NTNU – Trondheim
Norwegian University of
Science and Technology

Continuously adapting continuous Queries for Data Streams in Raincoat

Ken Oscar Grønnbeck
Steffen Rendahl Stenersen

Master of Science in Computer Science
Submission date: June 2013
Supervisor: Kjetil Nørvåg, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

Continuation of the specialization project 'Raincoat: High-level Framework for Twitter Storm', where a framework was created to simplify the process of creating Storm topologies. The thesis involves researching the state of the art within the query optimization field, and explore how it can be used in a distributed stream processing environment such as Raincoat.

The result of the thesis should be a optimizer that can optimize continuous SQL-inspired queries, which takes into account the volatile property of data streams, the real-time constraints such a system have and utilizes the parallelism of the system.

Assignment given: 27. January 2013
Supervisor: Kjetil Nørvåg

Abstract

In the last decade, the world wide web has grown from being a platform where users passively viewed content, to an active platform where the users themselves contributed with new content. With this came an explosion of available data that ventures could use to gain market advantage. Not only did the did the amount of available data grow massively, but also newly produced data started to arrive at immense speed. This spawned a new field of specialized computational framework being able to handle the change in the data paradigm. Now, one must be able to process the massive amount of incoming data within a reasonable response time, as well as be able to handle its high velocity. This spurred several ideas for processing fast data. One of these ideas uses SQL-like languages for processing fast data, taking advantage of the years of work on query optimization theory.

In the fall of 2012, we proposed and implemented the prototype of Raincoat. Raincoat was developed to ease developers without any experience with distributed programming, providing a familiar interface which they could use to deploy stream filtering jobs to a Storm cluster. As the prototype did not include any query optimization techniques it does not meet the expected performance requirements. In this thesis we research optimization techniques for scaling Raincoat. We explore optimization techniques from different fields including traditional, distributed, parallel, streaming and adaptive query optimization.

We propose an adaptive query optimizer, inspired by existing adaptive query optimizers. The focus of the optimizer lies in detecting when an optimization is needed and which optimization techniques that should be applied. In this thesis we explore the possibility of adaptively achieving better performance and scalability by carefully selecting the *join order*, *select order*, *merging of selection operators*, and applying *intra-operator parallelism* on operators.

Based on our results from experiments on the different implemented optimizers, we demonstrate their applicability and their significant contribution in increasing the performance of a Raincoat query.

Sammendrag

Tidlig i forrige tiår forandret world wide web seg fra å være en passiv innholdsplatform, til å bli en platform der brukere selv bidrar til å genere data dynamisk. Ut i fra denne forandringen begynte størrelsen på tilgjengelig data på webben å vokse ut av kontroll. Ikke bare må man håndtere størrelsen på den nye tilgjengelig dataen, men også den enorme hastigheten dataene ble tilgjengelig i. Ut i fra dette vokste det frem nye spesialtilpassede rammeverk for å håndtere prosessering av data fra dette paradigmet. Flere tilnærminger for å håndtere disse dataene ble foreslått. En av disse ideene var å bruke SQL-lignende språk for behandling av rask data. Ved å basere seg på SQL kunne man ta nytte av mange års forskning med spørreoptimaliserings-teori.

I prosjektoppgaven vår fra høsten 2012 foreslo vi en prototype av rammeverket Raincoat. Vi utviklet Raincoat som et system for å gi utviklere med lite erfaring innen distribuert programmering en enkel måte å prosessere rask data på. Dette ble realisert ved at utviklere kan bruke det velkjente grensesnittet til SQL til å distribuere prosesseringsjobber over en Storm kluster via Raincoat-rammeverket.

Siden prototypen ikke hadde støtte for optimalisering vil den ikke møte ytelseskravene til en bruker av systemet. I denne masteroppgaven forsker vi på forskjellige optimaliseringsteknikker for skalering av Raincoat-spørringer. Vi utforsker hvilke spørreoptimaliseringsteknikker kan tas i bruk i optimaliseringen av Raincoat-spørringer fra fagområdene til tradisjonelle, distribuerte, parallelle, strømmende og adaptive spørringer.

I denne masteroppgaven foreslår vi en adaptiv spørreoptimalisering som er inspirert av eksisterende optimaliseringsteknikker fra samme fagfelt. Vår tilnærming fokuserer på å både detektere om en optimalisering trengs og hvilken teknikk for optimalisering som skal tas i bruk for å oppnå ønsket ytelse. Vi undersøker hvordan vi adaptivt kan få bedre ytelse og skalerbarhet ved å med omhu velge hvilke teknikker for optimalisering man skal ta i bruk.

Med utgangspunkt i resultatene fra eksperimenter utført med de forskjellige optimaliseringsteknikkene, viser vi hver av teknikkenes anvendbarhet og bidrag til å bedre ytelsen for en Raincoat-spørring.

Acknowledgments

We would like to acknowledge the help provided by professor Kjetil Nørvåg at the Department of Computer and Information Science at NTNU. Kjetil has guided the thesis in the right direction, as well as given us valuable feedback and critiques of our work.

Contents

Abbreviations

API	Application Programming Interface
AQP	Adaptive Query Processing
DDSMS	Distributed Data-Stream Management System
DSMS	Data-Stream Management System
EBNF	Extended Backus-Naur Form
HDD	Hard Disk Drive
IEC	International Electrotechnical Commission
ISO	International Organization for Standardization
JVM	Java Virtual Machine
RDBMS	Relational Database Management System
SPJ	Select-Project-Join
SQL	Structured Query Language
TPC	Transaction Performance Council

Chapter 1

Introduction

1.1 Motivation

In the last decade, the world wide web has grown from being a platform where users passively viewed content, to an active platform where the users themselves contributed with new content. One can argue that Web 2.0 changed the web from static content to user-generated content. It introduced new web-based ventures where the product was the users and their generating content. Examples of such ventures are social networking sites, blogs, wikis, and video sharing sites. Needless to say, when millions of users having access to the internet suddenly starts producing content, the amount of new data available every day was immense in comparison to its predecessor.

The traditional relational database management systems are not designed to solve the new application domains introduced by the large quantity of data available on the internet. The exponential growth of data made available online has nurtured the need of finding better data processing tools. Having the better data processing tools has become a crucial factor for being a competitive web-based company.

New application domains means new fields of study. This has spurred the creation of new frameworks carefully adapted to a certain application domain. Google pioneered and standardized the idea of processing large amounts of data offline using MapReduce [16]. While the processing of fast streaming data has not yet been standardized and remains an open problem. In the past few years application frameworks such as Storm, S4 [32], STREAM [31], Aurora DB [9], Tribeca [40], and Gigascope [15] have been developed for simplifying the stream data processing. Generally, stream data processing frameworks can be divided into two main approaches the general and continuous query approach. Where the latter is basically the idea of a query that has no explicit end, thus, will process incoming stream data whenever data arrives. However, they all try to provide users with an application for processing fast streaming data.

Query optimization for data processing using continuous queries has been a subject of study for the last decade. Optimizing for a data stream environment is often very different than it has been for traditional database management systems. Therefore many new approaches has been proposed, one of these approaches is called Adaptive Query Processing (AQP). AQP is best summarized as a collection of techniques rather than a single technique of its own.

A problem brought to light by AQP is the process of continuously re-optimize continuous queries in a data stream environment, where the property of the data of a data stream might change over time. AQP tries to provide techniques and answers for questions in the domain of continuous query optimization. An example of such a question can be how often (or when) a continuous query should be optimized.

In the fall of 2012, we proposed a system to make processing of large data easier. Built upon Storm, a distributed and fault-tolerant real-time computation system, created by Nathan Marz, Raincoat allows the user to submit continuous queries described using a SQL inspired syntax which is translated into Storm topologies. These topologies are then distributed and ran over Storm cluster. The framework had however no optimization implemented, which makes it scale poorly with increasing data volumes. In this master thesis, we continue on the Raincoat system, focusing on researching and implementing query optimization techniques, better support for parallelization, and techniques for adaptive optimization.

1.2 Goals and research questions

With this master thesis we want to contribute with a set of optimization techniques for continuous select-project-join queries implemented upon Storm. We explores the possibility of dynamically and continuously optimize running queries supporting both soft- and hard real-time constraints.

The main goal is to have a system that can automatically optimize queries that are being processed in Raincoat. There are two requirements the optimization tries to meet:

1. Queries submitted to the system specifies a rate at which data should be outputted to the user. The system must be able to compute the answer within that time frame.
2. The system must be able to scale with increases in resources(nodes in the cluster) to handle increases in the rate at which the data arrives.

In order to meet the requirements, we want to explore how we can apply adaptive query optimization techniques to Raincoat. We want to answer the following questions:

RQ1 How can we determine the cost of a Storm topology, and how can we use that data to optimize the topology?

RQ2 Which methods exists in the field of query optimization, and can they be translated into our domain?

RQ3 When should the system perform optimization on the topology?

RQ4 How should we dedicate and fully utilize the resources we have available to the topology?

1.3 Outline

Chapter 2 covers the background knowledge required in the thesis, where we introduce the concept of Big Data and the different kinds of database system that exists. We also give an introduction to the MapReduce framework, and the concept of windows in streaming data applications.

Chapter 3 gives an overview of the state of the art techniques in traditional query optimization and adaptive query optimization, and answer the research question RQ2. Next we introduce the frameworks that is our basis for our thesis, in Chapter 4.

While designing the optimizer, we had to make several design decisions, and these the arguments behind them are presented in Chapter 5. This chapter partly answers the research question RQ4. The next chapter, Chapter 6, covers the optimization techniques we have applied in Raincoat, as well as the cost model for the topologies. It answers the research questions RQ1 and RQ3, and partly RQ4.

The testing and analysis of the optimization techniques is covered in Chapter 7. Finally, Chapter 8 concludes the master thesis and presents some notes on further work that was not covered in this thesis.

Chapter 2

Background

This chapter introduces the background knowledge required in the thesis. In Section 2.1, the concept of big data is introduced. We look at both traditional relational database systems methods and newer methods used in the field of distributed database systems in Section 2.2. Further, in Section 2.3 we introduce MapReduce, a programming model and framework for processing big data offline. Section 2.4 introduces the concepts of Windows.

2.1 Big data

Big data can be defined as data that exceeds the processing capacity of conventional database systems [17]. It is data that is difficult to both store and access, therefore specialized software is often used to be able to perform meaningful computations and analysis on it. This section is taken from our Raincoat paper written in the fall of 2012 [22].

2.1.1 Volume, velocity & variety

Data can be characterized in terms of three variables, volume, velocity & variety [17]. Volume is the size of the data. The size of the data becomes a problem when a conventional database system is unable to store all of the data coming in to the system. When this happens one might look at other ways to store the data than the conventional database systems or do some processing of the data in order to reduce it is size before storing it. Velocity is the rate at which data flows into the system. The faster the data comes into the system, the bigger the challenge it is to process and store it. Variety, which is that the data often is presented on different forms and its often hard to structure it into relational structures. If, for instance, a system gets data from multiple sources with different types of data, it might not be a easy way to structure the data. And if the data is structured into relational structures, it becomes a challenge to change those structures in the event that the incoming data changes or you add another data source to your system.

2.1.2 Streaming data

First of all we need to define what a data stream is. We use Golab and Özsu [20] definition of data stream, that is, a data stream is an unbounded data-set that is produced incrementally over time. Such data streams can come from a variety of sources. A few examples are given by the same authors. An excerpt of those examples are internet traffic analysis, senors networks, log mining of credit card transactions, and financial tickers and online trading. These sources comes from different areas of expertise but have one thing in common, a large stream of data. They can vary from a few gigabytes of data a day to several terabytes. An introductory example of such a data stream is the Twitter Firehose which streams millions of tweets on a daily basis, and at the time of writing the amount of tweets exceeded over 250 millions pr day. [1]

At any given time, a stream can be defined as a finite set N of points $x_1 ..., x_n$ that can only be read in the same order as they enter the system [33].

2.1.3 Batch vs real-time

We have defined batch-oriented data processing systems as systems that is not designed to support applications with real-time constraints. The most known and popular programming model and system that implements it is MapReduce and Apache Hadoop, respectively. MapReduce is discussed further in 2.3. On the other hand, a real-time system is designed to allow the users to get results from queries in near real-time on the most recent data [10]. Because the input stream can be indefinitely long, getting an exact answer on that data is in-feasible, but in many cases the systems that uses a real-time system requires the data right away [38]. This is discussed further in Section 2.2.

2.2 Database systems

2.2.1 Relational database management system

The Relational Database Management System (RDBMS) are the traditional database management system that are built on the relational model [35]. The relational model was introduced in 1970 by Codd [14], and is the basis for SQL [25]. Basically, for a system to be a RDBMS it must at a minimum be able to present the data stored in the system to the user as relations, and provide relational operators to manipulate the data.

2.2.2 Data-stream management system

A data stream is a real-time, continuous, ordered (by arrival date) sequence of items [21]. A Data-Stream Management System (DSMS) is a system to manage data streams, similar to how RDBMS manages relational data.

[1] http://news.cnet.com/8301-1023_3-57541566-93/report-twitter-hits-half-a-billion-tweets-a-day/

DSMS have grown in popularity due to the number of data sources are expanding and the size of the data grows. Examples of data sources can be sensors, satellites and stock feeds [1], or more recent streams like Twitter Firehose. Since streams may be of infinite size, storing all data that is needed for computing an exact answer may not be possible, so you can not treat the operations which are to be done on the data like you would in a relational database system. Another property of data-streams are that queries applied on them often come with some constraints to the response time. An example are systems that monitor network logs for intrusion attempts. These systems needs to respond in real time as the intrusion attempt is going on. Because of the mentioned properties, data-stream management system can use approximation techniques to give an answer to the user in the appropriate time frame.

Two examples of existing Data-Stream Management Systems are the Aurora [1] and The Stanford Data Stream Management System (STREAM) [2].

2.2.3 Distributed query systems

As data grows, more processing power is needed in order to process the data in a reasonable amount of time, and a database running on a single machine is not capable of meeting the same requirements for response time as a cluster of machines. Distributed database systems such as HBase [2] and batch processing tools like MapReduce, presented in-depth in Section 2.3, proposes two different strategies for processing large quantity of data where traditional database systems fails.

2.2.4 Distributed data-stream management system

The DSMS and distributed query systems solves two different problems. There are situations where we need the distributed query processing power on streaming data. Systems that are within this category are Yahoo! S4 [32], Walmarts Muppet [29] and Twitter Storm. Storm is introduced in Section 4.1.

2.3 MapReduce

The big data paradigm have been around for a while, but after the introduction of MapReduce in 2004, the field has gained more public attention, and the open source implementation of the MapReduce computational model, Hadoop, have made the task of processing large data sets easily available to the public. This section gives a brief overview of the MapReduce computational model and the architecture of the system, and introduces some of the issues that needs to be addressed in such systems. This section is taken from the Raincoat paper by Gronnbeck and Stenersen [22].

[2]Apache HBase: http://hbase.apache.org/

2.3.1 Introduction

MapReduce is a programming model (and a framework) developed by Google for doing computations on large amounts of data. The framework is basically a distributed execution of the two functions map and reduce. These two functions are implemented by the user of MapReduce. Both functions are inspired from methods in functional languages such as Lisp.

The run-time system of MapReduce takes care of partitioning of input data, scheduling the programs execution across a set of machines, handling machine failure, and managing the required inter-machine communication. By doing all this it allows programmers without any experience in parallel or distributed computing to fully take advantage of such computing.

It can be used to process a variety of data-types, and many different scenarios of large scale computations. MapReduce provides an abstraction for hiding complexity in a distributed system that process large amount of data, so that programmers without any experience with distributed systems can solve such tasks.

2.3.2 Programming model

The input for the computation is key/value pair and it produces a set of key/value pair as output. The user has to express the two functions: map and reduce.

The MapReduce library groups together all intermediate values associated with the same intermediate key I and passes them to the Reduce function. For examples of the map and reduce functions, see [16]. The functional representation of map and reduce are listed below.

```
map    (k1, v1)        →  list(k2,v2)
reduce (k2, list(v2)) →  list(v2)
```

In words, the map function takes in a key-value pair, performs an implementation specific action, and returns list of key-value pairs. The reduce function takes in list of values belonging to an inputted key, and performs an implemented analysis and recombination of inputs, and finally returns a new list values. Both the map and reduce functions implementations is specified by an user in form of a MapReduce job.

Example. MapReduce can be used to find URL occurrences in a large data set. In such a MapReduce job, the map tasks would usually use the URL as the key and set its corresponding value to 1. Then the reduce job would receive a list of 1s belonging to a particular URL, sum up that list, and the return the sum being the number of occurrences of that URL.

Other examples of what MapReduce has been used for are listed below. For a better description of those problems, see the paper by Dean and Ghemawat [16].

1. Distributed Grep

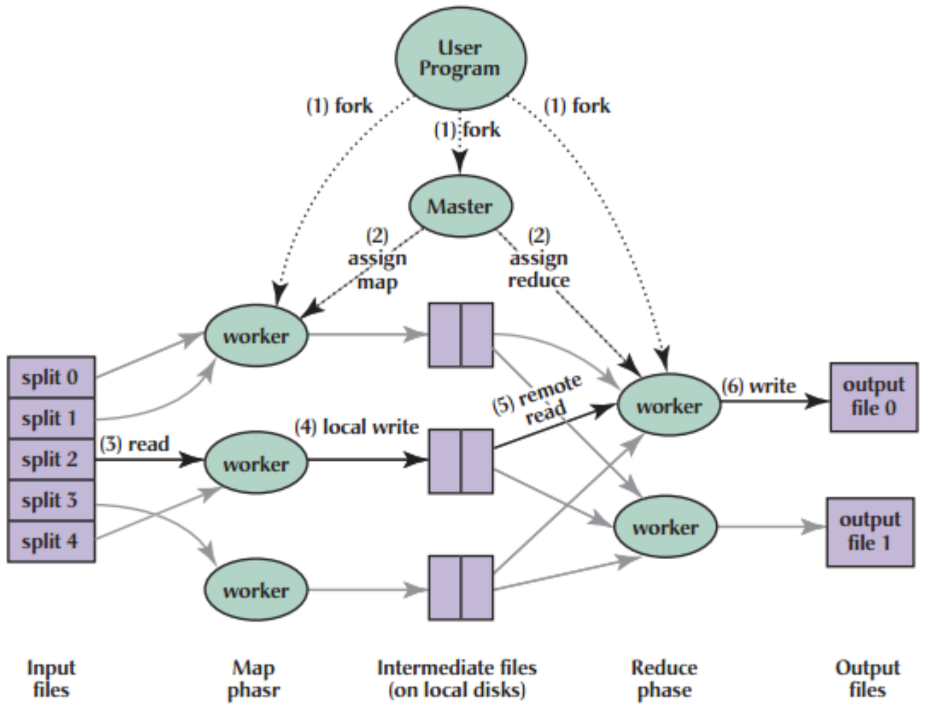


Figure 2.1: Overview of the execution of MapReduce. This figure is taken from the MapReduce paper [16]

2. Count of URL Access Frequency
3. Reverse Web-Link graph
4. Term-Vector per Host
5. Inverted Index
6. Distributed Sort

2.3.3 Execution

One of the nodes in the cluster is appointed as the master node. The rest are workers that get assigned work by the master. The input is split into a set of M splits, which can be processed in parallel. These splits are run through the given map function. The output of the map task is distributed to R reduce functions. The output of all map tasks are then sorted before delivered to the map functions, in order to group all occurrences of the same key together to speed up the reduce phase. Figure 2.1 gives a overview of the execution phase.

Master data structures

The master keeps track of some data structures in order to maintain the state of the tasks. For each completed map task the master stores the location and the sizes of the R intermediate file regions produce by the map tasks. The information is pushed incrementally to workers that have in-progress reduce tasks. For each map task and reduce task, it stores the state (idle, in-progress or completed), and the identity of the worker machine.

Fault tolerance

Fault tolerance for workers and masters are handled differently. The master pings workers periodically and awaits for an answer. If a worker fails to answer a ping the master re-schedules the tasks executed that the faulty worker was currently executing. By doing so MapReduce can guarantee that all tasks has been executed at least once. The master will ignore any duplicate results.

Handling master faults is far more difficult. The master does checkpointing and uses that to restart if a fault occurs. Dean & Ghemawat argues that since there is only one master such events are unlikely, and therefore in the presence of such events the current implementation (2004) of MapReduce just aborts the computation process.

Semantics in the presence of failures

When the user-supplied map and reduce operators are deterministic functions of their input values, MapReduce produces the same output as would have been produced by a non-faulting sequential execution of the entire program.

MapReduce does atomic commits of each task. Each in-progress task writes its output to private temporary files. A reduce task produces one local file, a map task produces R files (locally). When a map task completes, the worker sends a message to the master with the location of the local files (master ignores if it's a duplicate). When a reduce task completes, the reduce worker atomically renames its temporary output file to the final output file.

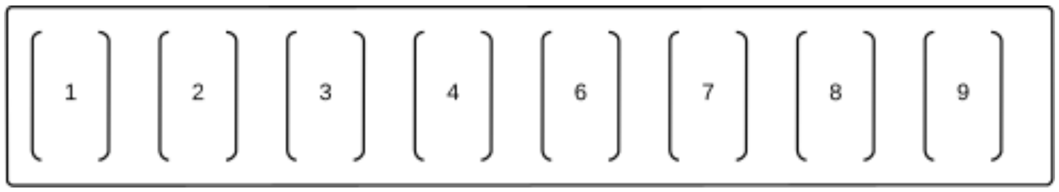
The main mechanism for fault tolerance is re-execution. Meaning, it will re-execute operations if a machine fails to deliver results, aka crashes.

2.4 Windows

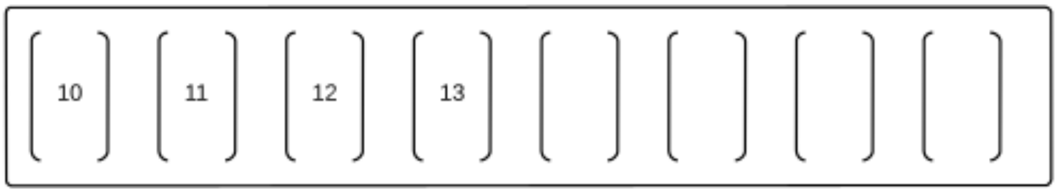
In traditional database management systems, one has access to the whole data-set and is able to calculate an exact answer to queries. When dealing with continuous data streams one has a completely different basis. Since the data set is potentially in-feasible to store, one can not do operations on all the data the system have seen. In order to get results from queries, the concept of windows have been introduced. When a query is submitted a window is defined to limit how much data should be used in each calculation.

There are two types of windows, sliding and tumbling [21]. In sliding windows, new tuples that enter are added in front of the window, pushing all other tuples one position back. When the window is full, the oldest tuple gets invalidated and are removed from the window. In a tumbling window, the queries wait until the window is full before executing the query. When the query is executing, the window is flushed and new tuples start entering the window. This result in that each tuple is only processed once.

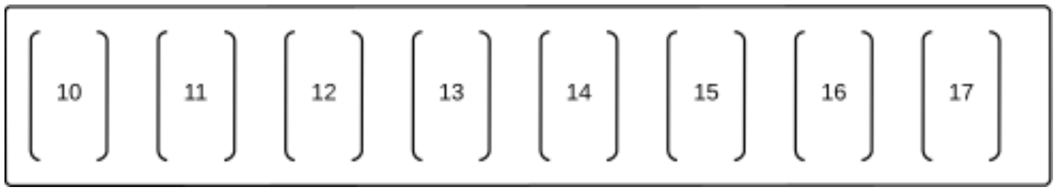
The biggest difference between the windows is the semantics they give us. In sliding windows, tuples can be used in several calculations, resulting in intersections between the output of the queries. On the other hand, tumbling windows will operate on independent data sets, which results in independent query results. Both tumbling and sliding windows has their uses, all depending on the results the user expects.



(a) The window's state after 9 tuples

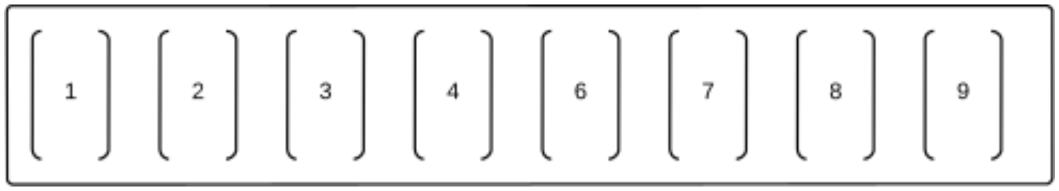


(b) The window's state after 13 tuples

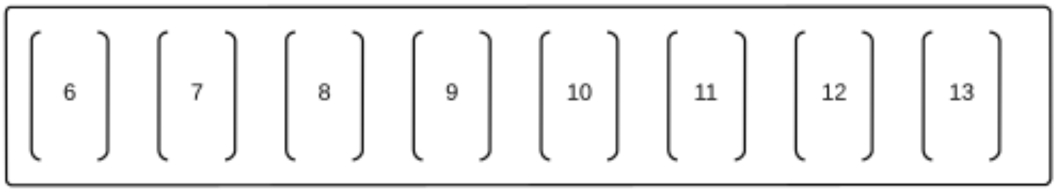


(c) The window's state after 17 tuples

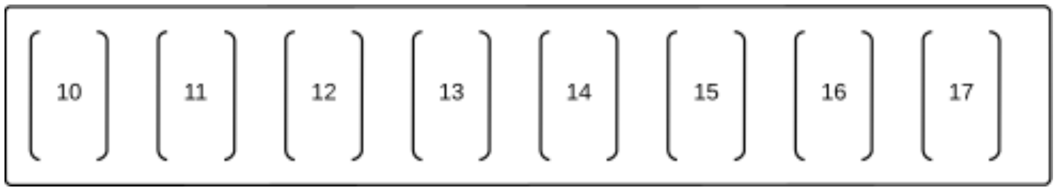
Figure 2.2: A visual representation of a tumbling window. The operation only executes over the tuples in the window when it is full.



(a) The window's state after 9 tuples



(b) The window's state after 13 tuples



(c) The window's state after 17 tuples

Figure 2.3: A visual representation of a sliding window. The operation can execute over the tuples at any time.

A window can be either time-based or count-based [21]. In a time-based sliding window, tuples expires after a given time-frame. In a time-based tumbling window, all tuples expires after a given time-frame. In time-based windows, there are no guarantee that there are any tuples in the window (this can happen if the data stream doesn't produce any tuples for a while). In a count-based sliding window, the oldest tuple expires when the window is full and a new tuple enters. In a count-based tumbling window, all tuples expires when the window is full.

Chapter 3

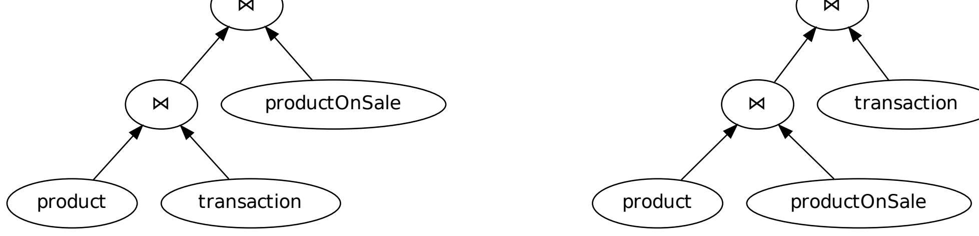
State of the art

This chapter gives an overview of the field of query optimization. Relational query languages such as SQL are parsed into relational algebra trees when submitted to the system. The relational algebra tree is a algebraic representation of the query, and consists of a set of operators nodes, such as select, project and join. The ordering of these operators have a huge impact on the time it takes to execute the query.

Consider the example query:

```
SELECT p.name, COUNT(*)
FROM product AS p, transaction AS t, productsOnSale AS ps
WHERE p.id = t.productId AND p.id = PS.id
GROUP BY p.id;
```

From it, we can derive (among others) the trees seen in Figure 3.1. If we say that only 10 of the products are on sale on any given time, and there is 100 000 transactions, then $join(p, ps)$ results in 10 tuples, and $join(p, t)$ results in 100 000 tuples. Then, the operation $join(join(p, ps), t)$ will produce a lot less tuples than $(join(t, p), ps)$.



(a) Join order 1 *(b) Join order 2*

Figure 3.1: Example of two possible join orders for a SQL query.

We introduce traditional query optimizing techniques in Section 3.1. Next, we introduce the concept of adaptive query processing in Section 3.2, both for traditional queries and for data streams. Further in Section 3.3, we look at techniques for optimizing distributed queries. In Section 3.4 we present different methods to estimate the different cardinalities for query optimizers. Query plans can be classified into two different structures, left-deep and bushy, Section 3.5 covers the difference between them. Finally, in Section 3.6 we present different ways to generate the query plans.

3.1 Traditional query optimizing

In traditional databases, queries are optimized before execution. The RDBMS usually have a system catalog, which contains a description of the database, such as tables, value ranges, indexes etc. It is based on this information the query optimizer decides how to optimize the given query. The goal of the traditional query optimizer is to find a low cost plan for executing a query [3]. The main variables that impacts the cost is disk usage and CPU processing time. So the optimizer tries to minimize the expected number of pages that needs to be fetched from secondary storage into the buffer and it tries to minimize how many tuple comparisons the query executor needs to perform. System R is a RDBMS which several newer RDBMS are based on. Its optimizer uses a cost model to estimate the cost of a partial or complete plan, and it estimates the size of the data outputted from each operator in the plan [12]. The cost model relies on formulas to estimate the selectivity of predicates, and formulas to estimate the CPU usage and I/O cost for each operator. Based on the cost model, the optimizer can choose the best plan for the query. In order to generate plans, it uses dynamic programming where it enumerated over all possible plans in a bottom-up fashion.

The system table only stores the sizes of the base relations. The challenge is to estimate the cardinality of selectivity for the various parts of the query. The problem of estimating cardinality is well-studied, and there exists various methods to do so [5]. Some of these methods are presented in Section 3.4.

3.2 Adaptive query optimizing

3.2.1 Adaptive query optimization on traditional queries

Traditional query optimization optimizes the queries before execution. Even though there has been a lot of research on pre-optimization of queries, the query optimizers can produce a sub-optimal result [27]. Adaptive query optimization tries to solve this problem by detecting sub-optimal query plans during query execution and improve the optimization.

The reason traditional query optimizers can produce sub-optimal results is because the query plans are generated based on estimates. These estimates are stored in the system catalogs, and might be inaccurate and not up-to-date. In addition, it does not take into account parameters that only are available at run-time, such as resource availability and system load [27].

Kabra and DeWitt [27] proposes a algorithm for doing adaptive query optimization, and it can be summarized in five steps:

1. **(Annotated) Query execution plans**. The pre-run-time query execution plan created by the optimizer does not store the information about the estimates used to create the query plan. By storing the information, it is easier to decide whether a runtime-generated plan is better than the current plan, as you can compare the estimates used against the actual data distribution.

2. **Runtime collection of statistics**. In order to support adaptive query processing, we must collect statistics from the running query. The collection is done by appending another node in the relational algebra tree which is responsible for the collection. The location of the node determines which part of the three can be optimized. Only the nodes that are processed after the statistics collection can be optimized.

3. **Dynamic resource re-allocation**. DBMS allocates memory to operators based on the estimated tuple cardinality coming into each operator. We can use the results from the runtime collection of statistics to better allocate memory based on the actual cardinality.

4. **Query plan modification**. There are two ways for the algorithm to change the running query plan. The simplest way is to discard the already processed results, and re-execute everything with a new query plan. This, however, is inefficient as it discards work that has been done. The other solution is to only re-optimize the parts of the query that has not yet started. The algorithm suspends the execution, and resumes the processing with a new plan that is optimized. One important question is when the re-optimization should be done. Kabra and DeWitt [27] suggest the following two heuristics to determine this:

$$\frac{T_{opt,estimated}}{T_{cur-plan,improved}} > \theta_1 \tag{3.1}$$

The heuristic presented in formula 3.1 determines whether it profitable to use time to search for an optimized plan. $T_{opt,estimated}$ is the estimated time taken to re-parse and re-optimize the query(based on the number of operators in the query). $T_{cur-plan,improved}$ is the estimate for executing the current plan based on collected runtime statistics. It is only profitable if 3.1 does not hold, i.e, the time required to optimize is significantly smaller than the time required to complete the current plan. Where $\theta_1 = 0.05$ or a similar value.

$$\frac{T_{cur-plan,improved} - T_{cur-plan,optimizer}}{T_{cur-plan,optimizer}} > \theta_2 \tag{3.2}$$

The heuristic presented in formula 3.2 checks if the current plan is sub-optimal. It is done by comparing the original cost based on estimates of the current plan against the improved cost based on the collected statistics. If the difference is bigger than $\theta_2 = 0.2$, we should re-optimize.

5. **Keeping overheads low**. The algorithm takes care not to collect statistics in situations where the probability that the estimates are wrong is low. The probability is based on the types of operators that are present in the query, so it for instance differs between non-equi-joins and equi-joins. The details can be found in [27].

3.2.2 Eddies

An Eddy is an operator that is used to continuously reorder operators in a query plan as it runs, and was introduced by Avnur and Hellerstein [4]. In a data stream environment where the characteristics of the data are unstable and hard to predict, a query plan that is optimal at one point might be sub-optimal right after. The concept of eddies was created on the assumption that data changes, and in order to efficiently compute the query one needs to consider which route the tuples should take on a tuple basis instead of using the same route for all the tuples.

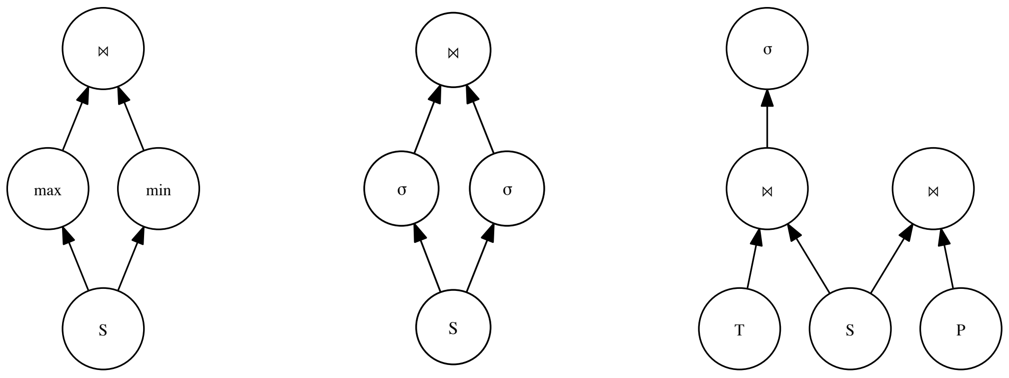
The eddy operator serves as a link between the input data and the operators. Tuples that enter eddies are associated with a tuple descriptor containing a vector of Ready bits and Done bits [4]. Eddies only route tuples to operators whose ready bit are on, and when all done bits are turned on, the tuple is sent to the output.

Moments of symmetry The Eddy operator assumes that the operators used in the query can be reordered without any consequences. Algorithms that are asymmetric, such as nested-loop joins can only be reordered when it reaches its synchronization barrier - that is, when it has reached the end of a scheduling dependency where the ordering of the tuples won't affect the result. Dependencies are present in operators that keeps state, such as join operators and aggregators.

Choosing the route The choice of route is affected by their consumption rate and production rate. The production rate is determined by cost and selectivity. The operators with high efficiency (high consumption rate and low production rate) gets tuples early in order to reduce the number of tuples and avoid bottlenecks. Eddies can be implemented with a learning algorithm such as Lottery Scheduling[42].

3.2.3 Common sub-queries

A idea of finding common sub-queries among a set of queries are presented by Roy et al. [36]. This strategy basically tries to reduce redundant computation time by reusing results of a sub-query that two or more queries has in common. They address the performance



(a) Inter operator parallelism (b) Intra operator parallelism (c) Inter-query parallelism

Figure 3.2: Parallelism approaches. The figure illustrates how the different parallelism approaches can be applied in a query execution plan setting. Inter-operator parallelism: max and min are two independent query operators which can be run in parallel and later be joined together to form the end result. Intra-operator parallelism: A query containing a cloned selection operator, where better performance is achieved by partitioning data between each operator. Inter-query parallelism: Two independent queries that can be executed simultaneously.

issues of earlier exhaustive and inefficient algorithms by introducing a cost based algorithm based heuristics.

3.3 Distributed query optimization

3.3.1 Parallel query execution

In the literature of parallel query execution there are three main approaches for achieving parallelism in a multi-query system. The first two are introduced by Ganguly et al. [19]. First we have *Inter-operator parallelism* which is basically two or more independent operators (or sub-expressions) that can be executed in parallel. The second is *Intra-operator parallelism* which is when an operator or a sub-query can be cloned and run in parallel. The third is introduced by Chen et al. [13] and is termed *Inter-query parallelism*, it is achieved by executing several queries simultaneously within a multiprocessor system.

For Intra-operator parallelism one can apply horizontal partitioning [8] on the source data. The idea behind horizontal partitioning is to split the source data and execute each subset of the source simultaneously, and it can be explained as follows. Let S be the source of tuples, and let S be split into n subsets S_i where $i = \{1, 2, ..., n\}$. Then execute each subset of S in a sub-query plan P_i, where S_i is executed on P_i. However, it may also execute each subset of S or in i cloned query plans in parallel [26].

In addition, one can route tuples through a query plan optimized for a particular tuple classification to obtain an overall better query execution performance. More formally, let $C = \{C_1, C_2, ..., C_m\}$ be the different tuple classification of S, $P = \{P_1, P_2, ..., P_m\}$, and let P_k be the current running plan. If for any classification C_j there exists an optimal query plan P_i for tuples in C_j for $0 \leq i, j \leq m$ then there exists a horizontal partitioned query plan that performs more optimal than P_k.

Another approach proposed is to run two or more different query plans in parallel, and at some point choose the best query plan based on collected statistics. By measuring their performance we can decide on the plan that is more optimal. However, such an approach may not be cost-efficient if the cost of running several non-optimal query is more expensive than pre- or progressively computing the same query plan.

Also, Ganguly et al. [19] mentions that data dependencies between operators and resource contention as two deterrents for parallelism. The former deterrent brings to light the issue of operators dependent upon the output data of a former operator. That is, an operator O_j that is dependent upon another operator O_i must wait for O_i's output before it can perform data processing on its own. This mainly limits the *Inter-operator parallelism* principle. The latter addresses the issue when two operators are running in parallel and are dependent upon the same resource. In such a scenario the two operators can create a bottleneck by competing for the same resource.

The concept of avoiding *low utility plans* on query optimization was introduced by [43]. If a query plan cannot fully utilize the parallel computing system it is by definition a low utility plan. In other words, if a query plan uses unique keys to partition data among the workers of the parallel computing system. Then a query plan that has low utility if it has fewer unique keys than parallel workers. The idea can be formally presented as follows. Let T_1, T_2, T_3 and T_4 be three tables in the following query:

$$((T_1 \bowtie T_2) \bowtie T_3) \bowtie T_4 \tag{3.3}$$

where T_3 and T_4 joins on some key k for some query plan. Let the cardinality of k be $|k| = n$, and the current available parallel workers to be m. Then a query plan has *low utility* if and only if $n < m$.

3.3.2 Distributed queries

In a centralized database systems, optimization focuses mainly on the ordering of operators, such as joins and filters. When dealing with distributed queries over data streams, one needs to consider multiple other factors that does not concern centralized database systems. In this section we will discuss these factors.

In a distributed database systems several queries might be running simultaneously. The placement of nodes in the network, the physical link between them will affect the overall cost of the query, as physical distance between nodes have a major influence on the cost. Example scenarios where different optimization techniques can be used is suggested by Seshadri et al. [37]:

Unique operators

Optimal plans needs to take into account both the query plan and the actual network placement of nodes. In order to decrease network communication, data-decreasing operators should be placed near the source, and data-increasing operators should be near the sinks. By putting data-decreasing operators near the source, we reduce the amount of data that needs to be processed later on in the tree, which leads to better performance.

Operator re-use

Several queries might use equal sub-operations. By reusing these operations between the queries the response time might go down. One has to consider plans which might be sub-optimal in a single-query environment but not in a multi-query environment as the end result might be better by re-using the operator. It is worth to mention that by re-using operators we might have to project more data, as the queries sharing the operator can project different attributes. Projecting more data will increase the network cost. One major factor to whether or not re-use is better is physical node placement. Let c be a cluster with $n = 5$ nodes. Let n_1 and n_2 share the same sub-query. If the nodes in the cluster are separated over significant distances, at opposite ends of the network, then re-using the operator might lead to a higher cost, as the cost of sending those tuples over the network topology is expensive.

Operator duplication

Operator Re-use might not be efficient in all situation. The nodes in the distributed system might be separated over long physical distances, and one needs to weight up the gain from re-using the operator vs the network cost, to figure out whether it's cheaper to re-use or duplicate the operator.

Delayed filtering

Consider a case where two queries share the same join operator, and one unique filter each, and the filters are different, and their selectivity rates are low (e.g 1%). It might seem like doing filtering early is optimal as it will reduce the amount of data sent to the join operators. But if you apply different filters you will not be able to reuse the join operator. In a situation like this, one needs to consider the benefits vs the cost of the operators.

When optimizing distributed queries over data streams, we need to consider the combination of execution plans and actual placement of operators, sources and sinks [37]. One needs to find the optimum over two conflicting objective functions, minimize the costs and minimize response time between sources and sinks. Where the reduction of cost can be achieved by reducing each cost component individually, and the reduction of response time comes from trying to do as many things in parallel as possible.

3.3.3 Parallel distributed query execution

By combining techniques used in both parallel and distributed query optimization one can achieve a better performance by executing a query using the parallel query execution technique over multiple nodes in a distributed system. Since the system is distributed one also has to take into account the techniques presented above for distributed query execution.

The main challenge is to decide which technique, or combination of techniques that will yield the best results.

3.4 Cardinality estimation

There exists several ways to estimate the cardinality of queries, and the problem has been widely researched. When deciding on a query plan for a given query, the estimates of the cardinality for the different parts of the query are used as a base, and the ordering of operations in a relational algebra tree will have a huge impact on the response time. As described in the introduction to query optimization, the order of operations affect the cardinality outputted from each operator, which in turn affect the response time. See Figure 3.1.

3.4.1 Histograms

Histograms are used in traditional databases for approximating the frequency distribution of values in the attributes of relations, which is used to estimate the size of the query result [24]. For each attribute, the histogram stores the number of tuples in that relation that correlates to it, different attributes are stored into different subsets (buckets). Building and maintaining histograms is expensive. You will have to do several read operations on existing data to get satisfying results, and the computations might equalize the gain you get from using histograms.

3.4.2 Self-tuning histograms

Ioannidis and Poosala [24] suggests self-tuning Histograms. The histograms are build not by examining the data in advance of the execution, but by using feedback information about the execution of the queries on the database. This way, you avoid extra read operations on the data, as you do the execution and histogram-update in the stream operation. When a query uses the histogram, it compares the estimated selectivity with the actual selectivity, and refine the histogram based on the results [24]. This way, you negate the initial cost associated with building traditional histograms. Self-tuning histograms needs to know the required number of buckets, the number of tuples in the relation and the minimum and maximum values of the attribute. However, knowing the minimum and maximum values are not critical, and can be estimated. The histogram is updated for every selection based on the estimation error.

Refining bucket frequencies Each bucket contains the frequency of the range of attributes it contains. At every histogram update, the bucket frequencies must be refined. To do this, we calculate the estimation error from the estimated selectivity and the actual selectivity. The main problem is distributing the 'blame' between the buckets that were involved in the calculation, that is, which attributes contributed the most to the estimation error. Ioannidis and Poosala [24] uses a heuristic where buckets with higher frequencies contribute more to the estimation error than buckets with lower frequencies. The amount of refinement the bucket need is based on the fraction of the bucket that overlaps the selection, the frequency and a damping factor to avoid adjusting to much.

$$frac = min(rangehigh, high(bi)) - max(rangelow, low(bi)) + \frac{1}{high(bi)} - low(bi) + 1 \tag{3.4}$$

$$freq(bi) = max(freq(bi) + \frac{damp \cdot esterr \cdot frac \cdot freq(bi)}{est}, 0) \tag{3.5}$$

We refer the interested reader to read the paper [24] for an explanation of the algorithms.

Restructuring buckets Having buckets with large frequency variations will result in a poor approximation, because the average frequency will be a poor approximation of the actual attributes. To solve this, buckets with large variations are split into several buckets. In order to keep a constant number of buckets, we apply a mechanism to merge other buckets together. Ioannidis and Poosala [24] purposes to restructure buckets every R selections. To select which buckets to merge, we set a merge threshold for the frequencies, and merge those buckets who are under the threshold. Likewise, to select which buckets to split, we set a split threshold.

3.4.3 Sampling

Random sampling is another method used to estimate cardinality. The advantages of sampling is that it needs to store less data compared to histograms. This is especially important in when the total data size is unknown, such as with data streams [41]. Another advantage of sampling compared to histograms is that it takes less time to compute the estimates as the data sizes are smaller. Sampling maintains a specified size k tuples that have arrived in the past, where the sample acts as a summary of the whole data set.

3.4.4 Reservoir sampling

Reservoir sampling [41] have traditionally been used in cases where insertions and updates of the sample sets are needed, as this procedure is too expensive to do in the normal sampling methods. Reservoir sampling is not however suitable for deletions as one must do a quite expensive scan over the data set [6]. It works by putting the first k tuples that are processed into a 'reservoir', which is basically a collection of tuples. Afterwards, tuples can be marked as candidates for replacing the existing reservoir tuples. The candidates are selected in a random fashion. One way to select tuples for the reservoir is to give each a $k/(t+1)$ chance of being selected, where t is the number of tuples processed so far. The tuple to be replaced is also selected on random.

The previously explained method does not handle expired tuples. In applications that execute queries based on tuples stored in windows, the traditional reservoir method does not work. Babcock and Chaudhuri [5] suggests keeping a reservoir sample of k tuples, and when new tuples arrives it causes an existing older element to expire. The downside of this method is that we end up with a periodic tuple list with only never tuples, as all the old ones have expired.

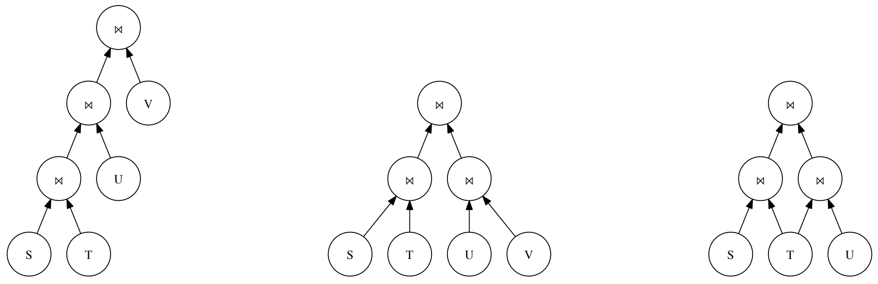
3.5 Query plan structure

Queries are represented as relational algebra trees. These trees can have different internal structures. The structures are mainly left-deep or bushy. Left-deep query plans are the subset of all possible query plans where the inner relation of each join is a base relation, i.e the inner relation is a leaf node [39]. Bushy query plans are all possible permutations of the base relations. The main difference between the plans are the search space. Let n be the number of relations in the query. The number of possible plans in the left-deep query plan space is $n!$. For the bushy plan space, it is $\binom{2(n-1)}{n-1}(n-1)!$, which is substantially larger. In traditional relational databases, left-deep queries are the preferred choice [43]. It simplifies the pipeline process, since at least one data-source is a base table, and even though the search space is smaller, good solutions are likely to exists.

For distributed systems, bushy plans might prove more useful. In addition to having a larger search space, and is therefore likely to find more optimal plans than in in the left-deep space, it also lets you exploit the parallelism of distributed systems. One of the downside in traditional systems is that you have to materialize more tuples, which can be a benefit in distributed systems as more operations can be ran concurrently. Franklin et al. [18] shows that bushy plans gain more performance with multiple nodes in a cluster, versus the left-deep plans.

3.6 Join order plan generation

Join operations are very expensive operations, and are often a target for query optimizers. As join operators generate data, the order of the operators will influence how much data must be processed in total. See Figure 3.1 for an example. Due to the size of the available search space, (see Section 3.5), several methods have been proposed which both focus



(a) Left-deep tree (b) Bushy tree (c) Directed-acyclic graph

Figure 3.3: Figure shows the different data structures that a query can be represented as.

on limiting the search space as much as possible while still finding near optimal plans. Steinbrunn et al. [39] divides the methods in 4 categories:

- *Deterministic Algorithms.* Step by step algorithms that uses exhaustive search or applies a heuristic to the search to determine the order. An example is the greedy search algorithm that orders the join operators after their selection rate.
- *Randomized Algorithms.* The solution space is defined as a set of moves, where moves is a edge between two solutions. The algorithm does random walks along the edges according to some rules, terminating after a given time limit or to some other rule, where the goal is to reach a global or local maxima.
- *Genetic Algorithms.* Random search algorithms that uses strategies from biological evolution in the search.
- *Hybrid Algorithms.* Which is a mixture between deterministic algorithms and randomized algorithms.

We refer the interested reader to the paper by Steinbrunn et al. [39].

Chapter 4

Frameworks used

This chapter introduces the frameworks we build our research on. Basically, there are two frameworks that we use in our thesis. Storm, which is a open source distributed real-time data processing system, presented in Section 4.1, and raincoat, which is a framework built around storm to ease the use of it, presented in Section 4.2.

4.1 Storm

Storm is a free and open source distributed real-time data processing system [1] created by Nathan Marz.

Being a distributed real-time data processing system makes Storm the logical counter part to MapReduce. Basically, that means while MapReduce processes data collected, and then computes results offline, Storm continuously collects and processes data without going offline. To be able to process data fast Storm has to do computations in-memory thus effectively sacrificing the computed results accuracy. That means Storm is forced to compute approximations relative to MapReduce who will provide a developer with the exact result.

While MapReduce provides developers with tools computing exact answers distributively offline, Storm provides developers with an abstraction from the real-time data processing paradigm. So without worrying about how data is passed between workers, the distribution of workers on nodes, and setting up the system architecture a developer is able to easily start a processing job fast on a large amount of data in no time. Storm as a framework tries to deliver the following six key properties to developers,

1. extremely broad set of use cases
2. scalability
3. guarantees no data loss,

[1] http://storm-project.net/

4. extremely robust
5. fault-tolerant
6. programming language agnostic

In this section we will present the fundamental mechanics behind Storm including, but not limited to, its architecture, key concepts and parallelism.

4.1.1 Concepts

This section introduces several important concepts that is used in Storm, which we refer to later in the report.

Topologies

A topology is analogous to a MapReduce job, except that a topology does not finish. Topologies defines the logic of the application. A topology gets its data from streams, and the data get routed from Spouts to Bolts through the topology. In Storm, one can use topologies as input to other topologies in order to create more sophisticated applications and re-use existing logic.

Streams & tuples

Storm topologies operates on tuples. A tuple is a defined set of key-value pairs. A stream is an unbounded sequence of tuples.

Stream groupings

Storm uses stream groupings to define how that stream should be partitioned among the bolt's tasks. One can implement custom stream groupings in Storm, but there are seven built-in groupings. We list the five most relevant groupings:

Shuffle grouping Randomly partitions tuples across bolt tasks. This is used when you want a uniform distribution of tuples across the tasks, without any requirements of which tuple goes where.

Fields grouping Partitioned based on the fields specified in the grouping. Can be used when you want to aggregate over tuples, e.g. a word count.

All grouping Tuples are replicated across all tasks. It is simply a broadcast.

Global grouping All tuples goes to a single one of the bolt's tasks, specifically it goes to the task with the lowest id.

Direct grouping The producer of the tuple decides which task should receive the tuple.

None grouping A undefined distribution of tuples. Currently implemented as a shuffle grouping.

Local or shuffle grouping If the target bolt has one or more tasks in the same worker process, tuples will be shuffled to just those in-process tasks. If not, it behaves like a normal shuffle grouping.

Bolts & spouts

Topologies are made out of two components, bolts and spouts. A spout is the stream source in a topology. Spouts can read data from external sources, or simply read data from existing topologies. Spouts are able to emit several streams. Spouts can be configured to be either reliable or unreliable, see section 4.1.4. Reliable spouts will re-transmit tuples who fail to be processed through the topology. Bolts are the processing units in Storm, and they can do work such as filtering, aggregations, joins etc. Depending on the complexity of the work, you can divide the logic into several bolts. As with spouts, bolts are also able to emit several streams.

Tasks

A task in Storm is an instance of a spout or bolt that performs the data processing. All tasks gets assigned a ID that can be used to locate the task. Spouts and bolts in a topology gets executed as several tasks across a cluster.

Workers

Workers are JVMs (Java virtual machine) which executes a subset of all the tasks for the topology, so each node in the cluster can have multiple workers. Workers spawn executor threads that can run one or more tasks for the same component. The number of executors for a component must be less than or equal to the number of tasks [2].

Tick tuples

Tick tuples are special tuples that Storm can emit in order to introduce a sense of time in the topology. One can configure the tick tuples to be emitted at a certain interval. Then, inside each bolt we can check if an incoming tuple is a tick tuple. See the following code snippet:

```
Config conf = new Config();
conf.put(Config.TOPOLOGY_TICK_TUPLE_FREQ_SECS, 60);

if(input.getSourceStreamId().equals("__tick")) {
    System.out.println("Got a tick tuple here!");
}
```

[2]https://groups.google.com/forum/?fromgroups=#!topic/storm-user/VvXCG-TqMx0

4.1.2 Architecture

The main architecture of Storm can be divided into three parts, Nimbus, ZooKeeper and Supervisors. In this section we present the aforementioned parts of Storm.

Nimbus

The first part of the architecture is Nimbus, which is the master node. Its responsibilities are listed below:

- Receiving topology submissions.
- Distributing code/configs around the cluster.
- Launch a coordinator sub-process(and supervise it) for each topology.
- Monitor for failures.

ZooKeeper

Storm uses a ZooKeeper cluster in order to coordinate the Storm cluster. ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services[3]. ZooKeeper stores the cluster configuration and state, such as topology, supervisor and task information, heartbeats from supervisors and tasks, task statistics. Supervisors and workers get their configurations from ZooKeeper.

Supervisors

The last main part of Storm is the Supervisors. Each worker runs a daemon called "Supervisor" which is responsible for listening for work from Nimbus and start and stop worker processes based on the messages it receives. New workers are launched when a task is assigned to an empty slot. The supervisor downloads the topology jar files and configuration files from Nimbus and stores them locally.

4.1.3 Life cycle of a topology

This subsection describes the whole process of running a Storm topology on a cluster, and what is done in the background. Specific details about the cost is presented in Chapter 6.

To run Storm topologies on a cluster, you must create a jar file out of your source code, with a main method which uses the *StormSubmitter.submitTopology()* method.

First, you run the command *storm jar topology-jar-path class-name* from a commando line. This command runs the main method of the jar file, as well as set the storm.jar environment variable for use by *StormSubmitter*. 1. *StormSubmitter* uploads the jar if it is

[3]http://zookeeper.apache.org/

not uploaded before. 2. *StormSubmitter* calls submitTopology which submits the topology to Nimbus, along with a serialized version of the topology configuration.

Next, Nimbus sets up the static state for the topology, which includes:

- Copying the jar and config file on the local file system because it is to big for ZooKeeper.
- Writes task-to-component mapping into zookeeper.
- Creates a zookeeper directory for heartbeats.

Nimbus then assigns tasks to machines. An assignment contains:

- The path to the jar and config files in nimbus.
- A mapping from the task id to the worker that task should be running on. A worker is identified by a node/port pair.
- A map from node id to hostname. Used for communication within the cluster.
- A mapping of task launch timestamps, used by Nimbus to monitor topologies.

Next, nimbus runs start-storm which writes data into zookeeper so that the cluster knows that the topology is active.

Each supervisor runs two functions in the background:

- Synchronize-supervisor downloads code from Nimbus for topologies assigned to the machine for which it does not have the code yet. It also writes a map from port to localAssignment. LocalAssignment contains topology id and a list of task ids for that worker.
- The Sync-processes, which read data that synchronize-supervisor wrote and compares that to what is actually running on the machine. Then starts/stops worker processes as necessary to synchronize.

Next, worker processes start up, and the following steps are executed:

- The worker connects to other workers and starts a thread to monitor for changes.
- The worker monitors whether a topology is active or not and stores the result. It is used by tasks to determine whether or not to call *nextTuple* on the spouts.
- Lastly, the worker launches the actual tasks as threads within it.

Nimbus is also monitoring the topology when it is running. In the event of a reassignment to the topology, the supervisors will trigger its synchronize method and start/stop workers.

When the *storm kill topology-name -w wait-time-secs* command is run, Nimbus changes the status of the topology to killed and schedules the remove event to run in the specified *-wait-time secs* seconds. Removal of a topology includes clearing assignment information in zookeeper and removing the jars/configs from the machines.

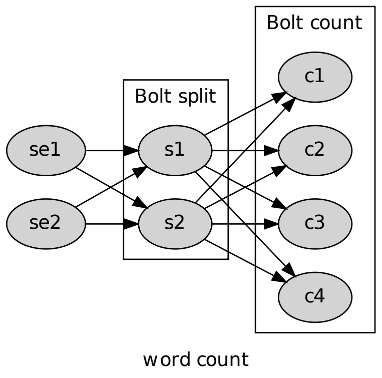


Figure 4.1: The word count topology with initial parallelism of two spouts, two split bolts and four count bolts.

4.1.4 Configuring Storm topologies

Below is an example of a topology in Storm. The implementation of the components RandomSentenceSpout, SplitSentence, WordCount are emitted. For a full implementation of the example code, see appendix A. *TopologyBuilder* is used for specifying a topology for Storm to execute. The given example is a simple implementation of a word count program. The spout "sentences" uses a input source who emits random sentences to the topology. The bolt "split" reads those sentences and splits them into words, and finally the count-bolt counts each occurrence of each word. It is worth to mention that *fields Grouping* is used on the count-bolt in order to route the same words to the same tasks in the cluster to obtain the correct result. See Figure 4.1 for an illustration of the topology.

```
TopologyBuilder builder = new TopologyBuilder();

builder.setSpout("sentences", new RandomSentenceSpout(), 2);
builder.setBolt("split", new SplitSentence(), 2)
    .shuffleGrouping("sentences");
builder.setBolt("count", new WordCount(), 4)
    .fieldsGrouping("split", new Fields("word"));
```

Configuring parallelism

The *setSpout* and *setBolt* methods have set the *parallelismHint* for their respective spouts and bolts to 2, 2 and 4. The *parallelismHint* sets the initial number of executors for the component. If nothing else is defined, the number of tasks will be equal to the *parallelismHint*, but you can set the number of tasks on a component with the *setNumTasks*-method.

```
builder.setBolt("split", new SplitSentence(), 2).setNumTasks(6);
```

By calling *setNumTasks* on the bolt, we change the maximum number of tasks for that bolt to 6. The number of executors will still be initially 2, but we are able to scale the number of executors up and down on the fly, to a maximum of 6 executors. To specify the number of workers, we can change the *Config.TOPOLOGY_WORKERS* field which sets how many workers a topology gets allocated. For the example above, we get a initial parallelism of 2 + 2 + 4 = 8 executors. Changing the parallelism must be done by the user, Storm does not scale parallelism automatically. The way it is achieved is by issuing a re-balance command to Nimbus, the master node. Re-balance will deactivate the specified topology for a specified timeout, and then change the parallelism and redistribute the workers evenly around the cluster. The re-balance command is on the following format:

```
    storm rebalance [topology-name [-w wait-time-secs]
    [-n new-num-workers] [-e component=parallelism]*]
```

Note that ZooKeeper does not support dynamic changes of the cluster size. That means that if you want to add new nodes to your cluster, they must already be defined in your storm.zookeeper.servers configuration when Nimbus is started. When you run up a supervisor on a node in the cluster, it will register itself with ZooKeeper. In order to utilize the node, we need to run the *rebalance* command, and Nimbus will take the new supervisor into account and redistribute the work. See the appendix for a more detailed explanation on how to configure a cluster.

Reliable vs unreliable topologies

Storm provides methods to guaranteeing that all tuples are processed in the topology. By default it will drop tuples that fails, which gives you at-most-once semantics guarantee for tuples. When you want a reliable topology, one needs to implement anchoring of tuples in the components in the topology. Anchoring gives you at-least-once semantics, which means that it guarantees that tuples are processed at least once through the topology. Failed tuples will be replayed until they are fully processed through the topology, but they might get processed several times. It is worth to mention that tracking tuples doubles the number of messages transferred in the system. In addition to anchoring, your input source needs to be reliable, that is, it must be able to replay messages in the event that they are lost in the Storm environment (e.g the nodes running the Spout crashes). A tuple is replayed when it fails to be processed through the topology before a timeout occurs. To specify that a topology should be reliable one anchors the input tuple along when you emit the output, and one must ack the tuple when the processing of it is complete.

```
        _collector.emit(old_tuple, new Values(word));
        _collector.ack(old_tuple);
```

Storm uses special acker tasks that track tuples through the topology. The number of acker tasks can be tuned in *CONFIG.TOPOLOGY_ACKERS*. The timeout is defaulted to 30 seconds, and is controlled by the property
CONFIG.TOPOLOGY_MESSAGE_TIMEOUT_SECS.

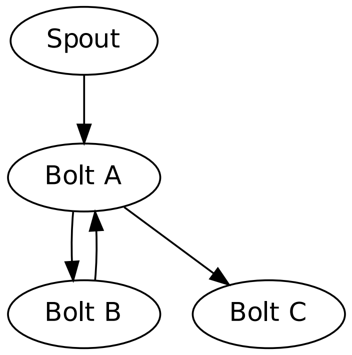


Figure 4.2: A topology that routes tuples in a cyclic fashion.

Transactional topologies

Transactional topologies lets you achieve exactly-once semantics in Storm. Storm achieves this by sending tuples batched up into the topology. So if a batch fails, that batch get replayed. The replay is possible by giving each batch its own unique id. Storms API handles the management of the state, it coordinates the transactions, it detects faults and handles the replay.

Cyclic topologies

If you have a topology where tuples are routed back to a component it already have visited in the topology graph, you have a cyclic topology, see Section 4.2. Cyclic topologies can cause performance issues. Storm has a maximum number of tuples that can be pending in the system (processing have started but not yet finished), and when it reaches that number no new tuples are sent into the system, which will lower the throughput. In addition, if the topology have operations that are waiting for more tuples, it can result in a deadlock. In order to prevent this, you can increase the maximum number of tuples that can be pending on a spout task, by setting the *Config.TOPOLOGY_MAX_SPOUT_PENDING* property. As this property applies to individual tasks, you can also increase the number of spout tasks in the topology. *TOPOLOGY_MAX_SPOUT_PENDING* only applies to reliable spouts.

Pluggable scheduler

Storm comes with a feature that lets you write your own task schedulers. The task scheduler decides how tasks get scheduled in the cluster. For instance, you are able to make sure a particular task runs on a particular machine, which can be very helpful if you want to reduce network cost. Also, it lets you separate CPU heavy bolts so they don't run on the same machine. With the scheduler, you can also prioritize topologies, to make sure a topology always get scheduled first. Using a custom scheduler gives you a lot of freedom, but also a lot of responsibility, as you have to make sure the tasks are scheduled in a way that utilizes the resources you have available.

4.2 Raincoat

Raincoat is a declarative scalable real-time data processing framework based on Storm [22]. Raincoat is designed to be scalable and modifiable, and to process high velocity streaming data using multiple distributed machines. It uses a custom SQL-like language for defining continuous queries for expressing real-time stream processing jobs declaratively. This stream processing jobs is executed on an already running Storm cluster, and does automatically utilize the parallelism made available by Storm.

In the Raincoat framework developers are provided with interfaces for easily extending the already existing functionality. Functionality such as adding new tuple sources, extension of the query language, and query optimization were made customizable.

Optimization was a main area of focus in [22]. Being able to execute long running continuous queries on a optimal query plan is essential for both performance and scalability of the framework. Both compile-time and mid-query optimization was brought into focus using some examples, but no explicit optimization solution was proposed.

4.2.1 Architecture

Raincoat is a framework built upon the Storm framework. Figure 4.3 gives an overview of the architecture. Raincoat comes with an API that is controlled by a client. The client submits queries to the API. Raincoat converts the queries to relational algebra, then to Storm topologies. Figure 4.4 gives an overview of how Raincoat is structured. The query butler receives the queries from the client, sends them to the parser, and passes the result to the topology controller. The topology builder builds the topology based on the input received and the available statistics. The statistics is used for optimizing the topology, and is collected by the query butler and stored in the Meta Storage module.

4.2.2 Query plan structure

A query plan is structured as a Storm topology, that mean that any running Storm cluster can execute a query expressed by Raincoats SQL. The processing plan modeled as a Storm topologies generated by Raincoat is structured as a directed acyclic graph (DAG). It is up to the optimizer to choose how a query plan is organized. It is here an optimizer makes the decision if two operators/sub-expressions are independent and should be ran in parallel, or not.

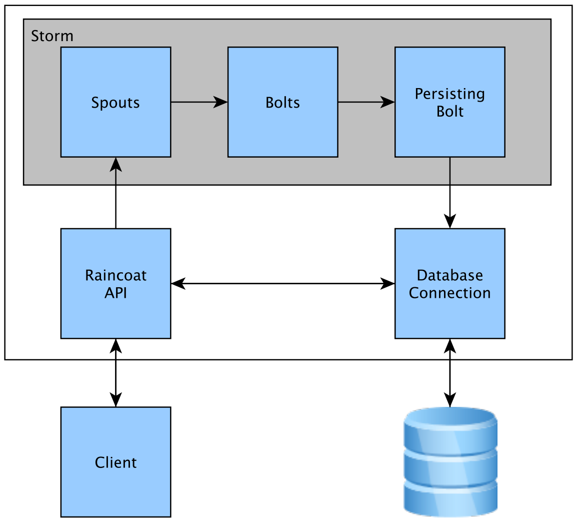


Figure 4.3: Shows the overall structure of the system, the arrows denotes a directed communication path. It shows the intended communication paths.

4.2.3 Adaptive query optimization

An idea of adaptive query optimization in Raincoat was proposed in [22]. Basically, the idea was to poll statistics from each operator of a running query, and periodically check the statistics if a topology was structured sub-optimally. If the currently running topology was sub-optimal it would stop the current one and replace that one with a new and optimal topology. However, a step by step description of how to adaptive migrate from one query plan to another was not in the scope of that project, but Raincoat is designed such that an optimizer easily can be implemented into the system.

4.2.4 Language syntax

Below is the formal syntax of the language used in Raincoat. It is presented on the Extended Backus-Naur form (EBNF)[4].

```
command = <create> | <select>

create =
    CREATE STREAM|STATIC <name>
    WITH FIELDS <name> (, <field_name>)*
TYPE <name>
[OPTIONS <name>=<name> (, <name>=<name>)*]
```

[4]ISO/IEC 14977

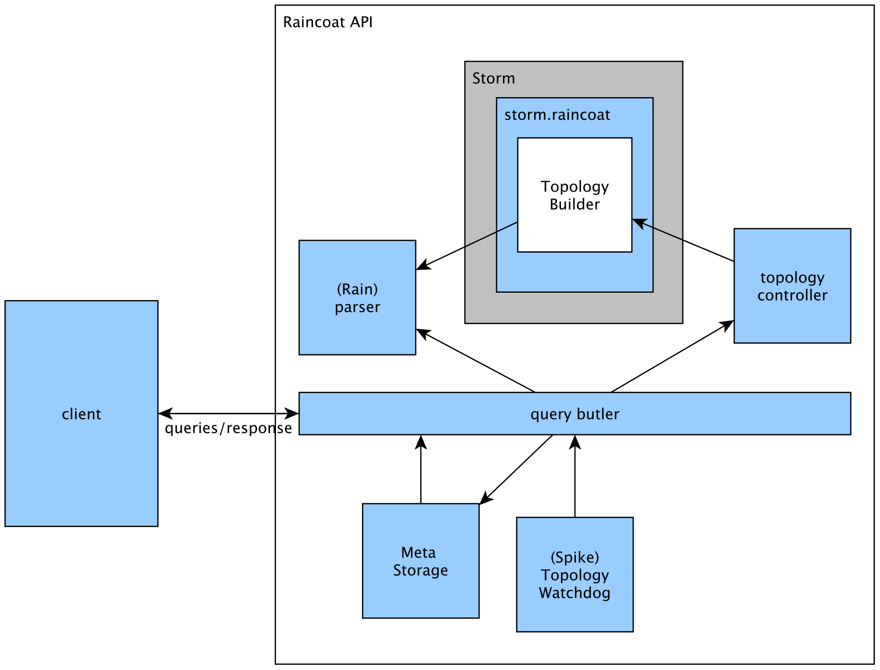


Figure 4.4: Shows the main components of the Raincoat API. The purpose of each component is explained below.

```
    ;

select =
    SELECT <select_expr> (, <select_expr>)*
    FROM <name> (, <name>)*
    [WHERE <where_condition> (, <where_condition>)*]
    EVERY <number> SECONDS|TUPLES
    SIZE <number>
    ;

select_expr = (name | AGGREGATOR)

where_condition:
    <condition> (AND|OR condition)*

condition =
    <name>["<"|">"|"!"]=<name>

name:
    (a-z, A-Z, ', ", .,*, '_', '-')*
```

```
number:
(0-9)*

AGGREGATOR:
    SUM(field_name) |
    AVERAGE(field_name) |
    COUNT(field_name) |
    MIN(field_name) |
    MAX(field_name)
```

As you can see from the query syntax, Raincoat only supports select-project-join queries its the current state.

4.2.5 Similar work

Similar approaches has been researched on implementing SQL for MapReduce [11] [30], and Pig [34] was commercial SQL approach for Apache Hadoop. These approaches has shown that the SQL-like approach for data processing system is preferred by many developers [34]. Thus, Raincoats tries to meet a need of being able to express real-time stream data processing jobs declaratively.

Chapter 5

Design decisions

In this chapter we discuss several challenges that we had to address, and we give the reasoning of architectural design choices and implementation choices we have made such as message semantics (Section 5.1), and operator implementation choice (Section 5.2), resource usage (Section 5.3), how to preserve state in operators between optimizations (Section 5.4), and memory usage (Section 5.5). We also present discussion on topics that are not implemented in the current version of Raincoat, but are relevant for further work, such as how to connect topologies together to create more complex queries (Section 5.6) and how to join static data with streaming data (Section 5.7).

5.1 Message semantics

In Section 4.1 we give an introduction to how at-least, at-most, and exactly-once message semantics are implemented in Storm topologies. In this section we will look into the pros and cons of supporting the different message semantics for the framework in the context of optimization.

The main consideration when choosing message semantics is the trade-off between performance and accuracy. Using at-most once messaging semantics one can process data at full speed without any regards for accuracy at all. Using such semantics we can calculate the drop rate of an operator but not determine if any tuples were lost, and thus there is no way one can recomputing state in the event of failure.

At-least once messaging semantics trades some performance for accuracy. In this messaging semantic tuples are marked as done through acking and anchoring. This process increases the network usage therefore sacrificing some performance for accuracy, as each tuple produces an extra message whenever a tuple is acked. However, the system gains knowledge of what tuples has been completed and is therefore able to replay tuples that failed so they can be fully processed through the topology. If order matters tuples can be replayed from some snapshot, and if not, single tuples can be replayed at any point in time, assuming that the input queue used is re-playable.

Exactly once messaging semantics will increase the complexity of the query migration process, since state has to be fully transferred from one topology to another before the newly started topology can commence processing data. This state includes both the state of each tuple in the topology, as well as inner state of every operator inside the topology. Typically operators such as aggregators and joins contains inner state.

For at-least once messaging semantics, we also need to transfer some state, but one does not have to worry if a tuple has been fully processed, thus it reduces the migration complexity. In this case, a tuple that is cached in a aggregating operator might be calculated several times.

Many applications of real-time processing of big data requires immediate answers, and can tolerate approximations rather than exact answers. Basically, we want to support applications that requires that all data gets processed, such as motoring applications, but can tolerate some errors (duplicated calculation of tuples). With the argument given above we choose to support at-least once semantics for our framework.

5.2 Operator algorithms

An important subject of query processing is the choice of operator algorithms. There exists several algorithms and we need to make sure that the choice of algorithms does not introduce any bottlenecks.

5.2.1 Join algorithm

The queries defined by Raincoat are contained inside defined windows, and the data that will be joined is the data currently saved in the window. The general joining of streaming tuples can be modeled as follows:

1. A new tuple enters stream *A*. 2. Scan stream *B*s window for matching tuples, propagate the result. 3. Insert the new value into stream *A*s window, and invalidate expired tuples.

Choosing the right operator for the right situation can give significant performance improvement [28]. However, doing so is an entirely different scope than the one we have in this thesis. We choose to use the one-way hash join operator. The optimizer we propose can be extended to support operator optimization, but it is listed as further work. We refer the interested reader to [28]. The hash algorithm works as follows:

Algorithm 1 One-way Hash Join

```
for each tuple $a \epsilon A$ do
    put $a$ in bucket $k = h(a.A)$
end for
for each tuple $b \epsilon B$ do
    for each tuple $a$ in bucket $k = h(b.B)$ do
        if $a.A = b.B$ then
            output $a \bowtie b$
        end if
    end for
end for
invalidateExpiredTuples(A)
```

5.2.2 Windows

In Section 2.4 we introduce the concept of windows in the context of streaming data. For raincoat we have implemented two different time-based sliding window. One implementation of the window uses a list as basis, and while the other is using a hashing structure. The former is used for operators that iterate over data, such as aggregators. The latter is an optimized version for operators using hashes, and is used in our implementation of hash joins.

The window divides its tuples into n buckets, where $n = window\_size/tick\_rate$. Let $t1$ be the time a arbitrary tick tuple enters the system, and $t2$ be the time the next tick tuple after $t1$ enters. All tuples that enters the window between $t1$ and $t2$ are put into the same bucket. p is a pointer that points at the oldest bucket. At every tick tuple, the operator holding the window iterates over all tuples in all buckets, and emits the result. It then invalidates the bucket p points at, removing all those tuples from the window.

Algorithm 2 Time-based Sliding Window

```
$n \leftarrow \frac{window\_size}{tick\_rate}$
$buckets \leftarrow createBuckets(n)$
$p \leftarrow 0$
procedure EXECUTE(tuple)
    if $isTick(tuple)$ then
        $executeOperation(buckets)$
        $invalidateExpiredTuples(buckets[p])$
        $p \leftarrow (p + 1) \bmod n$
    else
        $buckets[p].addTuple(tuple)$
    end if
end procedure
```

5.3 Network resources

When running Storm in a cluster, each machine needs to start a daemon called supervisor in order to take part of the cluster, and the machines must be registered in the Storm configuration file. Because the way Storm is designed, once the Supervisor is started, one can not configure whether to use the machine or not, it will be part of the Storm cluster. Because of this, including the cost of using the machines is omitted from the optimization phase as Storm will use all the Supervisors that is running. That is, we will always use all the resources available.

This design choice is based on the work that has to be done in order to start up supervisors and register new machines runtime. In order to control the cluster size, one would need to first gain access to the machines (via ssh) in order to start up the supervisor. Given that this machine is already registered in the distributed configuration file, all that is left is to restart the topology in order to make use of the new resources. In the case where the machine is not registered, one needs to distribute a new configuration file across the cluster with the ip-address of the new resources added to it. The process of doing this automatically is out of scope of this thesis and is marked as further work.

5.4 State transition

In the context of transferring state between topologies the topic of maintaining accuracy needs to be discussed. First, in Storm it is recommended that one transfer state by replaying tuples into the topology from some specified point in time. Other transition techniques must be implemented outside of the functionality provided by the framework.

Having accurate results or not is a trade-off decision made by developers. Therefore the optimizer must take a developers decision on accuracy into account before applying any optimizations on a running query. To determine if state transition is suitable or not we need to look into the different query scenarios. In some scenarios state transitioning might be so hard or time consuming to be beneficial. These are scenarios where topologies has to be restarted with resubmitting all tuples passed through a topology.

Replaying tuples Strategies for partial restarting queries in a distributed database management system is presented by Hauglid and Nørvåg [23]. They introduce an interesting way of thinking about the restart of relational algebra nodes. They categorize operators into two categories: stateless and stateful operators, and propose two different techniques for resuming an operator without losing state.

In a stateless operator, e.g. selection and projection, each tuple is processed independently. Basically, after a restart simply continue processing the remaining tuples after the last processed tuple. That is, let t_i, t_j be the i, j-th tuple in a tuple table, where $0 < i, j < n$. If the last processed tuple is t_j for some stateless operator, then the stateless operator will continue processing from t_{j+1}.

However, the result of stateful operators, such as join and aggregators, will be affected by the tuples held in-memory. Since all tuples are dependent upon each other in such an operator. They argue that for such operators all tuples inside the state of a failing or restarted operator has to be sent to the new operator. On restart Hauglid and Nørvåg [23] proposes to avoid re-sending tuples already sent by the operator. They do so by sending the new operator a number determining how many tuples that was received, and thus the new operator discards as many tuples.

In the context of Storm, continue processing in stateless operators after a fail or restart is straightforward, and is compatible with the replaying tuples scheme recommended by Storm. On the other hand, we have the stateful operators, these techniques cannot directly be implemented in our system. Considering state transition for stateful operators in the context of distributed stream data processing, we found a few different cases where simply replaying tuples would not be cost efficient for replicating state.

One such case occurs when pipelined and windowless aggregators such as maximum, minimum, summing, and finding average is used. Such aggregating operators is not performed as a batch process, but eagerly calculates and outputs results for each tuple arrived. Therefore, trying to recompute state by replaying tuples into a new topology is not cost efficient.

Moving state An approach is to apply the *Moving State Strategy* proposed by Zhu et al. [44]. This strategy transfer state by connecting and transferring operators state of an old query plan to its equivalent operator in the new query plan. However, transferring state between operators of different topologies are not supported in Storm, and implementing such functionality is out of scope of this thesis.

Saving state Another strategy is to save state somewhere outside of the run-time memory. Either persistently on the same node, on the same cluster or on another distributed node, where the new topology can read a serialized version of the state from disk to memory.

Choosing a strategy The three different state persisting strategies is basically a trade-off between performance and availability. Where we can use the first method to gain performance and moving to the last by sacrificing performance for availability. Note that a strategy for how often state should be persisted is also an issue. Persisting state too often will reduce the performance of a node either by accessing disk or using networking resources. Where the latter can result in lower overall performance since the system is a distributed system dependent upon scarce network resources.

Also, when choosing a state transition strategy one must take the type of query into consideration. In our case the query types continuous and ad-hoc queries must use different state transition strategies. For continuous queries one can argue that some aggregator operators do not need state transition. Operators such as average, max, min may over time converge to the same value as before over time. Also, if data is stored persistently divergent results can later be corrected by post-processing jobs. A strategy has to carefully consider the options available and choose the most cost efficient one.

In the case of ad-hoc queries the cardinality of data is finite and known. So ad-hoc queries will terminate after data sent from a static resource has been fully transmitted. In such a scenario performing run-time optimization might not be a good strategy, since the accuracy will be greatly reduced if a query is not running for a longer period of time. Operators such as the pipelined and windowless aggregating operators are examples of very state dependent operators. If such operators lose their state in the middle of a short running query, the accuracy will be reduced greatly. So the only state transition strategy that will make sense for ad-hoc queries will be to replay all previous tuples. As discussed above this might not be cost-efficient, thus, performing optimization that requires state transition does not seem to be beneficial.

With the information given above we choose to replay the tuples that has not yet been fully processed.

5.5 Windows & memory

The hardware on the servers in the cluster Raincoat runs on is assumed to be heterogeneous. That is, we can not make any assumptions on the size of the memory on the computers used in a Storm cluster. This becomes an issue when we need to store intermediate state in order to do operations such as aggregation over data. As the data coming into the system might be an infinite stream of tuples, we need make some limitations on how large the windows can be in order to prevent memory problems. This is a minor limitation to the system because you will not get the same accuracy as you would if you could create windows of any size, but the limitation does prevent the need to use database operations to save intermediate data in the cases where the memory is used up. If we were to save intermediate data to a database we would introduce overhead which would result in slower operations. Even though we limit the window size, if the user wants to aggregate over larger sets of data, they can aggregate over several window results returned by the system.

We do not propose a specific limit to the windows size, but assume that the window sizes given by the user will not result in memory issues. Figuring out a accurate limitation to the window size is out of scope of this thesis.

5.6 Connecting topologies

One of the features of Raincoat and Storm is the ability to use results from sub-queries to as the base to new queries. In this section we discuss two different ways for supporting this. One can either merge multiple topologies together to one large topology, or one can keep them as two separate topologies.

5.6.1 Merging sub-topologies

When merging multiple sub-topologies together into one topology, there are a couple of things you need to keep in mind. First, you need a way to get the result of the predecending sub-topologies to the end-user. This can be done by sending the result of each sub-topology directly to the source of a subsequent topology, and to the end-user through a message broker.

The benefit of this method is that you are able to optimize the topology as a whole, as opposed to only a sub-topology. This expands the search space, and we might find better plans. However, one needs to make sure when optimizing that the result of the original sub-topology doesn't get changed in the optimization phase.

5.6.2 Separate topologies

When one want to keep multiple topologies separated, one still need to send results through a message broker. However, instead of sending the result directly to a subsequent topology. The source of the subsequent topology listen to the message broker. But by keeping topologies separated one get more flexibility since you can run different queries on different machines, but it might add extra overhead since data between topologies has to be passed through a message broker, adding extra layer of complexity. Figure 5.1 gives an illustration of the different strategies.

Based on this, we choose to keep topologies separated. The reason being it is easier to reason over the topologies and we avoid having to keep in mind not to change the result of the sub-query which we have to if we merge the topologies together.

5.7 Joining static data

Even though Raincoat and Storm are built for data streams, there exists some use cases for when you want to join stream data with static data. One such example is when geotags from tweets needs to be joined with statically stored geo data. Such as finding cities corresponding to a geotags longitude and latitude.

There are several of approaches for joining streaming and static data. Many of them are simple but may be too naive to use in a large scale streaming application. Thus, one has to discuss some of the different implementation approaching to avoid unnecessary inefficiencies. An ideal static join operator should be scalable and not introduce unnecessary latency. That is, it should be highly parallelizable and fully perform join operations with in-memory data avoiding I/O latencies.

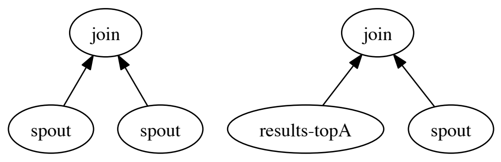
The naive approach To simplify this discussion we assume that all static data does not change over the lifetime of a running query/topology. We start by presenting a naive approach. Such an approach would be to query a database for each incoming tuple. This approach is simple but notice that it does not scale, and it is prone to network latencies.

It does not scale because complex queries with several static join operators, or a high tuple input rate will increase the the load of the single static database and the network causing an unavoidable bottleneck. Even without a bottlenecked database this approach will introduce network latency from querying the database for joinable static data.

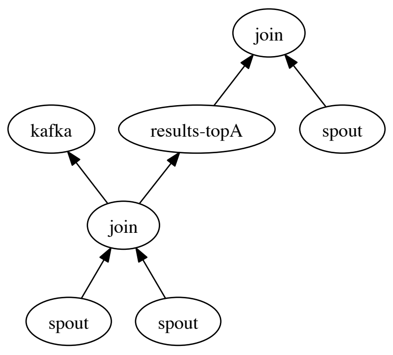
Caching and preloading Clearly, the approach given above was too naive. However, some modifications can be added to improve the performance of the current approach. Introducing caching of table rows will over time reduce the network latency introduced by querying the database. The modified version of the current approach would perform well if the cache hit rate is large. If the hit rate is low then it would be far better to load the whole static data table into memory. Of course this approach assumes that the table fits into memory. Since one only has to query for the table data at setup time the last modification will completely remove the network latency, and potential of bottlenecked databases.

This approach of loading the whole static data table into memory can be solved in the following way. Say we have a 1 GB static data table that we want to join with streaming data. Assume that the workers in our Storm cluster only has 200 MB of memory each. Then by splitting up the static data into more than five parts would reduce the number of table rows stored in-memory for each worker. Not only have we distributed the data successfully but we have also parallelized the static join operation. If an uniform hashing function is used to partition and distribute the data among the workers then the load on each operator will with a high probability also be uniformly distributed.

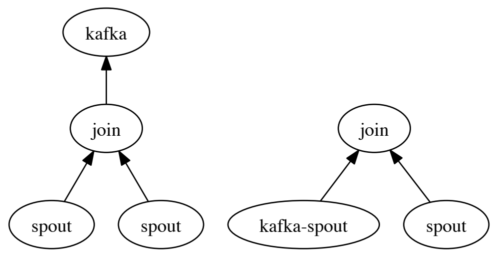
In Raincoat we use the third approach presented for performing static joins because of its simplicity while still being highly scalable assuming that the static data table is small.



(a) Two separate topologies.



(b) The topologies merged into one topology. The result from the left topology is outputted to a message broker.



(c) Keeping the topologies separate, communicating via a message broker.

Figure 5.1: An illustration of the two possibilities we have for connecting topologies. (b) shows you two topologies merged together. (c) shows you the same topologies kept as two, where the right topology gets the output from the left topology via a message broker.

Chapter 6

Query optimization

In this chapter we propose a query plan optimizer for the distributed real-time data processing system Raincoat [22]. The optimizer optimizes select-project-join queries, and focuses on maximizing throughput and keeping the response time within the user-specified time-frame.

We begin with describing the main challenges and the problem space for the optimizer in Section 6.1. In Section 6.2, we design a cost model for the queries to use as a basis for the query optimization and it is also used to reason about the effects of the optimization techniques. Further, we look at the costs associated with swapping running topologies, that is, taking down a running topology and uploading a new, optimized version, in Section 6.3. The adaptive query optimizer needs to decide when it is reasonable to do optimization, and our proposed heuristic is presented in Section 6.4. Section 6.5 presents the optimization techniques we have implemented in the system. Each optimization is designed to be independent of other optimizations, to make it easy to change the algorithms used. In the final Section 6.6, everything is tied together to form a adaptive algorithm.

6.1 Challenges and problem space

For a distributed realtime data processing system to perform optimally at all times can be though. It is so because of all the different requirements such a system can meet. Some of the requirements that distributed realtime data processing systems must meet are fault tolerance, data consistency, soft or hard realtime constraints, scalability, availability, high throughput, low response time, etc. Where many of these constraints can be in conflict with each other. In effect, when designing a distributed data processing system one has to make certain sacrifices to meet the requirements needed for the system at hand.

6.1.1 Optimizer requirements

When designing the adaptive optimizer for Raincoat, we have focused on maximizing throughput and keeping the query response time within a user-specified time frame. The user-specified time frame is the tick-rate specified in the query. The challenge here lies in the systems ability to handle increasing window sizes (increase in the number of tuples stored in the window), as the operators in the topology needs to operate on that data at each tick. Formally, our query optimizer have the following requirements:

1. Queries submitted to the system specifies a rate at which data should be outputted to the user. The system must be able to compute the answer within that time frame.
2. The system must be able to scale with increases in resources (nodes in the cluster) to handle increases in the rate at which the data arrives.

6.1.2 Challenges

There are several challenges that needs to be addressed when designing the optimizer, which are described below.

- **Time of optimization**. It is hard to determine at what time one has to perform query optimization. Only performing query optimization at compile time will reduce the overall cost of optimizing. However, the classification of incoming data tuple may change over time therefore an optimization calculated at compile time may not be optimal for the lifespan of a continuous query. That introduces the idea of runtime optimization presented earlier. However, questions such as when should such an optimization occur? How much data loss is acceptable? How long down time is acceptable for performing an optimization? These questions has to be answered before choosing a model for at-runtime query optimization.
- **The cardinality of data**. The size of input data is unknown, and may possible be infinite. The nodes performing calculations can run out of memory over the span of time if the input rate is high enough. In that event, what subset of incoming data should be used for calculating the results? What approximation techniques should be applied?
- **Unstable input rate**. Data is not received to the system continuously. The system must be able to handle bursts of incoming data. One has to chose between the conflicting requirements of high throughput and low response time.
- **Distributing work vs in-node computing**. We need to consider the gain of distributing work over nodes against utilizing the computing power of each node, mainly figuring out the amount of work a node can do without affecting the throughput.

6.1.3 Problem space

The problem space for our optimizer and the limitations we introduce are listed below.

- The optimizer optimizes select-project-join (SPJ) queries.

- The optimization is assumed to be used in environments with high tuple input rates, or on complex queries that takes considerable CPU-usage to compute, or both.
- The optimizer is designed to meet both soft- and hard real-time constraints enabling the user to configure the acceptable downtime of the system.
- We assume that approximations of answer to queries are good enough for the user. That is, our system will use at-least once semantics for computing data. The reason being the cost introduced by using exactly-once semantics. See Section 5.1 for a discussion on the subject.
- Operator optimization and which operator is optimal for the situation is out of the scope of this thesis. Our selection of operators is discussed in Section 5.2.
- The system should be able to handle large amount of traffic without sacrificing performance in terms of throughput.
- The system is assumed to be running on a cluster of nodes with heterogeneous commodity hardware, that is, hardware that is affordable, easy to obtain, and might differ in specifications.
- The input source of the data stream is assumed to be a re-playable message broker system, like Apache Kafka. A re-playable message broker system is needed to be able to restore the state of the system in the event of a system crash or if you want to swap topology plans.
- The system does not take resource cost into account, it is up to the user to define how many resources it want to use, which must be done before the system has started. See Section 5.3.

6.2 Cost model

In this section we present a cost model for query execution in terms of throughput. The cost model is used as a basis to reason about the cost associated with running a topology on a Storm cluster. Following is a summarized list of the costs taken into account when designing our cost model.

- **Order of operation**. The costs of running a query can be measured in the amount of data that has to be processed. From traditional query cost modeling we know that the order of operation have a huge effect on the overall cost of running a query. Taking selectivity into account is therefore an important aspect of the query cost model.
- **Distributed costs**. Since the system is distributed over several nodes we have to take into account and model network costs.

- **Parallelism**. Parallelism is a vital part of the system since it can increase throughput if introduced to a system. Thus, introducing parallelism to a query can reduce the cost (in terms of throughput).

The following subsections elaborates on on the aspects listed above.

6.2.1 Order of operation

As with traditional query optimizers we do also take operation selectivity into account when calculation the cost of a query. Similar approach has been discussed for drop-rates of stream filters in [7]. Changing the order of operations affects how much data is transmitted through the topology, and needs to be included in the cost model.

6.2.2 Distributed costs

Processing data over a distributed system with multiple nodes will introduce issues and costs related to networking. Issues such as network partitioning, congestions, race-conditions, latency and throughput will affect the cost of sending data in between nodes of a system. In our case the network throughout will affect our systems throughput. We can go as far as to say that the network throughput will be the lower bound of our system, thus, the cost model must reflect the network throughput for different topologies.

6.2.3 Parallelism

The last aspect of the cost model is parallelism. In a distributed system, the parallelism level affects the load the system can handle. If enough CPU resources is available, then executing the same task on several threads will increase throughput compared to running that same task on a single thread. Some techniques and issues related to parallel query execution was presented in Section 3.3.1. We propose a way to estimate the cost and cost reduction when paralleling operators.

6.2.4 Parameters

The parameters that we emphasize on in our cost model is shown in Table 6.1

Cost formula parameters	
CO:	Total cost
CO_i:	Cost of executing operator i
CO_n:	Cost of transferring a tuple over the network.
T:	Topology
i:	Operator $i \epsilon T$
C_i:	Child operators of operator i
N:	Incoming tuples
S:	Selectivity
p_i:	parallelism of operator i
k:	parallelism overhead

Table 6.1: Cost model parameters

6.2.5 Cost formula

The cost formula is broken down into three steps. Each of the steps adds new factor into the model. We begin with a simple model that only covers execution on a single machine without any parallelism, and end up with a cost model that both includes network cost and parallelism of components.

Step 1 Algorithm 6.1 shows the cost model for a topology that is running on a single machine without any component parallelism.

$$CO = \sum_{i \subset T} (CO_i * N_i) \tag{6.1}$$

$$N_i = \sum_{c \subset C_i} (N_c * S_c) \tag{6.2}$$

The cost of the topology is the sum of the cost of each operator in the topology. The cost of a operator in the topology is the cost of performing the operation on 1 tuple multiplied by the number of tuples coming into the operator.

Step 2 Step two adds network cost into the cost model. As we cannot track which tasks are running on which machines, we do assume the worst case; that each adjacent task in the topology tree are run on different machines. As we have to ack each tuple in Storm when they are done processing, the network cost must be applied twice. Algorithm 6.3 shows the cost model.

$$CO = \sum_{i \subset T} ((CO_i + 2 * CO_n) * N_i) \tag{6.3}$$

Step 3 Step three includes the cost of a parallelized component in a topology. We reduce the cost of the component by dividing the work by the parallelism level. There are some overhead by paralleling components; there will be extra messages to be sent over the network, this is covered by step two of the cost model. We might need to collect the results of each of the parallelized tasks into one component, for example with a parallelized sum operator, all the sums of the subsets needs to be iterated over in a new operator, but this is covered by step one of the cost model. Any extra overhead of paralleling the task (CPU cost and other parameters we don't measure) are labeled as a constant k.

$$CO = \sum_{i \subset T}((k * \frac{CO_i}{|p_i|} + 2 * CO_n) * N_i) \tag{6.4}$$

Limitations

The cost model does not take into account the cost from the running spouts in the topology. The spouts are the input source of the topology, and is assumed to to have a fixed cost that cannot be optimized. The network cost of the topology is simplified by assuming the worst case, in order to simplify the reasoning of the correctness of the cost model.

6.3 Cost of swapping topologies

An important question is how much does it cost to swap between two topologies. That is how much efficient computational time is lost due to performing a swap. We now look at the cost of applying the moving state strategy to Storm.

6.3.1 Moving state strategy

Since topologies are immutable we cannot connect old and new topologies together without using some intermediate tuple queuing system listening on all operators output streams. Such an approach will introduce unnecessary complexity to the construction of a topology, and is prone to introduce bottlenecks. We therefore choose to replay unacked tuples instead. Recall that in Section 5.1 we discuss the different message semantics and their corresponding strength and weaknesses. Also, recall that we choose to use at-least once messaging semantics guaranteeing tuples. This message semantics, in combination with a replayable message broker system [1], lets us easily keep control over the tuples that has not been fully acked.

With this in mind we can propose a moving state migration strategy as well as a corresponding cost model. Let every tuple input to Storm have a timestamp t associated with it. Also, let Y be the youngest tuple with the timestamp t_Y for the last output tuple of topology A. We define Y to be the last tuple in A that has been fully processed (the

[1] Apache Kafka is a message broker system that lets you replay tuples from a specified offset (http://kafka.apache.org/)

timestamp is managed by the message broker, so we assume that the message broker keeps the tuples in the same order as they arrived). The following steps gives a general description of the procedure for applying moving state strategy in Storm.

1. Bring a new topology B up.
2. Bring the old topology A down.
3. Replay tuples into B from timestamp t_C

After some time, B will have the same state as A when it was brought down, unless some tuples have been omitted. In the following paragraphs we will try to model that elapsed time.

A formula for calculating the cost of this strategy can be expressed as follows. Let C_{down} be the cost of bringing a topology down, C_{up} for bringing another one up, and C_{replay} be the cost of replaying the tuples to the new topology. We then get the following cost equation:

$$C = C_{down} + C_{up} + C_{replay} \tag{6.5}$$

Notice that the cost of bringing up a topology is partly hidden due to that we have already brought up the new topology before taking the old one down. Assuming that a topology has been uploaded to a running topology. We only need to model the cost of scheduling tasks omitting the cost associated with uploading the topology. We represent the cost of scheduling tasks as $C_{schedule}$, and we can now model the cost of swapping topologies as:

$$C = C_{down} + C_{schedule} + C_{replay} \tag{6.6}$$

The time it takes to reschedule (bring down and schedule) a topology on a Storm cluster is dependent on the number of tasks contained in a topology. Since the number of tasks is usually much lower than the number of unacked tuples that has to be replayed through a newly optimized query, we believe that the two costs C_{down} and $C_{schedule}$ can be assumed to be constant for all topologies of reasonable size. Thus, we can assume that the cost of rescheduling a topology is dominated by C_{replay}. Moreover, the process of replaying tuples through a known topology can be viewed as the cost of executing a query. We can therefore use the same cost model presented in Section 6.2 to estimate the cost of sending unacked tuples through a newly submitted topology.

6.4 When to optimize

The adaptive optimizer needs to figure out when it needs to optimize the running topology. Optimizing too often can negate the performance gained from the actual optimization, because of the overhead associated with re-optimizing. The offset includes time taken to generate new query plans, replaying tuples, taking down a topology and uploading a new one. In this section, we discuss how the system should detect that it needs to perform the adaptive optimization.

6.4.1 Query environment

Before we present the different strategies, we need to define the different types of queries that might be run. In some situations, it might not be desirable to perform optimization at run time because downtime is unacceptable. The situation is the same for ad-hoc queries, explained in Section 5.4, where accuracy is a more important property than efficiency. However, for long running continuous queries, where response time and throughput is more important than accuracy, it is desirable to do run-time optimization, as it is often more important to receive the answers to the queries in a timely fashion rather than getting a exact answer. For the reminder of this section, we assume that the queries are continuous and long-running queries that allows some downtime during execution.

6.4.2 Optimization time

In adaptive query optimization in DBMS, one can use a heuristics that compares the expected time it takes to generate a new plan against the estimated time it takes to complete the running plan to decide whether to spend computation time on generating new plans. If the query optimizer have decided that it want to generate a new plan, it must then decide whether it want to apply the plan or not. The latter decision can be determined by applying a heuristic that compares the expected running time of the current plan against the expected running time of the improved plan. See Algorithm 3.1 and 3.2 in Section 3.2.1 for a more in depth explanation.

The aforementioned heuristics are based on that you have access to a system catalog that have information about the data stored, and that the queries runs for a limited time. Neither of these conditions apply to our situation. In our situation we have a limited set of resources, and statistics about the throughput of tuples in the topology, as well as the number of unfinished tuples in the system. We have earlier assumed that the system should use all the resources available, so we will not optimize in order to limit the resource usage.

We want to optimize the topologies in order to provide the best possible response time. The response time is depending on how often a result is to be given to the user. This is defined when the user sends in the query, see Section 4.1.1. Then, a logical time to do optimization is when the response time is greater than tick tuple rate, T_r. We can detect this by looking at the input rate, I_{avg}, and the ack rate, A_{avg}. If we for example have a topology with an input rate of 1000 tuples per second, then system is not lagging if the rate of the number of tuples acked (ack rate) to be the same as the input rate. If on the other hand, the input rate is larger than the ack rate for some topology, it is time to try to optimize that topology. Formally,

$$I_{avg} > A_{avg} \tag{6.7}$$

The user is also able to define the tick rate to be based on the number of tuples that enters the system. Then both the window size and output rate will be size based. In this case,

Optimizing parameters	
T_r:	Tick rate
I_{avg}:	Average input rate per T_r
A_{avg}:	Average tuples acked per T_r
U_t:	Un-processed tuples in a Topology in time period t
t:	time period, e.g 1 min

Figure 6.1: When to optimize topology parameters

we do not have a time to relate to, so we need to look at other factors to determine if the system is responding fast enough. One way to do this is to look after tuple congestion in the topology. Here we can use the statistics gathered by the ack tasks, and look on the number of un-processed tuples in the system, U. If this value grows over time, we have a situation where we need to optimize. Formally:

$$\sum U_{t_1} - \sum U_{t_2} > 0 \tag{6.8}$$

Where t_2 is the time period right after t_1.

Detecting that the system needs to optimize the query based on the previous method will detect that we need an optimization, but it does not detect whether we can find a more optimal plan. In cases where Equation 6.8 does not hold, that is, the system throughputs data fast enough, there might be more optimal plans but there is no need for the system to optimize. This would have been the case if we also would optimize to reduce resources, but that is out of the scope of this thesis.

6.5 Optimization strategies

This sections present the optimization strategies we have implemented in Raincoat. We have split it up into four main categories, which are join order optimization, Storm relevant optimization, select order optimization and merging of operators, and lastly distributed optimization. We have split it up in separate strategies in order to easily measure the effect of each optimization. The time of optimization is decided by the heuristics described in Section 6.4.

6.5.1 Pre-optimization

At compile time, we do some lightweight optimization that does not require any runtime statistics.

- Merge project node into its child. As the project node does not do any extra work, only selects the fields that should be outputted, we will reduce the network traffic by merging it into its child.

- Re-order the tree so that select operations come before the join operations, in order to reduce the amount of data the join operators have to execute on.

6.5.2 Join order optimization

There are several ways to search the plan space of different join orderings, which are discussed in Section 3.6. As we in this step do not know whether or not the join operations can be run in parallel, we only search the left-deep plan space. Algorithm 3 describes a greedy algorithm that chooses the join operations with the lowest selectivity and joins them together, resulting in a optimized RA-tree. The selectivity is calculated by the cost model. The input is the set of join operators specified in the query. The selectivity of the operators are determined by the cost model, further discussed in Section 6.2.

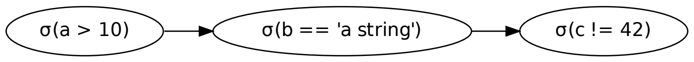
Algorithm 3 Greedy join order optimization based on selectivity

```
procedure OPTIMIZEJOINORDER(R)                    ▷ R: List of join relations
    while $R.length > 1$ do
        $a \leftarrow r.pop()$
        $b \leftarrow r.pop()$
        $c \leftarrow joinNode()$
        $c.addChild(a)$
        $c.addChild(b)$
        $R.append(c)$
    end while
end procedure
```

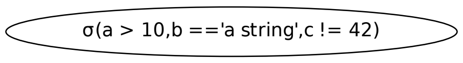
6.5.3 Storm optimization

Storm allows you to control how tasks get scheduled in the cluster. To do this, we need to implement a custom task scheduler, see Section 4.1. A custom task scheduler gives us control over network traffic, by specifying which operations you want to keep local on a machine. The Scheduler can control which nodes task gets executed on. So we can use it to schedule adjacent tasks on the same machine, or, in cases where we have CPU-heavy operations, we can make sure that two CPU-heavy operations does not run on the same machine. However, a proposed implementation of such a scheduler is out of scope of this thesis, and is listed as further work.

Another optimization that is Storm specific is the selection of stream groupings in the Topology. Stream groupings were introduced in Section 4.1. One of these groupings is the local or shuffle grouping, which schedules tasks locally if the bolt with the grouping have other worker tasks in the same process.



(a) Topology before merge



(b) Topology after merge

Figure 6.2: Select operators before and after merging.

6.5.4 Select order and merging of operators

We look at two different ways to optimize select operators. The first method is to merge operators together into fewer bolts. Each bolt in Storm can perform several operations, for instance several select operations. By merging operators, we reduce the amount of data that is sent over the network. We limit merging to select operators and aggregators, as join operators are defined as a heavy operation, and merging them together might result in a performance decrease. Figure 6.2 shows an example topology with the operators before and after optimization.

Another optimization we perform on select operators is to re-order the select operators, such that select operators with low selectivity comes first. This is the same principle as we use when optimizing the join order.

Algorithm 4 Sorted order of select operators

```
procedure OptimizeSelectOrder(R)                    ▷ R: List of select operators
    return sort(R)
end procedure
```

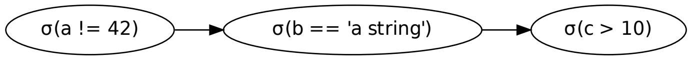
Figure 6.3 shows the operators before and after optimization.

6.5.5 Distributed optimization

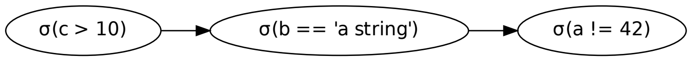
In distributed optimization we mainly look at two optimizations, running different operations concurrent and paralleling single operators, that is, dedicate more workers to certain tasks.

Concurrent calculations

In more complex queries, running independent operations concurrently can yield in a performance boosts. That is, searching the bushy tree for plans that are cheaper than the



(a) Select order 1



(b) Select order 2

Figure 6.3: Select operators before and after sorting. a != 42 has a 95% selectivity, b != 'a string' has 90% selectivity and c > 10 has 50% selectivity

equivalent in the left-deep tree. First, we look at the different operators we have:

- **Select.** The select operators are filters. Filters has to be run in a pipelined fashion to make sure that the filters are applied on all tuples, to ensure the right result from the query. We will not run different filters concurrently.
- **Project.** The project operation is a single operator that emits the result of the whole topology, so it makes no sense to run it concurrently with other operations.
- **Joins.** The join operator can be run in concurrently. We reduce the set of possibilities to join operations that does not share base relations(source).

So we can reduce the problem to finding join operators that does not share base relations.

When we have determined that two joins can be ran in parallel, we need to re-order the relational algebra tree to achieve this. Algorithm 5 describes how to create a bushy join order tree.

To illustrate the difference, we use the following query:

```
Select * from S,T,U,V WHERE S.a = T.a AND U.b = V.b AND V.a = T.a;
```

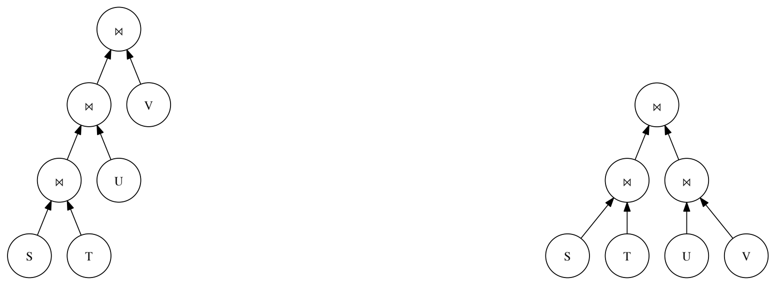
Its left-deep tree is shown in Figure 6.4a, and the bushy, parallelized tree is shown in Figure 6.4b.

Parallelism

Increasing the parallelism of a operator is done by changing its parallelismHint in the Storm topology. Mainly, we want to parallelize operators that has the worst execution time. As we have defined earlier, we do not optimize on resources used, so if there are idle workers, and the topology is performing sub-optimal we are able to increase the parallelism of bolts

Algorithm 5 This algorithms creates a bushy RA-tree of join nodes. It outputs a bushy RA-tree.

```
procedure OptimizeConcurrency(x,y)                ▷ x,y: base join relations
    $j1 \leftarrow joinNode()$
    $j1.addChild(x.A)$
    $j1.addChild(x.B)$
    $j2 \leftarrow joinNode()$
    $j2.addChild(y.A)$
    $j2.addChild(y.B)$
    $j3 \leftarrow joinNode()$
    $j3.addChild(j1)$
    $j3.addChild(j2)$
end procedure
```



(a) Left-deep tree *(b) Bushy tree*

Figure 6.4: Shows the difference between a pipelined join tree and a concurrent join tree

that are CPU-heavy. As the project and select operators are lightweight operators, these will not be taken into account. Only join operators and aggregators.

Figuring out the right amount of parallelism for heavy operators is a non-trivial task. First, we want to identify the operators that needs to be parallelized. As a heuristic, we look at the execution time vs the tick tuple interval. Remember that the tick rate is the interval at which the user gets results from the query. If the execution time is larger than the tick rate, the user will experience a delay from the system, which is unwanted behavior. Equation 6.9 shows the described heuristic.

$$Execution\_time > Tick\_rate \tag{6.9}$$

The next step is to decide how much parallelism the operator needs to perform fast enough. First, we calculate the percentage performance increase we need to perform the operation within the required time, as shown in Equation 6.10.

$$\epsilon = \frac{Execution\_time}{Tick\_rate} \tag{6.10}$$

When we know the percentage performance increase needed, we need to figure out how much we need to increase the parallelism to achieve that. Algorithm 6 is used to change the parallelism of an operator.

Algorithm 6 Algorithm for changing the parallelism of a operator

```
procedure OPTIMIZEPARALLELISM($T, E_t, P$)
    if $T > E_t$ then                              ▷ T: tick rate, $E_t$: Execution time
        Return P                                   ▷ P: current parallelism
    end if
    $\epsilon \leftarrow \frac{E_t}{T}$
    Return findParallelism($\epsilon, P$)
end procedure
```

Choosing a algorithm for $findParallelism$ is non-trivial. The challenge lies in choosing the right parallelism hint for the operator. We do not know beforehand how much performance gain an increase in parallelism will give us. A naive implementation is to increase the parallelism hint with a constant value every time, as shown in Algorithm 7.

Algorithm 7 Algorithm for estimating the needed parallelism, naive

```
procedure FINDPARALLELISM($\epsilon, P$)
    if local_minimum then
        Return $P$
    end if
    Return $P + 2$
end procedure
```

Another way is to use runtime-statistics to determine the increase needed for the operator to perform calculations fast enough. When we increase the parallelism, we log the performance gained. The next time we detect that we need to optimize, we use the history to determine the factor to increase with. It is worth to mention that this algorithm assumes that a increase in parallelism will yield a constant performance boost, and does not take into account any overhead that gets introduced.

Algorithm 8 Algorithm for estimating the needed parallelism, adaptive

```
procedure FINDPARALLELISM($\epsilon, P, A, P\_max$)          ▷ A: average performance boost
    if local_minimum then
        Return $P$
    end if
    for $i = 1 \rightarrow P\_max$ do
        if $i * A > \epsilon$ then
            Return $P + i$
        end if
        $i \leftarrow i + 1$
    end for
    Return $P\_max$
end procedure
```

The average gain is calculated after every optimization, with Algorithm 9.

Algorithm 9 Average gain from parallelism increase

```
procedure CALCULATEAVERAGEGAIN($Et_{new}, Et_{old}, a, \Delta P$)
    $gain \leftarrow \frac{Et_{new}}{Et_{old}}$
    $average\_gain \leftarrow \frac{gain}{\Delta P}$
    $a \leftarrow \frac{a + average\_gain}{2}$
end procedure
```

The $local\_minimum$ variable will set to true if an increase in parallelism does not yield performance gain. So both Algorithm 7 and 8 will stop their search at a local minimum if the desired performance is not yet achieved. Algorithm 7 and 8 are only proposed implementations. We evaluate the correctness of the algorithms in Chapter 7.

6.6 Adaptive optimization

After we have presented all the optimizations techniques, we tie them all together in an adaptive algorithm. Algorithm 10 is run at a specified interval to detect whether we need to optimize or not, and executes the optimization techniques presented in the previous section.

Algorithm 10 Adaptive optimization of topology

```
if NeedsOptimalization(topology) then
    OptimizeJoinOrder(raTree)
    if not IsSelectMerged(raTree) then
        MergeSelectNodes(raTree)
    end if
    OptimizeSelectOrder(raTree)
    OptimizeConcurrency(raTree)
    CalculateAverageGain($Et_{new}, Et_{old}, a, \Delta P$)
    if $idleWorkers > 0$ then
        for Each join in topology do
            OptimizeParallelism(join)
        end for
    end if
end if
```

There are some drawbacks to the algorithm. First of all, it never frees any resources. There are two consequences following; the first is that it affects all topologies running on the cluster, as they might need more workers to be able to handle the load, so they might have to continue to run sub-optimal. The other consequence is that the resources dedicated to the cluster can not be released to do other work. Addressing these issues is postponed and added to further work.

Chapter 7

Performance evaluation

In this chapter we present the performance evaluation of Raincoat. We begin with presenting our test approach in Section 7.1. The test environment is described in Section 7.2. Our problem and data model for the test are presented in Section 7.3 and the actual test plan is presented in Section 7.4. The results of the test can be found in Section 7.5 and we round up this chapter with some concluding remarks in Section 7.6.

7.1 Testing approach

In our tests we want to evaluate Raincoat's scalability potential. We do so by testing the different optimization techniques we presented, and apply them to a running query. Later on we present a few test cases and the results collected from those tests for evaluating the systems performance.

Originally, we wanted to use TPC-H for testing the different optimization techniques. TPC stands for Transaction Processing Performance Council, and is a non-profit organization. They define transaction processing and database benchmarks. TPC-H is a decision support benchmark system. TPC-H can be divided in two parts, the data generator, which populates 8 premade tables that is designed to be useful for a broad area of applications, mainly traditional database systems. The other part is the query generator which can generate queries based on the different input parameters. The main problem with TPC-H was the way the tables were designed. The tables they provide were customer, nation, region, supplier, part, orders, lineitem and partsupplier. Of those tables, only orders and lineitem can logically be used as a datastream, whereas customer, nation, region, supplier, part and partsupplier are tables with data that in a real life environment changes seldom. We want our tests to run on a more suitable example, so we have created our own test case, presented in Section 7.3.

7.2 Test environment

The test cluster consisted of 11 high performance commodity machines on a high performance local area network. Each instances run inside a rack with the following specification; PowerEdge R415 Rack Chassis for Up 4x 3.5" Hot Plug HDDs. 8 of 11 the machines had a AMD Opteron 4130 CPU with four 2.6GHz cores, 4x512K L2/6M L3 Cache, 32GB Memory, and had 2x2TB disks in RAID-1. The remaining 3 machines was slightly more powerful with 128GB Memory and 6-cores, instead of 32GB memory and 4 cores. Of the 11 machines, one was running a ZooKeeper instance and a Nimbus instance and the remaining 10 ran as supervisors. Each supervisor was running 8 workers.

7.3 Test problem description

Social networks such as Twitter, Instagram, Pintrest, LinkedIn and Facebook produces a lot of streaming data. By simulating the data generated by such web pages we can create an artificial test environment that resembles real streaming data. The data generated for this test case is mostly posts from the aforementioned social networks.

Below is a we present the data models as an entity-relationship diagram. Tweet, Pintrest, Facebook post, LinkedIn and Instagram are streams that can be queried. Username and Geotag is a table shared among these streams. To limit the amount of code needed to be written Pintrest and LinkedIn inherits their data model from Tweets and Facebook posts, respectively.

Below is a generic description of the different post models.

Twitter “tweets”
Username
Geotag
Message
Hashtags
Mention

Facebook Wallposts
Username
Message
Mention
Geotag

Instagram posts
Username
Hashtags
Geotag
Picture
Mention

Geotag
Geotag
Lat
Lng

Pintrest
Username
Geotag
Message
Hashtags
Mention

LinkedIn
Username
Message
Mention
Geotag

Both username and hashtags is generated from a list of unique usernames and hashtags. Thus the cardinality of these are known. We have a total of 400 usernames and 40 hashtags. We assume that an user uses the same username on all social networks. This assumption is made to simplify the querying model.

Using the data model presented above one can answer interesting question regarding the relationship between users of the different social networks. Below we have listed a few

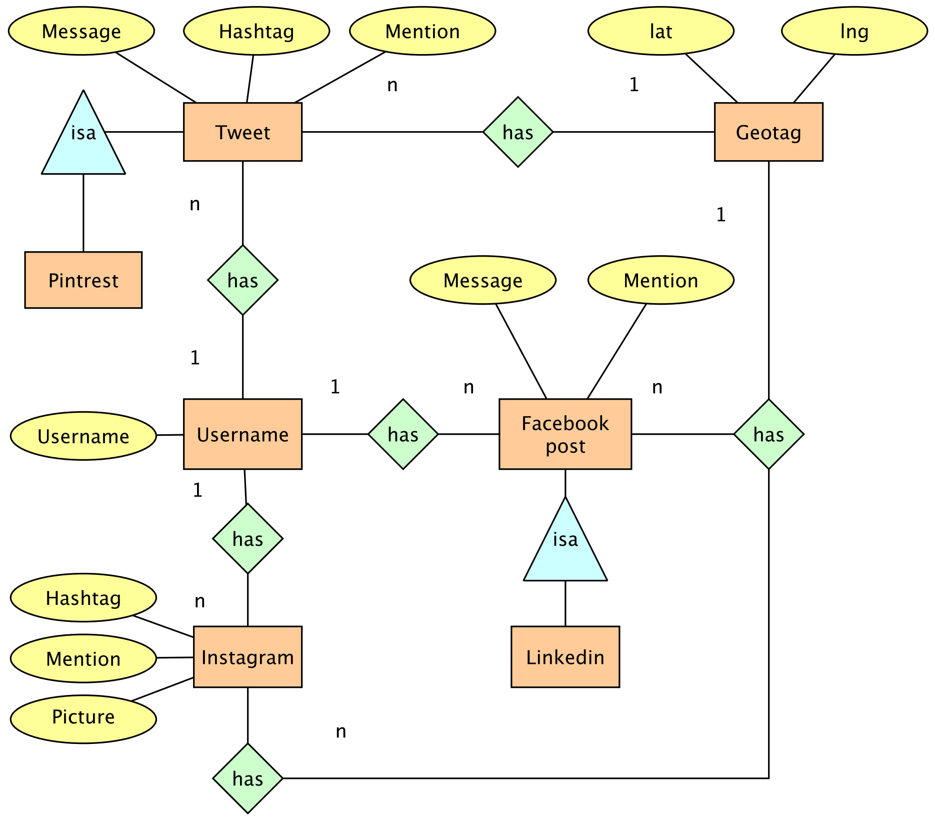


Figure 7.1: Enetity-relationship diagram for our data streams

of these questions.

1. Which tweet messages and pictures posts on instagram within the last 30 seconds shares the same mentions (assuming they have the same username on instagram and on twitter)

2. How many posts have been posted the last 30 seconds

3. Which users has posted on twitter, facebook, and instagram within the last 30 seconds,

4. How many posts one user have had in total on the different social networks within the last 30 seconds

Now that we have illustrated a few questions that can be addressed for the data model at hand. We can translate these question into queries returning the desired results. In the queries presented below we assume that *twitter, instagram*, and *facebook* represents the data model for tweet, instagram posts, and facebook posts, respectively.

1. `SELECT twitter_message, instagram_picture`

```
FROM twitter, instagram
WHERE twitter_mention = instagram_mention
EVERY 10 SECONDS SIZE 30;
```

2.
   ```
   SELECT count(*)
   FROM twitter
   EVERY 10 SECONDS SIZE 30;
   ```

3.
   ```
   SELECT username
   FROM twitter, facebook, instagram
   WHERE twitter_username = facebook_username
   AND facebook_username = instagram_username
   EVERY 10 SECONDS SIZE 30;
   ```

4.
   ```
   SELECT count(twitter_username)
   FROM twitter, facebook, instagram
   WHERE twitter_username = facebook_username
   AND twitter_username = instagram_username
   AND twitter_username = "<Insert username>"
   EVERY 10 SECONDS SIZE 30;
   ```

Observe that in the last query, all the usernames is the same. A smart optimizer would take this into consideration filtering the sources of both facebook and instagram as well. However, that optimization is not implemented at the current state of Raincoat. So by adding "facebook_username = '<Insert username>' and instagram_username = '<Insert username>'" one can expect a gain in performance compared to the original query.

7.4 Test plan

In Section 6.5 we presented a set of optimization theories that could be applied to Raincoat queries. With the performance tests we empirically check if the effectiveness of the implemented optimization theories. We focused on the performance testing for parallelism and query optimization. To keep test data clean and easier to reason with we ran each of the tests separately.

For the parallelism tests we run a complex join query with varying input rate, number of machines, parallelism hint, and window sizes. From these tests we want to show that increasing parallelism hint will decrease the execution time for join nodes on average. The test case is presented in detail in Table 7.2. We later we test the join order query optimization, a detail description of the test case is presented in Table 7.3. In this case we kept the parallelism hint constant at 1 while varying the input rate, number of machines, window sizes, and optimizers. In test case 3 (Table 7.4) we test the optimization techniques for select operators, such as merging select operators, and merging them. Finally, in test case 4 (Table 7.5) we explore the parallelism potential for select operators.

7.4.1 Log records

Before we present the different test cases we need to present the statistics collected. For each bolt we have implemented a statistics logger, it is used to log a row of statistical data whenever a tick tuple arrives. Each log record presents data collected by the logger between records.

timestamp	# of incoming tuples	avg	min	max	tot	window size	acked	emitted

Table 7.1: Log record

In this paragraph we explain the different fields in each log record presented in Table 7.1. The *timestamp* is the timestamp for when the log was recorded. Number of incoming tuples between the last record and now. *Avg*, *min* and *max* represent the average, maximum and minimum execution time pr incoming tuple. Where *tot* are the total execution time caused by all incoming tuples. *Window size* is the number of tuples contained in the window at this point in time. *Acked* and *emitted* is the number of tuples acked and emitted since last record, respectively.

Using the these logs records we can identify different properties of the test cases. E.g. one can identify that an operation is lagging by checking if the total execution time is longer than the tick rate specified by the queries. Or calculate the throughput of a tuple or topology by looking at the number of fully processed tuples (acked) over time.

7.4.2 Test variables

In this section we present the different parameters and variables for the test cases described below.

7.4.3 Test cases

Each query we test is run several time, with different variables. These variables and other aspects with the test cases are described in the following tables. The test variables row indicates which parameters that vary for each iteration of the tests.

Scalability of Join

This test evaluates the scalability of the join operator. The parallelism hint of the operator varies from 1 to 64. The result of the test gives us data about the percentage increase the operator gain by paralleling it.

Name	Short	Description
Parallelism hint	P	The level of parallelism a job presented as either a spout or bolt can achieve
Machines	M	The number of individual machines that is present in the test environments Storm cluster
Query	-	The continuously running query used for the test
Window size	W	How long tuples are allowed to live inside an operator
Tick tuple rate	-	The rate of the arrival of a specialized tuple. That can be used to either specify how often tuples should be evicted, or how often one operation should be performed. The hash joins uses the former case for evicting tuples from its window. While the latter is used for performing aggregation operations.
Tuple production rate	R	Represents how many tuples that are inputted from each Spout in the system
Query optimizers	-	The different query optimizers that was applied to the query tree before transforming it into a Storm topology

Test Case ID	1
Query	SELECT f_username FROM twitter, fb, insta, linkedin, pin WHERE t_username = f_username AND f_username = i_username AND i_username = l_username AND l_username = p_username EVERY 4 seconds SIZE <window_size>
Tuple production rate	2, 4, 10, 20 tuples per seconds per source
Tick rate	4 Seconds
Window size	8, 16, 32 seconds
Test run time	60 seconds (+ 30 seconds of shutdown time)
Machines	1, 4, 10
Parallelism Hint	1, 2, 6, 12, 16, 20, 32, 64
Query optimizers	None

Table 7.2: Table describing the details of test case 1

Join optimization evaluation

In this test we evaluate the performance gain of applying performance optimizations on a query. In this test we keep the parallelism hint constant and vary variables such as window size, tuple production rate, tuple tick rate, and number of cluster nodes (machines).

Test case ID	2
Query	SELECT f_username FROM twitter, fb, insta, linkedin, pin WHERE t_username = f_username AND f_username = i_username AND i_username = l_username AND l_username = p_username EVERY 4 seconds SIZE <window_size>’
Tuple production rate	2, 4, 10 tuples per seconds per source
Tick rate	4 seconds
Window size	8, 16, 32 seconds
Test run time	60 seconds (+ 30 seconds of shutdown time)
Machines	1, 4, 10
Parallelism Hint	1
Query optimizers	None, and join order

Table 7.3: Table describing the details of test case 2

Select optimization evaluation

In this test we wanted to look at the gain of ordering selection operations to see if tuples were significantly pruned or not. Then, we perform a merging of selection operation too see if the reduction of network traffic will increase system performance.

Test case ID	3

Query	SELECT f_username
	FROM fb
	WHERE f_username = ’creepytesting’
	AND f_mention = ’dummy’
	AND f_geotag != ’empty’
	EVERY 4 seconds 8 SIZE

Tuple production rate	10, 20, 40, 100, 200, 500 tuples per seconds per source
Tick rate	4 seconds
Window size	8 seconds, it does not matter since selection does not carry a window
Test run time	60 seconds (+ 30 seconds of shutdown time)
Machines	1, 4, 10
Parallelism Hint	1
Query optimizers	None, and select order and select merge

Table 7.4: Table describing the details of test case 3

Scalability of Selects

In this test case we explore the scalability of multiple select operators. The parallelism hint varies from 1 to 64. Using the result of this test we want to explore the parallelism potential of a select operator.

Test case ID	4

Query	SELECT f_username
	FROM fb
	WHERE f_username = 'creepytesting'
	AND f_mention = 'dummy'
	AND f_geotag != 'empty'
	EVERY 4 seconds 8 SIZE

Tuple production rate	10, 20, 40, 100, 200, 500 tuples per seconds per source
Tick rate	4 seconds
Window size	8 seconds, it does not matter since selection does not carry a window
Test run time	60 seconds (+ 30 seconds of shutdown time)
Machines	1, 4, 10
Parallelism Hint	1, 2, 6, 12, 16, 20, 32, 64
Query optimizers	None

Table 7.5: Table describing the details of test case 4

7.5 Results and analysis

In this section we present the interesting findings from the test cases given above. With each of the results we present why these results are included and provide a short interpretation and analysis of those data.

We have organized the rest of this section in the following way. In Section 7.5.1 we present results of increasing the level of parallelism. In Section 7.5.3 we look at how the different attributes of a join operator behaves over time, and provide an analysis of how one can reduce the average execution time of individual operators. In Section 7.5.4 we look at the concept of when to optimize presented in Section 6.4, and present the results and an analysis of data at hand. Finally, In Section 7.6 we combine the results and give an analysis of how this fits into Raincoat.

7.5.1 Results of parallel optimizations for joins

The following graphs shows the average execution time of all join nodes in the query presented in Table 7.2. It shows the gain of adding parallelism hint to all the join nodes with a pre-specified set of machines, input rate, and window size. Using the data collected we want to explore the systems parallelism capabilities. If the execution time is reduced on average for all join operators, without decreasing the systems throughput, when the level of parallelism is increased. Then we can say that the system will gain throughput by performing operations in parallel.

By superficially studying the different graphs we can observe that the system, in general, will experience a gain in performance by increasing the parallelism hint. The amount of gain achieved by introducing parallelism is any case limited by the window size, input rate and the query complexity. For this particular case we can observe that the queries with larger window sizes achieves higher performance gain by increasing the parallelism hint. E.g. in Figure 7.3 we see that the graph with $R = 250, M = 1$ increases performance by adding parallelism hint until it reaches its execution time minima at $P \approx 20$. Also, observe that if the rate is increased then the parallelism hint can, in general, be increased to reduce the average execution time of the join operators.

Observe that there are some abnormalities in some of the plots below. These are the graphs with plots that shows a decrease, increase, and then decrease again in execution time. Other plots increases in execution time for initial introduction of parallelism then decreases when the level of parallelism is further increased. Both these cases can be explained as a combination of bad scheduling of tasks and skewed shuffling between cloned join nodes ran in parallel. In such a case the distribution of tuples is skewed and will cause a few join tasks to be under much more load than the others. Thus, the load is not uniformly divided among the nodes. Causing the parallelized join operator to not fully utilize the available CPU cycles. Hence, causing a decrease in overall performance.

Also notice the graphs that reaches a minima in average execution time after increasing the level of parallelism to a certain value. Observe that the execution time stabilizes and does neither improve or worsen when parallelism hint is increased. Thus, it does not seem to introducing parallelism does not have any cost. However, observe from Figures 7.5, 7.6, and 7.7 that the total number of tuples acked decreases when the level of parallelism gets too high. These results indicates that the level of parallelism has a cost which has an effect on the throughput of an operator in Raincoat. This cost can be associated with the overhead of scheduling the multiple excess Storm tasks within one worker, and the extra network overhead caused by sending an additional packet of data for every worker used.

Also, remember that a Storm Cluster also has a pre-specified number of workers. Where each task is assigned to each worker. Each cluster have multiple of queries assigned to it. So if one query has a number of tasks equal or larger than the number of available workers for that cluster. No other queries will be able to process data on the same cluster. Causing the system to under perform since it cannot deliver the results as promised.

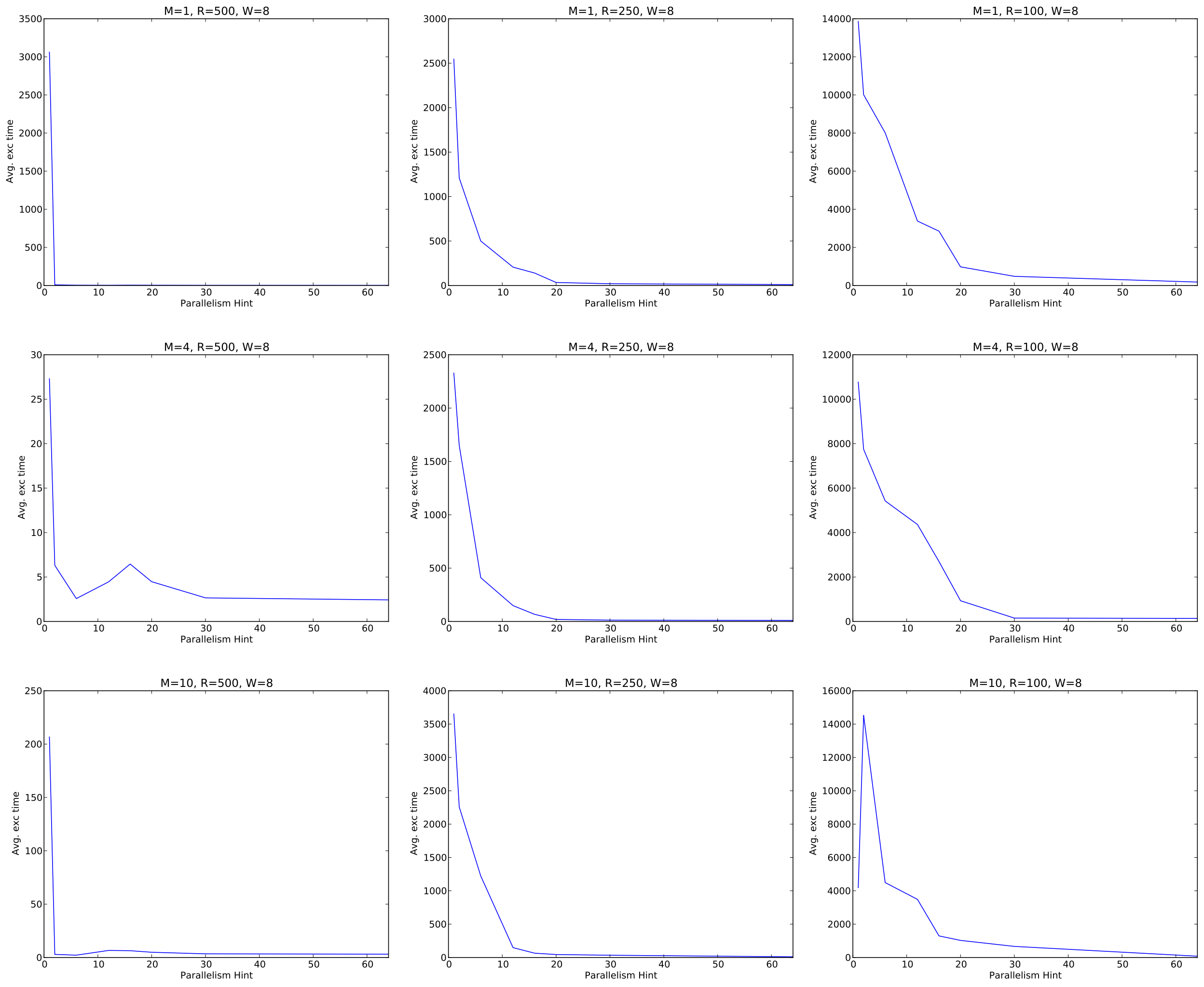


Figure 7.2: Increasing parallelism hint with $W = 8$ measuring average execution time

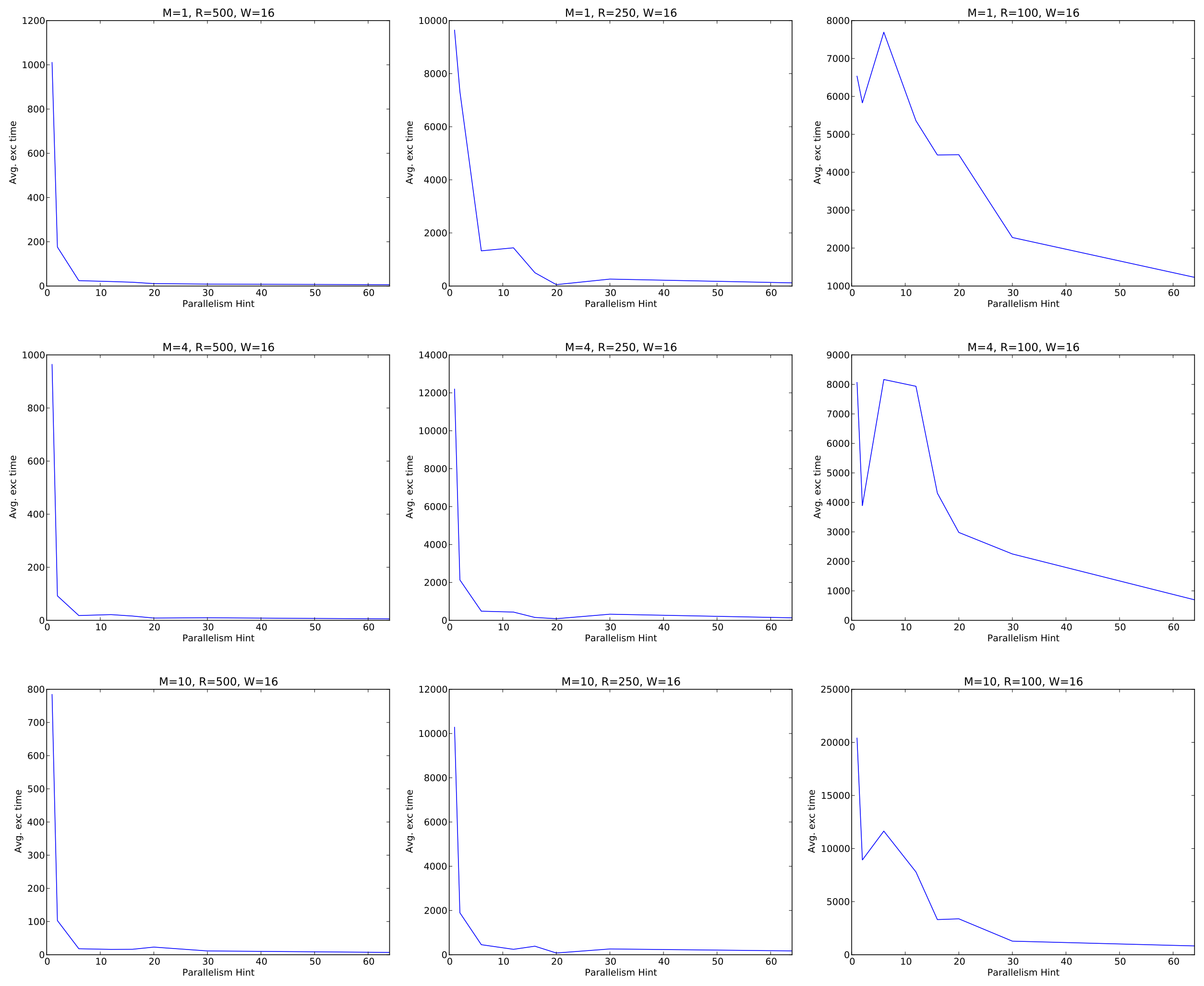


Figure 7.3: Increasing parallelism hint with $W = 16$ *measuring average execution time*

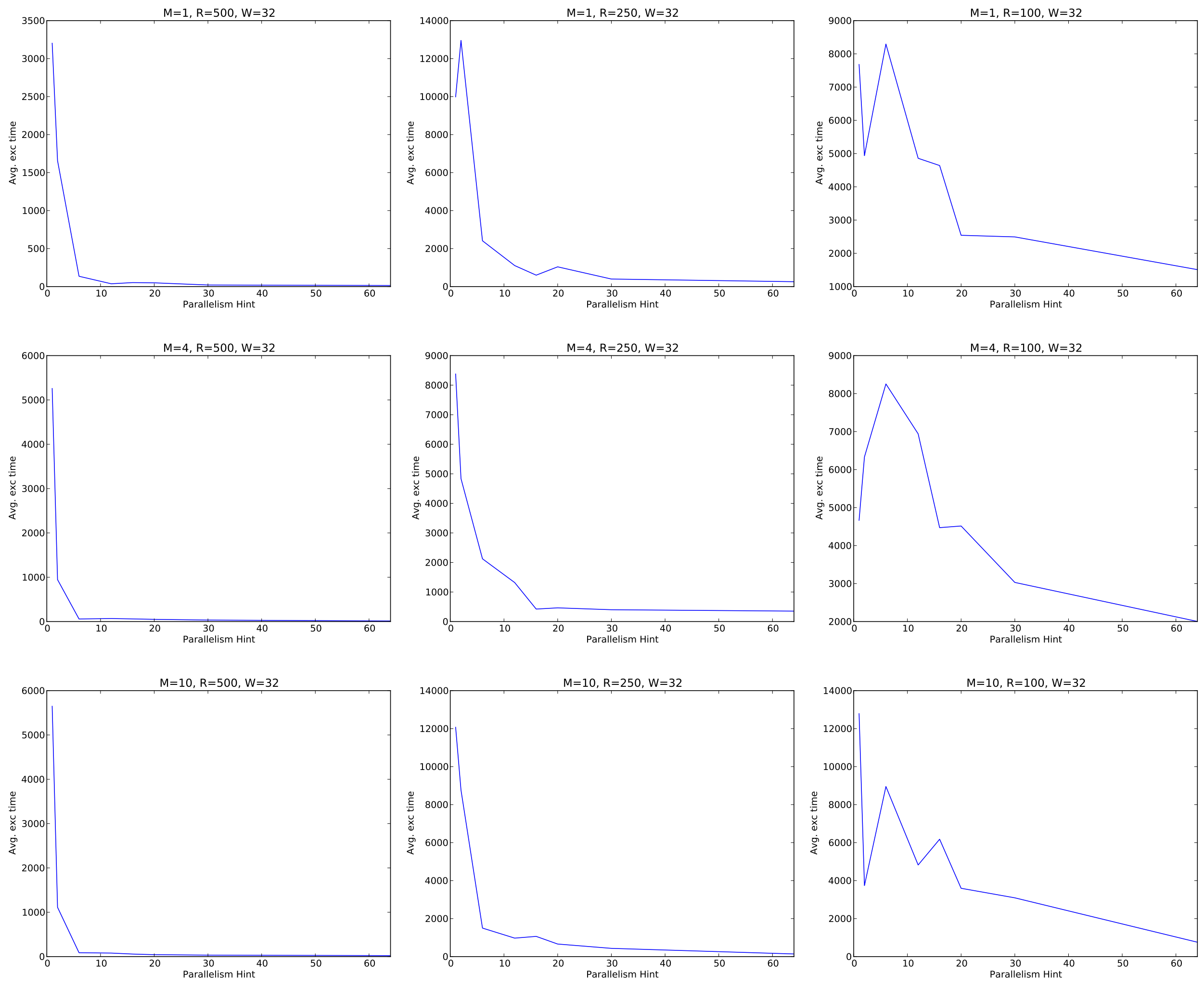


Figure 7.4: Increasing parallelism hint with $W = 32$ measuring average execution time

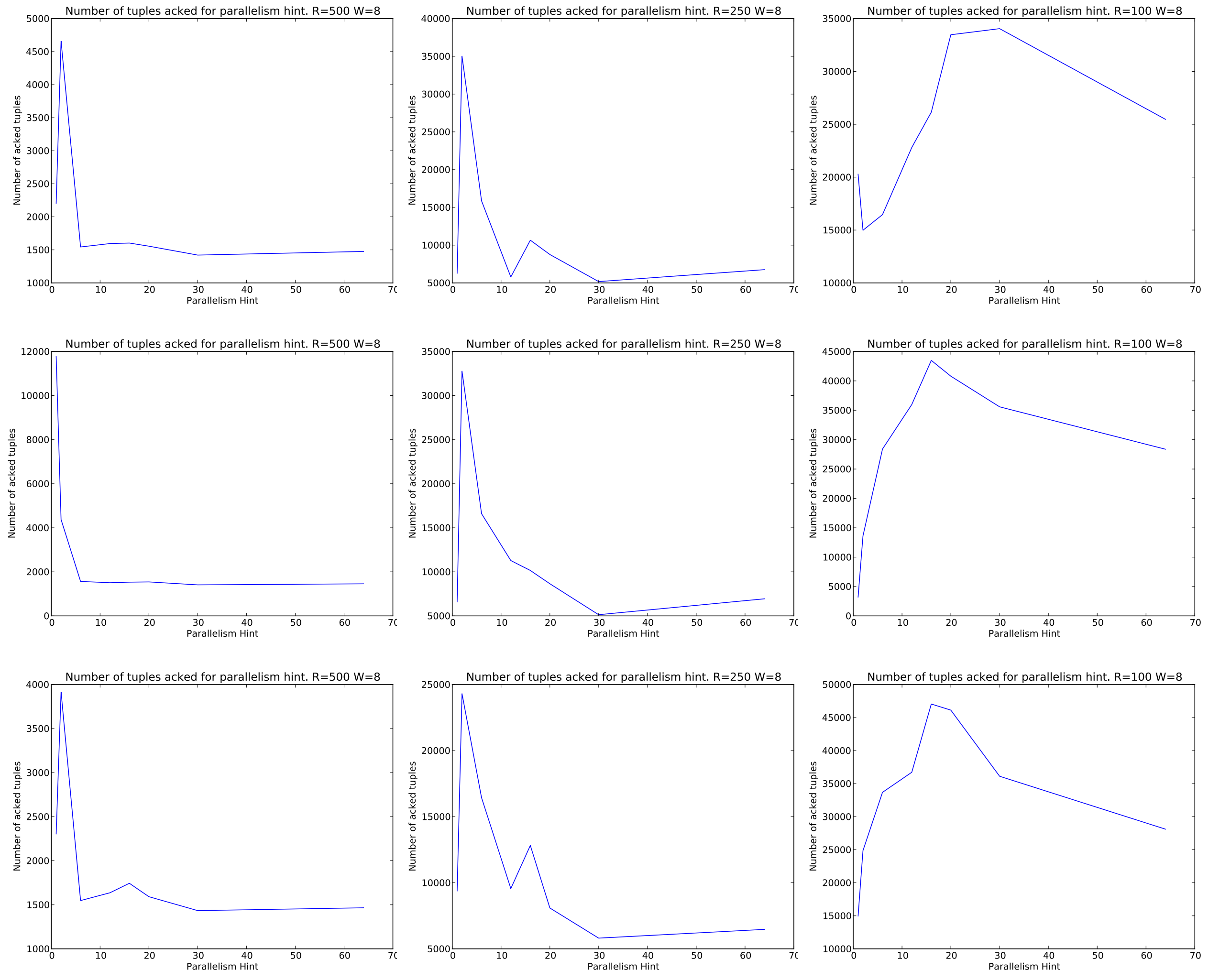


Figure 7.5: Increasing parallelism hint with $W = 8$ measuring the total acked tuples

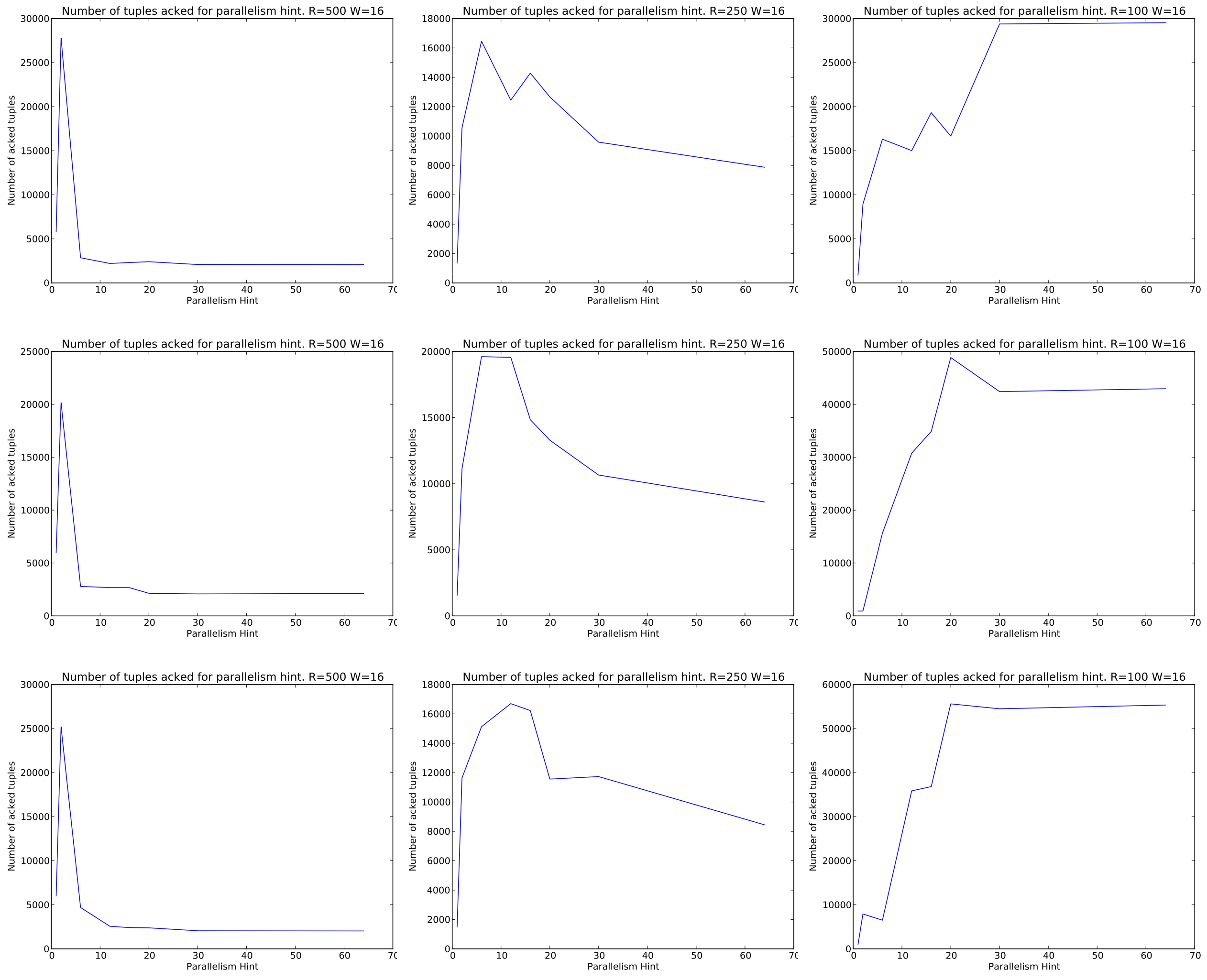


Figure 7.6: Increasing parallelism with over $\mathrm{W} = 16$ *measuring the total acked tuples*

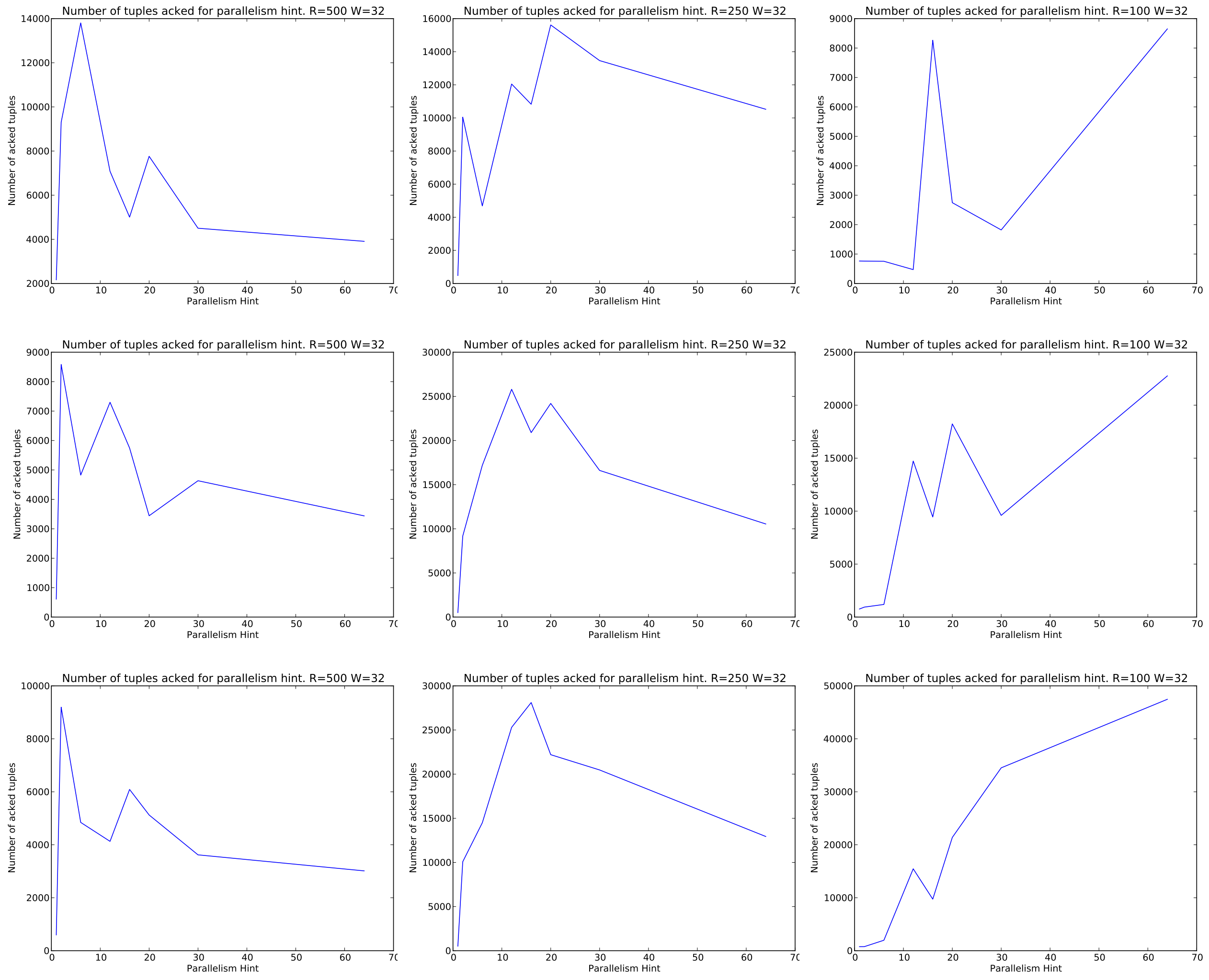


Figure 7.7: Increasing parallelism hint with $W = 32$ measuring the total acked tuples

7.5.2 Results of parallel optimizations for selects

In this section we analyze the results from the test case presented in Section 7.4.3. We analyze how an increase in parallelism affects selection operators of Raincoat. Recall that selection operators does not contain any window, therefore there is no need for analyzing the selection operations for different window sizes.

Observe from Figure 7.8 that increasing parallelism hint will decrease the average execution time and converge towards 0. Also notice that the number of acked tuples in Figure 7.9 does not experience a constant increase when parallelism hint is increased. Meaning that introducing parallelism for selection operators does also have a cost, which is the same cost as the one we found in Section 7.5.1.

Another observation worth mentioning is that the selection operation gains performance when increasing the parallelism hint as machines are added. In Figure 7.9 we see can observe that the graphs on the same horizontal line has approximately the same performance. However, we observe that the selection operators can handle more load if machines and parallelism hints are increased.

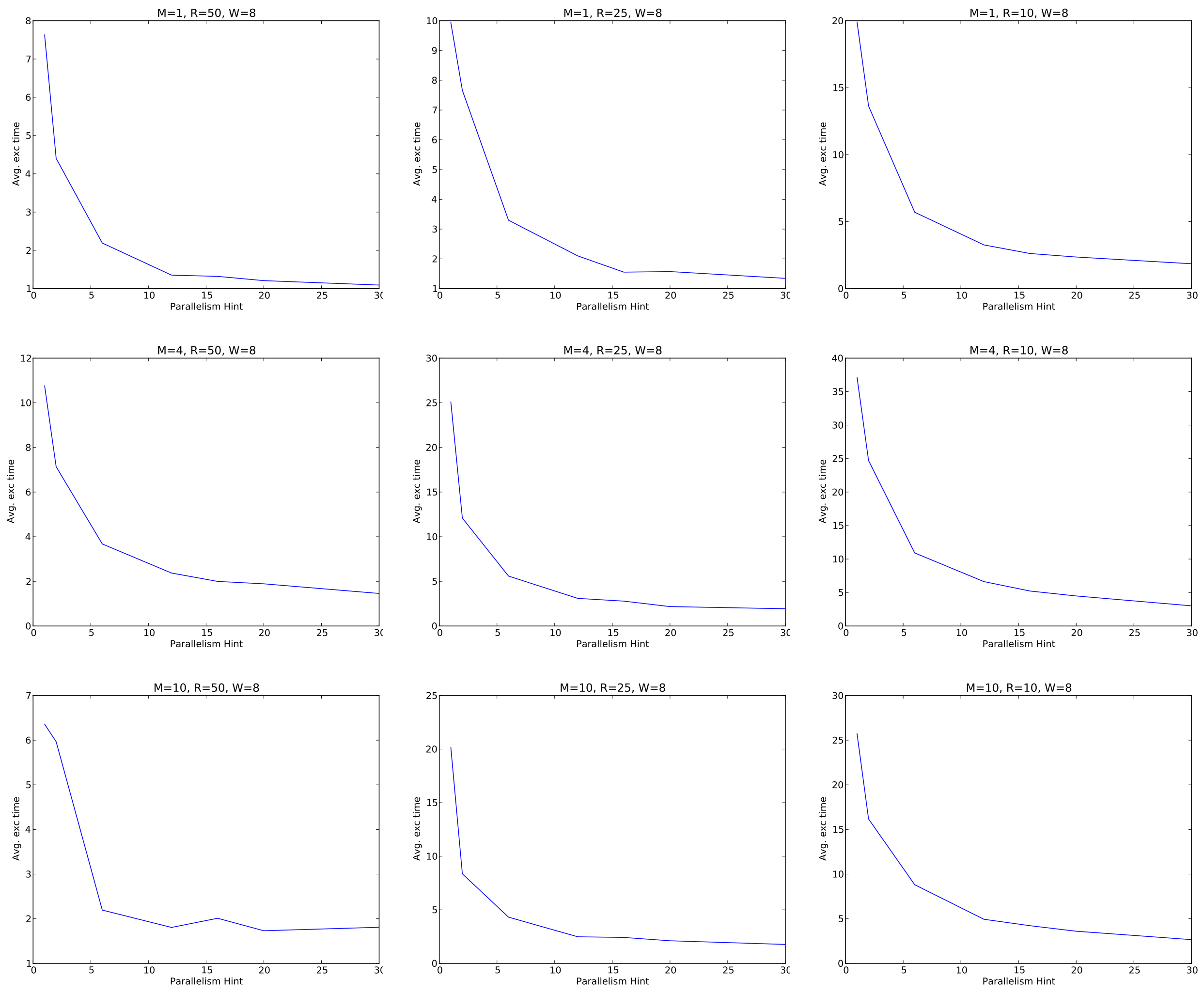


Figure 7.8: Increasing parallelism hint with $W = 8$ measuring average execution time

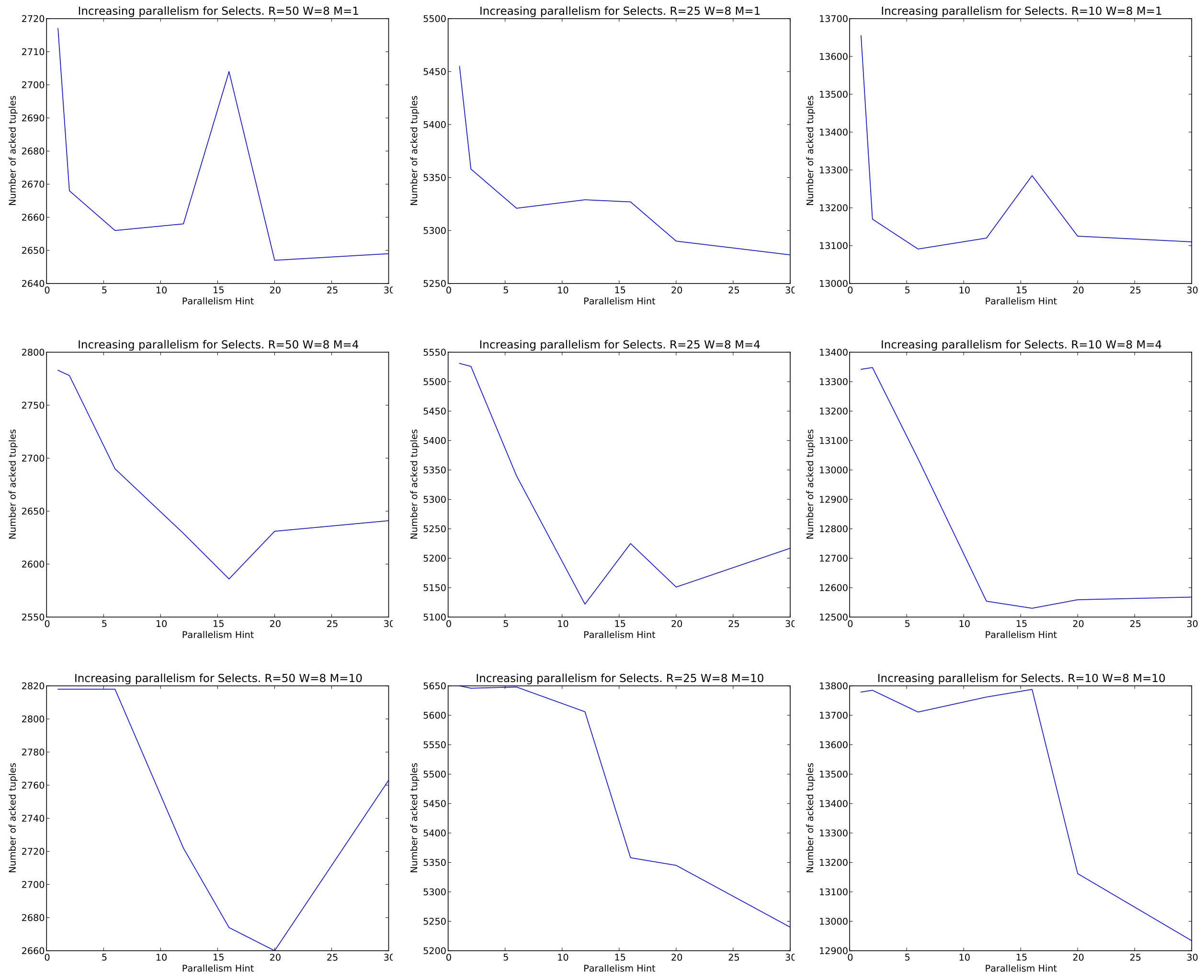


Figure 7.9: Increasing parallelism hint with $W = 8$ measuring the total acked tuples

7.5.3 Analyzing operator behavior over time

Below we present several plots exploring different aspects of join operators over time. The plots are extracted from data collected from running the tests presented in Section 7.4.3 using only the data where $P = 16, M = 1$ for all operators, varying the input rate and window size.

First, we examine the graphs superficially before going into a deeper analysis. It is easy to observe that the all the components measured increases significantly when the input rate is increased. Also, observe that the average execution time do also increase with the size of the sliding window. These observation is to be expected since increasing those variables will also increase the total amount of work the system has to perform.

Remember the computational heavy part of a join operator is to find matching tuples and merge them. Also, notice that the number of emitted tuples depends hit rate during joins, i.e. if no joins are performed then the number of tuples emitted from an operator would be 0. Therefore we should be able to find an correlation between average execution time for an operator, incoming tuples, window size, and emitted tuples. In the following paragraphs we will perform an analysis of the average execution times in the context of the different window sizes.

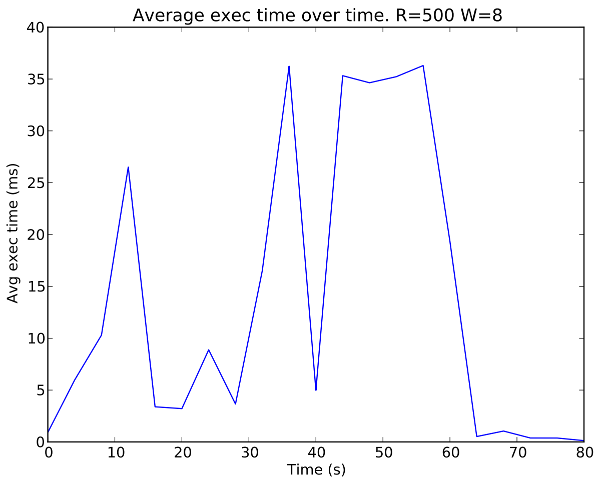
First we will start looking at the the average execution time for $W = 8$. Observe that in Figure 7.10a around 30 seconds into the life time of the query, we clearly see that that the this query has a spike in average execution time. This seems to be mainly because of the window size observed from Figure 7.11, as the average incoming tuples (Figure 7.12) and emitted tuples, Figure 7.13, are low around the same point in time. However, after about 45 seconds one can observe that both the number of tuples contained in the window and the execution time reaches another local maxima. These correlations are also present for different rates using the same W.

For $W = 16$ notice that some of the local maximas of the window size presented in Figure 7.15 are correlated with either maxima or local maximas of the execution time presented in Figure 7.14. However, also observe that the execution maxima for $R = 100$ in is not correlated to the it corresponding window size maxima. This leads us to believe that other factors than the size of the window can contribute to increase the execution time of the join operator. Examining the number incoming and emitted tuples in Figure 7.16 and 7.17, respectively. We observe that for this particular case that both have a significant increases about the same time as the spike in average execution time.

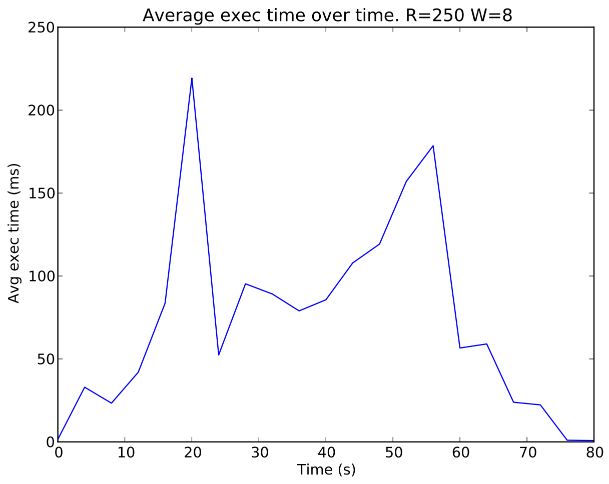
Finally, for $W = 32$ we still observe the phenomenon that window size and execution time is correlated. Also, notice that the execution time maxima is not correlated with the maxima of window size. This leads us to believe that the execution time of an operator is strongly dependent upon the window size of the operator, and weakly dependent upon the the number of incoming tuples.

Also observe that in the general case decreasing the window size can be beneficial in terms of decreasing execution time of an operation. This is expected and shows that the using the correct window size for heavy operators can be crucial for the system to perform. The window size cannot be adaptive be set by a system, since changing the window size will affect the accuracy of the results returned by a query.

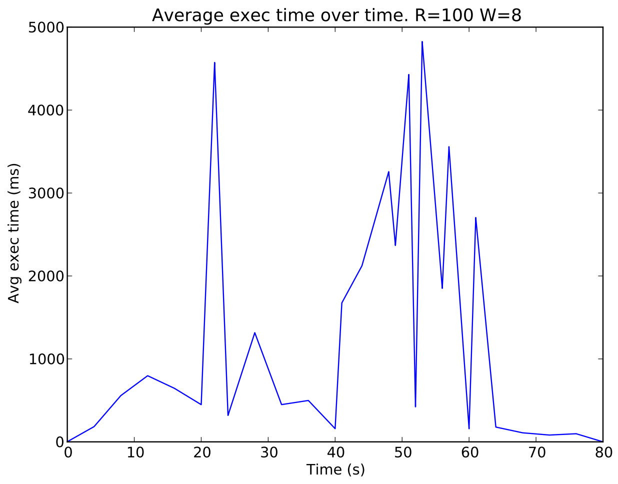
Also studying the number of tuples inside a window, one can observe that the size does not grow linearly with with the set tuples rate. This is probably due to that the number of emitted tuples from one join to another is larger than the number of incoming tuple for the former join. That also means that if the source of the input for the topology increases the window size will increase super-linearly (possibly polynomial or exponential). Thus, increasing the average execution time for the whole query.



(a)

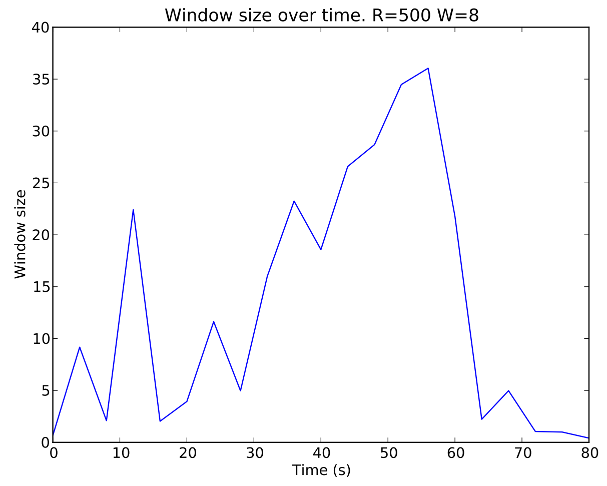


(b)

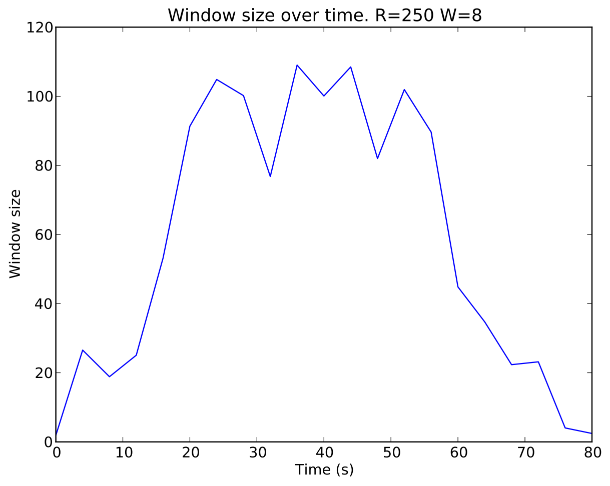


(c)

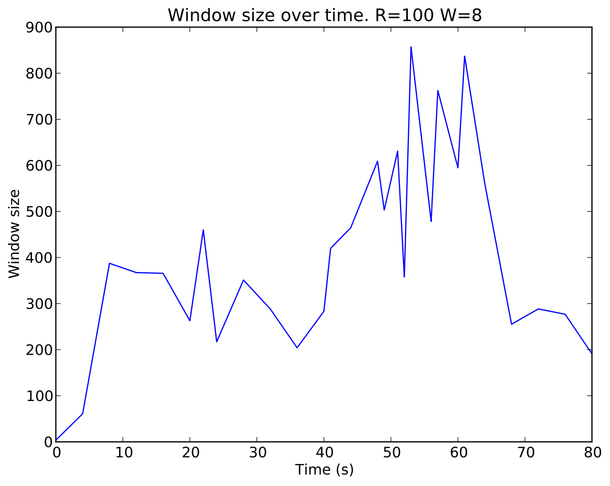
Figure 7.10: Average execution time for $W = 8$ with varying rates



(a)

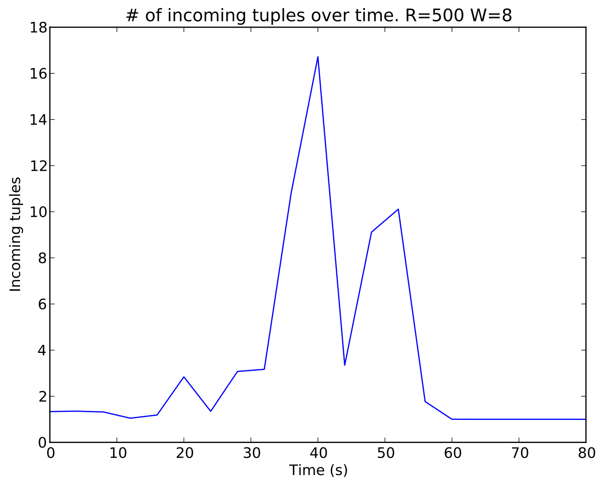


(b)

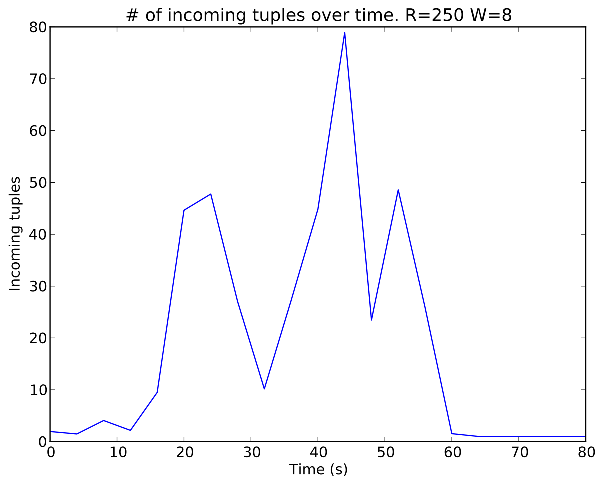


(c)

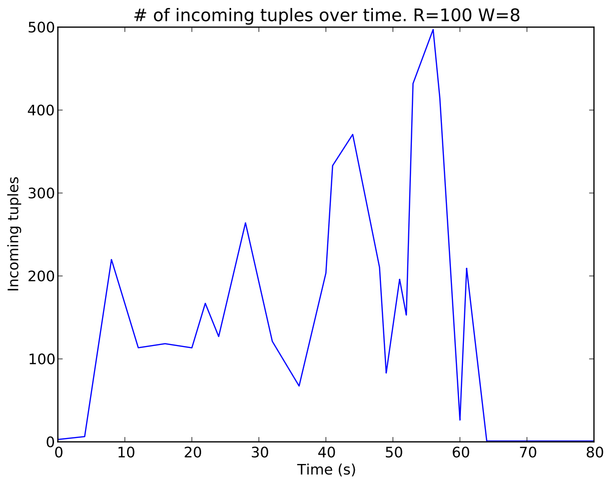
Figure 7.11: Number of tuples inside the window for $W = 8$ with varying rates



(a)

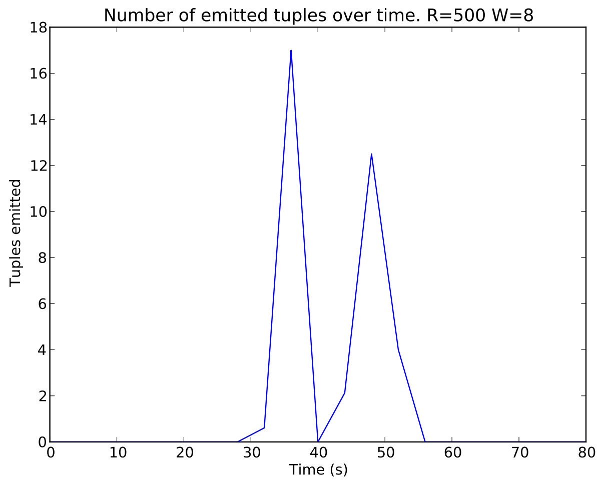


(b)

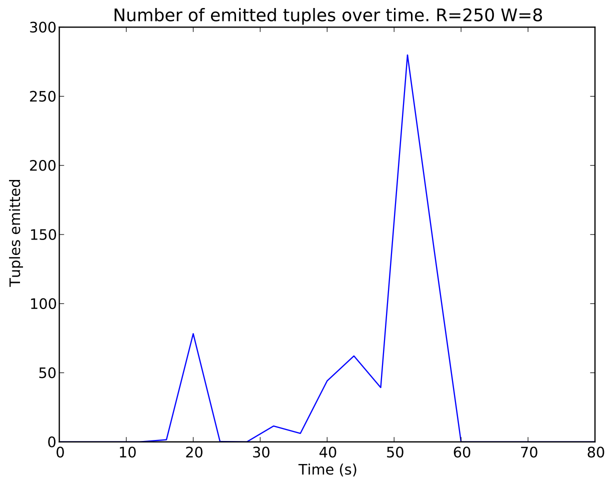


(c)

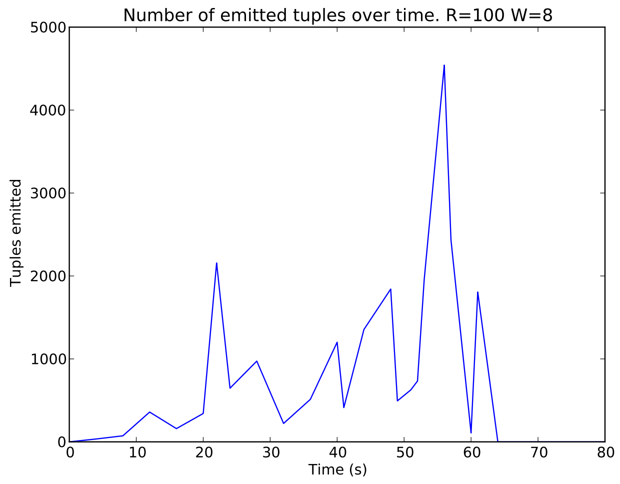
Figure 7.12: Incoming tuples for $\mathrm{W} = 8$ *with varying rates*



(a)

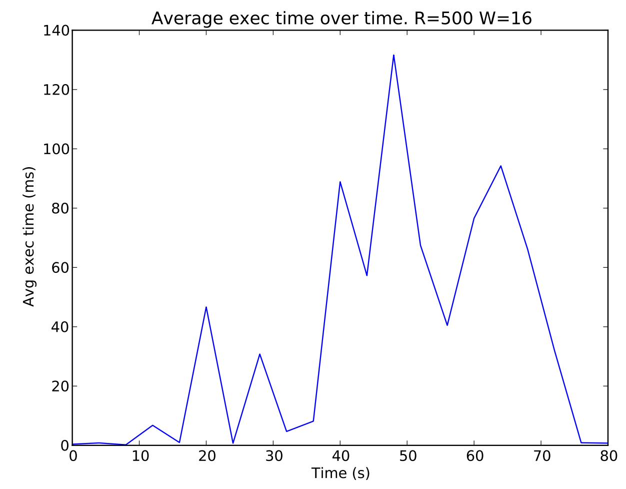


(b)

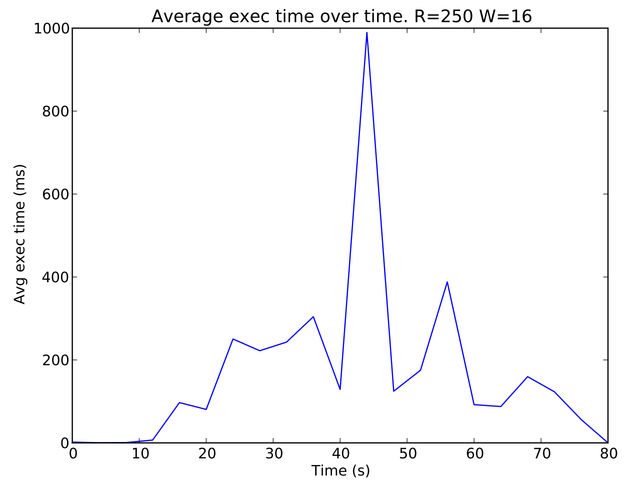


(c)

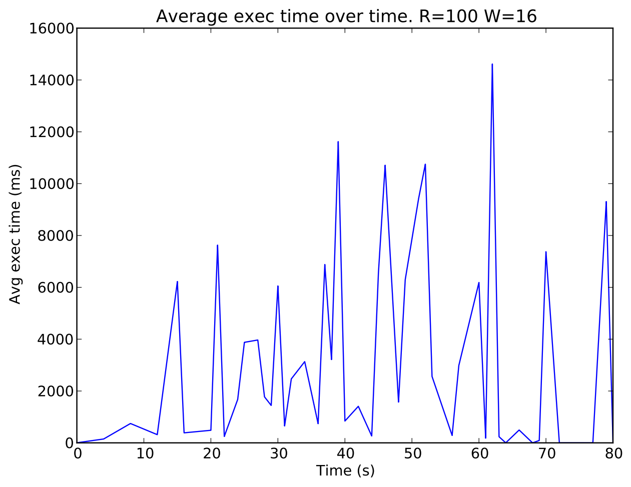
Figure 7.13: Emitted tuples for $W = 8$ with varying rates



(a)

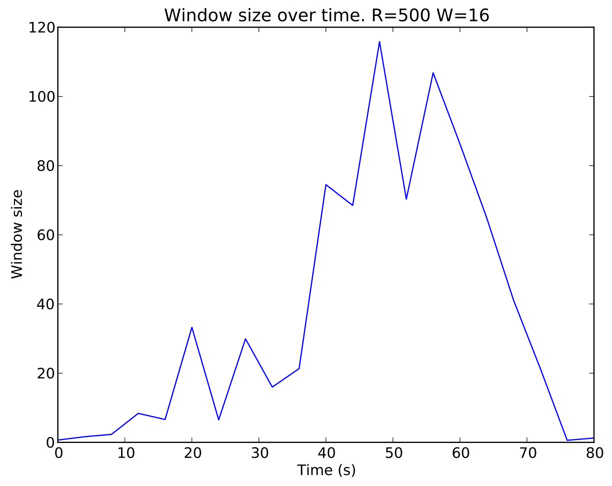


(b)

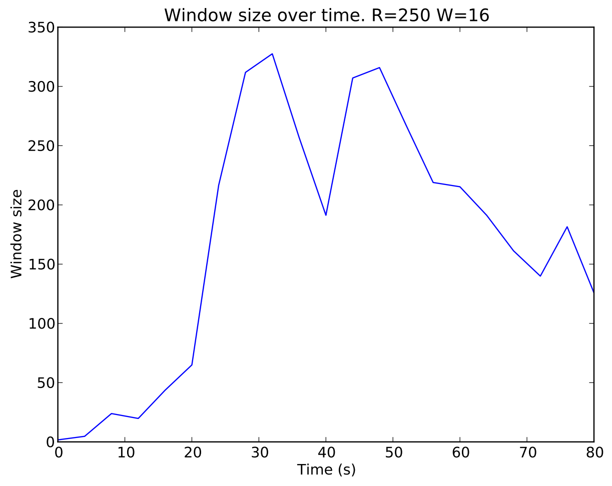


(c)

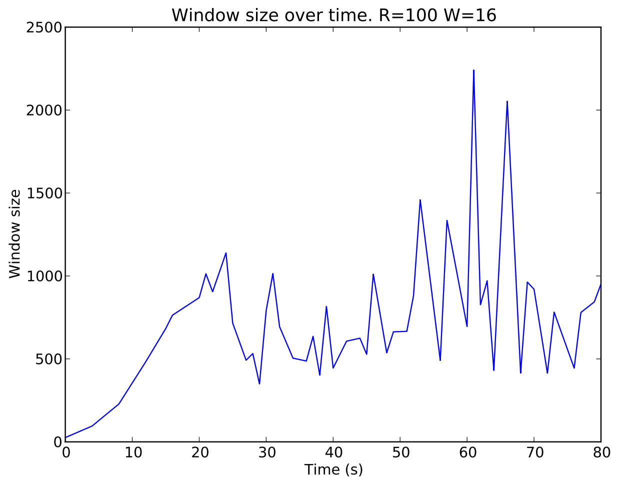
Figure 7.14: Average execution time for $\mathrm{W} = 16$ *with varying rates*



(a)

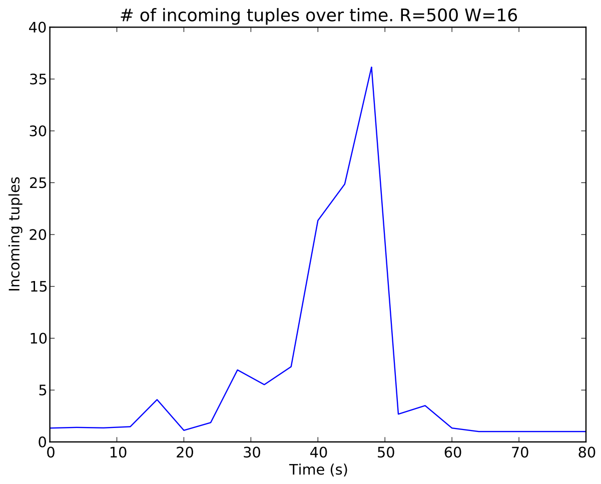


(b)

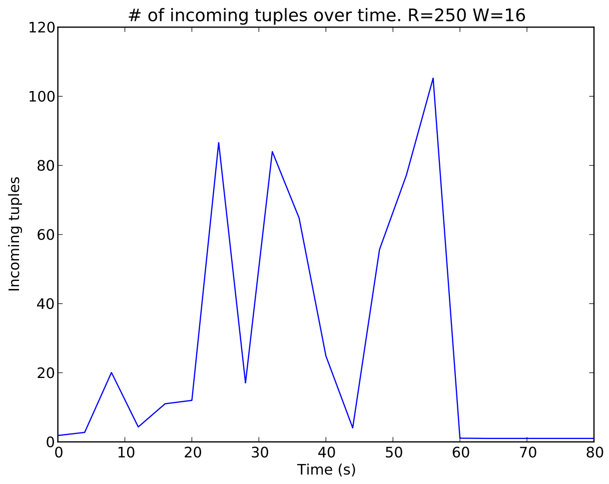


(c)

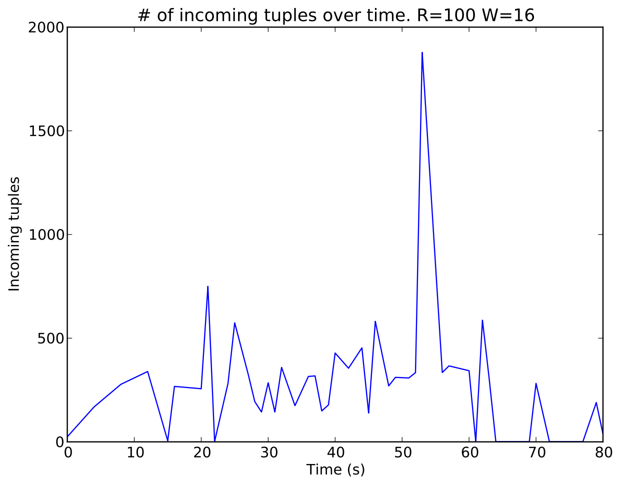
Figure 7.15: Number of tuples inside the window for $\mathrm{W} = 16$ *with varying rates*



(a)

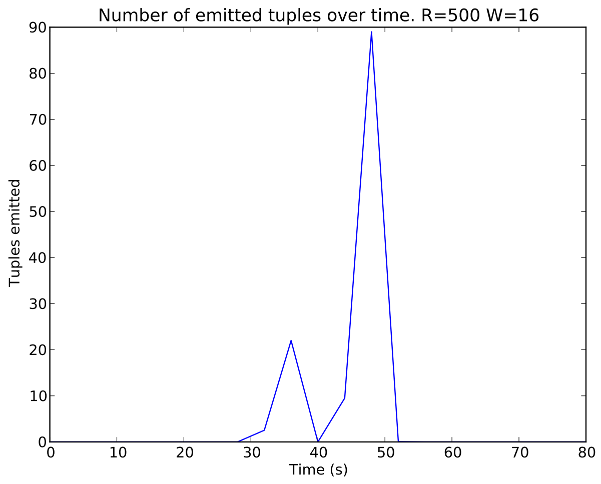


(b)

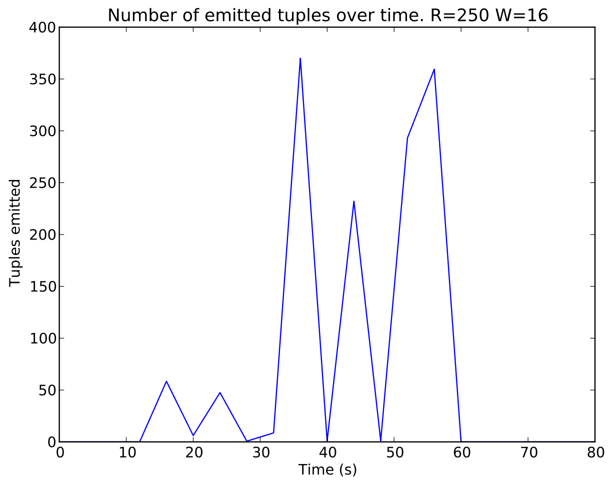


(c)

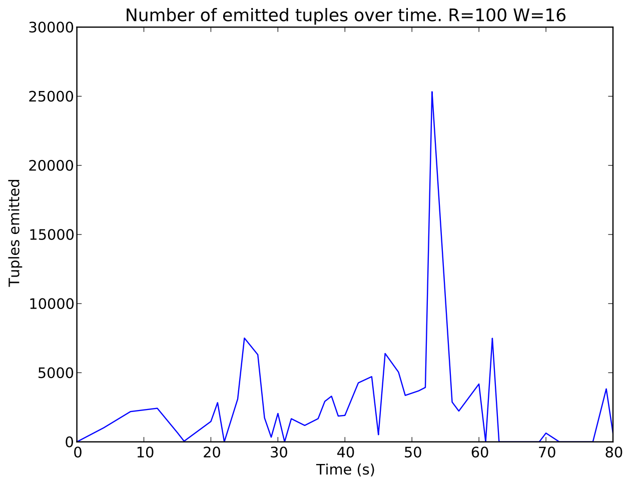
Figure 7.16: Incoming tuples for $W = 16$ *with varying rates*



(a)

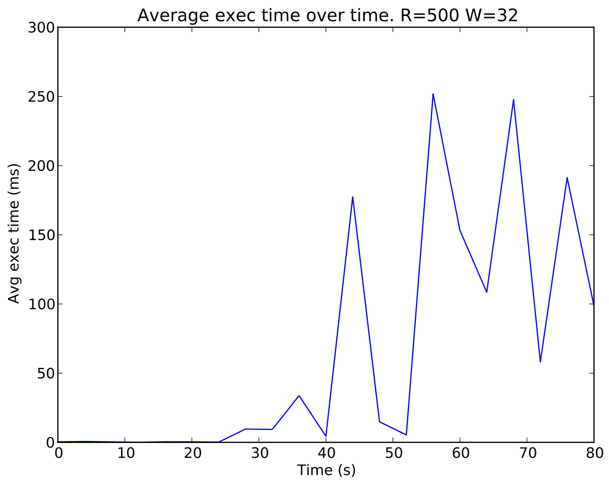


(b)

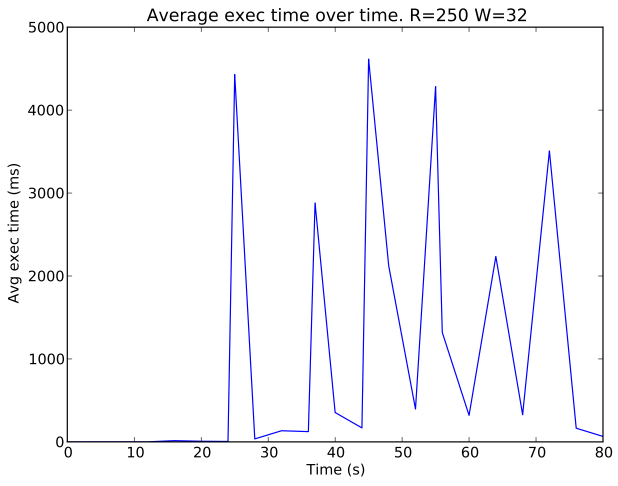


(c)

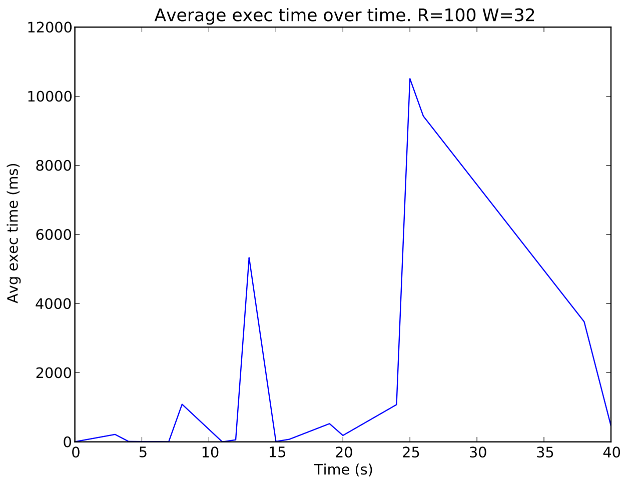
Figure 7.17: Emitted tuples for $\mathrm{W} = 16$ *with varying rates*



(a)

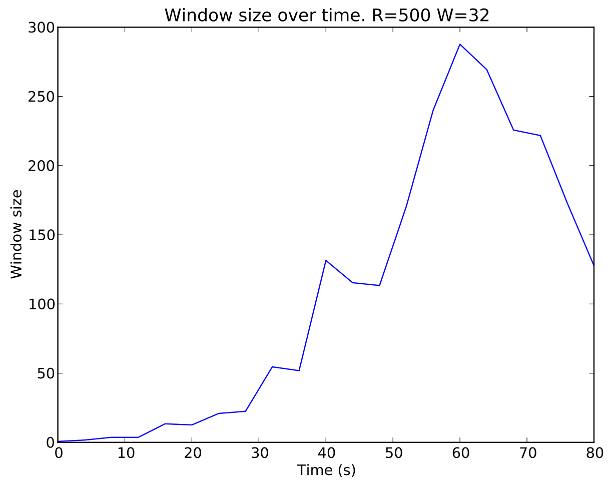


(b)

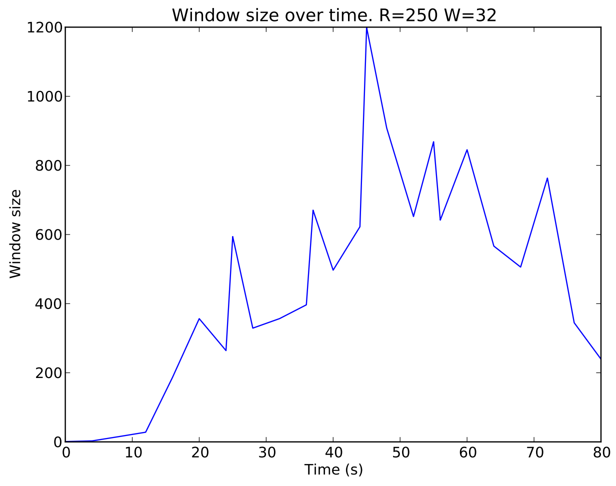


(c)

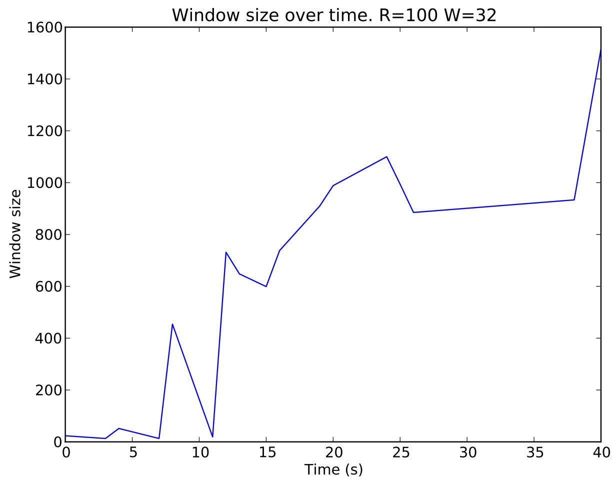
Figure 7.18: Average execution time for $W = 32$ with varying rates



(a)

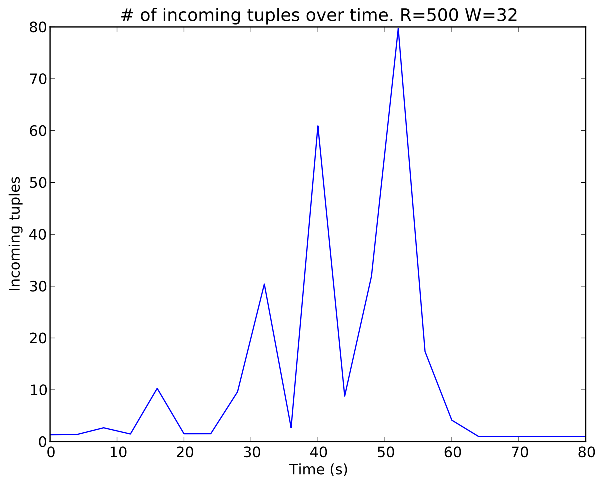


(b)

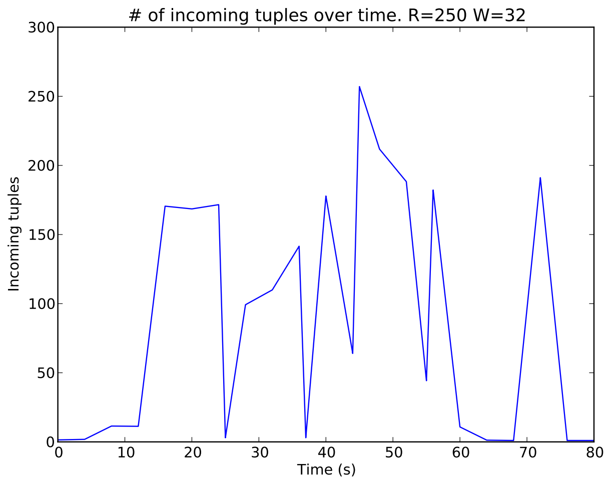


(c)

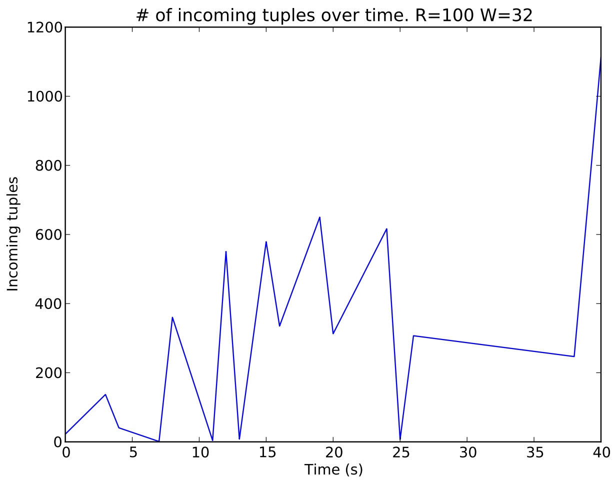
Figure 7.19: Number of tuples inside the window for $W = 32$ with varying rates



(a)

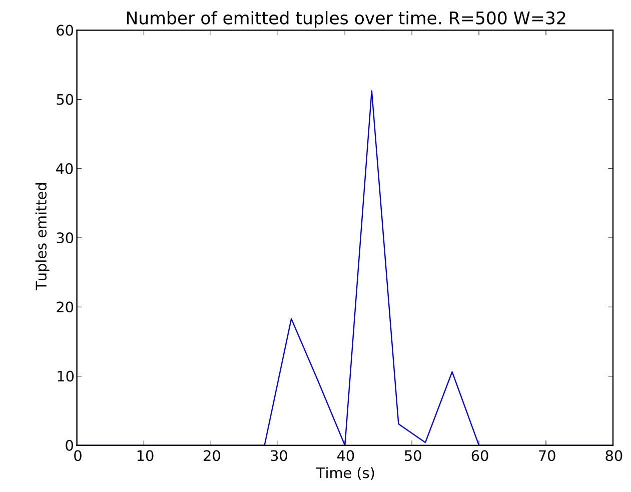


(b)

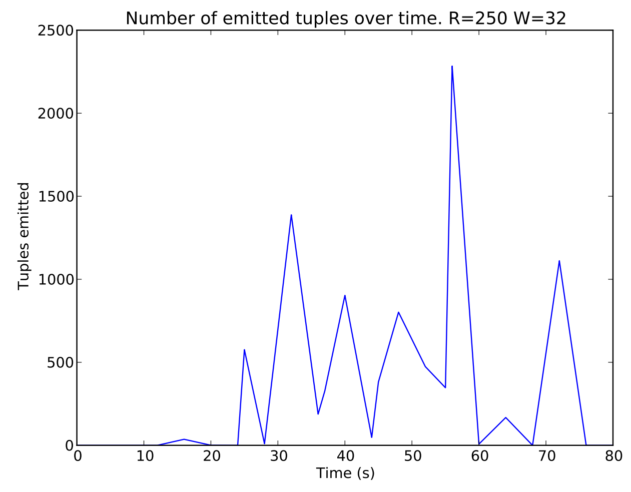


(c)

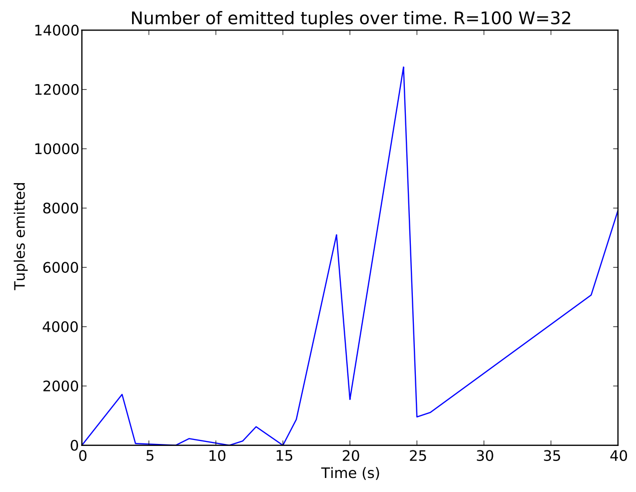
Figure 7.20: Incoming tuples for $W = 32$ with varying rates



(a)



(b)



(c)

Figure 7.21: Emitted tuples for $W = 32$ *with varying rates*

7.5.4 Time of optimization analysis

Below we presented a set of graphs with two plots, where the total number of incoming tuples is represented as the blue line, and the total number of acked tuples is presented as the green one. In this analysis we view the number of acked tuples for one operator as the total number of fully processed tuple by the same operator. Thus, if one is able to observe a significant difference between the number of incoming and acked tuples over time. One can tell if a query is lagging and therefore in need of an optimization.

Recall from Figure 7.18 presented in Section 7.5.3 that the query with $W = 32, R = \{100, 250\}$ was under-performing, and notice the corresponding graphs in Figure 7.24. We can clearly see that the number of incoming tuples exceeds the number of acked tuples by far. Using the graphs presented we can confirm that theory of when to optimize presented in Section 6.4 works in practice.

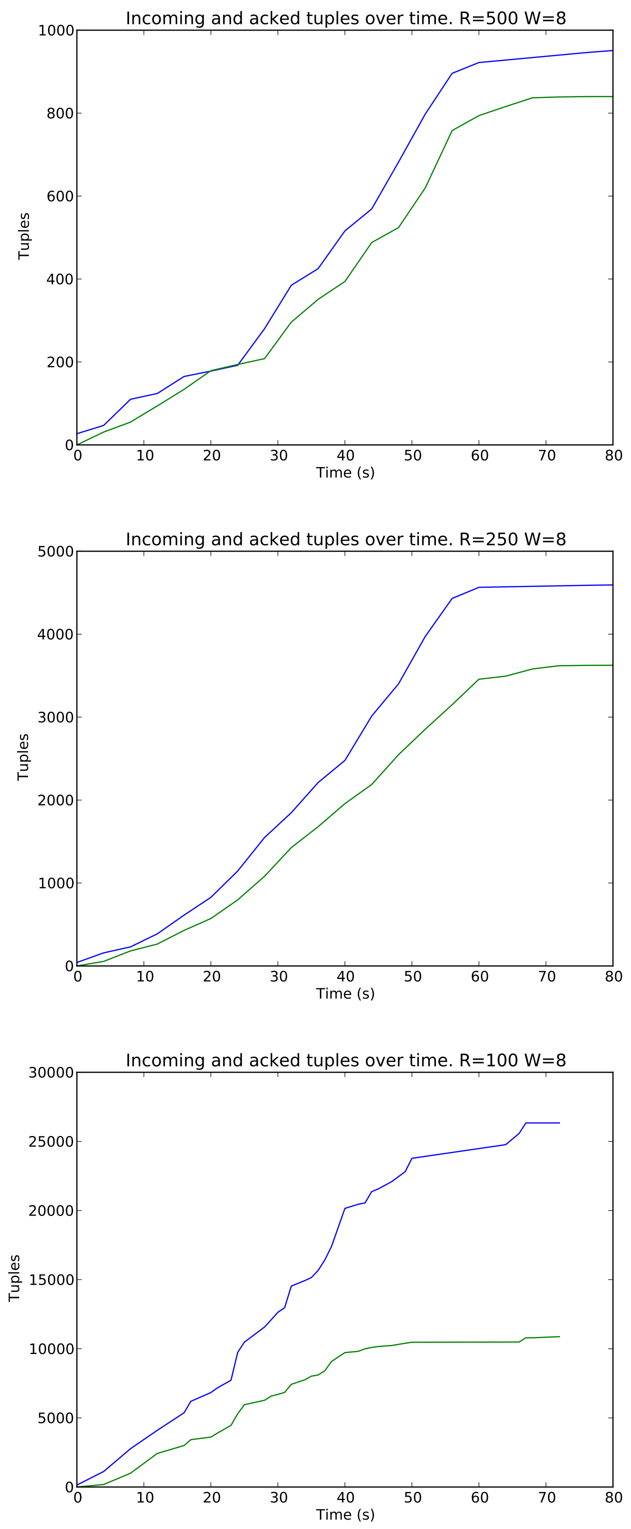


Figure 7.22: Emitted tuples for $W = 8$ with varying rates

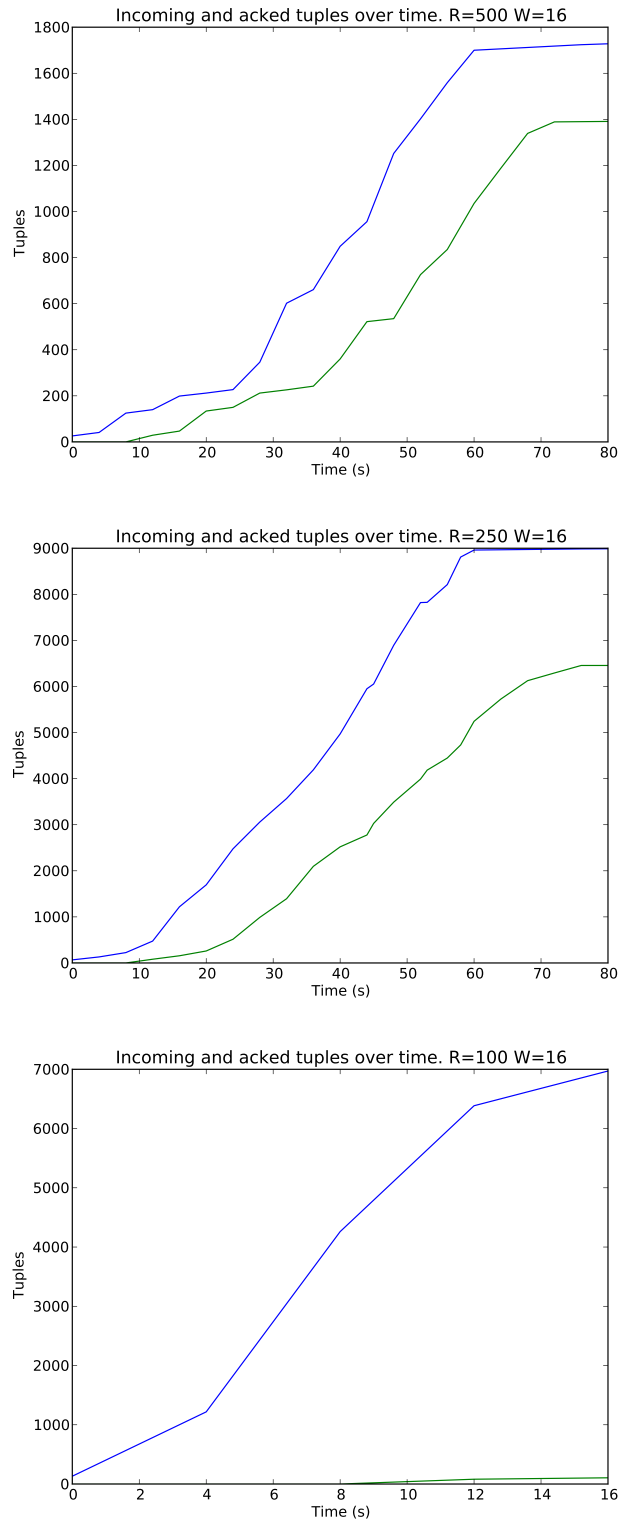
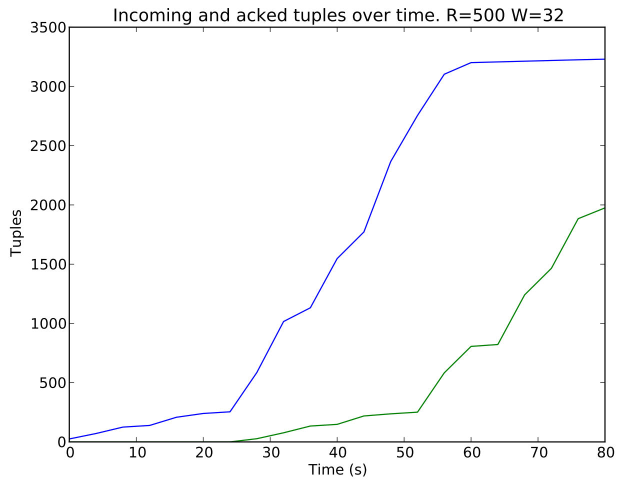
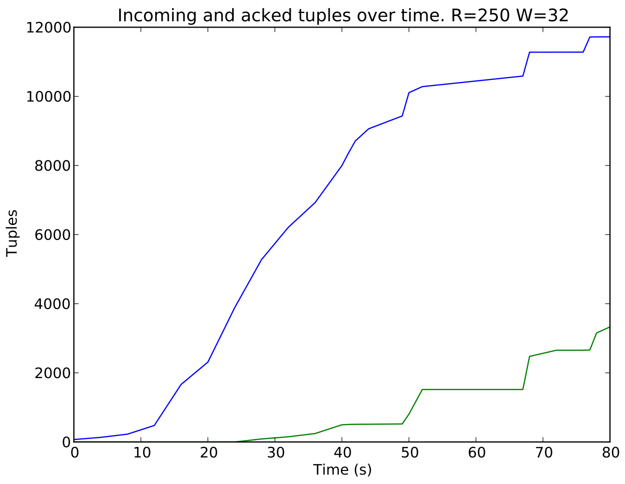


Figure 7.23: Emitted tuples for $\mathrm{W} = 16$ *with varying rates*





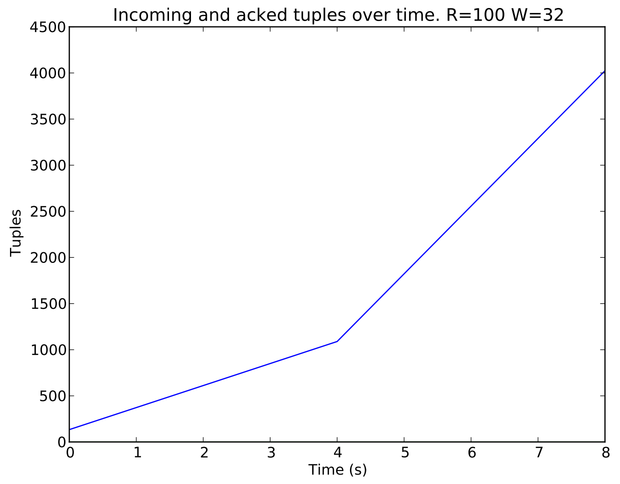


Figure 7.24: Emitted tuples for $\mathrm{W} = 32$ with varying rates

7.5.5 Results of query tree optimization

To analyze the effect of the query tree optimization we have plotted graphs that show the number of acked tuples before and after optimization. Figure 7.25 displays this for join order optimization, where the blue line represent the topology before optimization and the red one after. Figure 7.26 shows the execution time for the same cases as Figure 7.25.

What we try to achieve with the join order optimization is to order the operators in the relational algebra tree from low to high selectivity such that the operators with low selectivity gets executed first. When operators with high selectivity are executed before those with lower selectivity, we will see a spike in the number of acked tuples early on in the execution. We have achieved the desired result in the test cases where the number of acked tuples is higher before optimization than after, early on in the execution.

We can see that in three of the cases, $\{R = 500, W = 16, M = 1\}$, $\{R = 250, W = 8, M = 1\}$ and $\{R = 100, W = 8, M = 1\}$ we get a clear indication of improvement when join order optimization is applied. In one of the cases, $\{R = 500, W = 8, M = 1\}$ we observe the opposite effect, and in the rest of the cases the number of acked tuples are approximately the same.

If we look at the execution time, Figure 7.26, for the same three cases, we can see that we have a corresponding spike in execution time. Optimally, we would like to either have the same execution time, or lower execution time for the optimized trees. We did not achieve that for all the cases, but there are a couple of reasons that can explain that behavior:

1. We are using a rather naive sampling method that simply takes the 400 first tuples from each source. Sampling only the first tuples from the source is however feasible in our testing environment, as the data is coming from random data generators, and we do not expect any significant changes in the characteristics of the data. In a production environment, random sampling should be applied to capture any characteristic changes.

2. We are using a greedy algorithm for the select order. The greedy algorithm will not explore the whole search space available. Applying another algorithm, for instance a algorithm using dynamic programming can result in more stable results.

Nonetheless, the takeaway from that the order of join operators have a huge impact on the performance of the topology, and the order needs to be re-validated continuously to achieve the best optimization results.

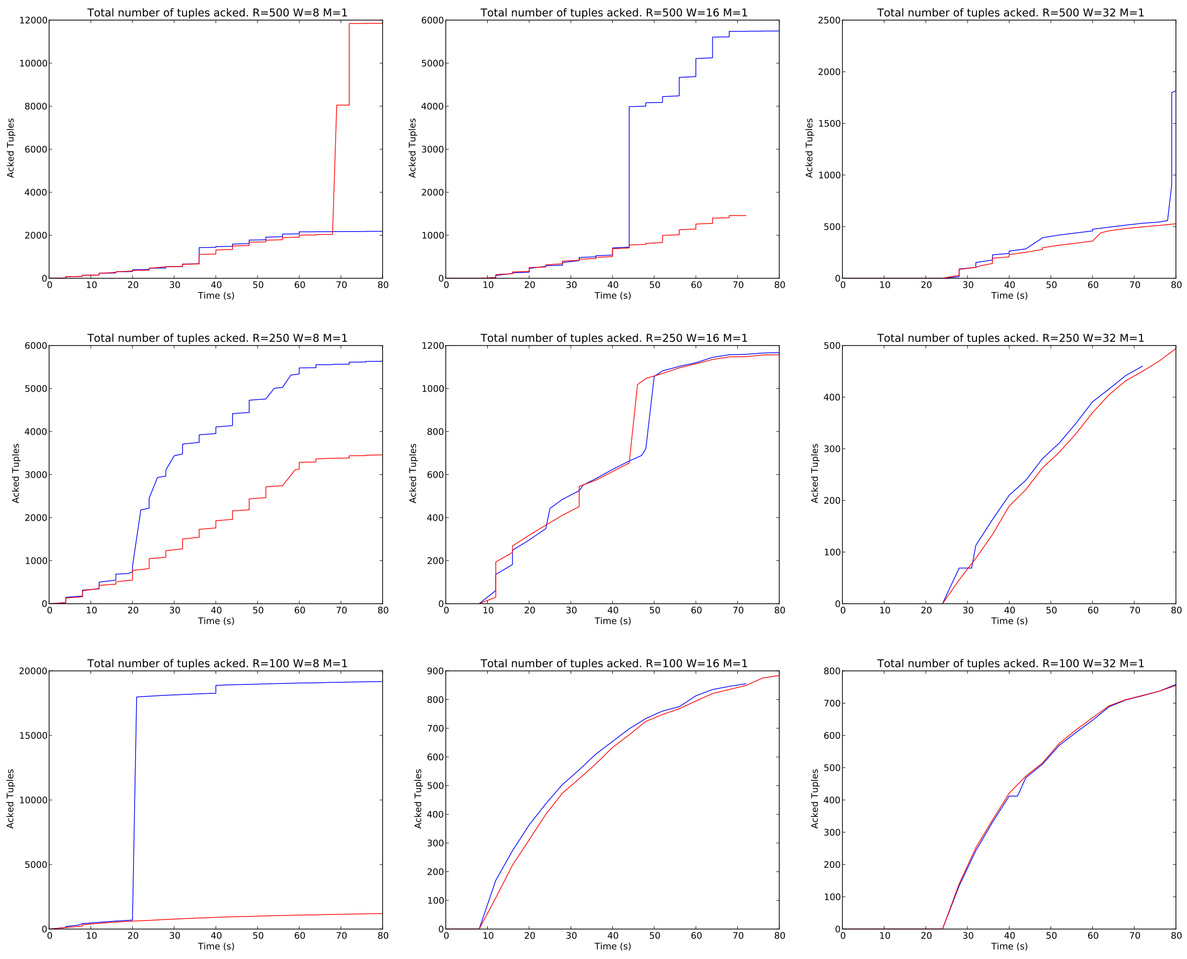


Figure 7.25: Total acked tuples over time before and after optimization

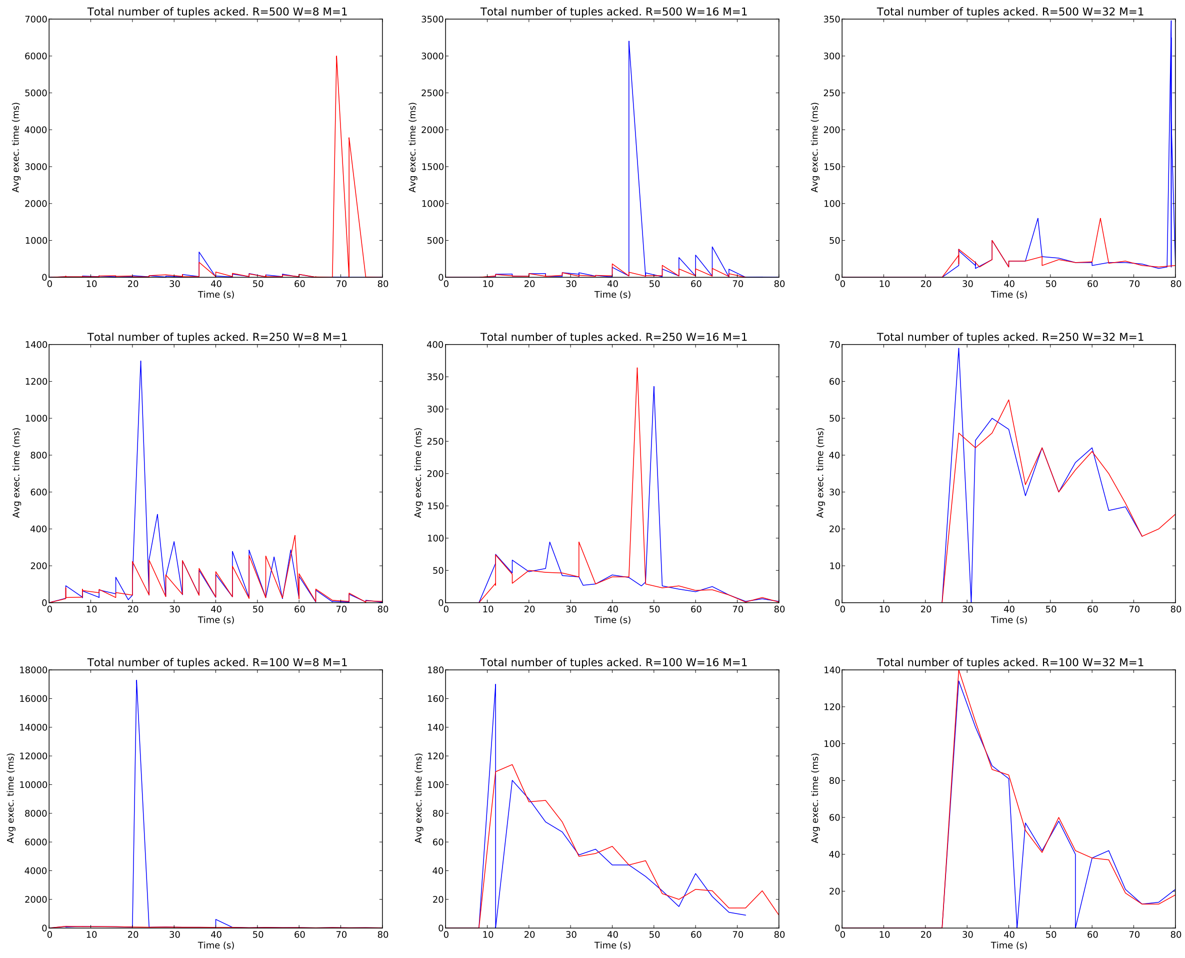


Figure 7.26: Execution time in ms over time before (blue) and after (red) optimization

Figure 7.27 shows the optimization of select operators. The blue line represent no optimization, red shows select order optimization, green shows when the select operators are merged together, and black shows when they are merged and ordered internally. The graphs on the left shows the execution time, and the right shows the number of acked tuples.

First, we need to point out that select operators does not produce any new tuples. So the only difference we can have in the number of acked tuples are when we reduce the number of operators. We can see that the number of acked tuples is lower when we merge the operators together, and it results in fewer ack-messages sent over the network which in turn reduces the network traffic.

When we look at the execution time graphs, we can see that there are no visible significant decrease in execution time between the unoptimized and the select order optimized query. This is mainly because the select operator is a cheap operator with a runtime complexity of $O(1)$ relative to the input size.

Next, we can compare no optimization with the merged version. Here we see a substantial decrease in execution time. Where the un-optimized version averages around 17ms with $R = 50$, 40ms with $R = 25$, and 60ms with $R = 10$, the merged version averages around 7ms, 12ms and 25ms.

Finally, comparing the merged and select-merged optimizations, there are no substantial difference. As we pointed out earlier, the running time of the select operator is too low to give an impact.

The takeaway from this analysis is the overhead associated with each node in the topology. By merging cheap operators together the total execution time goes down, and should be applied. But changing the order of select nodes has low-to-none impact on the execution time.

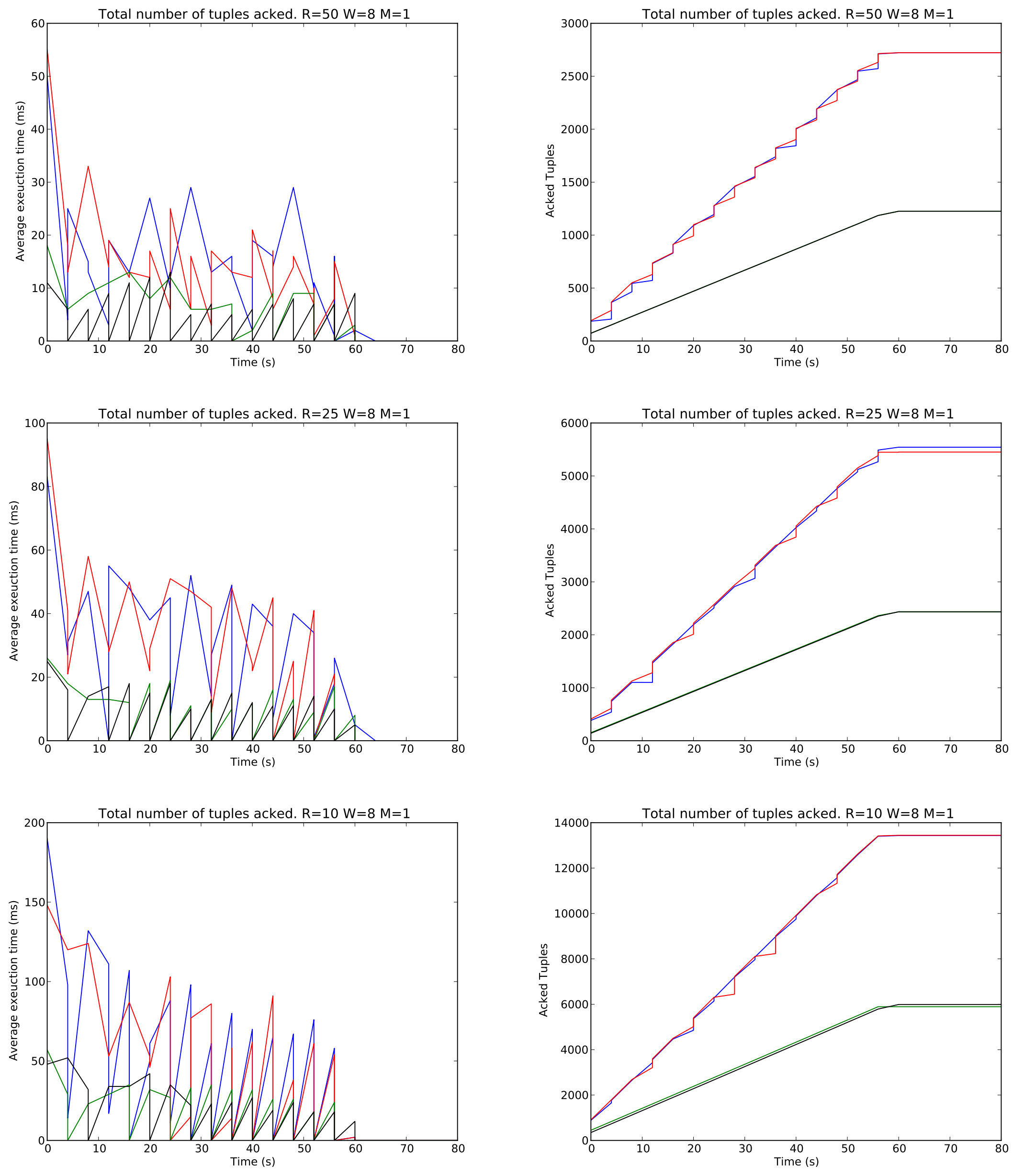


Figure 7.27: Execution time over time before and after optimization.

7.6 Concluding Remarks

Parallel optimization The results from Section 7.5.1 shows us that increasing parallelism can be used to achieve better system performance. More specifically, the results indicates that by increasing the parallelism hint for an operator would result in a decrease of the average execution time for that operator. We can use the analysis provided to propose a proper way of choosing parallelism hints for an operator while optimizing for parallelism.

Notice that the most naive algorithm presented in Algorithm 7 for estimating the needed level of parallelism will slowly converge to the ideal parallelism hint, if and only if it does not get stuck in a non-optimal local minima. If a local minima is found and it does not satisfy the $T > E_t$ condition, the algorithm will keep returning the same parallelism hint and never progress towards a potentially better performance. If no local minima is found we can observe that this algorithm will continue to increase the parallelism hint until it will find the global minima. As we can see from the analysis of Chapter 7.5.1 this algorithm will get stuck at a local minima undesirably often.

The next algorithm presented in Algorithm 10 has the same advantage as the naive one presented earlier. The main difference between these algorithms is that the latter will try to guess the ideal parallelism hint. Skipping the most naive increases in the level of desired parallelism. However, since this algorithm cannot go backwards, and by trying to guess a larger parallelism hint one might skip the a local minima that will satisfy the $T > E_t$ condition, and even skipping a global minima, resulting in a sub-optimal level of parallelism.

As we clearly can observe from the graphs presented in Section 7.5.1 we see that the system can have a global minima in average execution time at a parallelism hint less than maximum level of parallelism. Thus, an algorithm that is able to determine a parallelism hint that provide a good enough local, potentially global, minima in execution time is indeed desirable. Such an algorithm must be able to store history of average execution time for different parallelism hints tried and choose the one that results in the largest performance gain, not for each operator individually but for the system as a whole.

Window size and tick rate In Section 7.5.4 we looked at how the window size, incoming tuples, and emitted tuples affected the average execution time of an operator. We observed that it was the window size that affected the average execution time the most. We also propose that reducing the window size for queries under heavy load. Because of accuracy it might not be appropriate to reduce the window size adaptively. Therefore, assuming windows are time-based, an increase of parallelism for one or more operators under heavy load will effectively reduce the number of tuples inside the window of an operator. However, if this is not possible a user can either explicitly decrease the window size or increasing the tick rate. By increasing the tick rate a user is sacrificing response time for accuracy. While decreasing the window size one sacrifices accuracy for response time.

Time of optimization Section 7.5.4 gives us a clear indication that the theory behind when to optimize presented in Section 6.4 works. This results shows that by having access to the number of incoming and acked tuples for an operator over time one can successfully determine if either an operator solely or the query is in need of an optimization. In the former case one can determine if an operation is a candidate for parallelism optimization. In the latter case one can determine if the query is a candidate for query optimizations.

Join operator order The results of the join operator order optimization in Section 7.5.5 showed that the re-ordering had a huge impact on the execution time. However, using a good sampling technique and algorithm for creating the join order are important. In the tests we have used a naive sampling method, and a greedy select order algorithm. They have worked well for our test cases, but for a production environment where the characteristics of the data in the stream can change, an adaptive sampling method should be applied.

Select order and merge The results of the select order optimization tests in Section 7.5.5 shows us that the order of selects have little to none impact on the execution time. On the other hand, the merge optimization shows us that there are a lot of overhead for each bolt in the topology. When we merged the select bolts the execution time got reduced drastically. The overhead comes from the cost of scheduling tasks, spawning extra processes and sending tuples over network. Note that the network cost is not included in Figure 7.27, so the actual overhead is larger than the one presented in the graphs.

The takeaway is that cheap operations should be merged together to reduce the overhead in the topology, and therefore reducing the execution time. Care must be taken if you try to merge more expensive operations such as joins, and one must weight up the cost versus the benefit from such a optimization. An analysis of merging more expensive operators is out of the scope of this thesis.

Chapter 8

Conclusion and further work

8.1 Conclusion

In the fall of 2012 we created Raincoat, a high-level framework based upon Storm. Raincoat had no optimization when the first prototype was developed. In this thesis we wanted to look into the field of query optimization, and explore how existing query optimization techniques could be applied to it, as well as exploring what optimizations could be done within the Storm framework. This led us to the following research questions:

RQ1 How can we determine the cost of a Storm topology, and how can we use that data to optimize the topology?

RQ2 Which methods exists in the field of query optimization, and can they be translated into our domain?

RQ3 When should the system perform optimization on the topology?

RQ4 How should we dedicate and fully utilize the resources we have available to the topology?

In order to answer these questions, we gained an overview of the query optimization field. Based on the state of the art research, we decided to implement an adaptive query optimizer in Raincoat. The optimizer checks at a fixed interval if the system is in need of an optimization by comparing the ratio between incoming tuples and processed tuples. The aim of the optimizer is to ensure that the system is able to process inputted queries in such a way that the user does not experience any latency. The optimization is divided in two phases. The first phase performs optimization on a relational algebra tree, where a cost model is used as a basis. The second phase is to dedicate enough resources to each operator of a topology.

By investigating the parallelism capabilities of our system we naturally observed that heavy operators clearly gained more performance than lightweight operators such as selection when they were parallelized. Also, an interesting observation is that selection operators have a larger performance gain when adding machines while increasing parallelism hint. This observation was not as present when trying to parallelize join operators.

We observe that incrementally increasing parallelism hint could cause operators to get stuck in a local minima. The algorithms we have proposed in Section 6.5.5 will not be able to get past a local minima to find a parallelism hint sufficiently high for the system to perform optimally.

From testing the optimization on the relational algebra tree we found that changing the select order have little to none impact on the execution time. However, merging the select operators gave us a substantial performance boosts, so the overhead introduced by each bolt in a Storm topology affect the overall execution time of the system, and an optimizer should aim to minimize this overhead. The order of join operators did have a huge impact the execution time of the system, and care must be taken to use a sampling method that reflects the properties of the system.

In our analysis of the time of optimization we found that using the ratio between incoming and fully processed tuples is a valid heuristic. As this heuristic is based on the cardinality of tuples in the system, it can be applied to other data-stream management systems.

One aspect we have not focused on in our research is the impact the window size has on a running topology. The query optimizer can only reduce the execution time to a certain level, and the ratio between the window size and tuple input rate will determine the lower bound of the execution time of a topology.

Even though we were not able to implement all the aspects of the system discussed in Chapter 5 and 6, we were able to implement and evaluate key features of our system. We believe our contribution lies within our following three main findings (1) the algorithm for finding an ideal level of parallelism for a running operator, (2) our different approach on query optimization techniques compared to other DSMS, and (3) when to perform optimization.

8.2 Further work

Although a lot of aspects and issues of adaptive query optimization have been addressed in this master thesis, there are still some issues that we didn't have time to work on. This final section presents our thoughts on what can be done to improve Raincoat.

8.2.1 Query optimization

We have designed the optimizer to apply optimization in a pipelined fashion. So it is easy to add and replace optimization techniques. We have presented a way to do each optimization, but there exists other known methods in the field of query optimization that can be implemented in Raincoat, and there might be different situations where different optimization techniques might be better. Also, we have not addressed operator optimization at all. There has been a lot of research in this field, and existing techniques can be applied to Raincoat.

8.2.2 Custom task scheduler

Storm is a very versatile framework, which allows the user of the framework to control how the resources in the cluster is used. Exploring how to create a custom task scheduler and the effects of creating such a task scheduler should be done.

Another optimization in the Storm framework, is to create a custom fields grouping module. The fields grouping controls how tuples are routed inside the topology.

8.2.3 Resources

Resource usage is a issue that we have not focused on. In this thesis, we have assumed a fix number of nodes in the cluster. The effects of automatically adding and removing nodes in the cluster should be explored, as it can be of economic interest.

Another resource usage we have discussed but not handled is memory issues. A technique for handling the memory usage in nodes, against the window sizes should be implemented, so that Raincoat does not lose important data without the user being informed.

As mentioned in Section 6.6, the adaptive algorithm is only able to increase the number of workers, not decrease in situations where they are not needed anymore. An technique to minimize the worker usage should be applied.

8.2.4 Expanding raincoat

In the current state of Raincoat, it only supports select-project-join queries. In order for Raincoat to be useful in a production setting, it needs to support a larger subset of SQL, including operators such as group by, top-k, nested queries, the like operator etc. By expanding the language, Raincoat needs other optimization techniques to handle the new features.

Appendix A

Storm

Full code example of word count in Storm. The examples are taken from the storm-starter project on Github [1]

RandomSentenceSpout.java

```
package storm.starter.spout;

import backtype.storm.spout.SpoutOutputCollector;
import backtype.storm.task.TopologyContext;
import backtype.storm.topology.OutputFieldsDeclarer;
import backtype.storm.topology.base.BaseRichSpout;
import backtype.storm.tuple.Fields;
import backtype.storm.tuple.Values;
import backtype.storm.utils.Utils;
import java.util.Map;
import java.util.Random;

public class RandomSentenceSpout extends BaseRichSpout {
    SpoutOutputCollector _collector;
    Random _rand;


    @Override
    public void open(Map conf, TopologyContext context,
        SpoutOutputCollector collector) {
        _collector = collector;
        _rand = new Random();
    }

    @Override
    public void nextTuple() {
        Utils.sleep(100);
        String[] sentences = new String[] {
            "the cow jumped over the moon",
            "an apple a day keeps the doctor away",
```

[1] https://github.com/nathanmarz/storm-starter

```
            "four score and seven years ago",
            "snow white and the seven dwarfs",
            "i am at two with nature"};
        String sentence = sentences[_rand.nextInt(sentences.length)];
        _collector.emit(new Values(sentence));
    }

    @Override
    public void ack(Object id) {
    }

    @Override
    public void fail(Object id) {
    }

    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word"));
    }

}
```

SplitSentence.py

```
import storm

class SplitSentenceBolt(storm.BasicBolt):
    def process(self, tup):
    words = tup.values[0].split(" ")
    for word in words:
        storm.emit([word])

SplitSentenceBolt().run()

```

WordCountToplogy.java

```
public class WordCountToplogy {
    public static class SplitSentence extends ShellBolt implements IRichBolt {

        public SplitSentence() {
            super("python", "splitsentence.py");
        }

        @Override
        public void declareOutputFields(OutputFieldsDeclarer declarer) {
            declarer.declare(new Fields("word"));
        }

        @Override
        public Map<String, Object> getComponentConfiguration() {
            return null;
        }
    }

    public static class WordCount extends BaseBasicBolt {
        Map<String, Integer> counts = new HashMap<String, Integer>();

        @Override
        public void execute(Tuple tuple, BasicOutputCollector collector) {
            String word = tuple.getString(0);
            Integer count = counts.get(word);
            if(count==null) count = 0;
            count++;
            counts.put(word, count);
            collector.emit(new Values(word, count));
        }

        @Override
        public void declareOutputFields(OutputFieldsDeclarer declarer) {
            declarer.declare(new Fields("word", "count"));
        }
    }
    public static void main(String[] args) throws Exception {
        TopologyBuilder builder = new TopologyBuilder();
        builder.setSpout("spout", new RandomSentenceSpout(), 5);
        builder.setBolt("split", new SplitSentence(), 8)
            .shuffleGrouping("spout");
        builder.setBolt("count", new WordCount(), 12)
            .fieldsGrouping("split", new Fields("word"));
        Config conf = new Config();
        conf.setDebug(true);
        if(args!=null && args.length > 0) {
            conf.setNumWorkers(3);
            StormSubmitter.submitTopology(args[0], conf,
                builder.createTopology());
        } else {
            conf.setMaxTaskParallelism(3);
            LocalCluster cluster = new LocalCluster();
            cluster.submitTopology("word-count", conf,
                builder.createTopology());
            Thread.sleep(10000);
            cluster.shutdown();
        }
    }
}
```

Bibliography

[1] Daniel Abadi, Don Carney, Ugur Cetintemel, Mitch Cherniack, Christian Convey, C Erwin, Eduardo Galvez, M Hatoun, Anurag Maskey, Alex Rasin, et al. Aurora: a data stream management system. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 666–666. ACM, 2003.

[2] Arvind Arasu, Brian Babcock, Shivnath Babu, John Cieslewicz, Mayur Datar, Keith Ito, Rajeev Motwani, Utkarsh Srivastava, and Jennifer Widom. Stream: The stanford data stream management system. *Book chapter*, 2004.

[3] Morton M. Astrahan, Mike W. Blasgen, Donald D. Chamberlin, Kapali P. Eswaran, JN Gray, Patricia P. Griffiths, W Frank King, Raymond A. Lorie, Paul R. McJones, James W. Mehl, et al. System r: relational approach to database management. *ACM Transactions on Database Systems (TODS)*, 1(2):97–137, 1976.

[4] R. Avnur and J.M. Hellerstein. Eddies: Continuously adaptive query processing. *ACM SIGMoD Record*, 29(2):261–272, 2000.

[5] Brian Babcock and Surajit Chaudhuri. Towards a robust query optimizer: a principled and practical approach. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 119–130. ACM, 2005.

[6] Brian Babcock, Mayur Datar, and Rajeev Motwani. Sampling from a moving window over streaming data. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 633–634. Society for Industrial and Applied Mathematics, 2002.

[7] S. Babu, R. Motwani, K. Munagala, I. Nishizawa, and J. Widom. Adaptive ordering of pipelined stream filters. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 407–418. ACM, 2004.

[8] Stefano Ceri, Mauro Negri, and Giuseppe Pelagatti. Horizontal data partitioning in database design. In *Proceedings of the 1982 ACM SIGMOD international conference on Management of data*, pages 128–136. ACM, 1982.

[9] U. Cetintemel, D. Abadi, Y. Ahmad, H. Balakrishnan, M. Balazinska, M. Cherniack, J. Hwang, W. Lindner, S. Madden, A. Maskey, et al. The Aurora and Borealis

Stream Processing Engines. *Data Stream Management: Processing High-Speed Data Streams„ Springer-Verlag*, 2006.

[10] S. Chandrasekaran and M.J. Franklin. PSoup: a system for streaming queries over streaming data. *The VLDB Journal*, 12(2):140–156, 2003.

[11] Biswapesh Chattopadhyay, Liang Lin, Weiran Liu, Sagar Mittal, Prathyusha Aragonda, Vera Lychagina, Yonghee Kwon, and Michael Wong. Tenzing a sql implementation on the mapreduce framework. *PVLDB*, 4(12):1318–1327, 2011.

[12] S. Chaudhuri. An overview of query optimization in relational systems. In *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 34–43. ACM, 1998.

[13] Ming-Syan Chen, Philip S. Yu, and Kun-Lung Wu. Optimization of parallel execution for multi-join queries. *Knowledge and Data Engineering, IEEE Transactions on*, 8(3): 416–428, 1996.

[14] Edgar Frank Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 26(1):64–69, 1983.

[15] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk. Gigascope: A stream database for network applications. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 647–651. ACM, 2003.

[16] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[17] E. Dumbill. *Planning for Big Data*. O'Reilly Media, 2012.

[18] Michael J Franklin, Björn Thór Jónsson, and Donald Kossmann. Performance tradeoffs for client-server query processing. In *ACM SIGMOD Record*, volume 25, pages 149–160. ACM, 1996.

[19] S. Ganguly, W. Hasan, and R. Krishnamurthy. Query optimization for parallel execution. In *ACM SIGMOD Record*, volume 21, pages 9–18. ACM, 1992.

[20] L. Golab and M. T. Özsu. *Data Stream Managment*. Morgan & Claypool Publishers, 2010.

[21] Lukasz Golab and M Tamer Özsu. Issues in data stream management. *ACM Sigmod Record*, 32(2):5–14, 2003.

[22] K. Gronnbeck and S. Stenersen. Raincoat: Framework for declarative storm.

[23] Jon Olav Hauglid and Kjetil Nørvåg. Proqid: partial restarts of queries in distributed databases. In *Proceedings of the 17th ACM conference on Information and knowledge management*, pages 1251–1260. ACM, 2008.

[24] Y.E. Ioannidis and V. Poosala. Balancing histogram optimality and practicality for query result size estimation. *ACM SIGMOD Record*, 24(2):233–244, 1995.

[25] ISO ISO. Iec 9075: 1999: Information technology| database languages| sql. *International Organization for Standardization*, 1999.

[26] Z.G. Ives, A.Y. Halevy, and D.S. Weld. Adapting to source properties in processing data integration queries. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 395–406. ACM, 2004.

[27] N. Kabra and D.J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *ACM SIGMOD Record*, volume 27, pages 106–117. ACM, 1998.

[28] Jaewoo Kang, Jeffrey F Naughton, and Stratis D Viglas. Evaluating window joins over unbounded streams. In *Data Engineering, 2003. Proceedings. 19th International Conference on*, pages 341–352. IEEE, 2003.

[29] W. Lam, L. Liu, STS Prasad, A. Rajaraman, Z. Vacheri, and A.H. Doan. Muppet: MapReduce-style processing of fast data. *Proceedings of the VLDB Endowment*, 5 (12):1814–1825, 2012.

[30] Rubao Lee, Tian Luo, Yin Huai, Fusheng Wang, Yongqiang He, and Xiaodong Zhang. Ysmart: Yet another sql-to-mapreduce translator. In *Distributed Computing Systems (ICDCS), 2011 31st International Conference on*, pages 25–36. IEEE, 2011.

[31] Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Gurmeet Manku, Chris Olston, Justin Rosenstein, and Rohit Varma. Query processing, resource management, and approximation in a data stream management system. CIDR, 2003.

[32] Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. S4: Distributed stream computing platform. In *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*, pages 170–177. IEEE, 2010.

[33] L. O'Callaghan, N. Mishra, A. Meyerson, S. Guha, and R. Motwani. Streaming-data algorithms for high-quality clustering. In *Data Engineering, 2002. Proceedings. 18th International Conference on*, pages 685 –694, 2002. doi: 10.1109/ICDE.2002.994785.

[34] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110. ACM, 2008.

[35] Raghu Ramakrishnan and Johannes Gehrke. *Database management systems*. Osborne/McGraw-Hill, 2000.

[36] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. In *ACM SIGMOD Record*, volume 29, pages 249–260. ACM, 2000.

[37] Sangeetha Seshadri, Vibhore Kumar, and Brian F Cooper. Optimizing multiple queries in distributed data stream systems. In *Data Engineering Workshops, 2006. Proceedings. 22nd International Conference on*, pages 25–25. IEEE, 2006.

[38] J.A. Stankovic et al. Real-time computing. *Invited paper, BYTE, pp.(155-160)*, 1992.

[39] Michael Steinbrunn, Guido Moerkotte, and Alfons Kemper. *Optimizing join orders*. Citeseer, 1993.

[40] Mark Sullivan. Tribeca: A stream database manager for network traffic analysis. In *Proceedings of the International Conference on Very Large Data Bases*, pages 594–594. INSTITUTE OF ELECTRICAL & ELECTRONICS ENGINEERS (IEEE), 1996.

[41] Jeffrey S Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software (TOMS)*, 11(1):37–57, 1985.

[42] C.A. Waldspurger and W.E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, page 1. USENIX Association, 1994.

[43] Sai Wu, Feng Li, Sharad Mehrotra, and Beng Chin Ooi. Query optimization for massively parallel data processing. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 12. ACM, 2011.

[44] Y. Zhu, E.A. Rundensteiner, and G.T. Heineman. Dynamic plan migration for continuous queries over data streams. In *International Conference on Management of Data: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, volume 13, pages 431–442, 2004.