigned a different backup server. Because we each time checkpoint only one of these subgraphs, we call these subgraphs *checkpoint units*. We study the problem of forming checkpoint units as well as preserving safety against failures while the system reforms these units.

- **Backup Assignment.** We need an algorithm that finds an appropriate backup server for each checkpoint unit. Our backup assignment algorithm balances the checkpoint load and minimizes the expected recovery time.

- **Checkpoint Scheduling.** Because we rely on checkpointing for high availability, we need a

method that determines the order and the frequency of checkpoints. Our scheduling algorithm takes into account the characteristics of operators such as processing load and checkpoint cost.

In summary, our cooperative, fine-grained recovery approach has the following advantages.

1. Since the recovery load is distributed and balanced over multiple servers, the approach can take advantage of parallelism during recovery. This significantly improves the recovery speed.
2. Each server checkpoints only a small fraction of its query graph at each step. Therefore, compared to the previous techniques that checkpoint the entire state of a server, a checkpoint defers processing for a much shorter duration.
3. Each server strives to fully use its spare CPU cycles (i.e., those left after regular processing) to maximize the recovery speed.
4. If a server fails, each backup server takes over only a fragment of the query graph from the failed server. After recovery, each server thus experiences only a small increase in its processing load.
5. The framework is adaptive and does not require human administration (e.g., no primary/backup designation).

The rest of this chapter is organized as follows. We devise our backup framework in Section 4.1 and discuss forming checkpoint units in Section 4.2. Next, we present algorithms for checkpoint scheduling and backup assignment in Sections 4.3 and 4.4, respectively. In Section 4.5, we analyze stream processing operators and design efficient delta-checkpointing techniques. We demonstrate the experimental results in Section 4.6 and provide a summary of this chapter in Section 4.7. In Section 4.8, we formally define some important terms used in this chapter.

4.1 Our Backup Model

Each high-availability method described in Chapter 3 has unique benefits in terms of resource usage, recovery speed, recovery semantics, and the impact on regular processing. In this chapter, we focus on *checkpointing* (i.e., passive standby) as the underlying high-availability method. As we argue below, our choice is primarily due to the observation that checkpointing can effectively address a larger set of workload and configurations than other alternatives:

- Despite its fast recovery speed, we do not use active standby for server clusters. This is because it may not withstand *high load* situations that checkpointing can tolerate. In active standby, backups must consume the same amount of resources as primaries. In contrast, checkpointing requires far less resources because it each time copies only the *difference* between the current state and the state at the time of the previous checkpoint. For example, to checkpoint an Aggregate operator, we need to copy only the most recent summary value rather than all

the summary values overwritten since the last checkpoint. Similarly, checkpointing need not capture the values that newly entered but left the state of operators and the queues between operators.

- We do not use upstream backup as the general high-availability approach for server clusters. The reason is that it cannot not adequately support operators with large states. During recovery, for example, the backup of an operator with a window size of 10 minutes must re-process 10 minutes worth of input tuples.

The arguments above are experimentally demonstrated in Sections 4.6.2 and 4.6.3. As mentioned in Section 3.6.7, checkpointing also has an advantage that it can easily handle *non-deterministic* operators, whereas the other alternatives require complex solutions.

In this section, we state the assumptions behind our work. We also describe our recovery framework as well as its operation during non-failure and failure periods.

4.1.1 Assumptions

- **System Configuration.** We assume that a large number of servers are grouped into *logical clusters* (e.g., those of 5-20 servers) and study how the servers in each cluster can cooperate with each other to achieve fast failure recovery.

- **Communication.** We assume that servers are connected with a fast, reliable network (e.g., gigabit LAN). The communication protocol guarantees robust message delivery and also preserves message ordering. We do not consider network failures that isolate server clusters [22].

- **Failure Model.** We assume that all servers are subject to failure and a failed server stops functioning (i.e., fail-stop). We also assume that a server failure is a rare event and thus aim at protection against *single* server failures. It is generally acknowledged that a *1-safety* guarantee is sufficient for most real-world applications [62].

- **Processing Load.** We assume that the overall processing load is most of the time under the system's processing capability and well balanced over the servers [149, 148]. Based on this assumption, we focus on developing techniques that can use idle CPU cycles for high availability. We do not consider medium- to long-term overload situations because they in general necessitate load shedding [134] to favor timeliness over correctness, contradicting the principle of high availability.

4.1.2 The Basic Architecture

In distributed stream processing, stream operators are distributed over multiple servers in a scalable manner [41, 120, 149]. We call the mapping between the operators and the servers that execute them a *query deployment plan*. Formally, given a set of servers $\{\mathfrak{S}_i\}_{i=1}^{n}$ and a query network Q (i.e.,

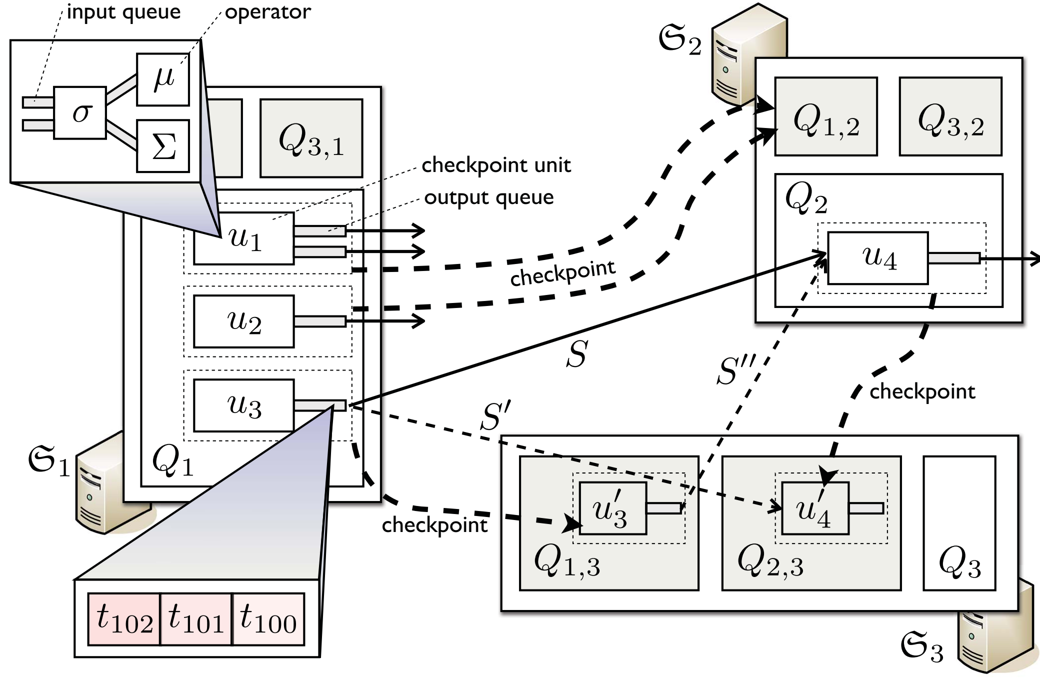


Figure 4.1: The Backup Model

the union of all queries submitted), we denote a query deployment plan with $\{Q_i\}_{i=1}^n$, where each Q_i represents the set of operators that server $\mathfrak{S}_i$ runs. We call Q_i the query on server $\mathfrak{S}_i$.

Once the system launches the query network, the query on each server can be viewed as a set of connected subgraphs. In Figure 4.1, on server $\mathfrak{S}_1$, a Filter (σ), a Map (μ) and an Aggregate (Σ) form a subgraph u_1. We call such a *maximal connected subgraph* a *checkpoint unit*. The reason behind this is that we take such a subgraph as an atomic unit for checkpoint because checkpointing only part of it would yield an inconsistent backup image. Notice that, for a chain where operator o_1 outputs to o_2, checkpointing only o_1 will leave (on the backup server) an image that would result from an invalid execution where some recent output of o_1 disappears without being fed into o_2. Hereafter, we specify the query deployment plan in terms of checkpoint units. In Figure 4.1, $Q_1 = \{u_1, u_2, u_3\}$, $Q_2 = \{u_4\}$, and so forth. Operators that belong to a checkpoint unit are called *constituent operators*.

We also regard that checkpoint units on the same server are *independent* (i.e., can be checkpointed onto *different servers* at *different times*) because they have no interdependency with each other. A *backup assignment*, a mapping between checkpoint units and their backup servers, is denoted by $\{Q_{i,j} : i \neq j, 1 \leq i, j \leq n\}$, where $Q_{i,j}$ is the set of checkpoint units that server $\mathfrak{S}_i$ executes and server $\mathfrak{S}_j$ backs up ($i \neq j$ implies that a server cannot back up itself). In Figure 4.1, $Q_{1,2} = \{u_1, u_2\}$, $Q_{1,3} = \{u_3\}$, and $Q_{2,3} = \{u_4\}$. u'_3 and u'_4 are the backup images built by checkpointing u_3 and u_4 onto $\mathfrak{S}_3$, respectively. Each shaded area $Q_{i,j}$ represents the collection of backup images that $\mathfrak{S}_j$ maintains for $\mathfrak{S}_i$.

It should be noted that the formation of checkpoint units will significantly affect the behavior of

our parallel recovery framework. We discuss this issue in Section 4.2.

4.1.3 No Loss Guarantee

Our parallel recovery framework masks a server failure by making other servers in the cluster collectively rebuild the latest state of the failed server. For this model, we assume that a backup can precisely repeat the pre-failure execution of its primary as long as it can obtain the input tuples that the primary has processed since the last checkpoint. Providing this guarantee for all kinds of stream processing operators including nondeterministic ones is discussed in Section 3.5.

In Figure 4.1, if $\mathfrak{S}_2$ fails, the processing that involves u_4 no longer continues. In this case, $\mathfrak{S}_3$ has to take over the processing because it is the only server that possesses the backup image u'_4 of u_4. In more detail, $\mathfrak{S}_3$ has to set up a new connection S' to feed its backup image u'_4 and start executing u'_4 to recover the state of u_4. To prevent data loss in this case, each backup image must be able to obtain the tuples that the primary has processed since the last checkpoint. In other words, if $\mathfrak{S}_2$ processed tuple t_{100} after checkpointing u_4 onto $\mathfrak{S}_3$, u'_4 on $\mathfrak{S}_3$ must be able to receive that tuple through S'. For this reason, each checkpoint unit has *output queues*, one for each output, to retain such tuples (i.e., those that the downstream backups are currently missing). Those tuples can be safely discarded when the downstream server processes them and checkpoints the effect onto the backup server. In our current implementation, both output queues and backup images are built in memory. This implementation can be extended to spill those components to disk under memory contention.

4.1.4 Non-Failure Time Operation for High Availability

As stated in Section 4.1.1, we assume that each server has spare CPU cycles and uses them for high availability purposes. An idle server (i.e., one that has no tuples to process at the moment) can perform one of the following tasks:

- **Capture:** The server chooses a checkpoint unit and sends to the backup server a message that captures the delta in the state (i.e., the difference between the current state and the state at the time of the previous checkpoint). We use the terms *capture* and *checkpoint message*, respectively, to refer to the task and the message that the task constructs.

- **Paste:** The server chooses one of the checkpoint messages that it received, and copies the content of the message to the corresponding backup image. We call this task *paste*. Once a paste finishes, the server notifies the sender of the checkpoint message (i.e., the primary server of the checkpoint unit). This enables the next round of checkpoint for that unit.

In the rest of this section, we describe checkpoint tasks in detail. We discuss the problem of scheduling them in Section 4.3.

Contents of a Checkpoint Message. If a server begins capturing a checkpoint unit at an idle time, the input queues of the checkpoint unit (i.e., those of the constituent operators) are empty.

For this reason, the server skips input queues and captures only the *constituent operators* and *output queues*. The details of capturing stream processing operators are discussed in Section 4.5. It should be noted that only a small part of each output queue needs to be captured. In Figure 4.1, for example, if $\mathfrak{S}_2$ acknowledged to $\mathfrak{S}_1$ that it received t_{101}, $\mathfrak{S}_1$ needs to capture only t_{102} among the tuples in the output queue of u_3. With t_{102}, the output queue of u'_3 on $\mathfrak{S}_3$ can guarantee that S'', a stream that will flow if $\mathfrak{S}_1$ fails, will not miss any tuple.

Checkpoint vs. Processing. Once a capture task begins, the server defers stream processing (i.e., only buffers arriving input tuples) until the capture ends. This is because (1) executing a checkpoint unit while it is being captured might introduce inconsistency in the captured image (i.e., capture and processing conflict semantically) and (2) interrupting a capture to execute other units will further suspend the checkpoint unit currently being captured. If an exceptional input burst appears during a capture, it may be desirable to abort the ongoing capture and immediately resume the processing to bound the growth of the latency. However, such an abortion is not always useful because the change in state tends to grow over time (i.e., a later capture is usually more expensive). In contrast to capture, paste can be interrupted to execute other operators. Notice that this task on backup can be done in parallel with the execution of stream processing operators.

Once a capture finishes, a *processing burst* appears until the buffered input tuples are consumed. Conceptually, the duration of capture (i.e., capture cost) implies the penalty of high availability (i.e., the additional processing latency due to capture). Furthermore, both the capture cost and the server's processing load affect the duration of the processing burst as well as the checkpoint interval. In practice, we can set a lower bound on the checkpoint interval to prevent checkpointing too frequently (i.e., to trade off processing against high availability). We can also set an upper bound that forces a checkpoint to bound the growth of recovery time (i.e., to trade off high availability against processing).

4.1.5 Failure and Recovery

In our parallel recovery framework, each server $\mathfrak{S}_i$ is monitored by a designated server that periodically (e.g., every 100ms) pings $\mathfrak{S}_i$. If $\mathfrak{S}_i$ does not respond for a timeout period (e.g., 300ms), the server assumes that $\mathfrak{S}_i$ has failed and broadcasts this to the other servers in the cluster. Each of these notified servers then searches for checkpoint messages from the failed server and pastes those messages to the corresponding backup images. Next, each server finds the backup images that it has maintained for the failed server and begins executing them as its new units, while redirecting the input and output streams as described in Section 4.1.3. If these new units become up-to-date, the server starts executing other units as well.

We use the term *recovery* to refer to the process during which the alive servers in the cluster take the actions described above. When the servers collectively rebuild the latest state of the failed server, we say that the cluster has *recovered* from the failure. Note that *having recovered* does not necessarily imply *being able to mask the next failure*. This is because the system may not be able to tolerate the next failure until it secures the checkpoint units taken over during recovery, by

checkpoints onto new backup servers. The *period of instability* refers to the amount of time, after failure, until all checkpoint units are again protected. Finally, if the failed server comes back up, it joins the system as a new member.

4.2 Formation of Checkpoint Units

As illustrated in Section 4.1.2, query deployment determines the formation of checkpoint units. We start this section by showing that the load management principles for stream processing are also beneficial to our parallel recovery framework. We then introduce a strategy that avoids managing too many checkpoint units. Finally, we discuss preserving the safety guarantee while the system reforms checkpoint units.

4.2.1 Impact of Load Management Principles

One principle of load management in stream processing is to distribute operators with highly correlated loads over different machines [149]. This is because placing them on the same machine will make it more vulnerable to load spikes. Adjacent operators (i.e., those connected by data streams) usually exhibit high load correlation. Therefore, unless they have very low processing load, we should place them at different servers. Furthermore, operators with heavy processing load are usually split into smaller pieces and distributed over multiple servers [41, 120] due to their negative impact on load management (refer to [148] for quantitative analysis). For the two reasons above, each server is likely to own many small-size operator chains (i.e., many fine-grained checkpoint units). This is advantageous to our recovery framework because (1) more checkpoint units, in general, lead to better backup distribution and (2) finer checkpoint units tend to have smaller capture costs (i.e., smaller disruption to processing). Note that checkpoint units with high capture costs can also be split in order to lower the costs.

4.2.2 Merging Checkpoint Units

While having many checkpoint units is usually beneficial in terms of backup distribution, managing too many checkpoint units may incur significant overhead. We address this problem by iteratively *merging* checkpoint units with similar characteristics as long as the capture cost remains under a threshold. In other words, we put all the operators that constitute those checkpoint units into a new checkpoint unit, even though those operators do not form a connected graph. We use the ratio of *processing load* over *capture cost* as the similarity metric. The reason behind this is that our checkpoint scheduling strategy (described in Section 4.3) expedites recovery by more frequently checkpointing units with high processing load and low capture cost. If we merge units that are dissimilar in terms of the aforementioned ratio, we can no longer checkpoint them with the frequencies that lead to the maximal recovery speed.

4.2.3 Safety during Checkpoint Unit Reformation

Safety against failure has to be preserved even when the system reforms checkpoint units due to operator splitting and migration. We achieve this by keeping the old backup images of the involved checkpoint units, until the reformation completes and the newly formed checkpoint units are again backed up. We describe the procedure for the following representative case. Suppose that an operator ρ is migrated from server $\mathfrak{S}_j$ to server $\mathfrak{S}_i$ and added (as a constituent operator) to a checkpoint unit u on $\mathfrak{S}_i$. In this case, the previous backup server $\mathfrak{S}'_j$ for ρ has to keep its backup image ρ' of ρ until $\mathfrak{S}_i$ checkpoints the expanded version of u onto the backup server $\mathfrak{S}'_i$ for u. This is because, before the checkpoint, $\mathfrak{S}'_j$ is only the server that possesses the backup image of ρ. $\mathfrak{S}'_j$ can safely remove its backup image ρ' after $\mathfrak{S}'_i$ receives the checkpoint message that captures $u \ni \rho$.

4.3 Checkpoint Scheduling

In our backup model, an idle server can perform either a *capture* task (i.e., among the units not being checkpointed, choose one, compose a checkpoint message for that one, and send the message to the appropriate backup server) or a *paste* task (i.e., among the checkpoint messages received, choose one and copy the content of the message to the appropriate backup image). In this section, we devise an algorithm that schedules such tasks in a manner that minimizes the expected recovery time. We first discuss how we can find the capture task that will most reduce the expected recovery time. Then, we describe how we choose the best from both capture and paste tasks. We conclude this section discussing the key properties of our scheduling algorithm.

4.3.1 Choosing the Best Capture Task

A capture task first finds the difference between a checkpoint unit's current state and its state at the time of the previous capture and then sends the difference to the backup server. After this, the backup server can freshen the backup image (i.e., reduce the amount of work to do during recovery) by simply copying the delta received. However, as we describe below, the recovery time is heavily dependent on how a server schedules capture tasks.

Figure 4.2 illustrates an example where server $\mathfrak{S}_1$ checkpoints its units u_1 and u_2 onto $\mathfrak{S}_2$, and u_3 onto $\mathfrak{S}_3$ in a round-robin fashion. To ease illustration, we do not consider the paste tasks that $\mathfrak{S}_1$ would perform and the capture tasks that $\mathfrak{S}_2$ and $\mathfrak{S}_3$ would do. We also ignore network latency and assume that units on $\mathfrak{S}_1$ have constant processing loads (in terms of CPU utilization): $l_{u_1}(t) = 11\%$, $l_{u_2}(t) = 10\%$, and $l_{u_3}(t) = 66.5\%$ for all time t. Finally, we assume that those units have (1) constant capture costs (in seconds): $c_{u_1}(t) = 0.125$, $c_{u_2}(t) = 0.25$, and $c_{u_3}(t) = 0.125$ for all time t and (2) the same paste costs as capture costs: $c'_{u_k}(t) = c_{u_k}(t)$ for $k = 1,\ 2,\ 3$. Notice that these costs are the amounts of time that the CPU would take when it performs high-availability tasks in isolation. The assumptions above are to ease illustration (refer to Sections 4.6.1 and 4.6.2 for the details in real cases).

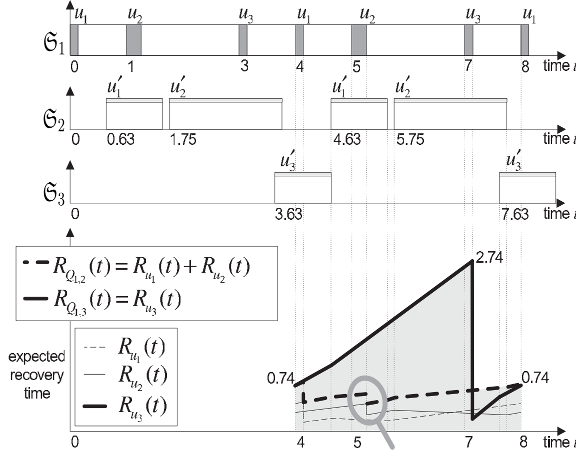


Figure 4.2: Recovery Time (Round-Robin)

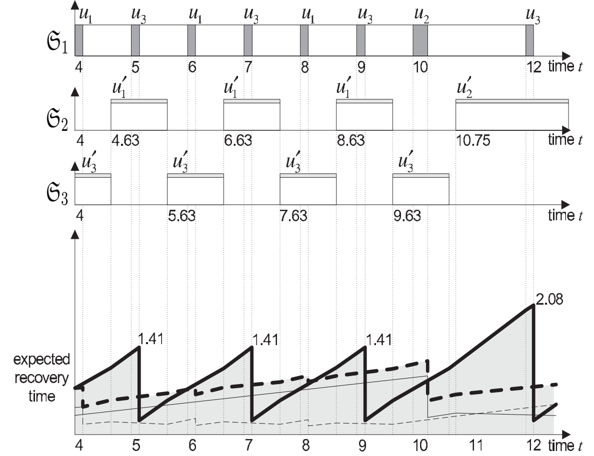


Figure 4.3: Recovery Time (Min-Max)

As described in Section 4.1.4, each capture (represented as a dark rectangle in Figure 4.2) defers query processing. For this reason, a processing burst (represented as an empty rectangle) appears after that. As mentioned before, the duration of such a burst, is a function of the capture cost and the server's processing load. For example, the duration of the burst after capturing u_1 is $\frac{\texttt{[capture cost][total load]}}{1-\texttt{[total load]}} = \frac{c_{u1}(t)\big(l_{u_1}(t)+l_{u_2}(t)+l_{u_3}(t)\big)}{1-\big(l_{u_1}(t)+l_{u_2}(t)+l_{u_3}(t)\big)} = \frac{0.125\cdot 0.875}{1-0.875} = 0.875$ (seconds). The figure also illustrates that each paste is deferred until its turn and, in contrast to captures, is interleaved with query processing (refer to Section 4.1.4 and the stacks of grey and empty rectangles in the figure).

Figure 4.2 also demonstrates how the expected recovery time changes over time for various entities such as units u_1, u_2, u_3, segments $Q_{1,2}$, $Q_{1,3}$ and the query Q_1 on server $\mathfrak{S}_1$. We use the convention that $R_*(t)$ represents the expected amount of time to recover an entity $*$ if the primary for $*$ fails at time t (the details are presented in Section 4.8.1). The figure shows that capturing u_2 during $[5, 5.25]$ reduces $R_{u_2}(t)$ from 0.43 to 0.28 when it finishes at 5.25. Notice that when the capture is about to complete, the expected amount of time to recover u_2 is $\int_1^{5.25} l_{u_2}(\tau)d\tau = 0.43$ because the backup image u_2' on $\mathfrak{S}_2$ has the state of u_2 as of time 1. However, when the capture finishes at time 5.25, $\mathfrak{S}_2$ receives a checkpoint message that captures u_2 as of time 5. Therefore, if $\mathfrak{S}_1$ fails, $\mathfrak{S}_2$ will take $c_{u_2}'(t) = 0.25$ seconds to consume the checkpoint message and $\int_5^{5.25} l_{u_2}(\tau)d\tau \simeq 0.03$ seconds to replay the execution of u_2 that occurred from time 5 to 5.25 on $\mathfrak{S}_1$.

The capture task for u_2 also reduces $R_{Q_{1,2}}(t)$ by the same amount. This is because $R_{Q_{1,2}}(t) = R_{u_1}(t) + R_{u_2}(t)$ in the example, as u_1 and u_2 on $\mathfrak{S}_1$ are backed up at $\mathfrak{S}_2$. In contrast, the capture task *cannot reduce* $R_{Q_1}(t)$ (see the upper bound of the grey area). Notice that the system will recover from $\mathfrak{S}_1$'s failure only if both $\mathfrak{S}_2$ and $\mathfrak{S}_3$ recover segments $Q_{1,2}$ and $Q_{1,3}$, respectively. Formally, $R_{Q_1}(t) = \max\{R_{Q_{1,2}}(t), R_{Q_{1,3}}(t)\}$. However, capturing $u_2 \in Q_{1,2}$ does not reduce the largest recovery load $R_{Q_{1,3}}(t)$ on $\mathfrak{S}_3$ (see in the figure that $R_{Q_{1,3}}(t) > R_{Q_{1,2}}(t)$). *To reduce* $R_{Q_1}(t)$, $\mathfrak{S}_1$ *at time 5 should have started capturing* $u_3 \in Q_{1,3}$ (as in Figure 4.3) *rather than* $u_2 \in Q_{1,2}$.

Based on this observation, we design an algorithm that selects a task that *will minimize the*

maximum recovery load among those spread over other servers. For this reason, we call our scheduling algorithm "*min-max*". In detail (also refer to Figure 1), each server $\mathfrak{S}_i$ looks for all checkpoint units $u \in Q_{i,j}$ such that (1) $u \in$ `c-q` and (2) $R_{Q_{i,j}}(t + c_u(t)) = R_{Q_i}(t + c_u(t))$, where `c-q` is a queue that remembers the units that $\mathfrak{S}_i$ can checkpoint immediately (i.e., those not being checkpointed) and $t + c_u(t)$ is the time when capturing u will complete. Condition (2) above implies that u's backup server will have *the maximum recovery load when capturing u is about to finish.* Among such units, the server finds unit u^* such that capturing it will *most reduce the recovery time*, relative to the capture cost. As the metric for this, the server uses $\frac{\Delta R_u(t)}{c_u(t)}$ for each checkpoint unit u, where $\Delta R_u(t)$ denotes the reduction in recovery time at the cost $c_u(t)$ of capturing u. We define $\Delta R_u(t)$ as $\int_{\alpha_u(t)}^{t} l_u(\tau)d\tau - c'_u(t)$, where $\alpha_u(t)$ denotes the start time of the previous capture. In the definition of $\Delta R_u(t)$, the first and second terms represent the *gain* of reducing the recovery load (i.e., freshening the backup image) and the *penalty* of consuming the checkpoint message, respectively. Notice that our algorithm prefers checkpoint units with *high processing load* (see $l_u(\tau)$ in the numerator of the metric) and *low checkpointing cost* (see $c_u(t)$ and $c'_u(t)$ in the metric).

4.3.2 Capture vs Paste

In principle, a server conducts *capture tasks* to better prepare for the failure of itself and *paste tasks* for the failure of others. To strike a balance between these goals, each server finds the task, whether it is a capture or a paste, that will assist the segment with the largest recovery load. In detail, server $\mathfrak{S}_i$ first computes $R_{Q_{i,j^*}}(u^*, t)$, the expected recovery time for the moment when it finishes capturing the best unit $u^* \in Q_{i,j^*}$ (details are presented in Section 4.8.2).

Then, for each backup segment $Q_{j,i}$, it computes the expected recovery time for the moment when it completely consumes the *oldest pending* checkpoint message from $\mathfrak{S}_j$ (this FIFO order is to abide by the decisions that $\mathfrak{S}_j$ already made). Using $u \in Q_{j,i}$ to denote the unit captured in the checkpoint message, we represent such expected recovery time as $R_{Q_{j,i}}(u, t)$ (again see Section 4.8.2 for the formal definition). If $R_{Q_{j,i}}(u, t) > R_{Q_{i,j^*}}(u^*, t)$, we assume that backup segment $Q_{j,i}$ is less prepared for failure than primary segment Q_{i,j^*} (i.e., the paste for $u \in Q_{j,i}$ is more urgent). The server selects the best from capture and paste tasks based on this rationale.

4.3.3 The Complete Scheduling Algorithm

Algorithm 1 summarizes the min-max algorithm. Whenever a server forms a new checkpoint unit, it pushes it into `c-q` (line 02). An idle server first finds the best capture task (lines 04-06). Next, it attempts to find the paste task that is more effective than all others (including the best capture task) (lines 07-11). Finally, it performs the best task found. If a capture task is chosen (lines 12-14), the server composes a checkpoint message and sends it to the relevant backup server (lines 21-22). If a paste task is chosen (lines 17-18), the server consumes the checkpoint message and then notifies the completion of checkpoint to the primary (lines 26-27).

Algorithm 1: Min-Max Scheduling (on server $\mathfrak{S}_i$)

1 **whenever** *server* $\mathfrak{S}_i$ *forms a checkpoint unit* $u \in Q_{i,j}$ **do**
2 　`c-q`.push(u, i); // enqueue the unit

3 **whenever** *idle* **do**
4 　$\mathcal{C} \leftarrow \{(u, j) \in \texttt{c-q} : R_{Q_{i,j}}(t + c_u(t)) = R_{Q_i}(t + c_u(t))\}$
5 　find $(u^*, j^*) \in \mathcal{C}$ such that // find the best unit to capture
6 　　$\frac{\Delta R_{u^*}(t)}{c_{u^*}(t)} = \max\{\frac{\Delta R_u(t)}{c_u(t)} : (u, j) \in \mathcal{C}\}$
7 　**for each** j $(1 \leq j \leq n,\ j \neq i)$ **do**
8 　　$(u, \Delta) \leftarrow$ `p-q`$[j]$.first() // oldest checkpoint msg from $\mathfrak{S}_j$
9 　　**if** $\big(R_{Q_{j,i}}(u, t) > R_{Q_{i,j^*}}(u^*, t)\big)$ *and*
10 　　$\Big(u' = \texttt{null} \mid\mid R_{Q_{j,i}}(u, t) > R_{Q_{j',i}}(u', t)\Big)$ /* *found the first or a better paste* */ **then**
11 　　　$(u', j', \Delta') \leftarrow (u, j, \Delta)$ // remember the found paste task

12 　**if** $(u' = \texttt{null})$ /* *no paste is better than the best capture* */ **then**
13 　　capture(u^*, j^*); // do the best capture
14 　　remove (u^*, j^*) from `c-q`;
15 　**else**
16 　　// if there exists a paste better than the best capture
17 　　paste(u', j', Δ') // do the best paste
18 　　remove (u', Δ') from `p-q`$[j']$;

19 capture(u^*, j^*)
20 **begin**
21 　copies into Δ the recent change in u^*;
22 　$\mathfrak{S}_{j^*}$.`p-q`$[i]$.push(u^*, Δ); // send Δ to the backup
23 **end**
24 paste(u', j', Δ')
25 **begin**
26 　copies Δ' into u';
27 　$\mathfrak{S}_{j'}$.`c-q`.push(u, i); // notify the primary
28 **end**

4.3.4 Discussion

Our min-max algorithm selects the task that will most reduce the largest recovery load. Figures 4.2 and 4.3 show an example where our scheduling algorithm maintains the recovery time at a 30% lower level than round-robin (in Section 4.6, we show the results from real test cases). The figures also show that min-max takes a longer time than round-robin until it checkpoints each unit at least once (we call such a period of time a *checkpoint cycle*). This is because min-max frequently checkpoints units with high processing load and low checkpoint cost, yielding a non-uniform schedule. Figure 4.3 shows that the recovery time under min-max gradually increases until it eventually drops at the end of a checkpoint cycle. This is because the backup images of uncheckpointed units get staler over time.

4.4 Dynamic Backup Assignment

In our parallel recovery framework, the recovery time depends on the checkpoint schedule as well as on the backup assignment. For example, a server with too many backups can easily be the bottleneck that delays the circulation of capture and paste tasks. If this happens, other servers cannot efficiently use spare CPU cycles and thus will poorly prepare for failures. In this section, we study how we can avoid this problem. We assume that servers with low backup load volunteer to back up new checkpoint units as soon as they are formed in the system. Hereafter, we focus on modifying the backup assignment to cope with varying system conditions introduced by operator migration, changes in input rates, and other reasons.

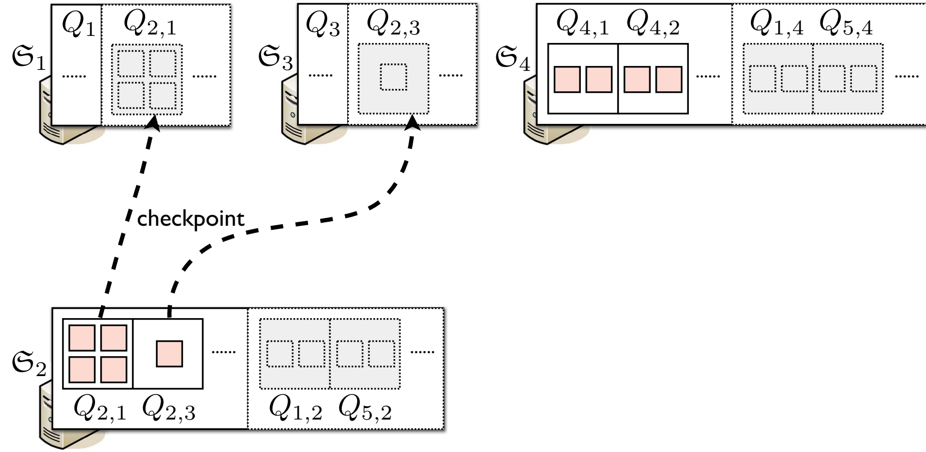
4.4.1 Determining Backup Load Imbalance

In principle, our algorithm reassigns backups to assist the server whose failure will cause the longest recovery. We call such a server *the worst point of failure*. We take this approach because (1) a server usually becomes the worst point of failure due to unbalanced backup distribution and (2) it is always advantageous to improve the worst-case disruption that a failure would cause. We use Figure 4.4 to illustrate two typical cases of backup imbalance. We assume that $\mathfrak{S}_2$ is the worst point of failure and, to ease presentation, all the checkpoint units (i.e., those represented as dark small rectangles) have the same processing load and checkpoint cost. Backup images are represented as small dotted-line rectangles. Figure 4.4(a) shows the case where $\mathfrak{S}_2$ has poorly assigned backups (see that $\mathfrak{S}_1$ backs up too much for $\mathfrak{S}_2$). In this case, $\mathfrak{S}_2$ can resolve the problem by transferring part of $\mathfrak{S}_1$'s backup responsibility to $\mathfrak{S}_3$. Figure 4.4(b) shows another case where $\mathfrak{S}_2$ maintains too many backup images for others (see $Q_{1,2}$ and $Q_{5,2}$). In this case, $\mathfrak{S}_2$ should not reassign backup servers because it cannot handle the imbalance in backup load. Instead, a different server (say $\mathfrak{S}_5$) should balance the backup load for $\mathfrak{S}_2$.

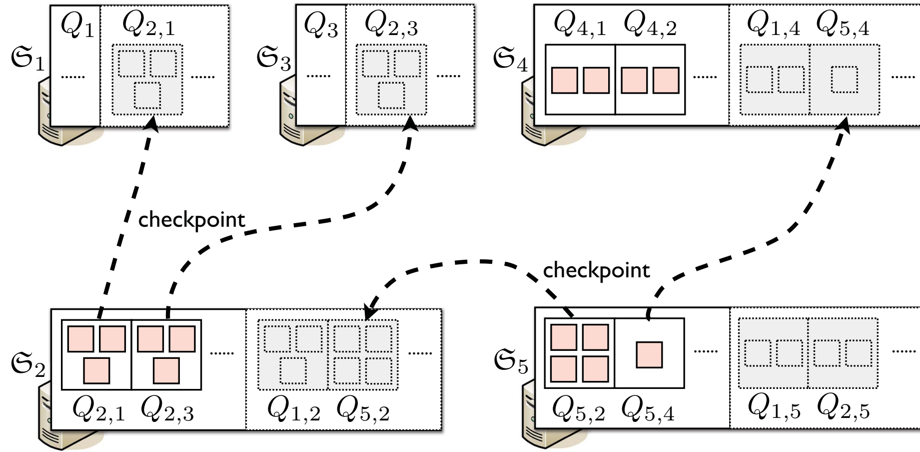
4.4.2 The Backup Reassignment Algorithm

Algorithm 2 summarizes our backup reassignment algorithm. At the end of *epoch* E (the period required for every server to finish at least one checkpoint cycle), the servers in the cluster determine the worst point of failure $\mathfrak{S}_{\bar{i}}$, based on the expected recovery times averaged over the epoch. We use this periodic approach because (1) it is hard to know the recovery time in the average sense before a checkpoint cycle ends (see Section 4.3.4) and (2) we should avoid changing backup assignment too frequently. Note that checkpointing a unit onto a new backup server (i.e., a whole checkpoint) is usually more expensive than an ordinary delta-checkpoint.

Server $\mathfrak{S}_{\bar{i}}$ (the worst point of failure) first finds its segments $Q_{\bar{i},\bar{j}}$ with the largest recovery load and $Q_{\bar{i},\underline{j}}$ with the smallest recovery load (note that $R_{Q_{\bar{i},\bar{j}}}(E) > R_{Q_{\bar{i},\underline{j}}}(E)$ by definition). It also finds another server $\mathfrak{S}_{\underline{k}}$ whose failure will result in the shortest recovery (line 02). If $R_{Q_{\bar{i},\underline{j}}}(E) < R_{Q_{\underline{k}}}(E)$, $\mathfrak{S}_{\bar{i}}$ assumes that $\mathfrak{S}_{\underline{j}}$ is assigned too low backup load and accordingly balances the backup load between $\mathfrak{S}_{\bar{j}}$ and $\mathfrak{S}_{\underline{j}}$ (lines 06-07; in Figure 4.4(a), $\mathfrak{S}_{\bar{j}}$ and $\mathfrak{S}_{\underline{j}}$ correspond to $\mathfrak{S}_1$ and $\mathfrak{S}_3$,



(a) Unbalanced Backup Assignment



(b) Too Many Backup Images

Figure 4.4: Backup Reassignment

respectively). Otherwise (line 08), it assumes that it has too many backup images. Thus, it locates a different server $\mathfrak{S}_{\overline{k}}$ that will effectively balance the backup load between $\mathfrak{S}_{\overline{i}}$ and $\mathfrak{S}_{\underline{k}}$ (lines 10-11). In Figure 4.4(b), $\mathfrak{S}_{\overline{k}}$ corresponds to $\mathfrak{S}_5$. $\mathfrak{S}_{\overline{i}}$ and $\mathfrak{S}_{\underline{k}}$ correspond to $\mathfrak{S}_2$ and $\mathfrak{S}_4$, respectively.

The **move**$(\overline{j}, \underline{j})$ phase in Algorithm 2 shows the details of reassigning backup servers. In the algorithm, $h_u(E)$ denotes the total CPU cycles used for updating the backup image of unit u during epoch E (we call this quantity the *backup load* of unit u). Similarly, $h_{Q_{\overline{i},j}}(E)$ represents the backup load of segment $Q_{\overline{i},j}$. To balance the backup load, $\mathfrak{S}_{\overline{i}}$ first computes the amount of backup load Δh to move from segment $Q_{\overline{i},\overline{j}}$ to segment $Q_{\overline{i},\underline{j}}$ (line 14). Then, it reassigns backup servers until the amount of backup load transferred reaches Δh (15-19). Note that this **move** phase only chooses the new locations to put backup images. The first checkpoints after reassignment in fact create the backup images.

Algorithm 2: Backup Reassignment

1 **whenever** *epoch E ends (On server $\mathfrak{S}_{\bar{i}}$ with the highest recovery time)* **do**
2 Find $\underline{k}, \bar{j}, \underline{j}$ such that
3 $R_{Q_{\bar{i},\bar{j}}}(E) = \max_{j=1}^{n} R_{Q_{\bar{i},j}}(E)$
4 $R_{Q_{\bar{i},\underline{j}}}(E) = \min_{j=1}^{n} R_{Q_{\bar{i},j}}(E)$
5 $R_{Q_{\underline{k}}}(E) = \min_{k=1}^{n} R_{Q_k}(E)$
6 **if** $R_{Q_{\bar{i},\underline{j}}}(E) < R_{Q_{\underline{k}}}(E)$ /* *the backup load is unbalanced* */ **then**
7 move($\bar{j}, \underline{j}$) // balance the backup load
8 **else**
9 // $\mathfrak{S}_{\bar{i}}$ has too many backup images
10 find $\bar{k} = \arg\max_{k=1}^{n} \left[R_{Q_{k,\bar{i}}}(E) - R_{Q_{k,\underline{k}}}(E) \right]$
11 $\mathfrak{S}_{\bar{k}}$.move($\bar{i}, \underline{k}$) // let $\mathfrak{S}_{\bar{k}}$ balance the backup load

12 $\mathfrak{S}_{\bar{i}}$.move($\bar{j}, \underline{j}$)
13 **begin**
14 $\Delta h \leftarrow \frac{R_{\bar{i},\bar{j}}(E) - R_{\bar{i},\underline{j}}(E)}{R_{\bar{i},\bar{j}}(E)} h_{Q_{\bar{i},\bar{j}}}(E)$ // backup load to transfer
15 **for each** $u \in Q_{\bar{i},\bar{j}}$ **do**
16 **if** $h_u(E) \leq \Delta h$ /* *found a checkpoint unit to reassign the backup* */ **then**
17 $\Delta h \leftarrow \Delta h - h_u(E)$ // update the backup load to move
18 $Q_{\bar{i},\bar{j}} \leftarrow Q_{\bar{i},\bar{j}} - \{u\}$ // reassign the backup server
19 $Q_{\bar{i},\underline{j}} \leftarrow Q_{\bar{i},\underline{j}} \cup \{u\}$
20 **end**

4.5 Delta Checkpointing

An efficient checkpointing mechanism will shorten the duration of capture and paste tasks. This implies better runtime performance (because the disruption to processing will decrease) and faster recovery speed (because more frequent checkpoints will be possible). In this section, we describe how to implement efficient, operator-specific delta-checkpointing techniques based on the details of operators. We also construct cost models relevant to these techniques. In Section 4.6, we demonstrate that checkpoint costs can be estimated accurately relying on the cost models. We also show that our min-max algorithm performs well due to such accurate cost estimations.

As described in Section 4.1.4, capturing a checkpoint unit requires incorporating the states of constituent operators and, for each output queue, a round trip worth of tuples. The latter however can be ignored safely by holding the checkpoint message until the downstream servers acknowledge the receipt of the tuples. Thus, we can represent the cost $c_u(\alpha)$ of capturing a checkpoint unit u as $\sum_{\rho \in u} c_\rho(\alpha)$, where α is the start time of the capture task and $c_\rho(\alpha)$ is the cost of capturing the internal data structure of a constituent operator ρ. Because stateless operators will not incur any checkpoint cost, we design (and analyze) the delta-checkpointing methods for two representative stateful operators, Aggregate and Join.

4.5.1 Aggregate

Aggregate splits input stream I into substreams $\{I[g]\}$, one for each group-by value g. For each substream $I[g]$, it assumes windows (sets of tuples) of w seconds that appear every s seconds (we can also define windows in terms of counts of tuples). Whenever a window expires, the operator outputs an aggregated value computed from the tuples contained in the window. In more detail, for each input tuple, the operator (1) reads (from the tuple) timestamp t and group-by value g. Next, the operator (2) uses its "map" to quickly locate *the list of windows* associated with g (if none exists, it creates a new list), and (3) determines if it needs to add new windows (e.g, if the timestamp t of the tuple is 401.5 and the timestamp of the most recent window is 400, an Aggregate with step size s of 1 second will add a window anchored at time 401). Then, the operator (4) iterates over the windows in the list updating the summaries (e.g., counts, sums, or histograms) that those windows maintain. Finally, it (5) closes expired windows while sending their summaries as output tuples.

Delta-Checkpointing. We use one dirty bit for each group-by value to mark those that have appeared since the last checkpoint (we clear dirty bits at the end of capture). We also use one dirty bit for each window to indicate whether or not it was created after the last checkpoint. To capture an Aggregate, the primary server finds all the windows associated with group-by values with their dirty bits on. Next, it copies into the checkpoint message (1) the entire content of each window with its dirty bit on (*full capture* for new windows) and (2) only the summary of each window with its dirty bit off (*partial capture* for updated windows). When the backup server consumes the checkpoint message, it checks the captured window images in the message. For a fully-captured image, it creates a new window from the image and associates the window with the appropriate group-by value (i.e., *full paste*). Otherwise, it copies the partial image onto the corresponding window that already exists in the operator (i.e., *partial paste*).

Cost Model. We can represent the cost of capturing this operator as $C_n c_f + C_u c_p$, where C_n is the number of windows created after the last checkpoint, C_u is the number of updated windows, and c_f and c_p are the costs of fully and partially capturing a window, respectively. We can similarly define the paste cost using per-window full/partial paste costs.

4.5.2 Join

Join has input streams I_1 and I_2. This operator searches for all pairs of input tuples (one from each input stream) that (1) belong to the same window of size w and (2) match the predicate defined for the operator. Whenever the operator finds such a pair of *matching input tuples*, it produces the concatenation of them as an output tuple.

Delta-Checkpointing. Since the recent change in state is the tuples newly entered the window, each checkpoint captures those tuples. It also captures the upper bound of the tuples that have left the window so that the backup server can remove those tuples from the backup image.

Cost Model. The number of tuples that have entered the window since the last checkpoint can be represented as $(\lambda_1 + \lambda_2) \min(t - \alpha, w)$, where t is the current time, α is the start time

of the last checkpoint, and λ_1 and λ_2 are the input rates (i.e., the number of input tuples per second) of input streams I_1 and I_2, respectively. Therefore, the capture cost can be expressed as $(\lambda_1 c_1 + \lambda_2 c_2) \min(t - \alpha, w)$, where c_1 and c_2 are the cost of capturing a tuple from input streams I_1 and I_2, respectively. We can similarly define the paste cost.

4.6 Experimental Results

In this section, we describe experimental results obtained from our Borealis prototype [22, 1] and a detailed simulator. First, we describe how we set up the experiments (Section 4.6.1). Next, based on the results from Borealis, we investigate how the cost of checkpointing varies depending on the frequency of checkpoints (Section 4.6.2). We also demonstrate how our technique effectively reduces the recovery time while being minimally intrusive to regular query processing (Section 4.6.3). Finally, we present supplementary results obtained from the simulator (Section 4.6.4).

4.6.1 The Setup

In our experiments, we used a five server cluster where a 1GBps router interconnects servers with an AMD Sempron 2800+ CPU and 1GB main memory. We used two different input streams. The first (Type 1) is a wide-area TCP trace obtained from [15]. We extract the timestamp and the source-IP address from each packet to form an input feed that runs at 2.0K tuples/sec on average. As commonly observed, the trace has a widely varying stream rate (std = 0.7K tuples/sec) and its source-IP addresses have a highly skewed distribution. The second (Type 2) is an artificial input stream with a source-IP-address field that ranges uniformly from 0 to 99, with a constant stream rate of 100 tuples/sec. These two input streams were designed to represent dissimilar loads.

Our test query consists of Aggregate operators each of which every second counts the number of tuples for each source-IP address over a window of 10 seconds. We form a checkpoint unit for two parallel Aggregates fed by a single input stream. On each server, we generate four checkpoint units with input type 1 and another four checkpoint units with input type 2. It should be noted that we use Aggregates since they are commonly implemented as described in Section 4.5.1 and therefore allow us to obtain results with generality. In contrast, the implementation of Join varies drastically, leading to significantly different processing loads for the same input. In Section 4.6.3, we show how sensitive the recovery time is to processing load.

4.6.2 Checkpointing Costs

In this experiment, we vary the frequency of checkpoints and observe how the cost of checkpointing an Aggregate operator varies both on primary and backup (Figure 4.5). As expected, when the checkpoint frequency decreases (i.e., the interval increases), both the time to form a checkpoint (capture cost) and the time to consume a checkpoint (paste cost) increase. This is because the state of the primary will increasingly diverge from the state of the previous checkpoint. Notice that the

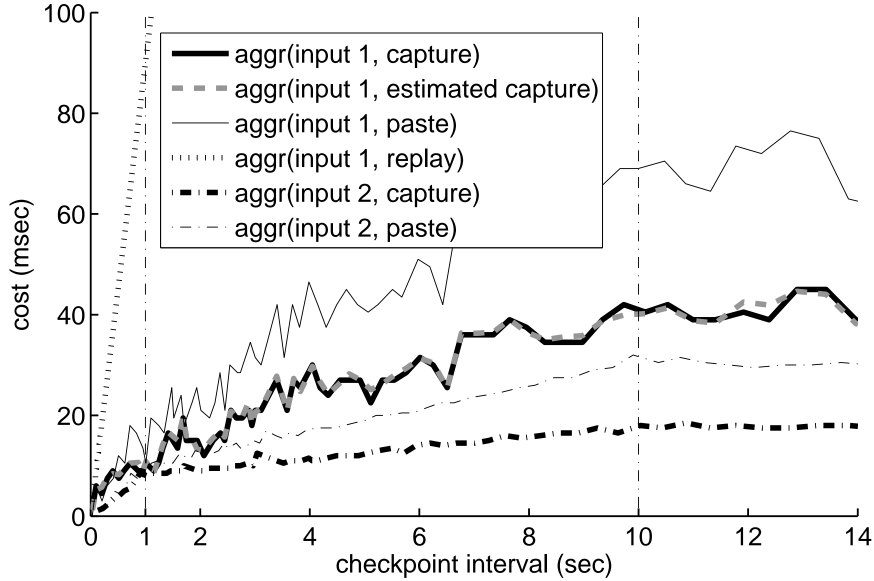


Figure 4.5: Checkpoint Costs

curves for the type 1 Aggregate have jitters, showing how much the checkpoint cost may vary each time due to the burstiness of real packet streams. The figure also shows that type 1 Aggregate has a *relatively lower checkpoint cost* than type 2 Aggregate: although the checkpoint cost of type 1 Aggregate is approximately twice higher than the checkpoint cost of type 2 Aggregate, the type 1 Aggregate has a significantly *higher processing load* than the type 2 Aggregate due to its 20 times higher input rate. The reason for the type 1 Aggregate to have a relatively lower checkpoint cost is that it maintains relatively fewer windows as the group-by values have a very skewed distribution.

The curve labeled "aggr(input 1, estimated capture)" shows that we can accurately estimate the checkpoint cost. Notice that this is important because our min-max algorithm operates on the basis of cost estimations. In the experiment, the estimated per-window full and partial capture costs (in Section 4.5.1, denoted as c_f and c_p) were 18.6 μsecs and 7.3 μsecs, respectively. We obtained these estimated values by linear regression over a collection of triples [# new windows (C_n), # updated windows (C_u), capture cost ($C_n c_f + C_u c_p$)]. The estimated per-window full and partial paste costs (in Section 4.5.1, denoted as c'_f and c'_p) were 30.2 μsecs and 7.2 μsecs, respectively.

The figure also shows the bounded nature of operator states: all curves tend to plateau after a 10 second checkpoint interval mainly because the operator can contain at most a 10-second worth of tuples. The curves for the type 2 Aggregate flatten out after a 1-second interval, as its input creates at most 100 group-by values. The gradual increase of those curves between 1 and 10 seconds accounts for the increase of new windows created after the previous checkpoint. Note that the type 1 Aggregate does not show such flattening (as it continuously observes new IP-addresses) until the checkpoint interval surpasses the window size. The figure also shows that paste costs are usually higher than capture costs because the backup allocates new windows (observe $c'_f > c_f$).

Finally, the difference between aggr(input 1, paste) and aggr(input 1, replay) shows the benefit

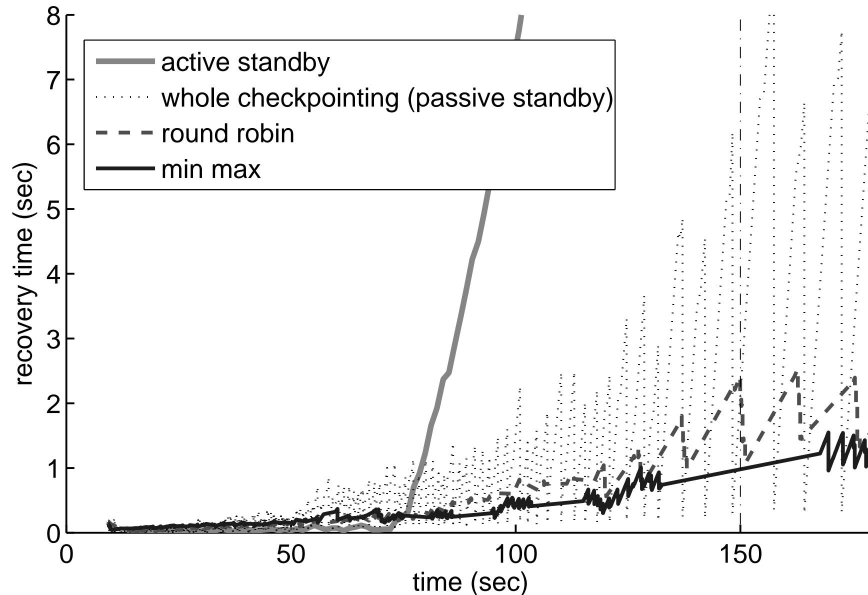


Figure 4.6: Recovery Time

of checkpointing over active standby. In particular, aggr(input 1, replay) shows how long the backup has to run to make up-to-date the stale backup image used in the corresponding checkpoint case. Checkpointing uses much fewer CPU cycles due to the reasons in Section 4.1 and the bounded nature of operators as argued above.

4.6.3 Recovery Time and End-to-End Latency

In this experiment, we study how the *expected recovery time* and *end-to-end latency* change due to checkpoints. We increase the stream rate of type 1 input uniformly from 0 (at time 0) to 2.0K tuples/sec (at time 150) while fixing that of type 2 input at 100 tuples/sec. After time 150, type 1 input is fed as described in Section 4.6.1 and each server is utilized approximately 90% for processing. We deployed a query network of 16 Aggregates on each of the five homogenous servers. On each server, we formed one checkpoint unit int the "whole checkpointing" case and 8 checkpoint units (as described in Section 4.6.1) in all the other cases. The checkpoint units were uniformly assigned backup servers to avoid imbalance in backup load.

Figure 4.6 shows how the expected recovery time changes as the input rate increases. We can see that active standby cannot withstand as the overall processing load increases beyond 50% of the cluster's computation capability (in the figure, backup processes start to fall behind after time 70 as they no longer can use the same amount of resources as primary processes). On the other hand, checkpoint-based methods continue their operations. In general, recovery time is sensitive to increase in processing load because (1) the recovery load increases and (2) fewer spare CPU cycles can be used for high availability. Both round-robin and min-max exhibit significantly faster recovery

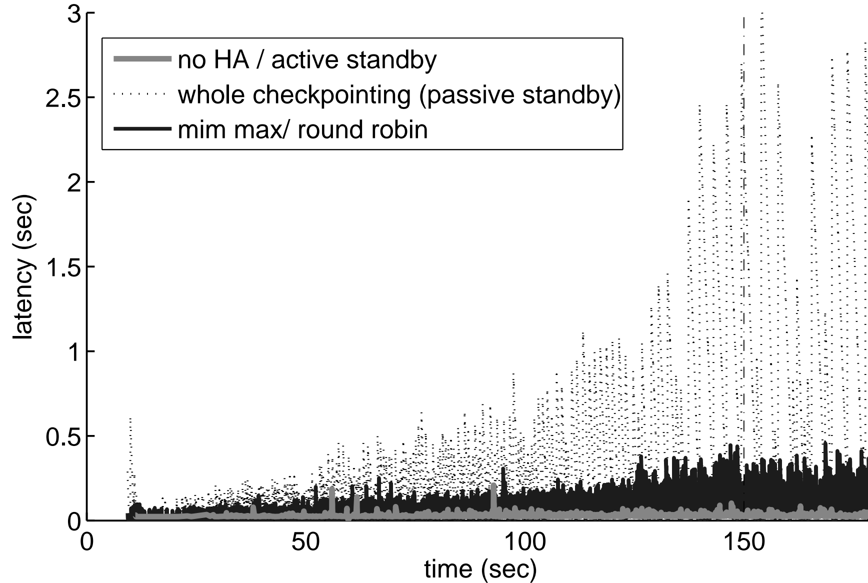


Figure 4.7: End-to-End Processing Latency

speed than whole checkpointing because the recovery load is distributed over multiple servers. Min-max achieves the fastest recovery speed by more frequently checkpointing Type 1 units than Type 2 units. The jitters in Figure 4.6 occur when Type 2 units eventually get checkpointed near the end of a checkpoint cycle (refer to Section 4.3.4).

Figure 4.7 shows how the capture and paste tasks affect query processing. We do not present the curve for active standby because its latency variation is similar to the latency variation without any high availability method. Notice that, to be fair, we assumed a distributed adaptation of the basic active standby model that can flexibly trade off processing against high availability in overload situations. The figure also shows that fine grained checkpoint techniques disrupt regular processing much less than the standard whole checkpointing approach. Each spike in latency is introduced by either a capture or a paste task. In terms of the impact on latency, round-robin is similar to min-max.

4.6.4 Scheduling and Backup Assignment

Figure 4.8 shows how recovery time changes as we increase the number of servers for combinations of scheduling algorithms (round-robin and min-max) and backup assignment techniques (random and balanced). This result was obtained from our detailed Borealis simulator. Round-robin scheduling and random assignment are considered to be the baseline cases. We compare the more robust algorithms, min-max (see Section 4.3) and balanced (See Section 4.4), with the baseline.

The first thing to notice is that since we only assign eight checkpoint units to each primary server, the recovery time does not improve as the number of servers increases past nine. At nine, each checkpoint unit can be backed up on its own server. When we fix the scheduling policy, the difference

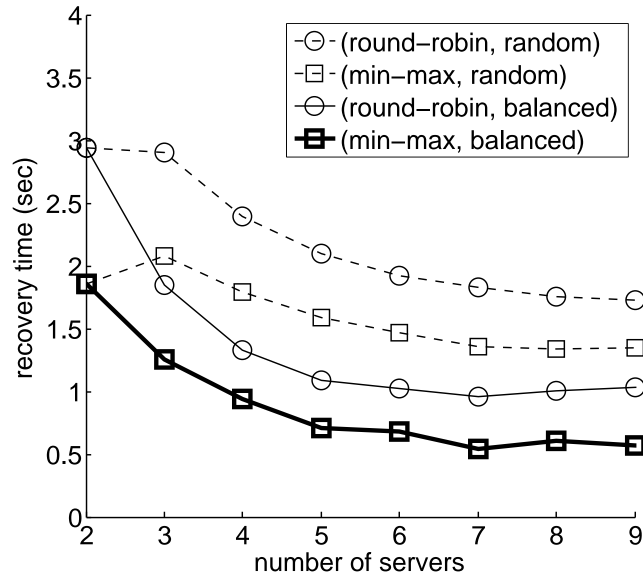


Figure 4.8: Scheduling & Backup Assignment Effects

between the random placement and dynamic placement is significant, yielding approximately 50% improvement in recovery time. This demonstrates the penalty of a random distribution in that such a distribution will not be balanced in general, and the overall recovery time is bounded by the worst case recovery time across all servers. Moreover, the scheduling algorithm can do better in assigning a checkpoint frequency when the backup servers are well balanced. This has the effect of also reducing recovery time as the checkpoint intervals will get smaller. Notice that min-max improves recovery time by only 25% with random, whereas it improves recovery time by 50% with balanced.

4.6.5 Effects of Different Query Networks

In this experiment, we study how the recovery time changes as we vary the mix of checkpoint unit types. Figure 4.9 shows that adding type 2 checkpoint units increases the processing load, the overall cost of checkpointing, and thereby the recovery time as well. At the far left with zero type 2 checkpoint units, round-robin and min-max have the same recovery time. In this case, all the checkpoint units are identical, so there is no advantage to be gained from a smarter scheduler. However, as we add different kinds of checkpoint units, min-max effectively bounds the increase in recovery time by less frequently checkpointing the newly added checkpoint units (recall that type 2 checkpoint units have higher checkpoint costs and lower processing loads than type 1 checkpoint units). In general, min-max does a better job than round-robin as the difference between checkpoint units increases.

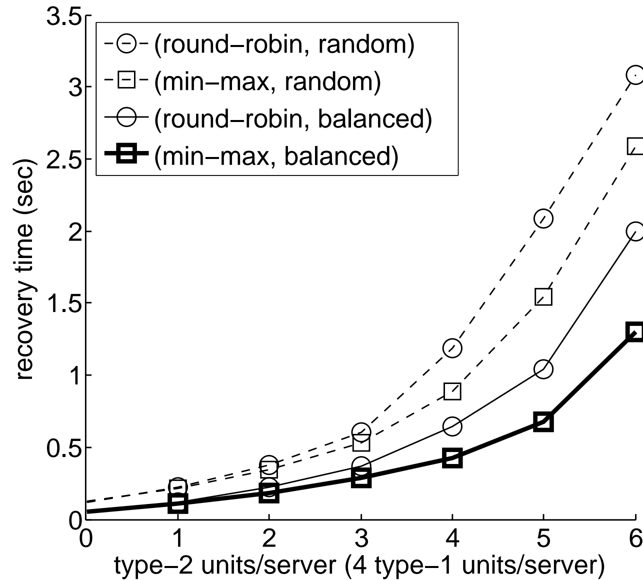


Figure 4.9: Query Network and Recovery Time

4.6.6 Processing Load

Section 4.6.5 varied the processing load by adding type 2 checkpoint units. In this section, we vary the procession load by varying the input rates. In order to effectively compare the results from both experiments, it is worth noting that our default configuration (4 type 1 checkpoint units and 4 type 2 checkpoint units lies on the x-axis of Figure 4.9 at point for 4 type 2 checkpoint units. In Figure 4.10, the same point occurs at 78.4% processing load (67.2% for the 4 type 1's and 2.8% for each type 2). Each load point on the x-axis here increases by 2.8%, thus, corresponding directly to the number of type 2 checkpoint units on the x-axis of Figure 4.9.

The plotted points in Figures 4.9 and 4.10 correspond to the same situations except that we are adding load in different ways. While adding load as in Figure 4.10 will always increase recovery time, adding checkpoint units with state as in Figure 4.9 will have an additional negative effect because of the increase in checkpoint costs. Note that each point in Figure 4.9 is higher than its corresponding point in Figure 4.10 for this reason.

4.7 Summary

In this chapter, we studied a checkpoint-based solution that addresses the needs of distributed stream processing through a parallel fine-grained backup and recovery approach that incurs low overhead and yields short recovery time. The key idea is to sub-divide the query at a given server into units that can each be backed-up on a different server. The approach has the advantage that each unit can be checkpointed separately and independently, thereby spreading out the checkpoint burden over time. It also reduces the overall recovery time because each unit can be rebuilt in parallel, making

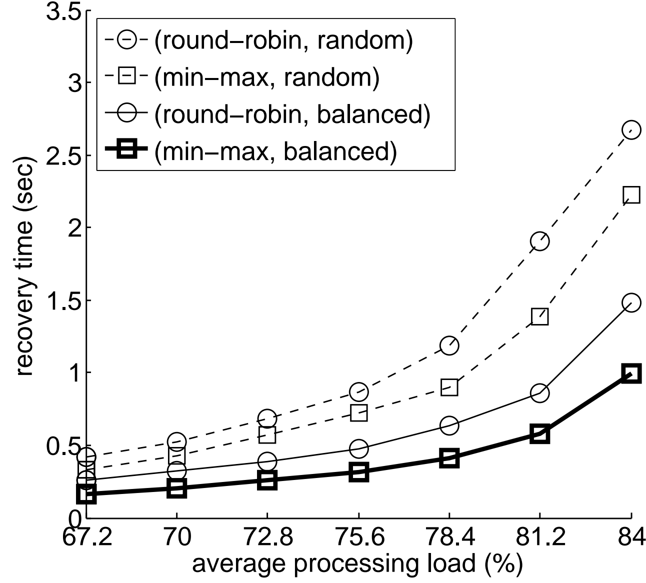


Figure 4.10: Processing Load and Recovery Time

the total recovery time equal to the recovery time of the slowest backup piece.

In this context, we studied how to distribute the backup load in order to minimize the expected recovery time. We also showed how our min-max algorithm adapts to changes in system processing load and performs significantly better than more standard approaches.

4.8 Formal Definitions of Terms used in Chapter 4

4.8.1 Formal Definition of Recovery Time

In this section, we formally define the expected amount of time to recover from a server failure.

Expected Time to Recover from a Server Failure

Let t denote the time when server $\mathfrak{S}_i$ fails. Then, we say that the cluster has *recovered* from the failure of server $\mathfrak{S}_i$ if each peer server $\mathfrak{S}_j$ has rebuilt, from its backup images, the states of the corresponding checkpoint units on $\mathfrak{S}_i$ as of time t. In Figure 4.1, for example, the cluster has recovered from the failure of $\mathfrak{S}_1$ if $\mathfrak{S}_2$ has rebuilt the states of u_1 and u_2 on $\mathfrak{S}_1$ as of t and $\mathfrak{S}_3$ has rebuilt the state of u_3 on $\mathfrak{S}_1$ as of t as well. Therefore, we formally define the *expected amount of time to recover from the failure of server* $\mathfrak{S}_i$ as:

$$R_{Q_i}(t) = \max_{1 \leq j \leq n, j \neq i} R_{Q_{i,j}}(t) \tag{4.1}$$

where $R_{Q_{i,j}}(t)$ is the amount of time that server $\mathfrak{S}_j$ would take to rebuild, from its backup images, the states of the corresponding checkpoint units on $\mathfrak{S}_i$ as of time t. Note that the definition uses "max" because the recovery time is indeed determined by the *slowest* among the parallel recovery

processes (i.e., a server failure is recovered only after all the other servers complete their recovery tasks). It should also be noted that the definition of $R_{Q_{i,j}}(t)$ can be formally expressed as:

$$R_{Q_{i,j}}(t) = \sum_{u \in Q_{i,j}} R_u(t) \tag{4.2}$$

where $R_u(t)$ is the amount of time that $\mathfrak{S}_j$ would spend to rebuild the state of checkpoint unit u as of time t. In Figure 4.2, $R_{Q_1}(t)$ corresponds to the polyline that upper bounds the grey area at the bottom. Note that $R_{Q_1}(t) = \max(R_{Q_{1,2}}(t), R_{Q_{1,3}}(t)) = \max(R_{u_1}(t) + R_{u_2}(t), R_{u_3}(t))$.

Furthermore, we can define the expected time to recover from the failure of an *arbitrary* server as $R_Q(t) = \frac{1}{n}\sum_{i=1}^{n} R_{Q_i}(t)$ assuming that each server has the same likelihood of failure. Finally, we express the expected recovery time over *period* $[a, b]$ as $R_Q[a, b] = \frac{1}{b-a}\int_a^b R_Q(t)dt$.

Expected Time to Recover a Checkpoint Unit

Hereafter, we discuss how long it would take to recover a checkpoint unit u assuming that $\mathfrak{S}_i$ possessed u until it failed at time t and $\mathfrak{S}_j$ wants to rebuild the state of u as of t, using its backup image u'. As described earlier, we denote such an amount of time with $R_u(t)$. First, we note that $\mathfrak{S}_j$ might have a checkpoint message that is not yet completely copied back to the backup image u'. We use $\gamma_u(t)$ to represent the amount of time to complete this task (if there is no such message, then $\gamma_u(t) = 0$). Second, we also note that the output queues at upstream servers would have buffered tuples for u'. In Figure 4.1, for example, the output queue of u_3 on $\mathfrak{S}_1$ would retain such tuples for u'_4. We use $\delta_u(t)$ to denote the amount of time that $\mathfrak{S}_j$ would take to consume those *buffered input tuples* until it rebuilds the state of u as of t. Therefore, we get that:

$$R_u(t) = \gamma_u(t) + \delta_u(t). \tag{4.3}$$

In what follows, we define each of $\gamma_u(t)$ and $\delta_u(t)$.

Expected Time to Consume a Checkpoint Message

In order to define $\gamma_u(t)$ (the expected amount of time to consume the pending checkpoint message formed from unit u), we first introduce related terms. Given unit u, let $\alpha_{u'}(t)$ denote the last time before t that $\mathfrak{S}_j$ *received* a checkpoint message for the backup image u'. Also, let $\beta_{u'}(t)$ be the last time before t that $\mathfrak{S}_j$ began *consuming* a checkpoint message for u'. If $\alpha_{u'}(t) \leq \beta_{u'}(t)$, then it is sure that $\mathfrak{S}_j$ has begun consuming the last checkpoint message for u'. In Figure 4.2, for instance, $\alpha_{u'_1}(5) = 4.125 \leq \beta_{u'_1}(5) = 4.63$ and $\mathfrak{S}_2$ at time 5 is consuming the checkpoint message for u'_1. Note that the condition will hold (even after the task finishes) until $\mathfrak{S}_j$ receives a new checkpoint message for u'_1. In other words, if $\alpha_{u'}(t) > \beta_{u'}(t)$, then $\mathfrak{S}_j$ must have a pending checkpoint message for u' that it has not yet responded to. Therefore, we get that:

$$\gamma_u(t) = \begin{cases} c_{u'}(\alpha_u(t)) & \text{if } \alpha_{u'}(t) > \beta_{u'}(t) \\ \max(\beta_{u'}(t) + \frac{c_{u'}(\alpha_u(t))}{1-l_{Q_j}} - t, 0)(1 - l_{Q_j}) & \text{otherwise} \end{cases} \tag{4.4}$$

where $\alpha_u(t)$ is the start time of the checkpoint that sent the checkpoint message to $\mathfrak{S}_j$ at time $\alpha_{u'}(t)$, and $c_{u'}(\alpha)$ is the amount of time that $\mathfrak{S}_j$ would take to consume (during recovery without being interleaved with regular processing) the checkpoint message that $\mathfrak{S}_i$ began forming for u at time α. Note that the start time of checkpoint (i.e., $\alpha_u(t)$) determines the contents of the checkpoint message and also the cost of consuming it (i.e., $c_{u'}(\alpha_u(t))$). In the definition, $\frac{c_{u'}(\alpha_u(t))}{1-l_{Q_j}}$ represents the duration of the paste when interleaved with regular processing and therefore $\beta_{u'}(t) + \frac{c_{u'}(\alpha_u(t))}{1-l_{Q_j}}$ expresses when the paste task will end. $\max(\beta_{u'}(t) + \frac{c_{u'}(\alpha_u(t))}{1-l_{Q_j}} - t, 0)$ represents how long it was supposed to take to finish the paste task if interleaved with regular processing. $1 - l_{Q_j}$ denotes the relative difference between the case where the paste task fully uses the CPU and the case where the paste task shares the CPU with regular processing.

Expected Time to Process Buffered Input Tuples

Once $\mathfrak{S}_j$ consumes all the pending checkpoint messages, each backup image u' at $\mathfrak{S}_j$ will have the image of u as of the time that the last checkpoint started (i.e., time $\alpha_u(t)$). To rebuild the image of u as of t (i.e., the moment of failure), u' then has to process input tuples preserved in the output queues at upstream servers. We define the expected time to process such buffered input tuples as:

$$\delta_u(t) = \int_{\alpha_u(t)}^{t} \frac{C_i}{C_j} l_u(\tau) d\tau \tag{4.5}$$

where C_i and C_j are the processing capabilities of $\mathfrak{S}_i$ and $\mathfrak{S}_j$, respectively, and $l_u(\tau)$ is the processing load (the percentage of CPU cycles used per time unit) of unit u on Server $\mathfrak{S}_i$ at time τ. Therefore, $\frac{C_i}{C_j} l_u(\tau)$ represents the percentage of CPU cycles that $\mathfrak{S}_j$ would consume processing the tuples that unit u on $\mathfrak{S}_i$ processed at time τ.

4.8.2 Expected Recovery Time after a Checkpoint

Expected Recovery Time after a Capture

We use $R_{Q_{i,j}}(u, t)$ to denote the expected amount of time to recover $Q_{i,j}$ when capturing unit u finishes. Using the definition of $R_{Q_{i,j}}(t)$ in Section 4.8.1, we get that $R_{Q_{i,j}}(u, t) = \sum_{h \in Q_{i,j}} R_h(u, t)$, where $R_h(u, t)$ represents the expected recovery time of unit h when capturing unit $u \in Q_{i,j}$ finishes. Note that capturing u will reduce the recovery time of h iff $h = u$. Therefore, we get that:

$$R_h(u, t) = \begin{cases} R_h(t + c_h(t)) - \Delta R_u(t) & \texttt{if } h = u \\ R_h(t + c_h(t)) & \texttt{otherwise.} \end{cases}$$

Note that $\Delta R_u(t)$ represents the reduction in recovery time due to capturing u (see Section 4.3.1) and $t + c_h(t)$ represents the time when capturing u will finish.

Expected Recovery Time after a Paste

We use $R_{Q_{i,j}}(u', t)$ to denote the expected amount of time to recover $Q_{i,j}$ when the backup server finishes a paste task for unit u. Using the definition of $R_{Q_{i,j}}(t)$ in Section 4.8.1, we get that

$R_{Q_{i,j}}(u',t) = \sum_{h \in Q_{i,j}} R_h(u',t)$, where $R_h(u',t)$ represents the expected recovery time of unit h when the backup server finishes a paste task for unit $u \in Q_{i,j}$. Note that doing the paste task for u (i.e., updating the backup image u') will reduce the recovery time of h iff $h = u$. Therefore, we get that:

$$R_h(u',t) = \begin{cases} R_h(t + \frac{c'_h(t)}{1-l_{Q_{j'}}}) - c'_h(t) & \texttt{if } h = u \\ R_h(t + \frac{c'_h(t)}{1-l_{Q_{j'}}}) & \texttt{otherwise,} \end{cases}$$

where $l_{Q_{j'}}$ is the average processing load of the backup server $\mathfrak{S}_{j'}$ of unit u and $\frac{c'_h(t)}{1-l_{Q_{j'}}}$ is the duration of the paste task if interleaved with regular processing. Note that those expected recovery times are the best guesses about the future and the actual recovery times may differ (e.g., if the primary server $\mathfrak{S}_i$ captures a unit in $Q_{i,j}$, the recovery time will change).

Chapter 5

Fast and Highly-Available Stream Processing over the Internet

In this chapter, we consider stream processing in a macro-scale that spans diverse areas of the globe. This will allow us to monitor various events occurring around the world and make smart decisions in near real time. To realize correct and timely processing in such a setting, however, we must address the following challenges:

1. As we use more servers, server failures are more likely to occur. A failed server cannot send data and may lose data essential for processing.
2. Computer networks are vulnerable to link failures and congestion. Communication outages sometimes last tens of minutes or more [35, 105].
3. A server can be overloaded due to unexpected surges of data streams [148] or by other applications that share the server. In this case, stream processing at subsequent servers also gets delayed.

We observe that previous techniques for reliable stream processing [68, 69, 118, 22] cannot successfully address the challenges above. These techniques commonly deploy, for each operator, k replicas on independent servers to tolerate up to (k-1) simultaneous failures. In these techniques, however, only one of the peer replicas can feed a downstream replica. If such a replica fails (or gets overloaded/disconnected), the subsequent processing stalls until the downstream replica notices the problem and acquires a new input connection from another functioning upstream replica.

To overcome the limitations of previous techniques, we propose a new approach where multiple replicas send outputs to each downstream replica so that it can use whichever data arrives first. To further expedite processing, our approach also allows replicas to independently process any available data. This may cause multi-input replicas to produce outputs in different orders. Despite such relaxation, our approach always delivers the results that non-replicated stream processing would produce without failures. We call this guarantee *replication transparency*.

Our approach uses more resources than previous approaches because all replicas send outputs downstream. However, our approach has a distinct advantage of improving performance since it always uses the fastest among multiple replicated data flows. Furthermore, the system naturally remains resilient against local congestions. It is also always operational without detecting failures and switching from failed replicas to functioning replicas.

The contributions made in this chapter are as follows:

1. We propose a new replication framework for fast and robust Internet-scale stream processing.
2. We define replication transparency as the key concept for our replication framework. We also devise stream processing primitives for replication transparency.
3. We develop an algorithm for managing replicas. This algorithm improves both performance and reliability, while efficiently using network resources.
4. We demonstrate the utility of our work through experiments on PlanetLab [107] and trace-driven simulations.

The rest of this chapter is organized as follows. We give an overview of our replication framework in Section 5.1 and design a semantic model for replicated stream processing in Section 5.2. In Section 5.3, we devise stream processing primitives for our replication model. Next, we discuss replica management in Section 5.4 and show experimental results in Section 5.5. We provide a summary of this chapter in Section 5.6.

5.1 Background

In this section, we first illustrate examples that describe how replication can improve both the performance and availability of a distributed stream processing system (Section 5.1.1). Next, we describe the assumptions behind our work (Section 5.1.2) and our replication framework (Section 5.1.3). Finally, we stress the specific problems tackled in this chapter (Section 5.1.4).

5.1.1 Motivating Examples

Figure 5.1 depicts a scenario where we are interested in finding long-latency communication paths among a subset of PlanetLab servers [107]. These servers ping each other every second. If a server detects another server that does not respond in a second, the former reports this in the form of a data stream. In Figure 5.1, server A reports that communication path A-C has been slow or down since 9:00:00 and that also A-B became so at 9:00:02. However, such a round trip delay might have appeared because the remote server slowly reacted to the ping message due to performing other important tasks. Therefore, we want to identify latencies resulting from network problems only (i.e., those certainly not caused by busy remote servers). In the example, the Join operator $\bowtie_3$ correlates the report of slow paths with the report of remote servers' load readings, based on timestamp as

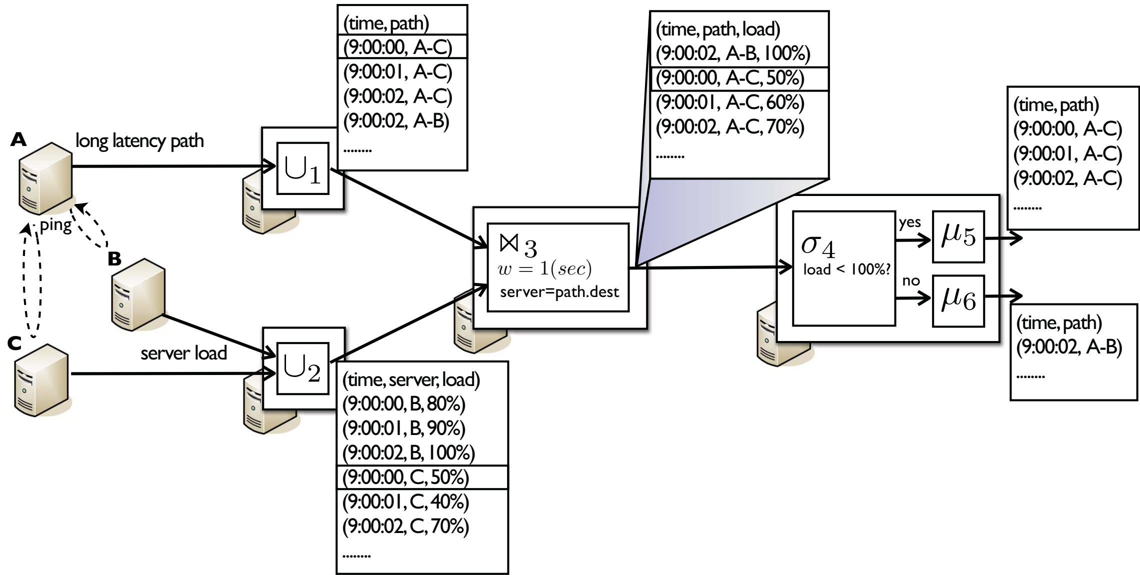


Figure 5.1: Non-Replicated Stream Processing.

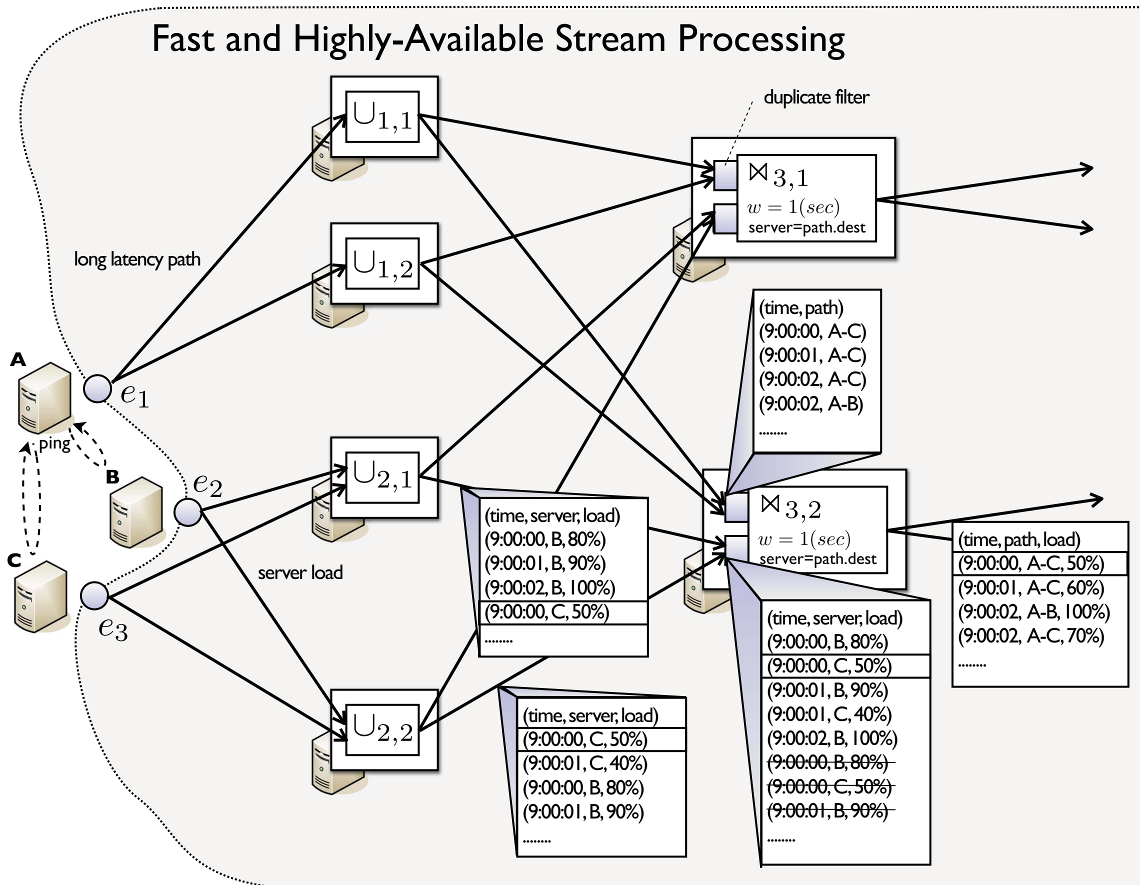


Figure 5.2: Replicated Stream Processing

well as server ID. The subsequent Filter σ_4 categorizes slow paths according to the load readings and then Maps μ_5 and μ_6 drop the load field values from the tuples from σ_4.

Impact of Slow Streams. As illustrated in Figure 5.1, a slow stream delays all processing downstream from it and may even delay the processing of other timely streams. In the example, the processing of a timely tuple (9:00:00, *A*-*C*) is delayed at $\bowtie_3$ as its matching tuple (9:00:00, *C*, 50%) arrives late. Tuple (9:00:00, *C*, 50%) was first hindered on the way to Union $\cup_2$ due to a network problem. In Figure 5.1, (9:00:00, *C*, 50%) arrives at $\cup_2$ after (9:00:02, *B*, 100%), a tuple generated even two seconds later than (9:00:02, *C*, 50%).

Benefits of Replication. $\bowtie_3$ in Figure 5.1 is a stateful operator (i.e., the production of an output tuple usually involves multiple input tuples). For this reason, the pace of its operation is determined by the slowest input flow. As Figure 5.2 illustrates, replicating input flows allows the operator to obtain input tuples at earlier times (through the fastest among the replicated input flows) than in the non-replicated case. In the example, replicas $\cup_{2,1}$ and $\cup_{2,2}$ both send their outputs to $\bowtie_{3,2}$ in parallel. Therefore, $\bowtie_{3,2}$ can use whichever tuple arrives first from $\cup_{2,1}$ and $\cup_{2,2}$, while eliminating duplicates (i.e., tuples whose copies already arrived at the operator through other stream replicas; see those lined-through). This type of replication can reduce the average and variance of result latencies. It also allows the system to continue its operation despite network link congestion, network failures, and server failures (even without detecting such problems).

5.1.2 Assumptions

In this chapter, we make the following assumptions:

- **System.** We assume the Internet as the substrate for stream processing. We assume that the network has abundant computation and communication resources.
- **Communication.** We assume that the network layer runs a reliable, in-order, point-to-point message delivery protocol such as TCP.
- **Failure Model.** We assume fail-stop server/network failures. We do not consider Byzantine failures where faulty components can behave in arbitrarily erroneous ways.
- **Query.** We assume that queries are translated into a directed acyclic graph of operators. In this chapter, we consider five representative stream processing operators, namely Filter, Map, Union, Aggregate, and Join.

5.1.3 The Basic Architecture

To manage the system in a scalable fashion, we group servers into logical clusters each of which comprises tens of servers. For each cluster to autonomously handle queries that span distant stream sources and applications, each cluster includes servers at *diverse* locations rather than those only within a small area. For this reason, clusters may overlap significantly with each other in terms of their geographic coverage. Hereafter, we focus on replicating operators within a logical cluster.

As illustrated in Figure 5.2, our approach guarantees fast and reliable processing by making multiple replicas feed each downstream replica (observe that replicas $\cup_{2,1}$ and $\cup_{2,2}$ both send data to the second input of $\Join_{3,2}$). To achieve reliable processing with unstable stream sources, we introduce entry points, which serve as the starting points for reliable stream processing (see e_1, e_2, and e_3 in Figure 5.2). Entry points buffer input tuples from external stream sources until they safely arrive at the downstream replicas. They also replicate input tuples at other peer entry points to improve availability. Finally, entry points have well-synchronized clocks [109] and can timestamp input tuples on behalf of unsynchronized stream sources.

5.1.4 Problem Statements

To complete the replication framework described above, we need to address the following research challenges.

1. What *execution semantics* should we consider for replicated stream processing?

   We take the position that replication must be *transparent* to users. In other words, we want to guarantee that replicated processing (with failures and delays) always produces the results that would appear without replication and failures. We discuss the details in Section 5.2.

2. How should we *extend processing primitives* for replication transparency?

   A simple way to ensure replication transparency would be to identically execute peer replicas. As shown in Section 5.5, however, such an approach introduces extra delays. Therefore, we let replicas run differently as long as replication transparency is achievable. To produce correct results, we also need to remove duplicates, as shown by lined-through tuples in Figure 5.2. In Section 5.3, we design primitives that remove duplicates in a non-blocking fashion.

3. How should we manage replicas?

   As demonstrated in Section 5.5, the deployment of replicas can significantly affect the network cost, performance (in terms of the average result latency), and availability (in terms of the probability that the system delivers results to client applications within a certain latency). In Section 5.4, we discuss managing replicas in a manner that improves performance and availability, while efficiently using network resources.

5.2 Replication Transparency

The central notion behind this work is *replication transparency.* Under this guarantee, each client application always receives the same results as in the *ideal non-replicated scenario* where the system is completely free from failures and delays. Because timestamps of tuples are usually used as essential elements for processing [2], replication transparency also requires the timestamps to represent the precise times when the tuples would be generated in the ideal non-replicated scenario. Our goal in this chapter is to deliver such results to client applications as soon as possible, by use of replication.

The definition of replication transparency is specified only in terms of *the results to client applications*. A natural question at this point is how we should instrument operator replicas to produce such results in the end. We first take the position that each operator replica must generate the tuples that would appear in the ideal non-replicated scenario. We, however, let each replica run independently, while processing any available data. As demonstrated in Section 5.5, this relaxation expedites processing, compared to previous approaches that execute replicas identically at distant servers. This, however, causes multi-way operators, such as Union and Join, to generate tuples in a random order. In Figure 5.2, replicas $\cup_{2,1}$ and $\cup_{2,2}$ produce outputs in different orders for this reason.

Despite the complications above, we achieve replication transparency as follows:

1. *We merge stream replicas into a non-duplicate stream using a non-blocking filter.* In our replication framework, each operator replica receives inputs from multiple upstream replicas. An operator such as a count aggregate, however, may produce incorrect results if it processes duplicate tuples. Furthermore, processing duplicates would waste CPU cycles. In Section 5.3.2, we devise non-blocking filters that eliminate duplicates from disordered stream replicas. In Figure 5.2, the second duplicate filter of $\bowtie_{3,2}$ merges two stream replicas into a non-duplicate stream that again contains the same tuples as its input streams. Because the input streams of the filter have different orders and because the filter operates in a non-blocking fashion, the output of the filter has a new different order.

2. *We sort disordered streams only when necessary.* Order-sensitive applications and operators, such as those with count-based windows, must process inputs in the order of the ideal non-replicated scenario. In Section 5.3.3, we discuss the details of sorting streams. Because sorting introduces delays, we sort streams only when necessary. In other words, we bypass the sorting phase for order-insensitive operators and applications.

3. *We redesign operators so that they can produce, from disordered input streams, the output tuples that would appear in the ideal non-replicated scenario.* The output stream need not be ordered, because downstream replicas can handle disorder. We call this property of operators *replica consistency* because it guarantees that replicas always produce consistent output streams from consistent input streams. We say that two streams are *consistent* if they contain the same tuples regardless of internal order. In Sections 5.3.4 through 5.3.6, we devise non-blocking implementations of Filter, Map, Union, and Join and a blocking implementation of Aggregate. All these operators guarantee replica consistency.

The arguments above state that we can achieve replication transparency by eliminating duplicates, minimally sorting data streams, and making every operator ensure replica consistency. Replica consistency can be defined formally as follows:

Definition 1. *Infinite streams S and S' are consistent (denoted by $S \equiv S'$) if they contain the same tuples regardless of internal order. Specifically, $S \equiv S'$ if there exists a permutation $\mu : \mathbb{N} \to \mathbb{N}$ such that $S[i] = S'[\mu(i)]$, where $S[i]$ denotes the ith tuple in stream S.*

Definition 2. ***(Replica Consistency)*** *Let $o(S_1, S_2, \cdots, S_n)$ denote the set of all possible output streams that operator o with n inputs can generate from input streams $S_1, S_2, \cdots, S_n$. Then, we say that operator o guarantees replica consistency if any possible output streams $O \in o(S_1, S_2, \cdots, S_n)$ and $O' \in o(S'_1, S'_2, \cdots, S'_n)$ are consistent (i.e., $O \equiv O'$) for any consistent input streams S_i and S'_i (i.e., $S_i \equiv S'_i, 1 \leq i \leq n$).*

It should be noted that a query guarantees replica consistency if all of its constituting operators guarantee replica consistency. Theorem 1 formally states this property.

Theorem 1. *Let Q denote a query with n inputs and one output. If all the constituent operators of Q guarantee replica consistency, Q also guarantees replica consistency (i.e., $R(S_1, S_2, \cdots, S_n) \equiv R'(S'_1, S'_2, \cdots, S'_n)$ for any replicas R and R' of Q and any input streams $\{S_i\}_{i=1}^n$ and $\{S'_i\}_{i=1}^n$ such that $S_i \equiv S'_i$).*

Proof: If Q contains only one operator, trivially Q guarantees replica consistency. Otherwise, denote the terminal operator of Q as o (i.e., o produces the output stream of Q). Then, for each input j ($1 \leq j \leq m$) of operator o, let Q_j be the query that consists of operators upstream from j. For a replica R of Q, let $R_j(S_1, S_2, \cdots, S_n)$ denote an arbitrary sequence of tuples that can appear on j if R processes input streams $S_1, S_2, \cdots, S_n$. Since each Q_j has less operators than Q (note that $o \notin Q_j \subseteq Q$), by the induction hypothesis, we get that $R_j(S_1, S_2, \cdots, S_n) \equiv R'_j(S'_1, S'_2, \cdots, S'_n)$ for another replica R' of Q and input streams $\{S_i\}_{i=1}^n$ and $\{S'_i\}_{i=1}^n$ such that $S_i \equiv S'_i$ ($1 \leq i \leq n$). Since o also guarantees replica consistency, the replicas of o in R and R' produce consistent output streams as they receive consistent input streams $\{R_j(S_1, S_2, \cdots, S_n)\}_{j=1}^m$ and $\{R'_j(S'_1, S'_2, \cdots, S'_n)\}_{j=1}^m$, respectively. As a result, we get that $R(S_1, S_2, \cdots, S_n) \equiv R'(S'_1, S'_2, \cdots, S'_n)$. $\square$

Theorem 1 can be generalized for a query with multiple outputs by forming sub-queries consisting of operators upstream from each output and applying the theorem to such sub-queries. In Section 5.3, we discuss extending operators so that they can guarantee replica consistency.

5.3 Extension for Replication Transparency

In this section, we devise the processing primitives for replication transparency. In Section 5.3.1, we introduce punctuations because many of our primitives use them. Next, we discuss filtering out duplicates in Section 5.3.2 and sorting streams in Section 5.3.3. In Sections 5.3.4 through 5.3.6, we extend stream processing operators for replica consistency. Any operator, including a user-defined one, can be safely used in our replication framework if it guarantees replica consistency.

5.3.1 Management of Punctuations

In our approach, either stream sources or their downstream entry points timestamp tuples using well-synchronized clocks. They also periodically send special values, called punctuations [141, 22]. Punctation p in input stream S guarantees that the timestamp of any subsequent tuple in S will be

Algorithm 3: Duplicate Filtering

```
1  whenever tuple t arrives from stream replica S_i ∈ {S_j}_{j=1}^k do
2      if t.timestamp > max_punctuation then
3          count[t][i] ← count[t][i] + 1;
4          if count[t][i] > max_count[t] then
5              output(t);
6              max_count[t] ← count[t][i];

7  whenever punctuation p arrives from any stream replica do
8      if p > max_punctuation then
9          output(p);
10         max_punctuation ← p;
11         remove all count[t][*] such that t.timestamp ≤ p;
```

larger than p. All streams in the system can also satisfy this property if each operator forwards p as soon as it receives p via all its inputs. This is because an operator in that situation will always receive tuples with timestamps larger than p. This implies that, in the ideal non-replicated scenario, the operator would process all these tuples after time p, and thus the timestamps of all later output tuples must be larger than p.

5.3.2 Duplicate Filtering

As pointed out in Section 5.2, we use duplicate filters to eliminate duplicate tuples from stream replicas. Duplicate filters must deal with disorder and multiple occurrences of the same tuple in each stream replica. They also should not block the data flow. Algorithm 3 describes the operation of our duplicate filter. For tuple t from stream replica S_i, we first check if t satisfies the condition in line 2. Otherwise (i.e., if t.timestamp $\leq$ max_punctuation), t is a duplicate because the filter received the current maximum punctuation (max_punctuation) from a stream replica before. This implies that the filter already received, from the same stream replica, all tuples t' such that t'.timestamp $\leq$ max_punctuation.

Next, the filter uses a variable count$[t][i]$ to remember how many times it received tuple t from stream replica S_i (line 3). If the filter received t from S_i more times than any other stream replica, it passes t to the operator as a non-duplicate (line 5). Otherwise (i.e., if $\exists S_j$ such that count$[t][i] \leq$ count$[t][j]$), t is a duplicate because the filter already received a corresponding tuple from S_j. Lines 7-11 describe that, whenever a new punctation arrives, the filter can safely remove count variables for all the known duplicate tuples. The life time of each count variable is thus bounded by the punctuation interval.

The following theorem proves the correctness of Algorithm 3.

Theorem 2. *Let $\mathfrak{D}(S_i)_{i=1}^k$ denote the set of all output streams that duplicate filter $\mathfrak{D}$ can produce from stream replicas $\{S_i\}_{i=1}^k$. If $\{S_i\}_{i=1}^k$ are consistent, any $O \in \mathfrak{D}(S_i)_{i=1}^k$ is also consistent with them (i.e., $O \equiv S_i$, $1 \leq i \leq k$).*

Proof: Since $\{S_i\}_{i=1}^k$ are consistent, all of them commonly contain an arbitrary tuple t the same number of times (say m). It suffices to prove that $\mathfrak{D}$ passes t, m times in any case. (1) if $m = 0$, $\mathfrak{D}$ cannot pass t since none of the streams $\{S_i\}_{i=1}^k$ contains t. (2) If $m > 0$, without loss of generality, suppose that $\mathfrak{D}$ receives the mth t from S_1 before it receives the mth t from S_j $(2 \leq j \leq k)$. Since this implies that $\mathfrak{D}$ has not yet received punctuation $p \geq t$.timestamp from any stream replica, t must satisfy the condition in line 2. Since it also implies that count$[t][1] >$ count$[t][j]$ $(2 \leq j \leq k)$, t must satisfy the condition in line 4. Therefore, D must pass t to the operator. This also must be the mth output of t because, by the induction hypothesis, $\mathfrak{D}$ must have passed t, $(m-1)$ times before this. $\mathfrak{D}$ then sets both count$[t][1]$ and max_count$[t]$ to m. Since count$[t][j] \leq$ max_count$[t] = m (2 \leq j \leq k)$, $\mathfrak{D}$ will filter out t afterwards . $\square$

5.3.3 Sorting Streams

If a stream S feeds an order-sensitive operator/application, we sort S using punctuations. We first insert each tuple from S into a sorted list $\mathfrak{L}$. This list contains tuples in the order of increasing timestamps. If multiple tuples have the same timestamp, we enforce a unique ordering, for example, by regarding these tuples as byte arrays and radix-sorting them. Whenever a punctuation p arrives from stream S, we remove tuples $\{t \in \mathfrak{L} : t.\text{timestamp} \leq p\}$ from $\mathfrak{L}$, while passing them, in the sorted order, to the next operator/application. This sorting phase each time observes the entirety of the tuples in S up to a punctuation p and passes them in a unique order. Therefore, it allows all peer replicas to process inputs in the same order, with extra delays that depend on the punctuation interval. This sorting phase is described in Algorithm 6 as part of an order-sensitive operator (refer to lines 1-6).

In the rest of this section, we discuss extending operators for replica consistency.

5.3.4 Stateless Operators and Replica Consistency

Stateless operators are those that produce each output tuple based on only the last input tuple. Filter forwards each input tuple if the tuple satisfies a pre-defined predicate. Map converts each input tuple into a different tuple. Union merges two or more streams into a single output stream.

Each stateless operator naturally guarantees *replica consistency* because it processes each tuple deterministically, regardless of the input order. In detail, all replicas of a Filter must pass a tuple if it satisfies the predicate. All replicas of a Map must identically convert each input tuple and replicas of a Union must forward each input tuple. Therefore, our replication framework uses the stateless operators as they are. Filter, Map, Union are all *non-blocking*. Union is *nondeterministic* because, as demonstrated by $\cup_{2,1}$ and $\cup_{2,2}$ in Figure 5.2, its output order can vary depending on the arrival order of tuples across its input streams.

Algorithm 4: Join

1 **whenever** *tuple* t_1 *arrives at input* I_1 **do**
2   **for each** $t_2 \in B_2$ *such that* $|t_1.\text{timestamp} - t_2.\text{timestamp}| < w \wedge P(t_1, t_2)$ **do**
3   output($t_1 \otimes t_2$);
4   B_1.add(t_1);

5 **whenever** *punctuation* p_1 *arrives at input* I_1 **do**
6     $B_2.\text{remove}(\{t_2 \in B_2 : p_1 - t_2.\text{timestamp} > w\})$;

* *Inputs to* I_2 *are processed symmetrically*

5.3.5 Extending Join for Replica Consistency

Join has two inputs I_1 and I_2, window size w, and predicate P. For any input tuples t_1 from I_1 and t_2 from I_2, it outputs the concatenation $t_1 \otimes t_2$ of them if they (a) belong to the same time window (i.e., $|t_1.\text{timestamp} - t_2.\text{timestamp}| < w$) and (b) satisfy predicate P (i.e., $P(t_1, t_2)$ holds). Here, we do not consider the extra features, called slack and timeout, of Join that handle disorder and silence [2]. This is because our approach tackles these issues by use of punctuations, while expediting processing through replication.

We implement Join as illustrated in Algorithm 4. Lines 1-4 output the concatenation of matching input tuples, while using B_i to buffer the tuples entered input I_i. The timestamp of $t_1 \otimes t_2$ is set to $\max(t_1.\text{timestamp}, t_2.\text{timestamp})$, which is the time when the tuple would be produced in the ideal non-replicated scenario. Lines 5-6 discard the buffered tuples that will no longer be used. Any tuple $t_2 \in B_2$ that satisfies the condition in line 6 cannot match with any tuple t_1 that will arrive at I_1. This is because $t_1.\text{timestamp} - t_2.\text{timestamp} > p_1 - t_2.\text{timestamp} > w$.

This Join implementation is *non-blocking* because it produces each output tuple as soon as it obtains both constituent input tuples. It guarantees *replica consistency* because, for each pair of matching input tuples, it produces the concatenation of them exactly once, regardless of the inter-arrival order of the input streams. Similar to Union, Join is *nondeterministic* and may introduce *disorder*.

5.3.6 Extending Aggregate for Replica Consistency

Aggregate [2] splits input stream I into substreams $\{I[g]\}_{g \in \mathcal{G}}$, where $\mathcal{G}$ is the set of groups and $I[g]$ is a subsequence of I that contains tuples belonging to group g. For each substream $I[g]$, this operator forms windows (sets of tuples) based on either *timestamps* or *the count of tuples*. If a window expires, Aggregate produces an output tuple computed from the tuples in the window. For the reasons described in Section 5.3.5, we do not consider slack and timeout.

For *replica consistency*, Aggregate must form and close windows uniquely, despite disorder in the input stream.

Aggregate with Time Windows. In this case, each group g forms windows of w seconds every s seconds. Therefore, for input tuple t, we can determine the set of windows $\mathcal{W}(t.\text{timestamp})$ that

Algorithm 5: Aggregate with Time Windows

```
1  whenever tuple t arrives do
2      for each window w ∈ 𝒢(t).𝒲(t.timestamp) do
3          w.update(t);

4  whenever punctuation p arrives do
5      for each g ∈ 𝒢 do
6          for each w ∈ g.windows such that w.expir_time ≤ p do
7              g.windows.remove(w);
8              output(w.get_summary());
```

Algorithm 6: Aggregate with Count-based Windows

```
1  whenever tuple t arrives do
2      𝔏.insert(t); // sorted list

3  whenever punctuation p arrives do
4      while L ≠ ∅ ∧ 𝔏.first().timestamp ≤ p do
5          agg_count_windows(𝔏.first());
6          𝔏.remove_first();

7  agg_count_windows(t)
8  begin
9      count++;
10     for each window w ∈ 𝒢(t).𝒲(count) do
11         w.update(t);
12         if w.expir_count ≤ count then
13             g.windows.remove(w);
14             output(w.get_summary());
15 end
```

t belongs to. For example, when $w = 10$ (sec) and $s = 5$ (sec), we get $\mathcal{W}(9{:}00{:}43) = \{[9{:}00{:}35, 9{:}00{:}45), [9{:}00{:}40, 9{:}00{:}50)\}$. Lines 1-3 in Algorithm 5 uniquely form windows regardless of the input order. Then, lines 4-8 use punctuations to find the windows that cannot contain more tuples. This allows Aggregate to produce the same output tuples from any consistent input stream.

Aggregate with Count-based Windows. This operator uses, for each group g, a window of w tuples that skips s tuples whenever it moves. Because this operation is order-sensitive, we sort input stream as described in Section 5.3.3 (lines 1-6 in Algorithm 6). After this, the operator forms and closes windows (lines 7-15) using the count of input tuples.

The two implementations above are *blocking* because they wait for punctuations to assure that they obtained all the required tuples. With time windows, we can minimize the delay by making stream sources or their entry points produce punctuations at expiration times of windows. With count-based windows, the delay depends on the punctuation interval.

Algorithm 7: Replica Deployment (for query $\mathfrak{Q}$)

```
1  for i = 1 to k_max do
2  |  deploy(Q);
3  deploy(Q)
4  begin
5  |  for each operator o ∈ Q do
6  |  |  C ← {s ∈ S : load(s) + load(o) < α · capacity(s) ∧
7  |  |         min(latency(s, s')) > d_min, ∀s' ∈ S(o)};
8  |  |  Find s* ∈ C such that ∀s ∈ C
9  |  |         cost_s*(in(o) ∪ out(o)) ≤ cost_s(in(o) ∪ out(o));
10 |  |  s*.deploy(o);
11 end
```

5.4 Management of Replicas

As demonstrated in Section 5.5, the deployment of replicas affects both result latencies and resource usage. In this section, we discuss managing replicas to improve performance and availability, while efficiently using network resources. In Section 5.4.1, we discuss deploying replicas initially in a resource-efficient fashion. In Section 5.4.2, we devise an algorithm that reduces the resource usage to a target level, while striving to minimize degradation in performance and availability. For this, the algorithm finds and then discards the least useful stream/operator replicas (i.e., those that make the smallest contribution to the downstream processing upto client applications). In Section 5.4.3, we consider reviving garbage-collected replicas to cope with changes in system conditions.

5.4.1 Deployment of Replicas

As illustrated in Section 5.1.3, our replication framework forms logical clusters each of which comprises servers at diverse locations. Servers in the same cluster elect a coordinator for them. In this subsection, we discuss how the coordinator of cluster $\mathfrak{S}$ should deploy a predefined number ($k_{\max}$) of replicas for each operator. Our strategy strives to *minimize the overall network cost* similarly to operator placement approaches in the non-replicated context [3, 106]. In our replication framework, for a collection of stream replicas $\mathfrak{R}$, the network cost of $\mathfrak{R}$, cost($\mathfrak{R}$), is defined as the sum of individual stream replicas' network costs. Formally, $\text{cost}(\mathfrak{R}) = \sum_{S \in \mathfrak{R}} \text{cost}(S)$ where cost(S) denotes the cost of stream replica S. cost(S) is in turn defined as $\text{rate}(S) \cdot \text{latency}(S)$ where rate(S) and latency(S) are the data rate and the network latency of stream replica S, respectively. This *bandwidth-delay product* is based on the idea that the longer data stays in the network, the more resources it tends to use. An optimal deployment under this metric also tends to choose fast network links, thereby accomplishing low-latency processing.

The Replica Deployment Algorithm. Algorithm 7 describes our replica deployment strategy. Each of the $k_{\max}$ deployment phases creates, for each operator o in query $\mathfrak{Q}$, a new replica on a server that minimally increases the network cost. For a new replica, it first finds good candidate

servers $\mathfrak{C}$ (lines 6-7). Such a server $\mathfrak{s}$ must not be busy (line 6). load($\mathfrak{s}$) and load(o) are the current load of server $\mathfrak{s}$ and the expected load of the new replica of o, respectively. capacity($\mathfrak{s}$) is the processing capacity of server $\mathfrak{s}$. With $\alpha < 1$, the condition in line 6 checks if $\mathfrak{s}$ is likely to have enough available CPU cycles even if it runs the new replica. A good candidate server $\mathfrak{s}$ also must have a low risk of falling into the same network partition with any $\mathfrak{s}'$ of servers $\mathfrak{S}(o)$ that currently run replicas of o. To ensure this, the heuristic in line 7 uses network latencies to ensure that replicas are always deployed on sufficiently distant servers. In several test cases, it turned out that 30ms to 70ms is a good range for the minimum latency $d_{\min}$ between peer replicas. After finding candidate servers $\mathfrak{C}$, the coordinator chooses the server $\mathfrak{s}^*$ that will minimize the network cost of the input streams in(o) and output streams out(o) of the new replica (lines 8-9). Specifically, in(o) denotes all possible input streams from the replicas directly upstream from any replica of o, and out(o) denotes all possible output streams to the replicas directly downstream from any replica of o. In line 9, $\text{cost}_{\mathfrak{s}}(\text{in}(o) \cup \text{out}(o))$ represents the network cost of these input and output stream replicas, provided that server $\mathfrak{s}$ runs the new replica. Formally, $\text{cost}_{\mathfrak{s}}(\text{in}(o) \cup \text{out}(o)) = \sum_{S \in \text{in}(o)} \text{rate}(S) \cdot \text{latency}(S.\text{source}, \mathfrak{s}) + \sum_{S \in \text{out}(o)} \text{rate}(S) \cdot \text{latency}(\mathfrak{s}, S.\text{destination})$. Finally, the coordinator deploys the new replica on $\mathfrak{s}^*$ (line 10).

Discussion. Our strategy above strives to find, phase-by-phase, the deployment that will minimally increase the network cost while providing the desired availability level. It is, however, hard to find the optimal deployment in the first phase because the data rate of each stream and the processing load of each operator are not yet known (these statistics are available in the later phases). The network cost also changes over time as the data rates and latencies of streams vary. For this reason, we use an approach that initially deploys replicas aggressively and dynamically garbage-collects/revives them afterwards.

5.4.2 Garbage Collection

Our replica deployment algorithm creates $k_{\max}$ replicas for each operator. Between $k_{\max}$ upstream replicas and $k_{\max}$ downstream replicas, it also creates $k_{\max}^2$ stream replicas. Although $k_{\max}$ is usually set to a small number (say 4 or 5 at most) in practice, using all of $k_{\max}^2$ stream replicas would waste system resources. Furthermore, faster operator replicas and those that feed many downstream replicas would have a higher impact on processing than others. Thus, we use a strategy that periodically discards the least useful stream and operator replicas. This is to reduce the resource usage, while minimally degrading performance and availability. To maintain the minimum fault-tolerance level, however, we ensure that at least $k_{\min}$ replicas of each operator survive.

The Garbage-Collection Algorithm. Algorithm 8 illustrates our garbage-collection strategy. Periodically, the coordinator computes the current overall network cost. If the cost is higher than a target utilization level θ (line 1), it finds the group of least useful replicas, relative to the network cost paid for them (lines 2-3). It then asks the related servers to discard them (line 4). dependents(S) in line 2 finds the group of replicas that must be discarded together with stream replica S. For example, if an operator replica $\mathfrak{o}$ has only one output stream S, removing S will make $\mathfrak{o}$ useless

Algorithm 8: Garbage Collection (for stream replicas $\mathfrak{R}$)

```
1  whenever ∑_{S∈𝔑} cost(S) > θ do
2      ℭ ← {dependents(S) : S ∈ 𝔑} − {∅};
3      Find a collection of stream replicas 𝔇* ∈ ℭ such that utility(𝔇*)/cost(𝔇*) ≤ utility(𝔇)/cost((𝔇)), ∀𝔇 ∈ ℭ;
4      discard(𝔇*);

5  dependents(S)
6  begin
7      return dependents(S, ∅);
8  end

9  dependents(S, 𝔇)
10 begin
11     𝔇.add(S);
12     𝔇 ← dependents(S.source, 𝔇);
13     if 𝔇 = ∅ then
14         return ∅;
15     else
16         return dependents(S.destination, 𝔇);
17 end

18 dependents(𝔬, 𝔇)
19 begin
20     if |ℭ(𝔬; 𝔇)| < k_min then
21         return ∅;
22     if need_to_remove(𝔬, 𝔇) then
23         for each S ∈ in(𝔬) ∪ out(𝔬) − 𝔇 do
24             𝔇 ← dependents(S, 𝔇);
25             if 𝔇 = ∅ then
26                 return ∅;
27     return 𝔇;
28 end
```

and therefore necessitates removing all the input streams of $\mathfrak{o}$ as well. In this case, dependents(S) must contain the input streams of $\mathfrak{o}$. Given replicas $\mathfrak{D}$ to discard, lines 09-17 check if removing stream replica S will require removing its source replica (line 12) or destination replica (line 16) for the reason above. Lines 13-14 handle the case where we cannot remove S because we would have fewer than $k_{\min}$ operator replicas if S was removed (see also lines 20-21). Lines 18-28 are to check an operator replica $\mathfrak{o}$. Lines 20-21 are to keep at least $k_{\min}$ operator replicas (in line 20, $\mathfrak{C}(\mathfrak{o}; \mathfrak{D})$ represents the number of replicas of o if we remove replicas in $\mathfrak{D}$). Line 22 checks if removing replicas in $\mathfrak{D}$ will make operator replica $\mathfrak{o}$ useless and thus require removing $\mathfrak{o}$. If so, the algorithm visits the input and output streams of $\mathfrak{o}$ while recursively applying the algorithm (lines 23-26). If removing such streams leads to having fewer than $k_{\min}$ replicas for some operator (line 25), it decides not to remove any streams (line 26).

Measuring the Utility of Each Replica. As shown above, our garbage-collection algorithm considers the utility of each stream. Intuitively, we define utility as the impact of a replica on the data flow towards applications. To compute this, our heuristic first uses duplicate filters to measure the contributions of stream replicas to the next operator. Specifically, each duplicate filter gives, for each input tuple, weights $\frac{1}{1}, \frac{1}{2}, \frac{1}{3}, \cdots$ to its input stream replicas based on how early they deliver the tuple (the fastest replica gets the highest weight each time). These weights, however, do not capture the impact after the next operator. Thus, our heuristic periodically computes the utilities of stream/operator replicas from applications to more upstream replicas. Specifically, for a group of stream replicas $\{S_i\}_{i=1}^{k}$, utility(S_i), the utility of S_i, is computed as $\frac{w(S_i)}{\sum_{j=1}^{k} w(S_j)}$utility($\mathfrak{o}$) where $w(S)$ is the accumulated weight of stream S and utility($\mathfrak{o}$) is the utility of the operator replica $\mathfrak{o}$ that $\{S_i\}_{i=1}^{k}$ commonly feed. utility($\mathfrak{o}$) is set to 1 if $\mathfrak{o}$ is an application. Otherwise, it is computed as $\sum_{S \in \text{out}(\mathfrak{o})}$ utility(S) where out($\mathfrak{o}$) denotes the output streams of $\mathfrak{o}$.

Discussion. As described above, our garbage-collection algorithm victimizes replicas with high cost and small contribution to downstream processing. Replicas of the opposite kind are likely to survive over time. Such surviving replicas are in general those with high popularity (i.e., those eventually connected to a large number of applications), those upstream, and those along fast data flows.

5.4.3 Adaptation to Changes

Our garbage-collection algorithm saves resources while striving to preserve the latency guarantee. If failures or local congestions occur, however, the surviving replicas may experience unexpected delays. We solve this problem by reviving garbage-collected replicas. In detail, if an operator replica observes delays longer than a threshold (say 10 seconds) across its input streams, it first finds garbage-collected input streams that connect to functioning upstream replicas. If there is such one, it revives the stream replica. Otherwise, it revives a garbage-collected upstream operator replica and acquires a new connection from that one. If the problem persists, we can either create more upstream replicas, revive/create peer replicas that could replace the hindered replica, or simply discard the hindered one.

If the coordinator finds that the current network cost is lower than the target level θ, it can also assist hindered replicas as described above.

5.5 Experimental Results

In this section, we present various results that substantiate the utility of our work. In Section 5.5.1, we describe how we set up the experiments and simulations. Then, we present results obtained from our prototype (Sections 5.5.2 through 5.5.4) and a trace-driven simulator (Sections 5.5.5 and 5.5.6).

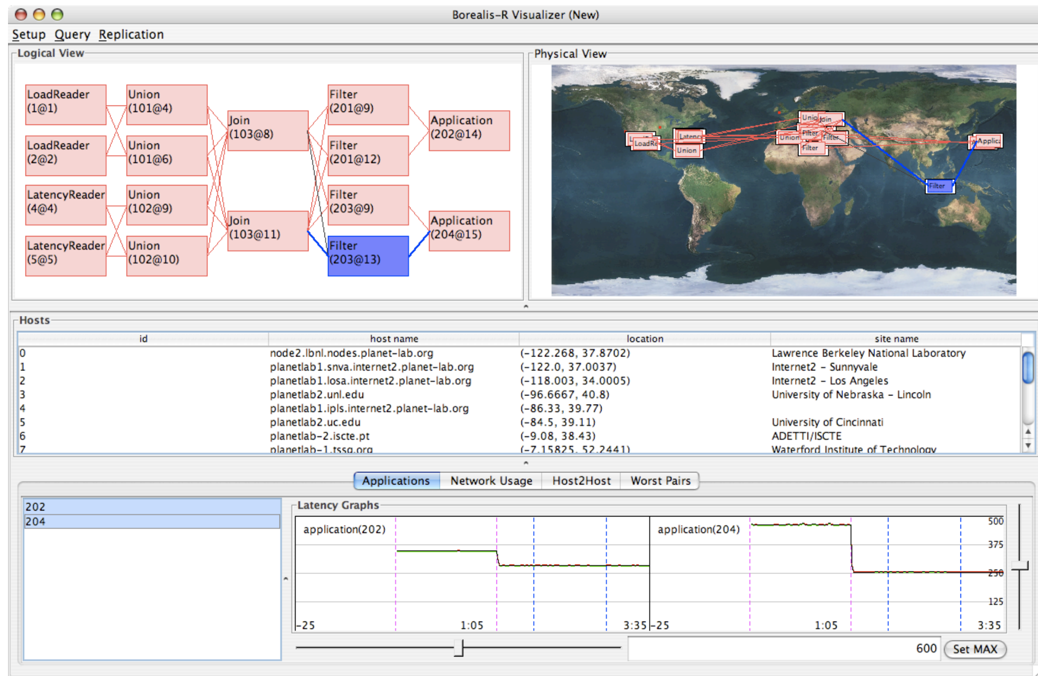


Figure 5.3: Borealis-R Visualizer (Latency)

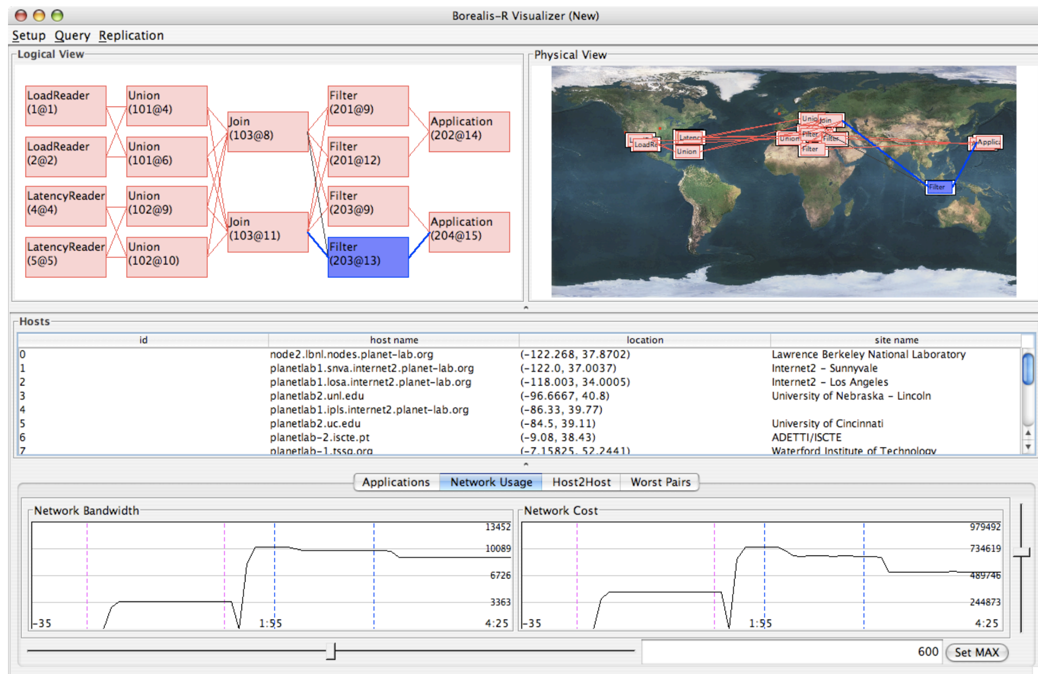


Figure 5.4: Borealis-R Visualizer (Network Cost)

5.5.1 The Setup

In all of our experiments, we assumed a system configuration where servers are grouped into clusters each of which consists of 30 servers. Thus, for our prototype, we chose 30 distant PlanetLab servers [107] that reliably communicate with others. The results obtained, however, varied each time because the servers were used intensively by many users. To compare various approaches under an identical condition, we also conducted simulations. Our prototype and simulator use the same source code except for the communication component. To send tuples and invoke remote procedures, our prototype uses TCP sockets. On the other hand, the simulator makes local calls while emulating network delays based on a trace. We obtained this trace by recording actual network delays between 100 PlanetLab servers every 10 seconds for a month starting from February 13, 2007. Figures 5.3 and 5.4 display the visualizer of our prototype.

Our experiments, except that in Section 5.5.4, used the query illustrated in Figure 5.2, while adding Filters and applications after the Joins. To easily detect data loss, however, we used stream sources that periodically generated tuples. We also made the Filters always pass their inputs. In detail, each server ran two kinds of stream sources, one that reported the server's CPU load and the other that reported the latencies of connections to other servers. Such input streams were first merged at Unions, one for each input type, and the subsequent Joins correlated load and latency readings. For the experiments in Sections 5.5.2 and 5.5.4, stream sources generated input tuples every half a second and 1 millisecond, respectively. In the other experiments, input tuples were generated every 10 seconds. Joins in Sections 5.5.2 and 5.5.4 used time windows of half a second and 100 milliseconds, respectively. In other cases, the window size was set to 10 seconds.

Given the query above, we deployed replicas. In Sections 5.5.2 and 5.5.4, we manually did the task. In other cases, the coordinator first obtained statistics on network delays between servers and the data rates of streams through a test run. Then, it deployed operators in a non-replicated fashion, using the spring relaxation algorithm [106]. This was to start with the best operator placement that has the lowest network cost. After this, the coordinator replicated operators and streams, according to the chosen replication method.

5.5.2 Comparison of Techniques for Reliable Stream Processing

In this experiment, we compare our replication technique with previous high-availability techniques. For this, we manually placed $\cup_{1,1}$ at WISC, $\cup_{2,1}$ at Purdue, and $\bowtie_{3,1}$ at OSU (see Figure 5.5).

In Figure 5.6, the curve labeled "no replication" represents how the latency, without replication, varied over time at the output of $\bowtie_{3,1}$. The latency of a tuple was defined as the difference between the wall-clock time and the timestamp of the tuple (i.e., the time when the tuple would be produced in the ideal non-replicated scenario, equivalently the earliest time when the tuple could be generated). In this experiment, we crashed the stream processing engine at WISC at time 60. After that, $\bowtie_{3,1}$ did not produce any output because it no longer received tuples from $\cup_{1,1}$. In contrast, other curves show that reliability techniques indeed provide protection against failures. The curve labeled

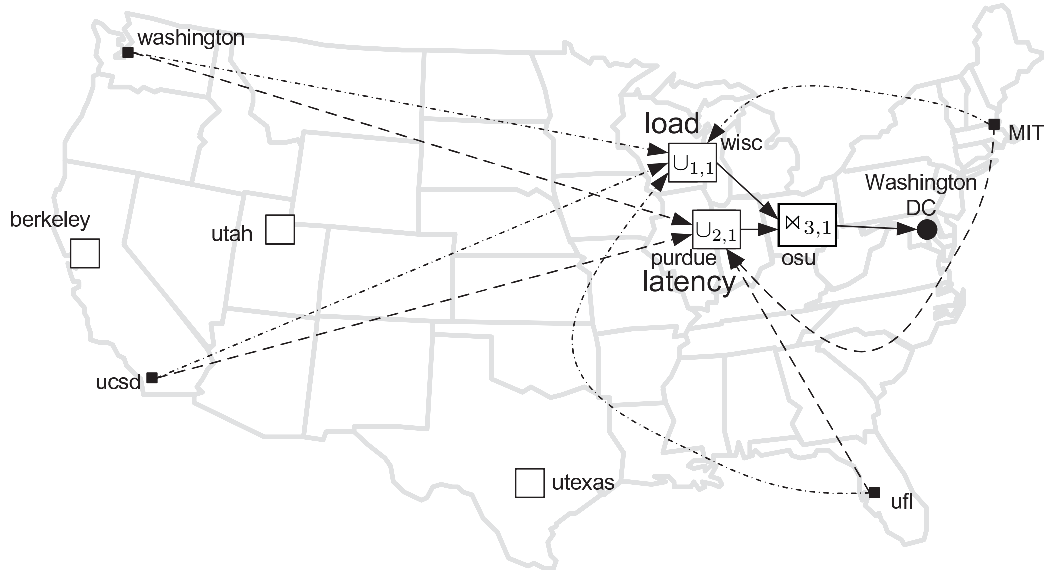


Figure 5.5: Experimental Setup

"replication" shows how our replication technique behaved when we added replicas $\cup_{1,2}$ and $\cup_{2,2}$ at Purdue and WISC, respectively. In this case, despite the failure at WISC, the processing continued relying on $\cup_{1,2}$ and $\cup_{2,1}$ at Purdue. After the failure, however, the latency increased because $\bowtie_{3,1}$ no longer benefited from replication. In Figure 5.6, "synchronization" shows the performance of a previous technique that always enforces an identical execution between primaries and backups. The details of this approach is presented in Section 3.5 of this dissertation and [68]. In this technique, each primary sends extra information, called *determinants*, to the backup so that the backup can mimic the processing of the primary. For Unions, we used the inter-arrival order of input tuples to generate determinants. This method introduces extra delays because primaries must hold output tuples until backups acknowledge the recept of determinants. Checkpoint-based techniques [68, 69] also show similar behavior because primaries hold outputs until one round of checkpoint finishes. After the failure, the latency dropped since $\cup_{1,2}$ and $\cup_{2,1}$ at Purdue no more had synchronization partners.

Finally, "deterministic" shows the variation of latency under a method that runs peer replicas (e.g., $\cup_{1,1}$ and $\cup_{1,2}$) identically by feeding the replicas in the same order [22]. As described in Section 5.3.3, sorting streams introduces extra delays because tuples are held until a relevant punctuation arrives. We can reduce extra delays by more frequently producing punctuations. The pace of $\bowtie_{3,1}$, however, is eventually determined by the slowest input flow.

In summary, the figure shows that our technique improves both *performance* and *reliability* because it always benefits from the best among multiple replicated data flows. On the other hand, previous approaches degrade performance because they identically run replicas at distant servers, thereby introducing extra delays. In previous approaches, the failure of a server also disrupts the processing until the downstream servers notice it and switch to another upstream server (observe

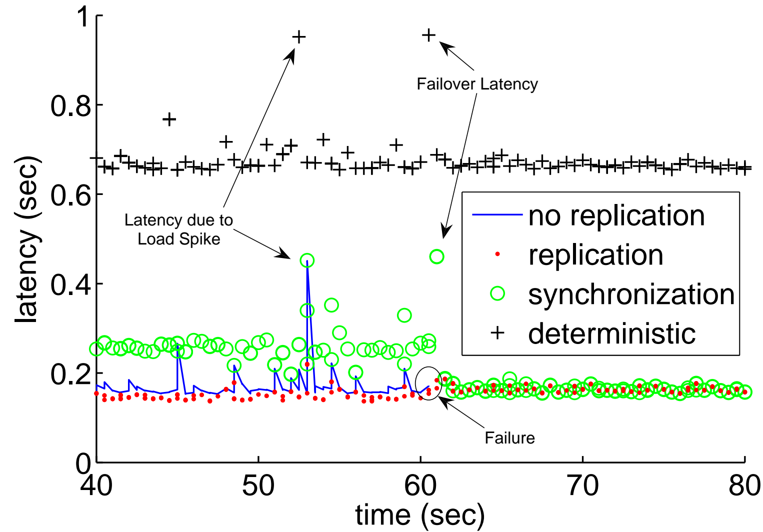


Figure 5.6: Comparison of Reliability Techniques

operators	no replication	degree of replication			
		1	2	3	4
Union	0.7	1.1 (1.43x)	1.8 (2.57x)	2.7 (3.86x)	3.5 (5.00x)
Filter	0.8	1.2 (1.38x)	1.9 (2.37x)	2.8 (3.50x)	3.6 (4.50x)
Aggregate	2.3	2.7 (1.17x)	3.6 (1.57x)	4.4 (1.92x)	5.2 (2.26x)
Join	9.5	10.3 (1.08x)	10.8 (1.14x)	11.6 (1.22x)	12.3 (1.29x)

Table 5.1: CPU cost of a Replica (% CPU cycles)

the failover latencies in Figure 5.6).

5.5.3 Impact of Replication on Latency

Figure 5.7 shows how the end-to-end latency at an application varies over time depending on the degree of replication. In this experiment, each server ran three stream processing engines and used each of them for a different degree of replication. For example, the curve labeled "$k_{\max}$=3" shows the latency results obtained from the engines that collectively deployed 3 replicas for each operator as described in Algorithm 7. The figure shows that the average as well as the variance of latency decrease as we deploy more replicas. This is because each operator in the system is provided more replicate input flows and thus can benefit from better ones.

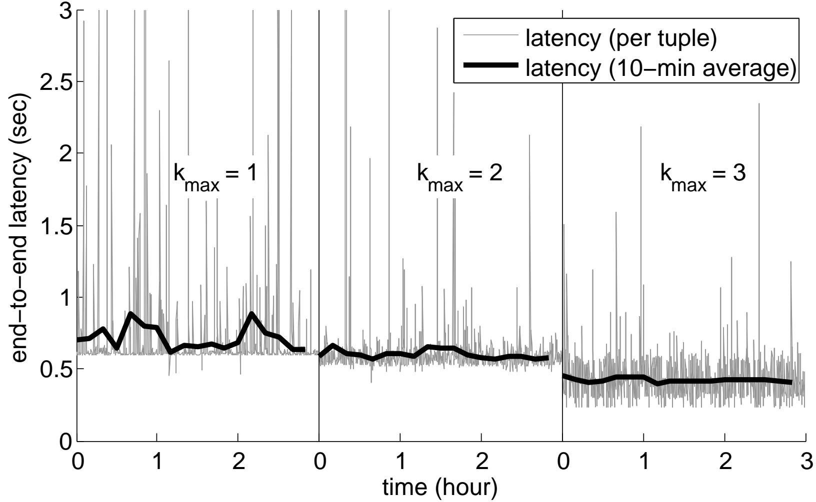


Figure 5.7: Impact of Replication on Latency

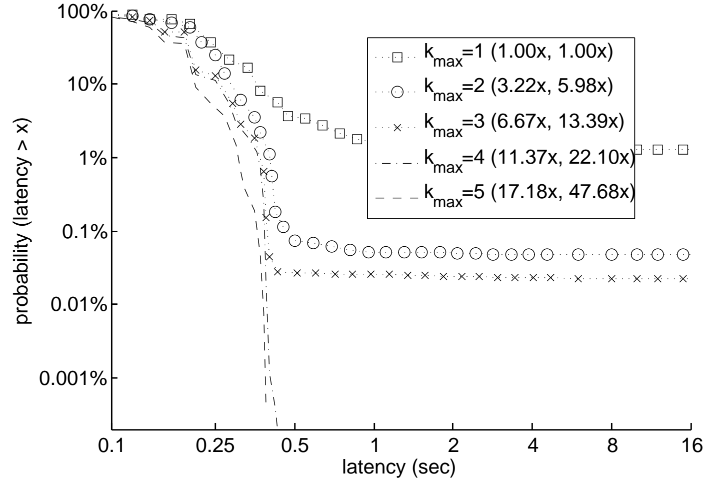
5.5.4 CPU cost for Replications

Using our prototype, we also measured the CPU cost for replication, using AMD Sempron 2800+ CPUs. In this experiment, we fed 1K tuples/sec to each input of the operators. The operators were instrumented to output 1K tuples/sec as well. Specifically, the Filter always passed input tuples after evaluating the predicate and the Aggregate computed the count of input tuples using a window of 10ms that slid every 1 ms. The Join matched each input tuple with 100 input tuples on the other input, but produced only one output tuple every 1 ms as the result of predicate evaluation.

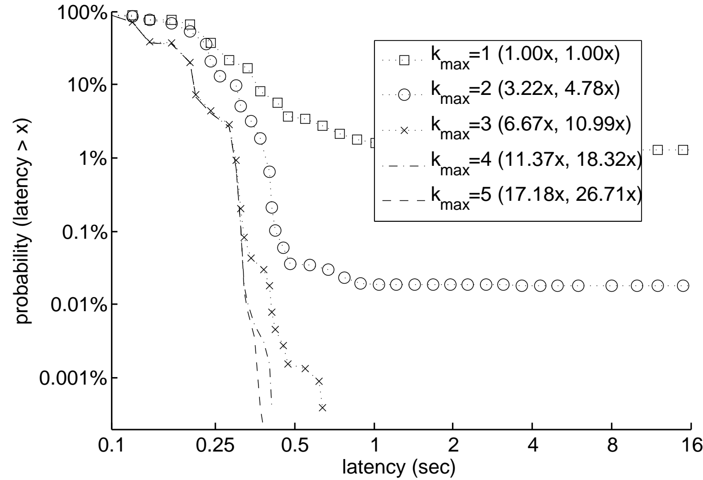
For each operator type, we first fixed the degree of replication (k_{max}) to 4 and gradually added replicas until approximately half the cycles of a CPU were used. After this, we decreased the degree of replication from 4 to 1, while finding the per-replica CPU cost by dividing the CPU usage by the number of replicas. We also measured the CPU cost for the non-replicated case. Table 5.1 summaries the results. The "no replication" column shows that each operator has a different processing cost. Each row of the table shows that the CPU cost per replica increases as we add more input/output stream replicas. This increase in the CPU cost corresponds to the overheads of removing duplicates as well as sending and receiving tuples via stream replicas. Finally, the table shows that the per-replica CPU cost increases at a different pace for each operator. The Join operator has the lowest growth rate because the processing cost itself dominates the cost of using more stream replicas.

5.5.5 Impact of Replica Deployment

Figure 5.8 shows the impact of replica deployment using results from our simulator. For the results, we ran the simulator for a month in simulation time. Then, we plotted the latency distribution for all the output tuples that appeared at 10 different applications. In both figures, k_{max} represents the degree of replication. The ratios within parentheses represent the relative bandwidth usage and network cost, respectively, compared to those in the non-replicated case (i.e., k_{max}=1). In



(a) random



(b) min-cost

Figure 5.8: Impact of Replica Deployment

Figure 5.8(a), $k_{\max} = 4$ $(11.37\mathtt{x}, 22.10\mathtt{x})$ illustrates the case where we replicated each operator at 4 random places and, as a result, consumed 11.37 times higher bandwidth and incurred 22.10 times higher network cost.

In all of the cases, the relative bandwidth usage was less than $k_{\max}^2$. This is because there were (1) $k_{\max}^2$ stream replicas between $k_{\max}$ upstream operator replicas and $k_{\max}$ downstream operator replicas and (2) $k_{\max}$ stream replicas between a stream source and $k_{\max}$ downstream replicas, and between $k_{\max}$ upstream replicas and an application.

As defined in Section 5.4.1, the network cost is a bandwidth-delay product. Because we started from an optimal, non-replicated deployment, stream replicas added later had longer delays than the previous ones. For this reason, the network cost ratio is usually higher than the bandwidth ratio.

Figure 5.8 shows that deploying replicas using Algorithm 7 (labeled “min-cost”) provides a better

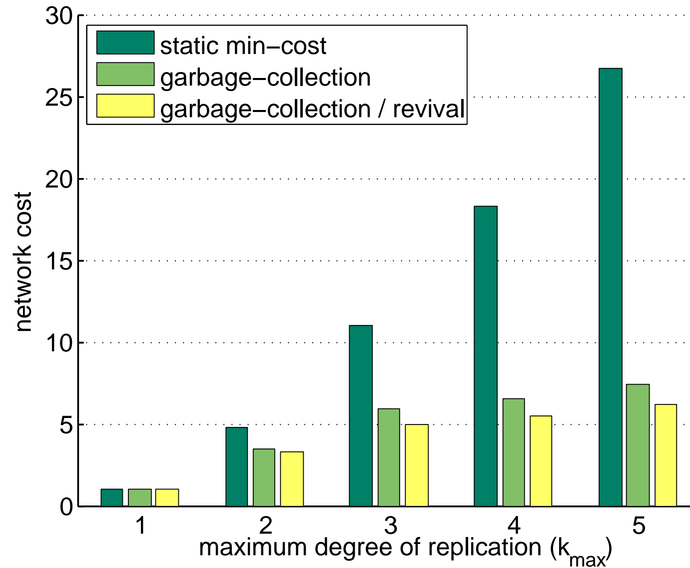


Figure 5.9: Impact of Garbage Collection

latency guarantee than deploying replicas at random servers (labeled "random"). In the random deployment case, there were latencies beyond 16 seconds even though a 13.39 times higher network cost was paid. In the min-cost case, the latency was always smaller than 1 second for a smaller network cost (10.99x). This is because our deployment algorithm finds, among the servers that are likely to achieve the desired availability level, those that minimally increase the network cost.

5.5.6 Impact of Garbage Collection

As described in Section 5.4.2, keeping all stream replicas may waste resources without any gain in performance and reliability. In this experiment, for each $k_{\max}$ value, we first achieved the latency guarantee in Figure 5.8(b) by running Algorithm 7. In Figure 5.9, the first bar for each $k_{\max}$ value represents the network cost in this case (labeled "static min-cost"). Then, we tested how much network cost could be saved through garbage collection without degrading the latency guarantee. The second bar for each $k_{\max}$ value represents the network cost after garbage collection. Thus, the difference between the first two bars in each case represents the network cost of the streams that did not contribute to performance and reliability. Figure 5.9 also shows that, as the degree of replication increases, only a smaller portion of stream replicas are useful. We also tested the case where replicas react to changes in system conditions as described in 5.4.3 (labeled "garbage-collection / revival"). In this case, we can more aggressively garbage-collect replicas because those garbage-collected can be reused whenever necessary.

5.6 Summary

Today's applications often require service level agreements (SLAs) on the latency of results. If the network is unable to deliver results within these SLAs, we can consider this as a failure. In this chapter, we introduced a replication-based approach that can cope with both fail-stop failures and unacceptable latencies. The central notion behind the approach is to replicate operators and let them flow outputs downstream in parallel. In this way, any replica in the system can use whichever data arrives first from upstream replicas. Therefore, the system naturally achieves low-latency processing as well as robustness against server and network problems.

For this replication framework, we also devised processing primitives that, despite the complications introduced by replication, can provide the same semantic guarantee as those devised for non-replicated scenarios. In particular, these primitives allow running replicas differently to avoid the overhead of previous approaches. They also merge stream replicas into a non-duplicate stream. If such a stream feeds order-sensitive operators/applications, our primitives can sort the stream to restore the order that would appear in the non-replicated scenario.

Another contribution made in this work is a strategy for managing replicas at distant servers. Our strategy improves performance and availability in a manner that efficiently uses network resources and copes with changes in system conditions.

Finally, we presented results obtained from our prototype as well as a detailed simulator. These results demonstrate how our approach can overcome the limitations of previous approaches in the Internet. They also show that, when resources allow, our replication technique is both feasible and correct.

Chapter 6

Related Work

In this chapter, we survey previous research efforts that are related to highly-available stream processing. We first summarize high-availability techniques developed for traditional database management systems (Section 6.1) and distributed systems (Section 6.2). Next, we present various problems addressed in the context of stream processing, including deploying operators at diverse geographic locations and achieving high availability (Section 6.3).

6.1 High Availability in Database Management Systems

A computer system, like any other mechanical or electrical device, is subject to failure. There are a variety of causes of such failures, including disk crash, power outage, and software errors. Therefore, an integral part of a database management system is a recovery scheme that can detect failures and lead the system to a state that would have existed if the failure did not occur. Research on failure recovery in database management systems can be dated back to early 70's [49, 30]. In this section, we review traditional disk-based rollback recovery techniques (Section 6.1.1), the process-pair approach (Section 6.1.2), persistent queues (Section 6.1.4), high-availability techniques for distributed database systems (Section 6.1.3) and workflow systems (Section 6.1.5).

6.1.1 Disk-Based Rollback Recovery

In addition to hardware failures and software bugs, a database management system can experience *transaction failures* for various reasons, including violation of integrity constraints and deadlocks. If such a failure occurs, the state of the database may no longer be consistent. Therefore, we need recovery schemes that can ensure the *atomicity* of transactions (i.e., either all operations or none of each transaction are reflected in the database). Techniques to ensure such atomicity are generally classified into the following approaches:

- *Shadow paging* uses two page tables, the shadow page table to keep the state as of the time when a transaction begins and the current page table to maintain the state that the transaction

manipulates [93]. The former (the latter) becomes the new page table if the transaction aborts (commits).

- In *log-based schemes* [61, 97], all updates are logged on stable storage (e.g., RAID [40]) so that, if a failure occurs, the system can restore the state of the system to a previous consistent state by undoing these logs. The system can reduce the number of log records that the system must process during recovery, by conducting *checkpoints*. A checkpoint saves a consistent state of the system on stable storage.

An advanced log-based recovery technique, called ARIES [97], has been widely used in database products, including IBM DB2 [121] and Microsoft SQL Server [45]. Recovery in Oracle [46] is described in [84].

As pointed out in Section 1.2, the aforementioned recovery techniques for database management systems are not appropriate for stream processing. The reason behind this is that they store data on disks with high overhead and the concept of database transactions is not applicable to the continuous stream processing model.

6.1.2 Process-Pair

In addition to the methods in Section 6.1.1, database systems often protect data from server failures by replicating data from a source database, called the *primary*, to a target database, called the *standby*. This model is frequently called the *process-pair* model [24, 60]. Process-pair has several variants that differ in runtime overhead and recovery speed. In the *cold-standby* variant, the primary periodically transmits a log of operations to the standby and, based on the log, the standby asynchronously performs the operations. Although logs of operations are commonly used, the primary can also checkpoint its state to reduce the amount of logs to process during recovery. In contrast to cold-standby, *hot-standby* makes the primary and standby perform all operations synchronously (i.e., perform every update before sending the result to the client). Hot-standby favors recovery speed over runtime performance. Cold-standby has the opposite characteristics.

The process-pair model is widely adopted in many existing database systems, including IBM DB2 HADR (High Availability Disaster Recovery) [73], Oracle 10g/DataGuard [111] and MS SQL 2005's Database Mirroring [104]. Oracle 10g/DataGuard is one such facility built on top of Oracle Streams [103]. DataGuard supports three recovery modes: maximum protection (MPR), maximum availability (MAV), and maximum performance (MPE). MPR synchronously applies the same update to multiple machines as part of the same transaction, providing precise recovery. MPE asynchronously transmits redo logs to the standby, providing gap recovery only. MAV switches between MPR and MPE based on the accessibility of the standby. In Chapter 3, we developed variants of the process-pair model, namely passive standby, active standby, and upstream backup, while taking into account the requirements of stream processing applications. These techniques can provide precise recovery as well as a new less rigorous recovery type called no loss recovery. Compared to precise recovery, no loss recovery can be achieved with a lower runtime cost and faster recovery speed. The

three developed recovery techniques also have different characteristics in terms of recovery speed and resource usage.

6.1.3 High-Availability Techniques for Distributed Database Systems

Recovery techniques in Section 6.1.1 also have been extended for distributed database systems. A distributed database system consists of a collection of sites, each of which maintains a local database system and also can participate in the execution of a global transaction (i.e., a transaction that accesses data in several sites) [59, 140]. To ensure atomicity of global transactions despite server and network failures, distributed database systems usually use the *two-phase commit protocol* [91, 58, 98]. This protocol is invoked when all the sites that have been related to a global transaction inform the completion of the transaction. The two phases of the protocol are the voting phase, in which the transaction coordinator checks whether or not each of the related sites is ready to commit, and the decision phase, in which the coordinator informs the sites of its decision (commit or abort) on the global transaction. The two-phase commit protocol may lead to situations where the fate of a transaction cannot be determined until a filed site recovers. The *three-phase commit protocol* reduces the probability of blocking [124]

Although the process-pair approach in Section 6.1.2 can protect a system from server failures, a distributed database in a wide area setting needs to deal with network failures and a large number of simultaneous server failures. For this, techniques that replicate databases at different remote locations have been proposed. These *replication* techniques can be classified into two categories: *eager replication* and *lazy replication*. Eager replication keeps all replicas exactly synchronized by updating all the replicas as part of one atomic transaction [56, 57]. This approach always leads to serializable execution and thus prevents concurrency anomalies. Eager replication, however, increases transaction response times because it adds extra updates and messages to each transaction. In contrast to eager replication, lazy replication trades off availability against consistency. In lazy replication, only one replica is updated by the originating transaction and updates to other replicas propagate asynchronously, typically as a separate transaction for each node. If replicas have inconsistent states, the system reconciles the states [81, 142, 139]. These replication approaches are prohibitive in Internet-scale stream processing because the notion of database transactions cannot be applied and the state of an operator replica can change too rapidly due to high input rates. For this reason, our replication approach in Chapter 5 concurrently executes operator replicas instead of propagating updates between operator replicas.

6.1.4 Persistent Queues

Bernstein et al. developed a fault-tolerance mechanism for transaction requests between clients and servers [26]. Making use of *persistent queues*, their approach ensures that each server processes each request exactly once and that a client processes each reply at least once. This work, however, is not appropriate in stream processing because it relies on transactional semantics and stores messages on

stable storage before they are get processed. Our entry points in Chapter 5 are similar to persistent queues in that they preserve input tuples from stream sources. Unlike persistent queues, entry points preserve tuples in their local memory as well as the memory of other entry points until the tuples are processed by downstream operators.

6.1.5 High Availability in Workflow Systems

High availability has been also studied in the context of workflow management systems (WFMSs). In a workflow system, such as IBM WebSphere MQ [65, 74], data travels through several execution steps that collectively accomplish a business goal [5, 67]. These workflow systems usually mask server failures by using standby machines. Many solutions commit the results of each execution step while storing messages on stable storage that both the primary and standby can access. To achieve fault tolerance in a scalable fashion, some workflow systems log messages on processing nodes rather than a central storage [6]. A variation of the process-pairs approach is used in the Exotica workflow system [80]. Instead of checkpointing process states, Exotica logs messages between workflow components. This approach is similar to upstream backup in that the system state can be recovered by reprocessing the logs. In general, high-availability techniques for workflow systems are not directly applicable to stream processing because they rely on transactional semantics which is not applicable to the stream processing context.

6.2 High-Availability Techniques in Distributed Systems

A large number of techniques have been developed to add reliability and high availability to distributed systems. In this chapter, we summarize two representative research fields in the area, namely rollback recovery and group communication.

6.2.1 Rollback Recovery

Techniques in this category treat a distributed system as a collection of application processes that communicate through a network. These techniques periodically checkpoint the states of these applications on stable storage [52]. Upon a server failure, application processes restart from the saved states, thereby reducing the amount of lost computation. These *rollback recovery* techniques form, as the result of recovery, a *global (i.e., system-wide) consistent state* in which if a process's state reflects a message receipt, then the state of the sender reflects sending that message. The next two paragraphs summarize these techniques.

Early rollback recovery techniques achieve fault tolerance by use of checkpoints. These *checkpoint-based techniques* can be categorized as follows:

- *Uncoordinated checkpointing* allows each process to take checkpoints independently [110, 27, 145]. This approach requires, upon a failure, rolling back application processes to earlier

checkpoints until a global consistent state is formed. This *cascaded rollback* is a major disadvantage of uncoordinated checkpointing. In uncoordinated checkpointing, a process may also take *useless checkpoints* that will never be part of a global consistent state.

- In *coordinated checkpointing*, processes coordinate their checkpoints in order to save a global consistent state. Coordinated checkpointing simplifies recovery (since processes only need to restart from their most recent checkpoints) and reduces storage overhead. This approach, however, causes large latency because a global checkpoint must be enforced before any message is sent to the outer world. Techniques in this category ensure global consistency by periodically making all processes take checkpoints before sending any application message [132, 38, 85, 82, 48].

- *Communication-induced checkpointing* piggybacks protocol-specific information on each application message so that the receiver process can determine whether or not it should take a *forced checkpoint* to prevent past checkpoints from being useless [101, 146, 64]. In addition to these forced checkpoints, each process can autonomously take *local checkpoints* (e.g., when its state is small and thus incurs a small checkpoint overhead). Because this approach does not require globally coordinated checkpoints, it can scale up with a large number of processes.

Log-based rollback recovery combines both checkpointing and logging of non-deterministic events (e.g., receipt of messages and hardware interrupts). Conceptually, log-based rollback recovery relies on *piecewise determinism*, an assumption that all nondeterministic events that a process executes can be identified and, by replaying these events during recovery in the exact original order, a process can recreate its pre-failure state [129]. Because this approach can recover a system up to the point of failure (beyond the most recent set of consistent checkpoints), it is particularly attractive for applications that frequently interact with the outside world. Log-based rollback recovery has the following three variants:

- *Pessimistic logging* saves each nondeterministic event on stable storage before the effects of the event can be seen by other processes or the outside world [77]. Due to this restriction, pessimistic logging degrades failure-free performance. To reduce this penalty, some protocols rely on special hardware [31] or log each message on the volatile memory of the sender [77].

- *Optimistic logging* asynchronously stores nondeterministic events on stable storage (i.e., first keeps these events in a volatile memory and then periodically flushes them to stable storage). Upon a failure, this approach thus can cause *orphan processes* (processes whose states depend on nondeterministic events that are not yet logged). Optimistic logging techniques roll back these orphan processes until no such processes remain. For this, they track *causal dependencies* between processes during failure-free execution [129, 76, 123, 125]. Causal dependencies are also used to log events before their effects are seen by the outside world.

- *Causal logging* has the failure-free performance advantages of optimistic logging while retaining most of the recovery advantages of pessimistic logging. Causal logging ensures that each

nondeterministic event that causally affects a process is always stored on either stable storage or the volatile log of the process [53, 7]. To achieve this, processes piggyback causal information on application messages and update their logs based on these messages.

The rollback recovery techniques above are related to our passive standby technique in Chapter 3 and the parallel recovery technique in Chapter 4. In principle, our techniques build a virtual backup storage on the memory of many servers, save logs on the sender processes, and conduct asynchronous checkpoints. In stream processing, data streams that connect operators represent the dependencies between operators. Therefore, without any complicated mechanism, groups of operators that are independent of each other can be easily found. Taking advantage of this property, our technique in Chapter 4 conducts fine-grained checkpointing, thereby significantly reducing the duration that a checkpoint blocks regular processing. Furthermore, the technique constructs a distributed backup framework that can parallelize recovery. This technique also distributes backups and schedules checkpoints in a manner that maximizes the recovery speed.

6.2.2 Reliable Group Communication

In distributed systems, it is often necessary that communication within a group of remote processes be reliable. For example, an application may require that every message is delivered to either all members of a process group or to none at all. In addition, it is also generally required that all messages are delivered in the same order to all processes. These two conditions above collectively define the *atomic multicast* guarantee. Atomic multicast is particularly useful in distributed database systems because it enables all the sites update their local databases in an identical way despite site crashes and network failures. The Isis [28], Horus, Ensemble [29], and Spread [10] toolkits provide generic group membership and atomic multicast functionalities. The Paxos algorithm achieves *distributed consensus* in a fault-tolerant manner [89, 90]. This capability is beneficial to failover-based recovery techniques because it allows a group of replicas to easily agree upon the health of each member. Our replication technique in Chapter 5 does not require group communication protocols. This is because it allows each replica to process whichever data arrives first from multiple upstream replicas. This technique protects the system from failures even without detecting failures and switching between replicas.

6.3 Stream Processing

Research on stream processing has begun to address the limitations of traditional database management systems in meeting the requirements of stream processing applications. This research has centered on investigating new techniques to provide low-latency processing of high-rate data streams. In this section, we first present a variety of stream processing systems developed and discuss issues that these systems aim to address (Section 6.3.1). Next, we summarize strategies for placing operators over diverse geographic locations (Section 6.3.2). Finally, we contrast our techniques for

highly-available stream processing with others (Section 6.3.3).

6.3.1 Stream Processing Systems

Tapestry [138] was one of the first systems that introduce the notion of stateful continuous queries. Tapestry, however, did not yet adopt the "on-the-fly" processing model in the sense that it still stores and indexes data in databases before running continuous queries. Although Tapestry has an advantage that it requires only small changes to a regular database management system, it does not scale well with the rate of the input streams and the number of continuous queries. In Tapestry, users submit continuous queries that identify some documents of interest. The users are then notified when such documents are inserted into the repository.

Tribeca [130] was an early stream processing system for network monitoring applications. Tribeca queries are expressed using a specific dataflow-oriented query language. Because queries can have only one input stream, Tribeca cannot support Join operations. Tribeca provides windowed Aggregates. It also supports other operators that split and merge streams. These operators are similar to Group-By and Union, respectively.

NiagaraCQ [39] focused on scalable continuous query processing over XML documents. NiagaraCQ aimed to support millions of simultaneous queries by dynamically grouping them according to their structural similarities. NiagaraCQ merges new queries into existing groups as they arrive. It can also reform groups in a manner that shares computation. In this way, NiagaraCQ can reduce resource usage and improve throughput. As part of the NiagaraCQ project, Tucket et al. proposed punctuations. Punctuations are special elements of a data stream that specify the end of a subset of tuples in the stream [141]. Puctuations are originally devised to adapt traditional blocking operators (e.g., Sort, Aggregate) and unbounded stateful operators (e.g., Join) to the context of stream processing. Our replication framework in Chapter 5 uses punctuations of its own style to eliminate duplicate tuples and ensure replica consistency.

TelegraphCQ [36] is one of the earliest systems that continuously process incoming data without first storing it. To adapt to varying conditions, groups of query modules can be connected together with an Eddy [16], which intercepts tuples as they flow between modules and makes routing decisions. TelegraphCQ also views stream processing as a Join operation between a stream of data and a stream of queries [37]. When a new query arrives, its predicate is used to probe the data. An incoming tuple can also be joined with relevant queries.

The STREAM project explored many aspects of stream processing, including a new data model and query language for streams [11], operator scheduling [17], resource management [19, 127], approximation [14], and distributed operation [102, 20]. The STREAM prototype supports continuous queries over both streams and stored relations [12]. To achieve this, STREAM supports three types of operators: stream-to-relation, relation-to-stream, relation-to-relation.

The Aurora project [2] also proposed a new data model and processing architecture for stream processing. In Aurora, users express queries directly as boxes-and-arrows diagrams where boxes and arrows represent operators and streams, respectively. The Aurora project addressed the problems of

efficiently scheduling operators [33] and shedding load in overload situations [134]. Borealis [1, 41] is a distributed stream processing system that builds upon Aurora. It borrowed communication capabilities from Medusa [23]. Borealis can generate revision tuples that correct the value of earlier tuples [113] and support time travel where a running query can go back in time and reprocess some data. Other problems such as operator placement [3], distributed load management [149, 148], and fault-tolerant distributed stream processing [22, 68, 69, 70, 71] are also studied in the context of Borealis.

Gigascope [47] is a stream processing system specially designed to support network monitoring applications. This system provides a query definition language, called GSQL, as well as a suite of user-defined operators. These operators enable Gigascope to leverage custom software modules that already exist. In Gigascope, queries are compiled into C and C++ modules and then linked to the runtime system. Gigascope performs data reduction (e.g., aggregation, selection) at data sources before sending the data to downstream components. Gigascope aims at efficient stream processing through sampling [44, 78] and real-world deployments [47].

In addition to the prototypes mentioned above, several commercial stream processing products have been developed. These systems include StreamBase [128], IBM System S [75, 9], Coral8 [43], Amalgamated Insight [8].

6.3.2 Operator Placement

A crucial issue in distributed stream processing is to determine the location to run each operator. The placement of operators affects both performance (in terms of result latencies) and network usage.

Srivastava et al. studied the problem of placing operators in a sensor network [126]. This work assumes that data is first collected by low-capability devices and then processed through a hierarchy of nodes with increasing computation power and network bandwidth. In order to reduce network bandwidth consumption, this system pushes operators to upstream nodes. This work defines the execution cost of an operator based on the processing load of the operator and the processing capability of the node that runs the operator. An operator has a larger execution cost as it is placed on a node with less processing power. This work strive to minimize the total cost of a query execution plan. This cost combines the execution cost of the query and the transmission costs of the related network links.

Ahmad and Uğur Çetintemel proposed a method to deploy operators in a wide area environment [3]. This work models the cost of a network link based on the amount of data transmitted as well as the transmission latencies. This work then determines the locations of the operators with the optimization goal of minimizing the total network cost.

Pietzuch et al. also studied the problem of operator placement in a wide area setting [106]. They developed a stream-based overlay network, called SBON, that maintains a virtual cost space where the distance between two nodes represents the network overhead of routing data between these nodes. SBON determines the placement of operators first in the virtual space using a spring

relaxation algorithm. This algorithm assumes that stream sources and client applications are fixed at specific locations whereas operators can move freely. By using network latency as the spring extension factor and data rate as the spring constant, the algorithm finds the locations in the virtual space that minimize the energy of the spring system (i.e., minimize the network usage). After this, the algorithm maps the locations in the virtual space to physical nodes and then places operators on those nodes.

Similar to the operator placement approaches above, our replica deployment approach in Chapter 5 also puts replicas at locations that minimize network usage. Our approach, however, considers the relationship between different locations to reduce the risk that replicas fall into the same network partition (while simultaneously isolated from the rest of the network). Our approach also can dynamically discard the least useful replicas to save network resources and can add more replicas and to cope with varying system conditions.

6.3.3 High Availability in Stream Processing

Most of the high-availability techniques developed for stream processing are based on failover [118, 68, 22, 69]. In these techniques, if a server fails, a group of pre-dedicated backups take over the failed execution.

Three general failover-based approaches is presented in [68] and Chapter 3 of this dissertation: in *passive standby*, each primary periodically checkpoints (i.e., copies only the change that occurred in its state since the last checkpoint) onto its backup; in *active standby*, each backup also receives and processes input data in parallel with its primary; in *upstream backup*, each primary logs its output data so that if a downstream primary fails, a backup can rebuild the lost state from scratch by reprocessing the logged data.

The passive standby approach was extended for local area clusters in [69] and Chapter 4 of this dissertation. The reason behind this was that passive standby can withstand *high load* situations while gracefully degrading the recovery speed. The developed technique partitions the query at each server into smaller pieces and backs them up onto different servers. Based on these distributed backups, the technique can parallelize recovery, thereby significantly reducing the recovery time.

The active standby approach guarantees faster recovery than passive standby in *low load* situations where all the backups can use the same amount of resources as the primaries. Shah et al. developed a technique, called Flux, that falls into the active standby approach [118]. Flux accomplishes loss-free and duplication-free failure/recovery semantics by use of sequences numbers assigned to tuples. Similar to our active standby technique in Chapter 3, Flux allows loose coupling between replicas where one replica can go ahead of the other and vice-versa, resulting in more opportunities to overlap their execution. In contrast our active standby technique, however, Flux cannot support divergent and convergent-capable operators. Balazinska et al. studied a variant of the active standby approach that can deal with network failures and partitions [22]. The developed solution yields tentative results when inputs are not available by a predefined time bound. If the deferred inputs become available, it starts sending correct revisions. This solution, however, increases output

latencies during failure-free periods because it forces operator replicas to run identically by sorting input streams to them. In contrast, our active standby technique in Chapter 3 and replication technique in Chapter 5 allow operator replicas to run differently while keeping them consistent with each other.

The third failover-based approach, called upstream backup, incurs very low overhead during failure-free periods because the backups remain idle. However, it may take a long time to recover large state queries. For example, recovering an Aggregate with a window size of 10 minutes requires re-processing 10 minute worth of tuples. This approach is thus beneficial when queries have small states, resources are scarce, and failures are very rare.

Recently, Murty and Welsh presented a high-level vision of a dependable architecture for Internet-scale sensing [100]. They proposed a replication technique that allows replicas to arbitrarily diverge and then reconciles results from such replicas by finding a representative value (such as the median). In contrast, our approach in [70, 71] and Chapter 5 of this dissertation makes replicas produce the same tuples, possibly in different orders. Based on this property, our approach guarantees that applications always receive the same results as in the non-replicated, failure-free case.

Chapter 7

Conclusion

In this dissertation, we presented our various techniques for highly-available stream processing. These techniques were motivated by the following observations:

- A failure in stream processing can have a fatal impact because it blocks downstream data flows and may result in losing data essential for processing.
- The larger the system, the higher the risk of having a faulty component.

The main challenge throughout our work was to develop resource-efficient and scalable fault-tolerance techniques that can consistently realize low-latency delivery of correct results. In this chapter, we summarize our contributions to date (Sections 7.1, 7.2, and 7.3) and describe future research directions (Section 7.4).

7.1 Basic Approaches for Highly-Available Stream Processing

In Chapter 3, we devised three approaches where a failed server can be taken over quickly by another server. These approaches commonly deploy, for each operator, k replicas on independent servers to mask up to $(k-1)$ simultaneous fail-stop server failures. They, however, differ in how they coordinate replicas to prepare for failures. In detail,

- Active standby executes all peer replicas in parallel, while allowing one of them to feed downstream replicas.
- In passive standby, only one of multiple peer replicas runs and periodically checkpoints its state onto the other passive replicas. Although the state of a replica includes any data that the replica maintains, each checkpoint copies only the difference between the active replica's current state and the state at the time of the previous checkpoint.

- In upstream backup, active replicas log output data so that, if a downstream replica fails, an empty replica can use the logged data to rebuild the latest state of the failed one.

For each of these approaches, we also devised a specific solution that guarantees precise recovery (i.e., the output with failure is always identical to that without failure) and other solutions that provide less rigorous recovery guarantees (e.g., no data is lost, but the output can differ from that without failure) at lower costs. Using a detailed simulator and our Aurora/Borealis prototype, we showed that these recovery techniques have very different performance characteristics and also identified the scenarios where each of them is most advantageous. The results are as follows:

- Active standby provides the fastest recovery at the highest cost. Therefore, it is desirable when failures are relatively frequent, applications require very low latency at all times, and the system has a large amount of idle resources.
- Upstream backup incurs the lowest overhead, but are not suitable for large-state queries. For this reason, it is advantageous when failures are rare, resources are very scarce, and queries have small states.
- Passive standby can strike a balance between performance and recovery speed by adjusting the checkpoint interval. Thus, it is beneficial when failures are relatively rare, applications can tolerate moderate latencies, and the system has slightly more resources than necessary for regular processing.

7.2 Highly-Available Stream Processing in Server Clusters

In Chapter 4, we developed a technique that realizes highly-available stream processing in commodity server clusters. The reason behind this was that such server clusters increasingly gained popularity due to their favorable price/performance ratio. For these environments, we adopted the passive standby approach because it can flexibly trade off between resource usage and recovery speed – less frequent checkpoints save resources, but, at the same time, the older the checkpoint the longer the recovery time. Other alternatives do not have such flexibility – active standby always requires k times more resources and upstream backup cannot accelerate recovery even if idle resources are available.

Under passive standby, our work focused on maximizing the recovery speed and minimizing disruption by checkpointing using idle CPU cycles. The main idea was to group servers into logical clusters and back up each server using others in the same cluster. Because individual operators on a server are backed up on different servers, they can be recovered in parallel. In this approach, a server failure is not fully recovered until all the other servers complete recovery. Therefore, each server determines the backups for its operators in a manner that balances the recovery load over the other servers. Whenever a server is idle, it begins the checkpoint that will most reduce the recovery load, relative to the CPU cost of checkpoint. This gain/cost ratio favors (i.e., causes to checkpoint more frequently) small-state operators with high processing load.

Through simulations and experiments on a server cluster at Brown University, we demonstrated the effectiveness of our backup assignment and checkpoint scheduling strategies. For these experiments, we used various workloads generated from real packet traces. We learned that:

- The distribution of recovery load over multiple servers significantly affects the recovery speed. If a server is assigned too much recovery load, it cannot as quickly complete its recovery as others. In this case, the overall recovery is slow, because it cannot take the full advantage of parallelism during recovery. We observed that, for this reason, randomly determining backup servers leads to poor recovery performance. On the other hand, our backup assignment strategy effectively improves the recovery speed.

- Our min-max checkpoint scheduling can accelerate recovery by a factor of two compared to round-robin for a real workload. This is because min-max scheduling finds the most beneficial checkpoint each time. Such prioritization tends to more frequently choose operators with high processing load and low checkpoint cost. We also found that the difference in recovery speed grows as operators have more different characteristics.

- Checkpointing a subset of operators each time suspends the regular processing for a much shorter duration than checkpointing all of the operators that a server runs. In this case, the result latencies are kept at a lower level.

- The recovery time is very sensitive to the input data rate. As an operator processes more input data, in general, both the state size and processing load of the operator increase. In this case, each checkpoint needs to transfer more data and less CPU cycles are available for checkpointing. Consequently, the checkpoint period as well as the recovery time can increase super-linearly with the input rate.

7.3 Fast and Highly-Available Stream Processing over the Internet

In Chapter 5, we discussed a replication-based approach for both fast and reliable Internet-scale stream processing. The Internet environment allows us to monitor various events occurring around the world and to make smart decisions in near real time. In this environment, however, previous techniques have limitations because a replica can receive data from only one of many upstream replicas. If the upstream replica fails (or its connections stall or fail), the processing gets delayed until downstream replicas notice the problem and acquire new input connections from another functioning upstream replica. To avoid such disruption, the developed approach creates multiple operator replicas at different geographic locations and lets them send outputs to each downstream replica. This replication-based approach can improve performance as well as availability because replicas can always use whichever data arrives first from multiple upstream replicas. To further reduce latency, replicas run without coordination, possibly processing data in different orders. Despite

this relaxation, our approach guarantees that applications always receive the same results as in the non-replicated, failure-free case.

For this replication-based framework, we also developed a strategy for managing replicas. Because it is in practice difficult to predict the future behavior of each operator, our strategy initially creates a predefined number of replicas at different locations and then gradually discards the least useful stream/operator replicas. When replicas are initially deployed, the system chooses locations that not only minimize the network cost, but also keep low the risk that peer replicas fall into the same network partition. To periodically find the least useful replicas, the system keeps track of the impact of each stream/operator replica on the subsequent processing up to client applications. If a replica observes delays in its input, it addresses the problem by creating more upstream replicas.

We demonstrated the utility of this work through experiments on PlanetLab, a worldwide network testbed. Our experimental results show that:

- Our replication framework can further reduce the average and variance of result latencies (i.e., improve performance and availability, respectively) as it uses more network resources. This sharply contrasts with previous solutions. In these solutions, adding more replicas generally does not improve performance because the regular processing relies on only one replica out of each group of peer replicas.
- Our replica deployment strategy outperforms random deployment because it selects locations that efficiently improve both performance and availability.
- Our garbage-collection technique can significantly save network resources while keeping latency low. This is because it removes the replicas that make the smallest contribution to downstream processing.

7.4 Future Directions

Our study on highly-available stream processing opens several research directions. We intend to study the following topics in the future:

- **Understanding the Nature of Failures.** Our previous research assumed that server failures are rare and independent. To prevent peer replicas from falling into the same network partition, we used a heuristic that considers network delays between servers. We expect that detailed statistics on server failures, link congestion and failures, network partitions, and their causes (e.g., power outages, human operator errors, and software bugs) would provide a new insight into the problem domain. To obtain such statistics, we plan to monitor PlanetLab using our prototype and consult other sources including data centers and failure data repositories [34].
- **Relaxed Fault-Tolerance Semantics.** Until now, we focused on approaches that ensure the processing of all of the input data despite failures. There are, however, situations where such

efforts are impractical. In sensor networks, for example, power constraints make it advantageous to send some form of summaries (e.g., samples, histograms) rather than the base sensor readings. In this context, while replication would still be the main means to improve reliability, we need to balance two conflicting goals – maximizing the quality of results and minimizing the resource usage. Interestingly, feeding replicas the same approximate data (e.g., the same samples) would not be a good idea, because feeding them different data and consolidating their outputs would yield better results (e.g., those with a higher entropy). This raises questions, such as "Would feeding each of many replicas less information be better than feeding each of fewer replicas more information?", "Should we uniformly or non-uniformly feed replicas?", "What would be the optimal routes for replicated streams?", "How can we consolidate the outputs of peer replicas?", and "How confident can we be about approximate results?"

- **Byzantine Failures.** As stream processing gains more popularity, we can envision situations where many custom-operators developed by various vendors and individual programmers are deployed in real applications. In such cases, the security vulnerabilities or bugs of these operators can cause catastrophic damage to the system. Just like the related work in other contexts [88, 143], a key idea would be to compare the outputs of either homogeneous replicas to handle non-deterministic failures, or heterogeneous replicas (e.g., those from different vendors or different versions from the same vender) to mask even deterministic failures. In stream processing, however, sophisticated optimization techniques (e.g., allowing replicas to process data in different orders) make it difficult to identify faults. To expedite data flow, optimistic approaches (i.e., those that process data as early as possible and address problems later) are preferable to pessimistic ones (i.e., those that defer processing until the input data is verified).

- **Hybrid Fault-Tolerance.** Our research to date used only one of the three fault-tolerance approaches for each specific scenario. It would be interesting to consider a technique that can apply a different approach to each operator to benefit from the unique advantages of the approaches – the recovery speed of active standby, the flexibility of passive standby, or the resource efficiency of upstream backup. To adapt to changes in system conditions, this hybrid technique should be able to seamlessly switch between different approaches.

Bibliography

[1] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Uğur Çetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag S. Maskey, Alexander Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stan Zdonik. The Design of the Borealis Stream Processing Engine. In *Proceedings of the 2nd Biennial Conference on Innovative Data Systems Research (CIDR)*, pages 277–289, 2005.

[2] Daniel J. Abadi, Don Carney, Uğur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: A New Model and Architecture for Data Stream Management. *The International Journal on Very Large Databases*, 12(2):120–139, 2003.

[3] Yanif Ahmad and Uğur Çetintemel. Network-aware Query Processing for Distributed Stream-based Applications. In *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB)*, pages 456–467, 2004.

[4] Yanif Ahmad, Uğur Çetintemel, Bradley Berg, Mark Humphrey, Jeong-Hyon Hwang, Olga Papaemmanouil, Alex Rasin, Nesime Tatbul, Ying Xing, and Stan Zdonik. Distributed Operation in the Borealis Stream Processing Engine. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, pages 882–884, 2005.

[5] Gustavo Alonso and C. Mohan. WFMS: The Next Generation of Distributed Processing Tools. *In Sushil Ja jodia and Larry Kerschberg, editors, Advanced Transaction Models and Architectures, Kluwer*, 1997.

[6] Gustavo Alonso, C. Mohan, Roger Gunthor, Divyakant Agrawal, Amr El Abbadi, and Mohan Kamath. Exotica/FMQM: A Persistent Message-based Architecture for Distributed Workflow Management. *In Proceedings of IFIP WG8.1 Working Conference on Information Systems for Decentralized Organizations*, 1995.

[7] Lorenzo Alvisi. *Understanding the Message Logging Paradigm for Masking Process Crashes.* PhD thesis, Department of Computer Science, Cornell University, 1996.

[8] Amalgamated Insight, Inc. `http://www.aminsight.com`.

[9] Lisa Amini, Navendu Jain, Anshul Sehgal, Jeremy Silber, and Olivier Verscheure. Adaptive Control of Extreme-Scale Stream Processing Systems. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, page 71, 2006.

[10] Yair Amir and Jonathan Stanton. The Spread Wide Area Group Communication System. Technical Report CNDS-98-4, Johns Hopkins University, 1998.

[11] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. Technical Report 2003-67, Stanford University, 2003.

[12] Arvind Arasu, Shivnath Babu, and Jennifer Widom. An Abstract Semantics and Concrete Language for Continuous Queries over Streams and Relations. *The International Journal on Very Large Databases*, 2005.

[13] Arvind Arasu, Mitch Cherniack, Eduardo F. Galvez, David Maier, Anurag Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. Linear Road: A Stream Data Management Benchmark. In *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB)*, pages 480–491, 2004.

[14] Arvind Arasu and Gurmeet Manku. Approximate Counts and Quantiles over Sliding Windows. In *Proceedings of the 23rd ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 286–296, 2004.

[15] Wide-Area TCP Traffic Archive. `http://ita.ee.lbl.gov/html/contrib/LBL-TCP-3.html`.

[16] Ron Avnur and Joseph M. Hellerstein. Eddies: Continuously Adaptive Query Processing. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 261–272, 2000.

[17] Brian Babcock, Shivnath Babu, Mayur Datar, and Rajeev Motwani. Chain : Operator Scheduling for Memory Minimization in Data Stream Systems. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 253–264, 2003.

[18] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and Issues in Data Stream Systems. In *Proceedings of the 21st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 1–16, 2002.

[19] Brian Babcock, Mayur Datar, and Rajeev Motwani. Load Shedding for Aggregation Queries over Data Streams. In *Proceedings of the 20th International Conference on Data Engineering (ICDE)*, pages 350–361, 2004.

[20] Brian Babcock and Chris Olston. Distributed Top-k Monitoring. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 28–39, 2003.

[21] Shivnath Babu, Lakshminarayan Subramanian, and Jennifer Widom. A Data Stream Management System for Network Traffic Management. In *Proceedings of the ACM Workshop on Network-Related Data Management (NRDM)*, 2001.

[22] Magdalena Balazinska, Hari Balakrishnan, Samuel Madden, and Michael Stonebraker. Fault-Tolerance in the Borealis Distributed Stream Processing System. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, pages 13–24, 2005.

[23] Magdalena Balazinska, Hari Balakrishnan, and Michael Stonebraker. Contract-Based Load Management in Federated Distributed Systems. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation (NSDI)*, pages 197–210, 2004.

[24] Joel Barlett, Jim Gray, and Bob Horst. Fault Tolerance in Tandem Computer Sytems. Technical Report 86.2, Tandem Computers, 1986.

[25] Jerry Baulier, Stephen Blott, Henry F. Korth, and Abraham Silberschatz. A database system for real-time event aggregation in telecommunication. In *Proceedings of the 24th International Conference on Very Large Data Bases (VLDB)*, pages 680–684, 1998.

[26] Philip A. Bernstein, Meichun Hsu, and Bruce Mann. Implementing Recoverable Requests using Queues. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, pages 112–122, 1990.

[27] Bharat K. Bhargava and Shy-Renn Lian. Independent Checkpointing and Concurrent Rollback for Recovery - An Optimistic Approach. In *Proceedings of the 7th Symposium on Reliable Distributed Systems*, pages 3–12, 1988.

[28] Kenneth Birman. ISIS: A System for Fault-Tolerant Distributed Computing. Technical Report TR86-744, Cornell University, 1986.

[29] Kenneth Birman, Robert Constable, Mark Hayden, Christopher Kreitz, Ohad Rodeh, Robbert van Renesse, and Werner Vogels. The Horus and Ensemble Projects: Accomplishments and Limitations. In *Proceedings of the DARPA Information Survivability Conference and Exposition (DISCEX)*, pages 149–161, 2000.

[30] Lawrence A. Bjork. Recovery Scenario for a DB/DC System. In *Proceedings of the ACM Annual Conference*, pages 142 – 146, 1973.

[31] Anita Borg, Wolfgang Blau, Wolfgang Graetsch, Ferdinand Herrmann, and Wolfgang Oberle. Fault Tolerance under UNIX. *ACM Transactions on Computing Systems (TOCS)*, 7(1):1–24, 1989.

[32] Don Carney, Uğur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Monitoring Streams: A New Class of Data Management Applications. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB)*, pages 215–226, 2002.

[33] Don Carney, Uğur Çetintemel, Alexander Rasin, Stan Zdonik, Mitch Cherniack, and Michael Stonebraker. Operator Scheduling in a Data Stream Manager. In *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB)*, pages 838–849, 2003.

[34] The Computer Failure Data Repository (CFDR). `http://cfdr.usenix.org`.

[35] Bharat Chandra, Mile Dahlin, Lei Gao, and Amol Nayate. End-to-end WAN Service Availability. In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and System (USITS)*, pages 97–108, 2001.

[36] Sirish Chandrasekaran, Amol Deshpande, Mike Franklin, and Joseph Hellerstein. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *Proceedings of the 1st Biennial Conference on Innovative Data Systems Research (CIDR)*, 2003.

[37] Sirish Chandrasekaran and Michael Franklin. Streaming Queries over Streaming Data. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB)*, pages 203–214, 2002.

[38] K. Mani Chandy and Leslie Lamport. Distributed Snapshots: determining Global states of Distributed Systems. *ACM Transactions on Computing Systems (TOCS)*, 3(1):63–75, 1985.

[39] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 379–390, 2000.

[40] Peter M. Chen, Edward L. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. RAID: High-Performance, Reliable Secondary Storage. *ACM Computing Surveys (CSUR)*, 26(2):145–185, 1994.

[41] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Don Carney, Uğur Çetintemel, Ying Xing, and Stan Zdonik. Scalable Distributed Stream Processing. In *Proceedings of the 1st Biennial Conference on Innovative Data Systems Research (CIDR)*, 2003.

[42] Enzo Cialini and John Macdonald. Creating Hot Snapshots and Standby Databases with IBM DB2 Universal Database$^{(TM)}$ V7.2 and EMC TimeFinder$^{(TM)}$. DB2 Information Management White Papers, 2001.

[43] Coral8, Inc. `http://www.coral8.com/`.

[44] Graham Cormode, Theodore Johnson, Flip Korn, S. Muthukrishnan, Oliver Spatscheck, and Divesh Srivastava. Holistic UDAFs at Streaming Speeds. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, pages 35–46, 2004.

[45] Microsoft Corporation. *Inside microsoft SQL server 2000.* Microsoft TechNet, 2003.

[46] Oracle Corporation. *Oracle9i Database Release 2 Product Family.* An Oracle white paper, 2003.

[47] Charles D. Cranor, Theodore Johnson, Vladislav Shkapenyuk, and Oliver Spatscheck. Gigascope: A Stream Database for Network Applications. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 647–651, 2003.

[48] Flaviu Cristian and Farnam Jahanian. A Timestamp-based Checkpointing Protocol for Long-Lived Distributed Computations. In *Proceedings of the 10th Symposium on Reliable Distributed Systems*, pages 12–20, 1991.

[49] Charles T. Davies. Recovery Semantics of a DB/DC System. In *Proceedings of the ACM Annual Conference*, pages 136–141, 1973.

[50] Amol Deshpande, Carlos Guestrin, Samuel Madden, Joseph M. Hellerstein, and Wei Hong. Model-Driven Data Acquisition in Sensor Networks. In *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB)*, pages 588–599, 2004.

[51] Holger Dreger, Anja Feldmann, Vern Paxson, and Robin Sommer. Operational Experiences with High-Volume Network Intrusion Detection. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS)*, 2004.

[52] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. *ACM Computing Surveys (CSUR)*, 34(3):375–408, 2002.

[53] Elmootazbellah N. Elnozahy and Willy Zwaenepoel. Manetho, Transparent Rollback-Recovery with Low Overhead, Limited Rollback and Fast Output commit. *IEEE Transactions on Computers*, 41(5):526–531, 1992.

[54] Klaus Finkenzeller. *RFID Handbook: Radio-Frequency Identification Fundamentals and Applications.* John Wiley and Sons, 1999.

[55] Michael J. Franklin, Shawn R. Jeffery, Sailesh Krishnamurthy, Frederick Reiss, Shariq Rizvi, Eugene Wu, Owen Cooper, Anil Edakkunni, and Wei Hong. Design Considerations for High Fan-In Systems: The HiFi Approach. In *Proceedings of the 2nd Biennial Conference on Innovative Data Systems Research (CIDR)*, pages 290–304, 2005.

[56] Hector Garcia-Molina and Daniel Barbara. How to Assign Votes in a Distributed System. *Journal of the ACM (JACM)*, 32(4):841–860, 1985.

[57] David K. Gifford. Weighted Voting for Replicated Data. In *Proceedings of the 7th ACM Symposium on Operating Systems Principles (SOSP)*, pages 150–162, 1979.

[58] Jim Gray. Note on Data Base Operating Systems. *Lecture Notes In Computer Science*, 60:393–481, 1978.

[59] Jim Gray. The Transaction Concept: Virtues and Limitations. In *Proceedings of the 7th International Conference on Very Large Data Bases (VLDB)*, pages 144–154, 1981.

[60] Jim Gray. Why Do Computers Stop and What Can be Done About It? Technical Report 85.7, Tandem Computers, 1985.

[61] Jim Gray, Paul McJones, Mike Blasgen, Bruce Lindsay, Raymond Lorie, Tom Price, Franco Putzolu, and Irving Traiger. The Reocvery Manager of the System R Database Manager. *ACM Computing Surveys (CSUR)*, 13(2):223–242, 1981.

[62] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann Publishers, 1993.

[63] Andy Harter, Andy Hopper, Pete Steggles, Andy Ward, and Paul Webster. The Anatomy of a Context-Aware Application. In *Proceedings of the 5th Annual ACM/IEEE International Conference on Mobile Computing and Networking (MOBICOM)*, pages 59–68, 1999.

[64] Jean-Michel Helary, Achour Mostefaoui, and Michel Raynal. Preventing Useless Checkpoints in Distributed Computations. In *Proceedings of the 16th Symposium on Reliable Distributed Systems (SRDS)*, pages 183–190, 1997.

[65] Mark Hiscock and Simon Gormley. Understanding High Availability with WebSphere MQ. An IBM white paper, May 2005.

[66] B. Hoffmann-Wellenhof, H. Lichtenegger, and J. Collins. *Global Positioning System: Theory and Practice, Fourth Edition.* Springer-Verlag, 1997.

[67] Meichun Hsu. Special Issues on Workflow Systems. *IEEE Data Engineering Bulletin*, 18(1), 1995.

[68] Jeong-Hyon Hwang, Magdalena Balazinska, Alexander Rasin, Uğur Çetintemel, Michael Stonebraker, and Stan Zdonik. High-Availability Algorithms for Distributed Stream Processing. In *Proceedings of the 21st International Conference on Data Engineering (ICDE)*, pages 779–790, 2005.

[69] Jeong-Hyon Hwang, Uğur Çetintemel, and Stan Zdonik. A Cooperative, Self-Configuring High-Availability Solution for Stream Processing. In *Proceedings of the 23rd International Conference on Data Engineering (ICDE)*, pages 176–185, 2007.

[70] Jeong-Hyon Hwang, Uğur Çetintemel, and Stan Zdonik. Fast and Reliable Stream Processing over Wide Area Networks. In *Proceedings of the 1st ICDE Workshop on Scalable Stream Processing Systems (SSPS)*, pages 604–613, 2007.

[71] Jeong-Hyon Hwang, Uğur Çetintemel, and Stan Zdonik. Fast and Highly-Available Stream Processing over Wide Area Networks. In *Proceedings of the 24th International Conference on Data Engineering (ICDE)*, pages 804–813, 2008.

[72] Jeong-Hyon Hwang, Sanghoon Cha, Uğur Çetintemel, and Stan Zdonik. Borealis-R: A Replication-Transparent Stream Processing System for Wide-Area Monitoring Applications. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 1303–1306, 2008.

[73] IBM. IBM DB2 - Online Documentation. `http://www.ibm.com/software/data/db2/udb/hadr.html`.

[74] IBM Corporation. IBM WebSphere V5.0: Performance, Scalability, and High Availability: WebSphere Handbook Series. IBM Redbook, July 2003.

[75] Navendu Jain, Lisa Amini, Henrique Andrade, Richard King, Yoonho Park, Philipe Selo, and Chitra Venkatramani. Design, Implementation, and Evaluation of the Linear Road Benchmark on the Stream Processing Core. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, pages 431–442, 2006.

[76] David B. Johnson. *Distributed System Fault Tolerance using Message Logging and Checkpointing*. PhD thesis, Department of computer Science, Rice University, 1989.

[77] David B. Johnson and Willy Zwaenepoel. Sender-based Message Logging. In *Proceedings of the 17th Annual International Symposium on Fault-Tolerant Computing*, pages 14–19, 1987.

[78] Theodore Johnson, S. Muthukrishnan, and Irina Rozenbaum. Sampling Algorithms in a Stream Operator. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, pages 1–12, 2005.

[79] Joseph M. Kahn, Randy H. Katz, and Kristofer S. J. Pister. Next Century Challenges: Mobile Networking for "Smart Dust". In *Proceedings of the 5th Annual ACM/IEEE International Conference on Mobile Computing and Networking (MOBICOM)*, pages 271–278, 1999.

[80] Mohan Kamath, Gustavo Alonso, Roger Guenthor, and C. Mohan. Providing High Availability in Very Large Workflow Management Systems. In *Proceedings of the 5th International Conference on Extending Database Technology (EDBT)*, pages 427–442, 1996.

[81] Leonard Kawell, Steven Beckhardt, Timothy Halvorsen, Raymond Ozzie, and Irene Greif. Replicated Document Management in a Group Communication System. In *Proceedings of the 1988 ACM conference on computer-supported cooperative work (CSCW)*, 1988.

[82] Richard Koo and Sam Toueg. Checkpointing and Rollback-Recovery for Distributed Systems. *IEEE Transactions on Software Engineering*, 13(1):23–31, 1987.

[83] Christopher Kruegel, Fredrik Valeur, Giovanni Vigna, and Richard Kemmerer. Stateful Intrusion Detection for High-Speed Networks. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pages 285–, 2002.

[84] Tirthankar Lahiri, Amit Ganesh, Ron Weiss, and Ashok Joshi. Fast-Start: Quick Fault Recovery in Oracle. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, pages 593–598, 2001.

[85] Ten H. Lai and Tao H. Yang. On Distributed Snapshots. *Information Processing letters*, 25:153–158, 1987.

[86] Leslie Lamport. The Implementation of Reliable Distributed Multiprocess Systems. *Computer Networks*, 2:95–114, 1978.

[87] Leslie Lamport. The Part-Time Parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[88] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[89] Butler W. Lampson. How to Build a Highly Available System using Consensus. In *Proceedings of the 10th International Workshop on Distributed Algorithms (WDAG)*, pages 1–17, 1996.

[90] Butler W. Lampson. The ABCD's of Paxos. In *Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, page 13, 2001.

[91] Butler W. Lampson and Howard Sturgis. Crash Recovery in a Distributed Data Storage system. Technical report, Computer Science Laboratory, Xerox Palo Alto Research Center, 1976.

[92] Ulf Leonhardt and Jeff Magee. Multi-sensor Location Tracking. In *Proceedings of the 4th Annual ACM/IEEE International Conference on Mobile Computing and Networking (MOBICOM)*, pages 203–214, 1998.

[93] R. A. Lorie. Physical Integrity in a Large Segmented Database. *ACM Transactions on Database Systems (TODS)*, 2(1):91–104, 1977.

[94] Samuel Madden and Michael J. Franklin. Fjording the Stream: An Architecture for Queries over Streaming Sensor Data. In *Proceedings of the 18th International Conference on Data Engineering (ICDE)*, pages 555–566, 2002.

[95] Dahlia Malkhi and Michael Reiter. Byzantine Quorum Systems. In *Proceedings of the 29th annual ACM symposium on Theory of computing (STOC)*, pages 569–578, 1997.

[96] Mesquite Software, Inc. CSIM 18 User Guide. `http://www.mesquite.com`.

[97] C. Mohan, Donald J. Haderle, Bruce G. Lindsay, Hamid Pirahesh, and Peter M. Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Transactions on Database Systems (TODS)*, 17(1), 1992.

[98] C. Mohan and Bruce G. Lindsay. Efficient Commit Protocols for the Tree of Processes Model of Distributed Transactions. In *Proceedings of the 2nd Annual ACM SIGACT-SIGOPS Symposium on Princiles of Distributed Computing (PODC)*, pages 76–88, 1983.

[99] Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Gurmeet Manku, Chris Olston, Justin Rosenstein, and Rohit Varma. Query Processing, Approximation, and Resource Management in a Data Stream Management System. In *Proceedings of the 1st Biennial Conference on Innovative Data Systems Research (CIDR)*, 2003.

[100] Rohan Narayana Murty and Matt Welsh. Towards a Dependable Architecture for Internet-scale Sensing. In *Proceedings of the 2nd Workshop on Hot Topics in Dependability (HotDep)*, 2006.

[101] Robert H. B. Netzer and Jian Xu. Necessary and Sufficient Conditions for Consistent Global Snapshots. *IEEE Transactions on Parallel and Distributed Systems*, 6(2):165–169, 1995.

[102] Chris Olston, Jing Jiang, and Jennifer Widom. Adaptive Filters for Continuous Queries over Distributed Data Streams. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 563–574, 2003.

[103] Oracle Inc. Oracle 10*g* High Availability Solutions. `http://otn.oracle.com/deploy/availability`.

[104] Michael Otey and Denielle Otey. Choosing a Database for High Availability: An Analysis of SQL Server and Oracle. A Microsoft white paper, 2005.

[105] Vern Paxon. End-to-End Routing Behavior in the Internet. *IEEE ACM Transactions on Networking*, 5(5):601–615, 1997.

[106] Peter Pietzuch, Jonathan Ledlie, Jeffrey Shneidman, Mema Roussopoulos, Matt Welsh, and Margo Seltzer. Network-Aware Operator Placement for Stream-Processing Systems. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE)*, pages 49–58, 2006.

[107] PlanetLab. `http://www.planet-lab.org`.

[108] Nissanka B. Priyantha, Allen K. L. Miu, Hari Balakrishnan, and Seth Teller. The Cricket Compass for Context-Aware Mobile Applications. In *Proceedings of the 7th Annual ACM/IEEE International Conference on Mobile Computing and Networking (MOBICOM)*, pages 1–14, 2001.

[109] NTP: The Network Time Protocol. `http://www.ntp.org`.

[110] B. Randell. System Structure for Software Fault Tolerance. *IEEE Transactions on Software Engineering*, 1(2):220–232, 1975.

[111] Ashish Ray. Oracle Data Guard: Ensuring Disaster Recovery for the Enterprise. An Oracle white paper, March 2002.

[112] Frederick Reiss and Joseph M. Hellerstein. Data Triage: An Adaptive Architecture for Load Shedding in TelegraphCQ. In *Proceedings of the 21st International Conference on Data Engineering (ICDE)*, pages 155–156, 2005.

[113] Esther Ryvkina, Anurag Maskey, Mitch Cherniack, and Stan Zdonik. Revision Processing in a Stream Processing Engine. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE)*, page 141, 2006.

[114] Andre Schiper and Sam Toueg. From Set Membership to Group Membership: A Separation of Concerns. *IEEE Transactions on Dependable and Secure Computing*, 3(1):2–12, 2006.

[115] Fred B. Schneider. Implementing Fault-Tolerant Services using the State Machine Approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[116] Fred B. Schneider. What Good Are Models and What Models Are Good? In *Distributed Systems*, pages 17–26. ACM Press/Addison-Wesley Publishing Co, 2nd edition, 1993.

[117] Loren Schwiebert, Sandeep K. S. Gupta, and Jennifer Weinmann. Research Challenges in Wireless Networks of Biomedical Sensors. In *Proceedings of the 7th Annual ACM/IEEE International Conference on Mobile Computing and Networking (MOBICOM)*, pages 151–165, 2001.

[118] Mehul A. Shah, Joseph M. Hellerstein, and Eric Brewer. Highly-Available, Fault-Tolerant, Parallel Dataflows. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, pages 827–838, 2004.

[119] Mehul A. Shah, Joseph M. Hellerstein, Sirish Chandrasekaran, and Michael J. Franklin. An Adaptive Partitioning Operator for Continuous Query Systems. Technical Report CS-02-1205, UC Berkeley, 2002.

[120] Mehul A. Shah, Joseph M. Hellerstein, Sirish Chandrasekaran, and Michael J. Franklin. Flux: An Adaptive Partitioning Operator for Continuous Query Systems. In *Proceedings of the 19th International Conference on Data Engineering (ICDE)*, pages 25–36, 2003.

[121] Tetsuya Shirai, John Barber, Mohan Sabo jr, Indran Naick, and Bill Wilkins. *DB2 Universal Database in Application Development Environments*. Prentice Hall, 2000.

[122] Abraham Silberschatz, Henry Korth, and S. Sudarshan. *Database Systems Concepts*. McGraw-Hill, 4 edition, 2002.

[123] A. Prasad Sistla and Jennifer L. Welch. Efficient Distributed Recovery using Message Logging. In *Proceedings of the 8th Annual ACM SIGACT-SIGOPS Symposium on Princiles of Distributed Computing (PODC)*, pages 223–238, 1989.

[124] Dale Skeen. Non-Blocking Commit Protocols. In *Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data*, pages 133–142, 1981.

[125] Sean W. Smith and David B. Johnson. Minimizing Timestamp Size for Completely Asynchronous Optimistic Recovery with Minimal Rollback. In *Proceedings of the 15th Symposium on Reliable Distributed Systems*, pages 66–75, 1996.

[126] Utkarsh Srivastava, Kamesh Munagala, and Jennifer Widom. Operator Placement for Internet Stream Query Processing. In *Proceedings of the 24th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 250–258, 2005.

[127] Utkarsh Srivastava and Jennifer Widom. Memory-Limited Execution of Windowed Stream Joins. In *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB)*, pages 324–335, 2004.

[128] Streambase, Inc. `http://www.streambase.com`.

[129] Rob Strom and Shaula Yemini. Optimistic Recovery in Distributed Systems. *ACM Transactions on Computing Systems (TOCS)*, 3(3):204–226, 1985.

[130] Mark Sullivan. Tribeca: A Stream Database Manager for Network Traffic Analysis. In *Proceedings of the 22nd International Conference on Very Large Data Bases (VLDB)*, page 594, 1996.

[131] Robert Szewczyk, Joseph Polastre, Alan M. Mainwaring, and David E. Culler. Lessons from a Sensor Network Expedition. In *Proceedings of the first European Workshop on Wireless Sensor Networks (EWSN)*, pages 307–322, 2004.

[132] Yuval Tamir and Carlo H. Sequin. Error Recovery in Multicomputers using Global Checkpoints. In *Proceedings of the International Conference on Parallel Processing*, pages 32–41, 1984.

[133] Nesime Tatbul, Mark Buller, Reed Hoyt, Steve Mullen, and Stanley Zdonik. Confidence-based Data Management for Personal Area Sensor Networks. In *Proceedings of the VLDB Workshop on Data Management for Sensor Networks (DMSN)*, pages 24–31, 2004.

[134] Nesime Tatbul, Uğur Çetintemel, Stan Zdonik, Mitch Cherniack, and Michael Stonebraker. Load Shedding in a Data Stream Manager. In *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB)*, pages 309–320, 2003.

[135] Nesime Tatbul, Ugur Cetintemel, and Stan Zdonik. Staying FIT: Efficient Load Shedding Techniques for Distributed Stream Processing. In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB)*, pages 159–170, 2007.

[136] Nesime Tatbul and Stan Zdonik. Window-aware Load Shedding for Aggregation Queries over Data Streams. In *Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB)*, pages 799–810, 2006.

[137] Crossbow Technology. `http://www.xbow.com/`.

[138] Douglas B. Terry, David Goldberg, David Nichols, and Brian Oki. Continuous Queries over Append-only Databases. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, pages 321–330, 1992.

[139] Douglas B. Terry, Marvin Theimer, Karin Petersen, Alan J. Demers, Mike Spreitzer, and Carl Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, pages 172–183, 1995.

[140] Irving L. Traiger, Jim Gray, Cesare A. Galtieri, and Bruce G. Lindsay. Transactions and Consistency in Distributed Database Management Systems. *ACM Transactions on Database Systems (TODS)*, 7(3):323–342, 1982.

[141] Peter Tucker, David Maier, Tim Shreard, and Leonidas Fegaras. Exploiting Punctuation Semantics in Continuous Data Streams. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 15(3):555–568, 2003.

[142] Randy Urbano. Oracle Streams Replication Administrator's Guide, 10g Release 1. *Oracle Corporation*, 2003.

[143] Ben Vandiver, Hari Balakrishnan, Barbara Liskov, and Sam. Tolerating Byzantine Faults in Database Systems using Commit. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, pages 59–72, 2007.

[144] Giovanni Vigna, William Robertson, Vishal Kher, and Richard A. Kemmerer. A Stateful Intrusion Detection System for World-Wide Web Servers. In *Proceedings of the 19th Annual Computer Security Applications Conference (ACSAC)*, pages 34–45, 2003.

[145] Yi-Min Wang. *Space Reclamation for Uncoordinated Checkpointing in Message-Passing Systems*. PhD thesis, Department of Computer Science, University of Illinois, 1993.

[146] Yi-Min Wang. Consistent Global Checkpoints that Contain a Set of Local Checkpoints. *IEEE Transactions on Computers*, 46(4):456–468, 1997.

[147] Arthur T. Whitney and Dennis Shasha. Lots of Ticks: Real-Time High Performance Time Series Queries on Billions of Trades and Quotes. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, page 617, 2001.

[148] Ying Xing, Jeong-Hyon Hwang, Uğur Çetintemel, and Stan Zdonik. Providing Resiliency to Load Variations in Distributed Stream Processing. In *Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB)*, pages 775–786, 2006.

[149] Ying Xing, Stan Zdonik, and Jeong-Hyon Hwang. Dynamic Load Distribution in the Borealis Stream Processor. In *Proceedings of the 21st International Conference on Data Engineering (ICDE)*, pages 791–802, 2005.

[150] Yunyue Zhu and Dennis Shasha. StatStream: Statistical Monitoring of Thousands of Data Streams in Real Time. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB)*, pages 358–369, 2002.