Imperial College of Science, Technology and Medicine

Department of Computing

Stream Processing in the Cloud

Wilhelm Kleiminger

Submitted in part fulfilment of the requirements for the
MEng Honours degree in Computing of Imperial College, June 2010

Abstract

Stock exchanges, sensor networks and other publish/subscribe systems need to deal with high-volume streams of real-time data. Especially financial data has to be processed with low latency in order to cater for high-frequency trading algorithms. In order to deal with the large amounts of incoming data, the stream processing task has to be distributed. Traditionally, distributed stream processing systems balanced their load over a static number of nodes using operator placement or pipelining.

In this report we propose a novel way of doing stream processing by exploiting scalable cluster architectures as provided by IaaS/cloud solutions such as Amazon's EC2. We show how to implement a cloud-centric stream processor based on the MapReduce framework. We will then design a load balancing algorithm which allows a local stream processor to request additional resources from the cloud when its capacity to handle the input stream becomes insufficient .

Acknowledgements

I would like to thank:

- My supervisor Peter Pietzuch for the project proposal and his great support and encouragement throughout the project.
- My second supervisor Alexander Wolf for the insightful discussions on MapReduce.
- Eva Kalyvianaki for her help throughout the project and all the great suggestions leading to this report.
- Nicholas Ng, for proof-reading the script and coming up with the ingenious acronym M.A.P.S. for the custom Python MapReduce implementation. M.A.P.S. stands for "My Awesome Python Script".
- My sister Lisa for proof-reading and my parents Elke and Jürgen for their great support throughout school and university. Vielen Dank!

Contents

Chapter 1

Introduction

Today's information processing systems face formidable challenges as they are presented with new data at ever-increasing rates [19]. In response, processing architectures have changed with a new emphasis on parallel architectures at the on-chip level. However, research has shown that an increase in the number of cores cannot be seen as a panacea. As cores increase in number and speed, communication becomes increasingly a bottleneck [4]. Alternative solutions like clusters are still preferred for high performance applications. So while the traditional PC has seen advances in multi-core architectures, much of this effort is complemented by a move from local to cloud processing. Cloud-based computing seeks to address the issue that while in most cases today's computational resources are idling, they may still not be adequate in peak load situations. By sharing resources and requesting more power when needed, we can overcome these bottlenecks. The result should improve both latency and reduce the cost to the user. This projects seeks to evaluate a novel way of load balancing data intensive stream processing queries into a scalable cluster. The goal is to exploit the scalability of a cloud environment in order deal with peaks in the input stream.

Cloud computing has certainly been one of the most hyped trends of the last few years. The result is that companies of tacked this name to a variety of different service offerings. This makes it difficult to come up with a single, concise definition. Kunze and Baun [27] have derived a good definition from Ian Foster's definition of grid computing [24]:

> Cloud computing uses virtualised processing and storage resources in conjunction with modern web-technologies to deliver abstract, scalable platforms and applications as ondemand services. The billing of these services is directly tied to usage statistics.

We can distinguish between three applications of Cloud Computing: *Infrastructure as a service* (IaaS), *Platform as a service* (PaaS) and *Software as a service* (SaaS) [27]. IaaS describes a service that offers computational resources for distributed applications. The infrastructure is flexible enough for the user to run his own operating system and applications. The administration of the system lies mostly with the user. PaaS takes some of the administration away from the user and allows some (limited) programming of the resources. An example for this is Google's App Engine. Finally, and probably most exposed to the general public are SaaS applications. These are offerings such as Apples MobileMe and Google's email and text processing applications. They offer little to no customisation but the convenience of storing data off-site. We are interested in applying IaaS services to the computation of financial algorithms.

Recent years have seen a massive increase in algorithmic trading. Billions of pounds are traded by software [40]. More than a quarter of all equity trades at Wall Street come down to algorithms

with little to no human intervention [21]. Financial markets emit hundreds of thousands of messages per second [31]. The exchanges have responded to the demand. The delay between the time a trade is placed and filed at the Singapoore exchange for example has dropped to around 15 milliseconds and other exchanges are following suit [33]. An algorithmic trading systems must therefore process real time data streams with very low latency to in order to stay competitive. The arms-race over ever faster responses to changing market conditions necessitates highly scalable stream processing systems.

Stream processing systems are fundamentally different to ordinary data processing systems. Streams are often too large, too numerous or the important events too sparse [29]. This means data has to be processed on the fly by pre-installed queries. In most cases only a small number of tuples is interesting to the trader. The job of a stream processing system is to find these and make them available in a manner similar to traditional publish/subscribe systems.

A number of systems have emerged to deal with these problems. The distributed systems community coined the term Complex Event Stream Processing for evaluating the output of sensor networks. A similar approach taken by database vendors such as Oracle is simply called Event Stream Processing. The difference between the two is that the former advocates a publish/-subscribe approach [20], whereas the database vendors promote SQL and distributed databases. Current systems distribute the query over a number of nodes [13], thus focussing on the complexity of the query itself. We feel that these techniques are too rigid to dynamically scale in a cloud environment. Instead we have chosen to extend the MapReduce paradigm to enable streaming queries.

MapReduce is based on ideas from functional programming. Two functions, map and reduce take over the task of implementing the query. This technology has been successfully employed by Google [19] and others [10] [25] [35] and is supported in by various IaaS providers [5] [6] [7]. As MapReduce has orginally been designed for batch-processing, it will have to undergo some changes to be applied to stream processing. Recently, a first step towards streaming MapReduce has been made with the Hadoop Online Prototype (HOP) [18]. We will build on this work to show how a MapReduce stream processor can be implemented.

The choice to use the MapReduce framework is motivitated by our goal to provide efficient load balancing into the cloud. As we will show later in this report, the data rate of a stream is likely to vary a lot. From our data set, we found the highest demand on the stream processor occurs in the morning, presumably as many trades are carried over from the previous day. The whole trading session lasts 6.5 hours. This is only just over a quarter of a day. Most trading is done on work days. To provide resources 24/7 would not be economical [16]. Instead we opt to design a load balancing algorithm which dynamically responds to bursts in the input stream and relieves the strain on a small-scale, local stream processor by out-sourcing some of the computation to the cloud. The MapReduce implementation on the cloud should then scale as more computational power is required.

1.1 Contributions

In this report we seek to complement the current state of the art in stream processing techniques with the following contributions:

1. **Streaming extension for the Hadoop framework** We will design and implement the network components necessary to run a streaming MapReduce query on top of the Hadoop framework.
2. **MAPS: A Lightweight MapReduce framework** written in Python. Starting from the origins of MapReduce in functional programming, we describe the design and implementation of a simple MapReduce stream processor written in Python. The design draws from the lessons learned while working on the Hadoop framework.
3. **Loadbalancing strategies** to use the cloud in an existing stream processing setup. We show the design and implementation of a minimal version of a single node MapReduce stream processor and how its resources can be complimented by our cloud implementation with two load balancing strategies.
   (a) **Always-on** approach: The cloud's resources are always used to complement the local stream processor.
   (b) **Adaptive** approach: The cloud's resources are used on-demand to assisst the local stream processor.
4. We will **evaluate the benefits and limitations** of MapReduce for stream processing applications. We will compare our two cloud-based stream processing solutions. Taking the results into account, we will conclude by evaluating our loadbalancing techniques with respect to their ability to assist a local stream processor.

1.2 Outline of this report

In Chapter 2 we will look at existing stream processing solutions, cover the background for our MapReduce implementation and discuss some existing load balancing strategies. We will finish with a short introduction to the financial concepts behind our streaming queries. Having laid the theoretical foundations, Chapter 3 shows how the HOP/Hadoop framework can be extended to process streaming queries. Building on the experiences from the Hadoop stream processor, Chapter 4 focuses on the design and implementation of a custom prototype for a MapReduce stream processor. With a cloud-based solution in place, Chapter 5 focuses on the design of suitable load-balancing algorithms. In Chapter 6, we will evaluate the MapReduce paradigm in the context of financial queries. We will conclude this report by looking at the performance of the load balancing algorithms designed in Chapter 5.

Chapter 2

Background

The goal of this project is to enable stream processing in the cloud by using a scalable MapReduce framework. In addition, we are going to evaluate a load balancing algorithm which allows us to utilise the resources of an IaaS provider on-demand. For the purpose of implementation and evaluation we will use a set of local nodes in the college's datacentre. However, since this is a homogeneous cluster, the results should be easily transferable to a real IaaS service. In order to introduce this ultimate goal of running the MapReduce stream processor with an IaaS provider, we introduce Amazon's EC2 offering in §2.2.5.

We will start this chapter with a short detour into finance to gain an insight into possible areas of applications of a stream processor and to explain the reasoning behind *Put/Call parities* - our chosen test query.

After this, we will formally introduce the MapReduce algorithm as designed by Google [19]. The MapReduce algorithm enables data processing on a large number of off-the-shelf nodes. This means that it has now become a focal point of any discussion on cloud computing in general. We will discuss its limitations and why we have initially chosen one of its variations (see §2.3.4) to be part of our implementation. We will then introduce Sawzall, an effort by Google to create a domain specific language for MapReduce and proceed to Hadoop, an open-source implementation of the MapReduce framework. After having laid the algorithmic foundations, we will discuss the underlying infrastructure. Having covered the basics of MapReduce and the underlying platforms, we will concentrate on the current state of the art in stream processing systems and discuss why we have chosen to follow the rather novel path of utilising the MapReduce paradigm. We will conclude this chapter by looking at techniques currently employed in load-balancing and explain how these are applicable to our problem.

2.1 Financial Algorithms

As laid out in the introduction, the financial services industry constitutes a major area of application for stream processing. In our tests we are planning to use stream processing in the cloud to compute financial equations. We have a set of financial data available. This set contains the quotes for options at various stock exchanges for a single day. In order to formulate a sensible query, we must understand the rationale behind the financial data given. In the following we will discuss the concepts behind put and call options as well as the concept of futures trading. At the moment, we are aiming to deploy a single query over the MapReduce network. We must thus formulate a query which makes sense both under the MapReduce as well as financial paradigms.

2.1.1 Foundations

Options An options contract can be described as follows. Farmer Enno sells investor Antje the right to buy next year's harvest at a specific *strike price*. Obviously, neither knows the actual value of the harvest. Enno produces biofuel. Now say the economy dips into recession the following year and substitute goods such as oil become cheaper. Then Enno's biofuel will drop in price as well and Antje is going to drop her option. She has lost the commission and any other associated fees. However, if the economy is well and Enno has an exceptional harvest, Antje will exercise her option. The price of the option is below the actual market price. Antje will be able to sell the fuel at a premium.

Futures In contrast to options, futures are binding contracts over the purchase of a good. If Antje buys a futures contract on Enno's harvest, she is obliged to buy it at the *spot price*. This guarantees Enno a specific price for his harvest and insures Antje against rising costs. Our dataset only includes options data and thus we will not discuss futures any further. Below we define a few financial terms necessary for the further discussion.

Short selling A buyer *borrows* a position (eg. shares) from a broker, betting its price will fall. The broker receives a commission. The buyer immediately sells the position (going short). With prices falling, the buyer can now re-acquire the position and return it to the broker. The buyer has made a profit of the difference between the initial sell and final buy actions minus the commission. The buyer makes a loss if prices rise as he has to buy at a premium to his initial sell price.

Long buying This is the opposite to short selling and describes buying positions and betting that prices will rise in future.

Bonds Bonds are similar to shares. Shareholders are owners of the company and shares can be held indefinitely. Bondholders are creditors of the company and bonds are usually associated with a due date. This time period is called *maturity*. As shares usually pay dividends, bonds have an associated interest rate called *coupon*.

2.1.2 Put and call options

In options trading we distinguish two types of financial produces. *Put* and *Call* options. Put and call options are short selling and long buying applied to options. If Antje thinks that the price of Enno's harvest will decrease in future, she can buy a *put*. A put means that Antje is going short on the right to buy Enno's harvest. Should the weather outlook dictate a higher price for the option, Antje will have to buy back the option at a premium and therefore lose money. However, if prices of biofuel seem likely to fall, Antje is most likely to be able to buy back the option at a cheaper price and return it to the broker at a premium. In options trading, the seller, previously referred to as broker is called *writer*. Obviously, the writer's profit is maximised if the buyer chooses not to exercise the option.

A *call* option is the name for acquiring the right to buy shares of stock at a specific strike price in the future. In the above paragraph on options, we have described this simple case of options already.

2.1.3 Arbitrage opportunities

Put-call parity

Put-Call parity is a relationship between put and call options which mainly applies to European options[1] [12]. This concept is important for valuing options but can provide a arbitrage opportunity.

Portfolio A We define the following variables for the put:

- **K** Strike price at time T
- **S** Share price on day of expiry (unknown constant at time T)

Now, assume we have a portfolio with a single put position at strike price K (short option) and a single share at time T. Should the (unknown) share price S, be the same or exceed K, the value of our portfolio is S as we do not wish to exercise the option. Should the strike price, however, be greater than S we would like to exercise the option and our portfolio is worth the value of the put, $K - S$ plus the value of the share S: $K - S + S = K$.

Portfolio B For the call we define the following variables:

- **K** K bonds each worth 1 unit (constant value)
- **S** Share price on day of expiry (unknown constant at time T)

A portfolio with a single call position (normal option) and K bonds (each worth 1) is worth the same as A if their strike price and expiry are the same. K always remains the same. Like above if at time T, the strike price K is less than the (unknown) share price S, we wish to exercise the right to buy stock at K and make a profit of $S - K$. The total value of our portfolio in this case is S. If the strike price K is greater than S, we will chose not to exercise the option and therefore have a portfolio worth K. This shows that at time T, both portfolios have the same value regardless of the relationship between T and S.

If one of the portfolios was cheaper than the other, then there would be an arbitrage opportunity since we could go long on the cheaper one and go short on the more expensive one. At the expiration T the portfolio will have zero value. But any profit made before is kept.

Relation to our project For our purposes, when we talk about "put-call parity", we simply wish to find out if we can find two markets with a put and a call options at the same strike price and with the same expiry date. As we do not have any information about the rest of the market, we cannot evaluate the financial formulae using prices for bonds, shares and dividend/coupon payments. We envisage that the full implementation will be possible in a system using multiple queries.

[1] option cannot be exercised before expiration

2.2 Cloud computing

2.2.1 MapReduce

MapReduce was first introduced by Google in 2004 [19]. It is a framework designed to abstract the details of a large cluster of machines to facilitate computation on large datasets. The inspiration for the MapReduce concept was taken from functional programming languages such as Haskell. Nowadays, many companies have implemented their own MapReduce frameworks. Open source implementations exist. In the following we will describe how MapReduce works and how it can be applied to our task.

Google File System (GFS) MapReduce works by distributing tasks over a large number of individual nodes. Therefore, the implementation is often accompanied by a distributed file system. In the original Google implementation this is GFS [19], the Google File System. GFS is particularly suited for MapReduce as the framework assumes that files are large and updated through concatenation rather than modification. In GFS, files are divided into chunks of 64MB each and then distributed across several chunk servers [39]. Replication is used so that we can recover from failures such as a machine becoming disconnected from the network.

How it works The main goal of MapReduce is to prevent the user from creating a solution that requires a lot of synchronisation. All synchronisation is done within the MapReduce framework. This way we avoid the pitfalls induced from race conditions and deadlocks. We can focus on the actual computation of values. In order to simplify data handling, the programming model also specifies the input and output as sets of key/value pairs.

The MapReduce algorithm then follows a divide and conquer approach to compute a set of output pairs for a given set of input pairs. This is done through two functions: Map and Reduce. Using the divide and conquer approach, the master node distributes the input tuples over the set of worker nodes by splitting the problem into a smaller sub problems. The map phase can form a tree structure in which problems are recursively split into smaller sub-problems before being passed to the reduce phase. Similarly, the output of the reduce phase can be fed back into the system and start another map reduce iteration. The necessary work is done within the MapReduce library.

In Haskell notation, one would describe the map and reduce functions in the following way:

```
map :: (key_1, value_1) -> [(key_2, value_2)]
reduce :: (key_2, [value_2]) -> [value_3]
```

The map function takes a `(key, value)` pair and produces a list of pairs in a different domain. The framework takes these pairs and collates values under the same key. The resulting pair of `(key, [value])` is then processed by the reduce function to produce output values in a (possibly) different domain. The original C++ implementation uses string inputs and outputs and leaves it to the user to convert to the appropriate types. The following map and reduce functions illustrate how the MapReduce framework can be used to count the occurrences of each word in a large set of documents.

This example mirrors the working of the map and reduce functions. For each word, the map function emits a tuple with the word as the key and the number 1 as its value. As the reduce

```
1 def map (name, document):
2     for word in document do
3         EmitIntermediate(w, 1)
4
5 def reduce (word, partialCounts):
6     res = 0
7     for pc in partialCounts:
8         res = res + pc
9     Emit(res)
```

Listing 2.1: MapReduce Example

function is operating on all values of a single key, it just sums up the 1's from each individual word and returns the total number of appearances. The partialCounts variable is an iterator which walks over the output from the map function as written to the distributed file system.

Besides the map and reduce functions we further have an **input reader** which reads from the file system and produces the input tuples for the map function. These tuples are split into a set of M splits and processed in parallel by the worker nodes. The partitioning for the reduce nodes is similar. We specify a **partitioning function** like `hash(key) mod R` to do this. The partitioning function is used to split the data into R regions when buffered data from the map tasks is written to stable storage. The master is notified and then notifies the reduce tasks to start working. Because the hash function can map multiple keys to one region, we need to sort the data at the reduce node before we can process region R. Finally an **output writer** takes the output of the reduce function and writes it to stable storage.

In some cases it is advisable to put a combiner function in between the map and reduce stages to limit the amount of data passed on to the network layer. This can be seen in the word counting example, where given the input language of the documents is English, there will be a lot of `("the", 1)` pairs emitted. The combiner function acts like the reduce step, the difference being that the algorithm writes its output to an intermediate file, whereas the output of the reduce step is written to the final output file.

As is tradition with Google, MapReduce clusters typically consist of large clusters of commodity PCs networked via ordinary switched Ethernet. Network failures are common, therefore, the algorithm has to be very failure tolerant. By ensuring that individual operations are independent, it is easy to reschedule map operations in case of failure. However, we must be more careful when output of a failed map has already been partially consumed by a reduce operation. In this case the reduce has to be aborted and rescheduled, too.

So far we have assumed that the map and reduce operations are independent. This restriction allows us to make full use of the distributed system as these operations can all be executed in parallel without any significant locking overhead. The restriction is that the data has to be available to the processing units without any significant delay. In the original implementation, a reduce task for example needs all the data for a particular key to be presented at the same time. To be able to process streams efficiently, we need to get rid of this restriction. As this restriction was implemented using a write to the distributed file system GFS, we need to stream data from mappers to reducers instead. The details of this are discussed in §2.3.4.

Criticism / Evaluation Our choice in favour of MapReduce is motivated by the ease of use of its programming model and the tight integration with the cloud paradigm. The MapReduce design provides natural support for scalability with its mappers and reducers. However, there has been some criticism voiced over its implementation. Notably, David DeWitt and Michael

```
1 count: table sum of int;
2 total: table sum of float;
3 sum_of_squares: table sum of float;
4 x: float = input;
5 emit count <- 1;
6 emit total <- x;
7 emit sum_of_squares <- x * x;
```

Listing 2.2: Sawzall Example

Stonebraker, two proponents of distributed databases have criticised MapReduce for its lack of expressiveness [22]. They argue that the programming model is outdated and not able to utilise any of the performance improvements known from traditional databases such as indexing. We argue that although DeWitt and Stonebraker are right on the limited expressiveness, they miss the point on the purpose of MapReduce. MapReduce is an excellent way to process a large amount of data that comes in a relatively unstructured form. Once we have processed the data, we can store it in relational tuples and evaluate it further using a traditional database approach. The two approaches are therefore complimentary, a result that Stonebraker et al. acknowledge in a later paper [36]. They refer to the pre-processing stage as *extract-transform-load* tasks and call them a MapReduce speciality. In our project we will evaluate relational tuples as we expect the difference between MapReduce and distributed database systems to be negligible in single-query environments.

2.2.2 Sawzall

DeWitt and Stonebraker also criticised MapReduce for its lack of high level query language. Recently, there have been several efforts to remedy this. Sawzall is a domain specific procedural language developed by Google to lie on top of the MapReduce framework. This language takes the MapReduce approach further to make sure that its programmers write parallel code. It uses a set of aggregators "to capture many common data processing and data reduction problems" [34]. As Sawzall is an interpreted language, we might think this would pose a significant constraint in a high performance environment. However, as the algorithms mainly spend time on I/O and the underlying methods are implemented natively, this does not incur any significant penalty. The increase in performance when adding machines is almost linear. As we need more control over the communication between the hosts to implement our stream processing paradigm, we cannot use Sawzall to program the cloud infrastructure. However, this is an interesting approach and it might be considered to extend Sawzall to enable stream processing as the authors suggest.

The following simple Sawzall program (Listing 2.2) takes as input a set of records containing one floating point number each. It then produces three results. The total number of records, the sum of all the floating point values and the sum of the squares of the floating point numbers.

This looks very similar to the map phase as we have seen in the discussion of MapReduce in §2.2.1. Here we define three aggregators: `count`, `total` and `sum_of_squares`. By defining these aggregators as `sum` tables we tell Sawzall what the reduce part of MapReduce should be doing. By pre-defining the aggregators, Sawzall relieves the programmer of the reduce task but also somewhat limits the expressiveness of map reduce. However, the upside is that the aggregators can now be implemented natively. Furthermore, the parallelism is hidden from the user. Aggregators have to be commutative, associative and efficient for distributed programming. Extending the set of aggregators is therefore difficult.

Evaluation Sawzall is a very interesting approach to take MapReduce more en par with traditional database systems. By designing a domain specific language it is possible to further abstract and simplify parallel programming. At this stage, Sawzall is not yet able to express streaming queries. However, we will keep MapReduce DSL approaches in mind for further projects.

2.2.3 Hadoop

Hadoop is the umbrella project for several Apache projects which have the aim to provide a reliable platform for distributed computing. The Hadoop project can be split into two parts. The first is an implementation of MapReduce. Although the framework is written in Java, it accepts MapReduce tasks written in other languages [1]. Similarly to Google's Sawzall effort, the Hadoop project also includes Pig [28], a high level data-flow language to describe parallel processes. Pig's compiler produces a chain of MapReduce operations. The language layer is called Pig-latin. The second part of the Hadoop implementation is the Hadoop Distributed File System (HDFS), a file system, similar to GFS as employed in MapReduce [19]. Both HDFS and GFS are optimised for batch processing.

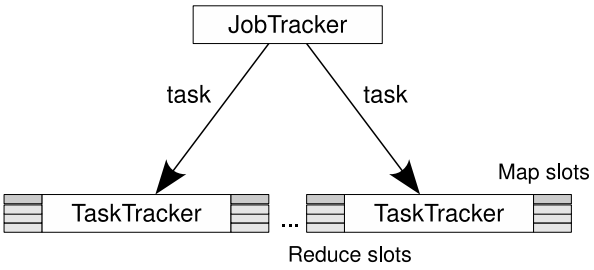


Figure 2.1: Hadoop implementation

Architecture

Fig.2.1, shows the organisation of a Hadoop installation. The master node is called *JobTracker* and waits for incoming jobs. The jobs are split into tasks and assigned to worker nodes - called *TaskTrackers*. A TaskTracker has a fixed number of slots to run map and reduce tasks. The execution of these is managed by the TaskTrackers themselves. The JobTracker uses a heartbeat protocol to keep record of free slots on the TaskTrackers and to schedule new tasks [18].

Hadoop job submission

A new MapReduce job is submitted through the *JobClient* interface. The JobClient provides facilities to monitor the MapReduce cluster, submit jobs and view their status. When a job is submitted to the JobTracker via the JobClient, the following steps are executed [2]:

1. Checking input and output configuration.
2. Computing the InputSplit[2] values for the job.

[2]Input files are split up prior to the distribution over the mappers

3. Copying the job's jar and configuration to the MapReduce system directory on the file system.

4. Submitting the job to the JobTracker and monitoring its status.

Evaluation Hadoop is available on as cloud images and a standalone application and therefore interesting to our project. This and the amount of supporting tutorial material for Hadoop convinced us to choose it for the MapReduce implementation. In its original design it does not allow for streaming MapReduce. This limitation is covered in §2.3.4.

2.2.4 Remote Procedure Calls

As distributed systems like Hadoop have to communicate intermediate results and to coordinate their behaviour, they need a reliable way to communicate. In Hadoop, a lot of inter-process communication is done using RMI, the Java version of a Remote Procedure Call (RPC). RPC is a technology to lookup functions outside the address space of the current process [30]. This could mean a function on the same node, implemented by a class running in a different process, or code running on a different machine altogether. In either case, once the function has been bound by the RPC framework on the local machine, a call behaves (almost) exactly like you would expect it to on a local machine.

Implementation

A remote procedure call initialised and executed through message-passing [39]. The client node first sends a request to the server process containing a unique identifier for the function to be called. In order to find the server implementing the function, the calling process might either broadcast a message on the local network or request a list of implementations from a well known name server. In the latter case, it is necessary for the server process to register its implementation first. Once the client has completed the lookup phase for the particular function, it is bound to a local stub and can be used by the rest of the program. On the server side, a daemon ensures that all communication request are dealt with by either spawning a new thread or picking one from a pre-initialised thread pool.

Serialisation

RPC libraries can easily deal with most basic data types such as integers and strings. However, if we wish to call methods on complicated objects by reference, we need to make sure that the remote host is aware of these objects first. We need to serialise the object [39]. Serialisation of an object such as a linked list means traversing all the pointers and essentially flattening the datastructure such that it can be either saved to disk or transmitted over the network. Flatting complex datastructures is not always an easy task as we must follow all references without looping forever due to back edges in the graph. Furthermore, we cannot merely send the contents of the objects over the network as any pointer variables will not have the same meaning on a remote host. The solution to this problem is called pointer swizzling. In our example of a linked list, we would for example introduce a new field to every node holding a unique identifier. We can then set the pointer to the next node to its identifier rather than the address in memory. This process can be easily reversed at the other side. However, there is an cost associated with this operation. Thus, serialisation should not be used extensively

when communicating in an environment where runtime is critical. When applicable, simpler datastructures should be used.

Java RMI

Remote method invocation (RMI) is Java's implementation of the RPC protocol. Instead of publishing information on a single method with a nameserver, the callee registers its remote object with the RMI-Registry under a unique identifier [37]. The client program can now query the RMI-Registry with this identifier to get a reference to the remote object. This reference has to be an implementation of its remote interface. The remote interface defines the methods that are guaranteed to be implemented by the remote object. The client can now call the methods on the remote object as published by the remote interface. It is possible for the method headers of the remote object to contain references to interfaces. Any class implementing these interfaces may then be used on the remote host including class definitions from the caller. This means that Java gives the user the opportunity to dynamically load new code as part of a RMI call. In order to make sure that this is not exploited for malicious use, a security manager is always installed as part of the RMI implementation. Any special requests have first to be granted by the designer of the server application.

2.2.5 Amazon EC2

Despite having chosen to implement our stream processor on cluster nodes in the college, our ultimate aim is to run stream processing on an IaaS platform such as Amazon EC2. The decision to develop and test locally has been purely a development one. However, as all our machines are single-purpose and only used for this project, our setup is not too dissimilar to a "cloud". The participating nodes are running the same specifications. In fact, the EC2 technology is available as part of the Ubuntu Eucalyptus packages and can be integrated into an Ubuntu server installation. It could thus be easily installed on our test cluster.

Amazon's EC2 offering, introduced in 2006 [11], was one of the first commercial cloud infrastructures. EC stands for Elastic Cloud. Customers are allowed to create an Amazon Machine Image. This image is used to start a virtual machine on the Amazon cluster. The infrastructure allows customers to start and terminate new virtual machines as required by the application, hence the term elastic. Additional nodes can be accessed within minutes. This ease of use makes it very interesting for our purposes of load balancing. Amazon charges by the hour or the data transfer rate. Further charges are possible. The various virtual machines instances as of December 2009 are shown below. These instances are based on the notion of EC2 Compute Units. The units mirror equivalent capacity of physical hardware. It is possible to specify the geographical location of the servers in order to optimize network latency and fault tolerance.

1. **Small Instance (Default)** 1.7 GB of memory, 1 EC2 Compute Unit (1 virtual core with 1 EC2 Compute Unit), 160 GB of local instance storage, 32-bit platform

2. **Large Instance** 7.5 GB of memory, 4 EC2 Compute Units (2 virtual cores with 2 EC2 Compute Units each), 850 GB of local instance storage, 64-bit platform

3. **Extra Large Instance** 15 GB of memory, 8 EC2 Compute Units (4 virtual cores with 2 EC2 Compute Units each), 1690 GB of local instance storage, 64-bit platform

The Amazon CloudWatch service gives information about the current load on the system. This information can be accessed via a web service or web APIs. Unfortunately, this service incurs a charge for the number of monitoring instances used. However, the CloudWatch service is necessary for using the auto scaling feature of EC2. Auto-scaling enables us to specify triggers which cause Amazon to add or remove instances automatically.

Persistent storage on EC2 systems is provided using the Amazon Simple Storage (S3) service. The EC2 implementation itself only stores temporary data. If an instance is rebooted, this data is lost. As this can happen explicit via an API call or a failure in the system, we cannot ensure fault-tolerance without a form of persistent storage. Google's MapReduce algorithm requires a distributed file system to store the intermediate results. The S3 service makes data available directly to the EC2 instances and over the network via http. This makes it suitable for ordinary MapReduce. For low latency access, we must use S3 and EC2 services in the same area. There is no extra cost for S3 to EC2 bandwidth. Despite its advantages and close link to the original MapReduce idea, the distributed file system is more of an obstacle towards efficient stream processing as both input and output should be written to sockets rather than to stable storage.

2.3 Stream processing

In this section we are covering several research prototypes of stream processing, ranging from single node implementations to distributed versions. The goal of the project is to focus on the load-balancing between the front-end node and the processing power of the cloud. In this light, we will evaluate the state of the art and rationalise our decision to use a MapReduce framework to do stream processing in a cloud environment.

2.3.1 Mortar

Logothethis and Yocum have developed Mortar - "A distributed stream processing platform for building very large queries" [29]. The Mortar platform focuses on applications with thousands of distributed sources. It takes into account unreliable sources and adverse network connections. Mortar is designed to manage streaming queries across large federated systems[3]. The Mortar infrastructure takes over the task of creating and removing operators as well as organising the data flow between them. For this reason, one of the main contributions of the above paper is to provide robust overlay networks to run queries. Mortar also addresses the problem of time synchronisation between the different nodes. Stream processing cannot work without reliable time information as the timestamps used for windows become useless. Mortar differentiates itself from the other approaches shown here, as not only are the data-supplying nodes involved in the processing, but the designers want them to be the only ones to do the work. Logothetis and Yocum speak of *scoped queries* in this context. We now discuss the overlay networks and routing algorithm employed by Mortar.

Overlay network The overlay networks in Mortar are static. The authors justify this by the claim that in federated systems, machines are rarely added or removed. They can only become unavailable for a short time. It is assumed that there is personnel in charge of the sites which quickly rectifies any problems. However, in order to deal with failures, multiple static trees are created and operators are connected across them. The static aggregation trees in the Mortar framework are build such that the majority of nodes is close to the root node.

[3]Federated systems typically include heterogeneous hardware and different authorities

This is necessary to ensure the latency bounds required by streaming applications. Once the primary tree is built, the sibling trees are derived through successive random rotations of internal subtrees. The authors have confirmed by empirical observation, that this is unlikely to change the clustering.

Dynamic tuple striping The routing scheme proposed by Logothetis and Yocum is called *dynamic tuple striping*. Dynamic striping is a multi-path routing scheme. Routing is done up the trees towards the root. Since scoped queries are used, each node needs to know about all live parents for locally installed queries. These are found using a heartbeat protocol. Children are updated in the same way. Tuples are routed through a parent only if this moves them closer to the root. The algorithm first tries to use the parent on the tree on which the tuple arrived. If this fails, it tries a parent on a different tree which must not be further away from the root as the current tree. This avoids cycles. However, if no further progress could be made, a tuple is allowed to move down to a child and try a different path. As this now introduces the possibility to incur cycles, a time to live field is introduced (TTL) to restrict the number of tries.

Evaluation Mortar has somewhat limited applicability to our cloud approach. As our chosen platform is homogeneous and under a single administration we have eliminated most of the problems stemming from federated systems. Indeed, the implementation of our chosen stream processor should be modifiable to run on virtual machines which might be added or removed during the computation. This form of scalability is not what the authors of Mortar envisaged. Mortar is a very reliable platform for applications such as sensor networks with a large number of data providing nodes. In these cases an overlay network is justified. In our case, however, it violates the low latency requirements. Applying Mortar to stream processing in a cloud setting would require too many modifications.

2.3.2 STREAM

STREAM is a data stream management system (DSMS) developed by the University of Stanford [9]. Ordinary database management systems (DBMS) deal with one-time queries. Network monitoring, financial analysis and sensor networks, however, emit a continuous stream of data. Thus in addition to ordinary relations, we have bags of tuple, timestamp pairs, called streams. Streams can only be handled by continuous queries. The STREAM project aims to fill this gap. The prototype DSMS [9], supports a "large class of declarative continuous queries over continuous streams". It evaluates the queries by translating the declarative queries into a physical query plan. This plan is then processed by the DSMS. A query plan consists of operators, queues and synopses.

Operators Arasu et al. have developed the Continuous Query Language (CQL) [9]. CQL looks very much like an extension to SQL. There are three types of operators: *relation-to-relation*, *relation-to-stream* and *stream-to-relation*. The *relation-to-relation* operators are the well-known SQL operators. The *stream-to-relation* operators are more interesting. They offer the ability to turn the continuous stream into a relation which can be modified by ordinary relational algebra. The stream-to-relation operators are following the sliding window concept. A *tuple-based* sliding window is expressed using `[ROWS N]` after the steam identifier. This returns a relation with the last N tuples/rows in the stream. The `[Range N]` parameter gives a *time-based* sliding window. The relation returned includes all tuples with more than N timesteps in

```
1 Select IStream(*) From S [Rows 100] Where S.A > 10
2 Select IStream(*) From S [Rows Unbounded] Where S.A > 10
3 Select IStream(*) From S [Range 10] Where S.A > 10
4 Select IStream(*) From S [Now] Where S.A > 10
```

Listing 2.3: New operators in CQL

the past. The abbreviation `[Now]` returns the window with $N = 0$; i.e. only tuples that have the same timestamp as the clock of the DSMS are returned. Examples are shown in Listing 2.3.

The last type of stream-to-relation operators is a little more complex. A *partitioned* sliding window, takes a set of attributes over which it splits the input stream into separate substreams `[Partition by A,...,Z ROWS N]`. `N` specifies the number of rows used in this process. The individual windows are combined using the SQL union operator to generate the output relation. If we want to turn the output relation back into a stream, we can use *relation-to-stream* operators.

CQL defines three relation-to-stream operators. *Istream* or insert stream, contains all the tuples inserted into the relation. Every time a new tuple is added to the relation, it is forwarded to the stream (unless it is a duplicate). *Dstream* or delete stream, contains all the tuples deleted from the relation. Like for the insert stream, deleted tuples are forwarded to the stream. *Rstream* or relation stream contains all tuples in the relation at all time instants. The first line of Listing 2.3 therefore selects 100 rows from the stream S, turns it into a relation, selects all tuples where $S.A > 10$ and returns these as a stream.

Queues As queries are converted to physical query plans, we need queues between the operators. Queues are connecting the operators and can be empty. Due to the nature of the processing, queues have to be read in non-decreasing timestamp order. Only this way, an operator can be sure that all the necessary data is present to close the window and begin processing.

Synopses Synopses are associated with operators and further describe the data. An example given by Arasu et al. is that of a windowed join of two streams. In this case the synopsis are a hash table for each of the input streams. A synopsis can be shared amongst multiple operators.

Evaluation The STREAM project has led the way in the field of stream processing. The expressiveness of CQL is great and the available optimisations manifold. Arasu et al. [9] identify operator reuse and replication as two performance enhancing components of placement algorithms. This is because in large static queries certain sub-queries are often replicated leading to a waste of resources if we were to re-instantiate the operators for these queries. These optimisations and the full-fledged implementation of stream processing make STREAM a good reference for our project. We are trying to take the work done in the database community to the MapReduce world. Note, we are not aiming to achieve the expressiveness of CQL. We are focussing on simple interfaces and scalability.

2.3.3 Cayuga

Cayuga is a publish/subscribe system which handles stateful subscriptions [20]. While ordinary publish/subscribe systems deal with individual events, Cayuga extends to handling subscriptions that involve relations between multiple events. This makes it ideal for handling complex

		Number of concurrent subscriptions	
		few	many
Complexity of subscription	low	(trivial)	pub/sub
	high	DSMS	stateful pub/sub

Table 2.1: Trade-offs between pub/sub and DSMS (from Demers et al.)

streaming data such as sensor networks or stock exchanges. As an example, Cayuga can handle sequences of events such as the following subscribe query taken from Demers et al. *"Notify me when the price of IBM is above $100, and the first MSFT price afterwards is below $25"* **??**. In this case the stream processor has to store state in order to evaluate the query.

The authors compare Cayuga to Data Stream Management Systems (DSMS), which offer query languages to the same effect, but fail to scale over a large number of subscriptions. The difference between a traditional DSMS such as STREAM and Cayuga is shown in Table 2.1 [20]. Cayuga, however, is still closely related to the database community. The query language called *Cayuga Event Language* (CEL) is similar to SQL [20]. Like STREAM, Cayuga works with event streams - temporally ordered sequences of tuples. However, instead of using a single timestamp, Cayuga uses a start and end timestamp during which the event was active. This gives a duration as well as a "detection time" [13].

Cayuga includes unary relational algebra operators such as selection, projection and renaming. It supports union, but it excludes Cartesian products and window join. These unrestricted joins are less useful in a stream setting as they are not taking into account the timestamps of the events. Instead Cayuga offers a *sequencing* and an *iteration* operator.

- **Sequencing operator** ($;_\theta$) The sequencing operator is a forward-looking combining join that processes the event stream in sequence and tries to satisfy the filter predicate θ.
- **Iteration operator** ($\mu_{\xi,\theta}$) The iteration operator allows more complex joins. Here ξ can be any unary operator such as projection.

The implementation behind Cayuga is a single node system based on non-deterministic finite automata. The approach used to evaluate queries is very similar to the one used in regular expressions. CEL queries are compiled into state machines and then loaded into the query engine via an intermediate XML format [13]. Predicates are mapped to edges and events are affecting the state of the automaton. Like with regular expressions, an edge is only traversed if the incoming event satisfies the predicate. If no edges are traversed, the event cannot satisfy the query and is thus dropped.

Distributed Cayuga

Brenna et al. have developed a distributed version of Cayuga [13]. We will discuss two of the techniques they used - *Row/Column scaling* and *Pipelining*.

Row/Column scaling A simple technique to scale query processing in stateless systems is to spread the queries over n machines. This *row* is then replicated m times to form a $n \times m$ matrix. Events now can be routed to any row. Usually the routing is done in a round-robin fashion, however. This causes a problem with stateful queries. In order to make use of states

we must route events to the same row. This is infeasible. Brenna et al. [13] propose two ways of partitioning the workload.

The first technique is to partition the input event stream into several substreams of related events. These substreams are then assigned to rows and processed individually. The partitioning itself can be expressed as a Cayuga query and done by a separate machine.

The second technique is to partition the query workload. Thus instead of sending substreams to the rows, each row receives a full copy of the input stream.

Pipelining Cayuga provides the possibility for queries to be split into sub-queries if the former can not be run on a single machine. The output of one subquery to another is controlled via a feature called re-subscription [13].

Evaluation Cayuga and the effort to run it over large distributed networks are two interesting approaches. In the single node implementation Cayuga with its very expressive CEL language makes a lot of sense. However, in the distributed case, the situation is more complicated. For our project we are planning to start with a single query which means that the ideas concerning row/column scaling are currently not applicable.

2.3.4 MapReduce Online

MapReduce Online enables "MapReduce programs to be written for applications such as event monitoring and stream processing"; it has been proposed in a technical report by Condie et al. in 2009 [18]. In contrast to the original Google implementation of MapReduce [19], Condie et al. propose a pipelined version in which the intermediate data is not written to disk. The prototype for this concept is a modification of the Hadoop framework and called *Hadoop Online Prototype* (HOP).

The first change the authors introduced into stock Hadoop is pipelining. Instead of writing data to disk, it is now delivered straight from the producers (map tasks) to consumers (reduce tasks) via TCP sockets. If a reduce task has not been scheduled yet, the data can be written to disk as in plain Hadoop. The number of connections is also bounded to prevent creating too many TCP connections. Combiners are facilitated by waiting for the map-side buffers to fill before they are sent to the reducers. This enables us to pre-process the data before it is sent to the reducers. The processed data is written to a spill file which is monitored by a separate thread. This thread transmits the data to the reducers. The spill files enable simple load balancing between the mappers and reducers as their size and number reflects the load difference in the system. If spill files grow, we can adaptively invoke a combiner function on the side of the mapper to even the load.

The Hadoop Online Prototype further allows for inter-job pipelining. It must be noted that it is impossible to overlap the previous reduce with the current map function as the former has to complete before the latter can start. However, pipelining reduces the necessity to store intermediate result in stable storage which could be costly. Condie et al. have extended Hadoop to be able to insert jobs "in order". This helps to preserve dataflow dependencies. The introduction of pipelining into Hadoop makes the system ready for running streaming applications.

Evaluation The Hadoop Online Prototype is a very interesting approach for stream processing. The changes over ordinary MapReduce implementations are subtle but powerful. However, like all other technologies visited in this section, HOP does not automatically scale during runtime. The number of map and reduce tasks is fixed. This impairs the ability of the MapReduce implementation to cope with varying load. We have chosen to use Hadoop as a starting point to design a stream processor utilising the MapReduce paradigm.

2.4 Load balancing

Load balancing is an integral part of our system's design. A single stream processing node should automatically move computation to the cloud when its own resources are not sufficient anymore. In this project we aim to do this in a way which avoids an increase in latency and gives a guarantee on QoS properties of the system. To find possible strategies to achieve this, we have studied several load balancing strategies for webservers. First there are client side policies in which the client decides which server to use. Client-side policies include NRT (network round-trip-time) and latency estimated from historical data [14]. Selection algorithms are also based on hop count or geographical distance. We are also interested in server side policies which use a load index to determine how to route requests. Both are applicable as we run the local stream processor and evaluate its load on the same machine. However, none of the research mentioned in this section is 100 percent applicable to our situation. We are not aware of a load-balancing solution that evaluates computational load at a single front-end node and moves the overhead into a scalable cluster.

2.4.1 Load-balancing in distributed web-servers

Cardellini et al. discussed the state of the art in locally distributed web-server systems [15]. While our load-balancing or load-sharing algorithm is located at layer 7 (i.e. application layer of the OSI model) this paper gives insights into routing at all layers. We are especially interested in the discussion of a proxy for routing the requests (TCP gateway) and the discussion about dispatching algorithms. In our use-case we are looking for a load sharing algorithm that smooths out transient peak overload. This algorithm must be dynamic as we require it to have information about the state of the system. Finally, we must decide between centralised and distributed algorithms. As we will do the load balancing on the single machine serving as a controller to the cloud computations and the initial worker node, we opt for a *centralised* algorithm here.

As mentioned in the article, fundamental questions are which server load index to use, when to compute it and how this index is shared with the orchestrating server. Ferrari and Zhou found that load indices based on queue length perform better than load indices based on resource utilisation [23]. They define the load index as a positive integer variable. It is 0 if the resource is idle and increases with more load. We must further acknowledge that there is no general notion of load. For example the CPU might be idle in a heavily I/O bound process, while in other cases processes are competing for CPU time. To summarise, the following load indices are mentioned in Ferrari and Zhou [23].

- CPU queue length
- CPU utilisation
- Normalised response time[4]

[4]response time of job on loaded machine / response time on idle machine

Evaluation Our load balancing exercise will mainly concentrate on the local node as for the cloud this should be done by the MapReduce framework. This means that we can easily calculate the normalised response time. If this time exceeds a certain threshold we can start moving computational load to the cloud. Queue length is also a good indicator of load. In the evaluation chapter of this report we shall look at both response time and queue length.

2.4.2 Locally aware request distribution

Pai et al. suggest a load balancing based on data locality [32]. In this system called LARD (locally aware content distribution), a single frontend receives the requests from clients. These requests are usually for static web content. This static content is stored on a number of back-end webservers. Once the first request for a particular resource comes in, the front-end examines it and routes it to the server with the least load. Any subsequent requests for this file are (if possible) routed to the same server. This makes sure that this back-end server can efficiently cache the data. Furthermore, it enables the design of a scalable storage system. We can spread the data over multiple back-end servers with little replication. Each request will be routed to a different server thus resulting in an efficient architecture. This works especially well for websites as http is a connectionless protocol and for every file, a new connection is opened.

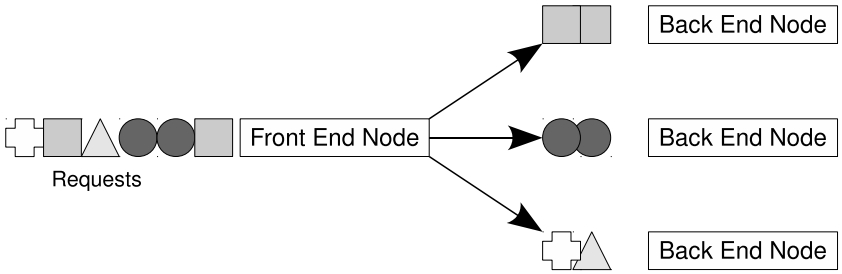


Figure 2.2: Structure of LARD system

In order to implement the LARD system, the front-end server has to examine the incoming requests. In contrast to routing at OSI level 4, the front-end has to examine the request to know which backend-server should deal with the request [32]. Furthermore, the front-end needs to know about the individual loads at the backend. In their discussions, Pai et al. assume that the front-end and the network have no degrading effect on the performance of the overall system.

Evaluation LARD is only remotely applicable to our stream processing application. LARD is distributing the requests over servers in order to reduce cache misses. The only stable data in our system are the queries. The streaming data must not be stored but processed immediately. An additional complication is introduced by the fact that in our case the front-end takes part in the processing. This breaks with the assumption that the performance is independent of the front-end.

2.4.3 TCP Splicing

TCP slicing is a technique implemented in layer 7 switches to improve performance [17]. Like the LARD protocol above, layer 7 switches are able to do URL aware redirection of HTTP traffic. The benefits are the aforementioned cache hit rate and the ability reduce the need for replication in the backend. In this scheme, the switch acts as a proxy and redirects incoming

HTTP requests based on the URI in the GET request. Incoming requests are examined and and the switch queries the right backend server for the response.

While layer 4 switching is based on port and IP, layer 7 switches work at the application layer. As application data is only transferred once the connection has been established we cannot do URI aware routing without first establishing a connection between the client and the switch. This means that we need to establish two connections. The first connection is between the client and the switch. The second connection is between the switch and the backend server. The most common form of handling this problem is called TCP gateway. The TCP gateway solution simply forwards the packets from the backend over the switch at the application layer. However, this is unnecessary. Once we have established the connection between the client and the switch and located the right backend server, any subsequent reply from the backend can be forwarded over layer 4. This technique is called TCP splicing. Packets from the server to the switch are translated to look as if they had passed through the application layer in the TCP gateway protocol. As we do not have to examine the content but only to rewrite sequence number addresses, this is relatively simple and can be done with little to no overhead. TCP slicing is very effective for large files as the number of outgoing packets is much greater than the incoming requests. But even for data as small as 1Kb, significant performance benefits can be obtained [17].

Evaluation TCP slicing only concerns how the data is being handled by the switch. The actual load balancing is not affected. This technique is very interesting for us as we will need to redirect the output from the cloud to the clients. However, the response is likely to be shorter than the incoming data (as discussed above). Furthermore, we cannot afford to create too much overhead in the local server as we would like to use it for processing as well. An interesting case arises when the amount of work done in the cloud eclipses the work done locally by a few times. In this case, the local machine may become merely a coordinator and the TCP slicing solution viable.

2.5 Summary

In this chapter we have covered the design principles of the MapReduce framework. We have discussed the original paper by Google [19] and a streaming variant called HOP based on the Hadoop framework [18]. We will be using the MapReduce paradigm to design a scalable stream processor. The discussion on remote procedure calls (RPC) serves to give the necessary background for the custom MapReduce prototype introduced in Chapter 4. The short introduction to Amazon's EC2 should remind us of the ultimate goal which is to move the auxiliary stream processor to an IaaS provider. We have reviewed the current state of the art in stream processing and hinted why none of the existing systems are designed to be used on a cloud infrastructure. This chapter was concluded with an overview of some existing load balancing techniques and their limitations when applied to our specific situation. We are not aware of a load-balancing solution which supports a single-node stream processor through an auxiliary, scalable, cloud-based stream processor.

Chapter 3

Stream Processing With Hadoop

In this chapter we will discuss our efforts to extend the MapReduce paradigm to continuous queries. We will show the design and implementation process leading to a MapReduce stream processor which can be run on a cloud infrastructure. We will start by showing how we can extend Hadoop to accept a TCP stream as a data source. We will then discuss how the concept of jobs can be extended to continuous queries. This chapter will be concluded with a discussion on the architectural limitations of Hadoop as a stream processor. The evaluation of our results will follow in Chapter 6.

3.1 From batch to stream processing

We have chosen Hadoop as the basis for our streaming MapReduce as it has already been adopted in a range of industry projects [10] [25] [35]. The main advantage of this is that the acceptance of a streaming solution is likely to be greater with this already proved software. Hadoop is open-source software and therefore amenable for extensions and modifications.

The key to our evaluation of the MapReduce framework will be the guarantees given on latency and the capability to scale efficiently as the load is increased.

Hadoop is designed for batch-processing. A typical Hadoop job takes hours and is run on dozens of machines [19]. A query in a stream processing system must typically be executed in seconds or milliseconds. Therefore, some modifications are necessary to adopt Hadoop for stream processing. As discussed in the background section (see §2.2.3 and §2.3.4), we shall be using a modified version of Hadoop, the Hadoop Online Prototype [18] (HOP), for our experiments.

There are two main tasks, we need to achieve before we can use Hadoop to process continuous data streams:

1. **Network I/O** Hadoop needs to be able to forgo the distributed file system and read its data straight from a network socket. Likewise, the output of the MapReduce operation has to be written to a socket for delivery to the client.

2. **Persistent queries** Hadoop is designed to accept one job per input directory. We can submit many jobs in succession, but there it is currently impossible to install a persistent query.

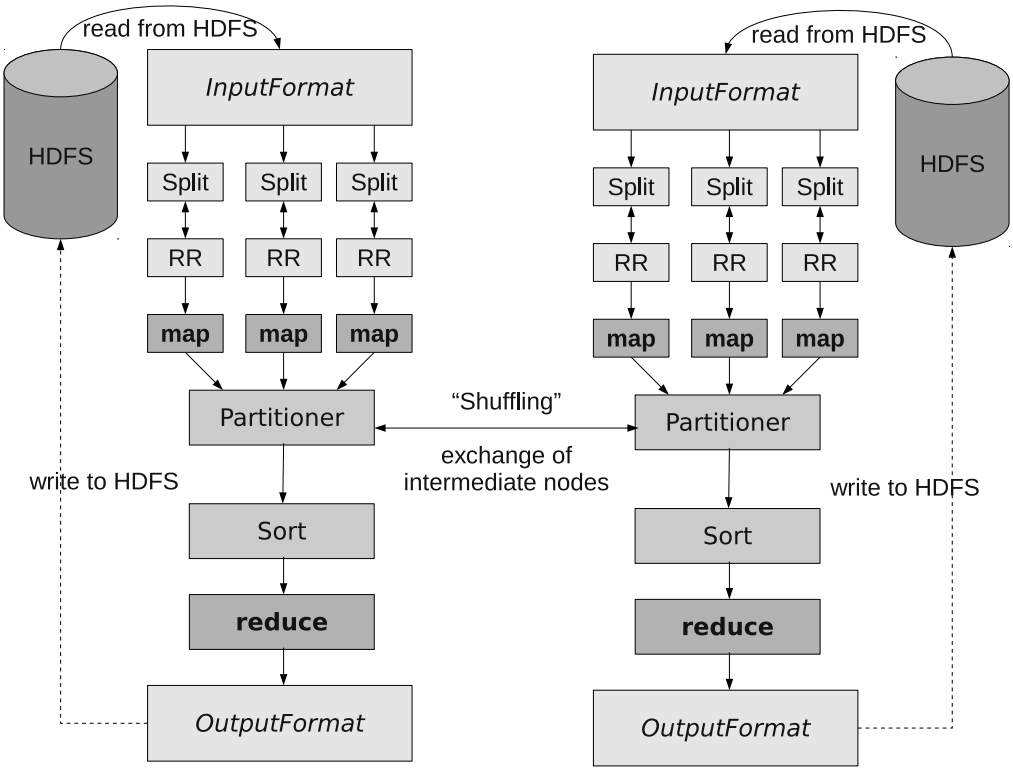


Figure 3.1: Hadoop dataflow on two nodes. InputFormat determines InputSplits and RecordReaders (RR). Map tasks get their data from the RRs and pass them to the Partitioner for grouping prior to the reduce stage. The reduced data is collected by RecordWriters as defined in the OutputFormat. Diagram adapted from http://developer.yahoo.com/hadoop/tutorial/module4.html

3.2 Network I/O

Traditional Hadoop approach

Hadoop relies heavily on its distributed file system to provide the data to mappers and reducers (Figure 3.1). Streaming applications could use the ZooKeeper [3] tools to control the distributed file system and monitor changes. We could simply write a server which uploads incoming data to the file system and a corresponding ZooKeeper task to act on any changes to the HDFS. With our low latency constraint, however, we cannot afford to store the window on the distributed file system data prior to processing. Use of the distributed file system would incur a significant overhead in processing and prevent us from being able to give any sort of latency guarantee. The distributed file system must be eliminated from the MapReduce cycle (Figure 3.1) in order to accomodate streaming queries.

Stream server

The first step in making the HDFS obsolete is to extend the Hadoop framework to obtain input splits from a network socket. The HOP implementation ensures that the intermediate data is only stored if really necessary. The output stream is written to a socket as well. All data is transient and the system completely stateless.

In order to implement the input and output streams, we need to introduce an additional stream server into the HOP framework (Figure 3.2). The StreamServer is invoked once and then handles

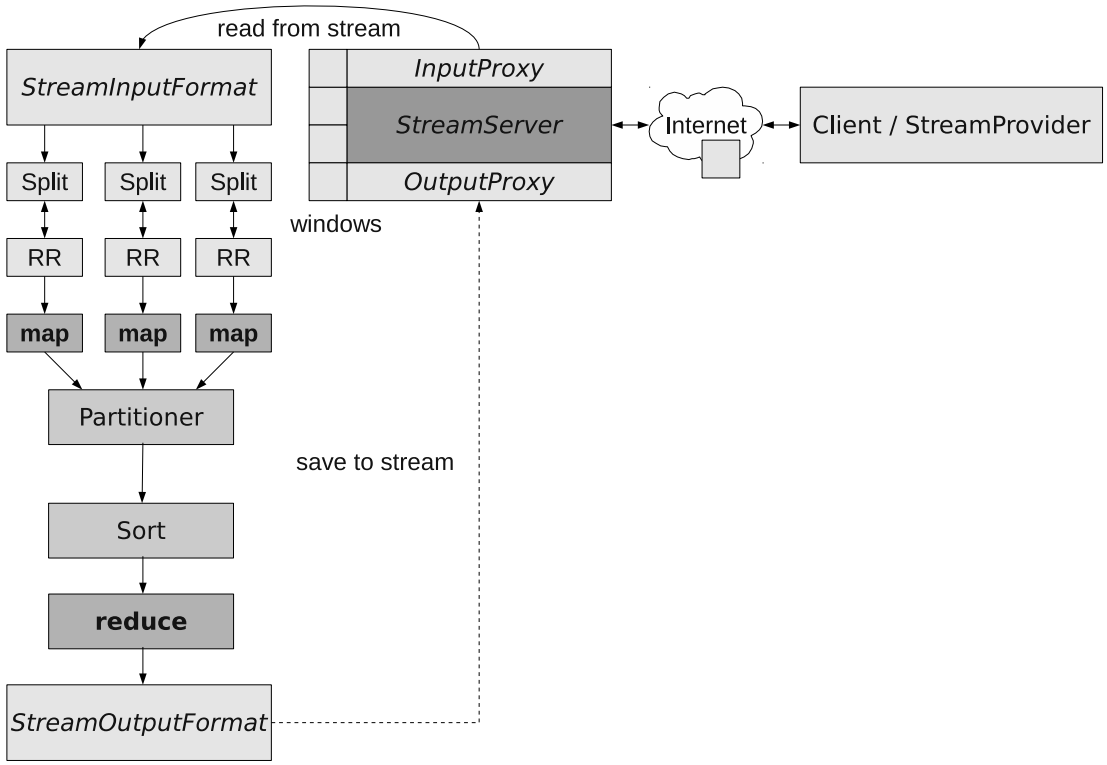


Figure 3.2: Hadoop dataflow with StreamServer component

the communication with the mappers and reducers. As the StreamServer and the MapReduce job do not know the size of the window in advance, this information is communicated as part of the control information of every window. This information can then be used with the split size to create input splits for the mapper processes. Once a query is invoked, the mappers block until data is available from the StreamServer. The result is then processed and handed over to the reducers. Once the reducers have finished, the information is relayed back to the StreamServer and communicated back to the invoker of the query.

3.2.1 Implementation of the changes

Listing 3.1 shows how the implementation of streaming queries has changed the way in which we have to write a Hadoop MapReduce job. The changes have necessitated the development of a new way of handling the input.

StreamInputFormat

Hadoop uses the `JobConf` class to assert which `InputFormat` has been chosen by the user (Listing 3.3). The InputFormat class tells the Hadoop framework where to get the input splits from. In the original implementation this could be for example a file or folder on the distributed file system or a distributed database. The InputFormat class is generated on the fly by reflection from the JobClient. We therefore have to set its parameters using reflection, too. This can be achieved by using a static method as shown on line 8 in Listing 3.1. As the the given InputFormat classes do not contain provision for sockets, we have to create our own implementation. We have done so in the form of a new `StreamInputFormat` class. Like the existing implementations, StreamInputFormat is a factory for the RecordReader class ("RR" in Figures 3.2 and 3.1). The

```
1  StreamServer.startServer(50002);
2  // Create a new job/query configuration
3  JobConf conf = new JobConf(StreamDriver.class);
4  // Set the input to a stream
5  conf.setInputFormat(StreamInputFormat.class);
6  conf.setOutputFormat(StreamOutputFormat.class);
7  // Connect the stream to the StreamServer
8  StreamInputFormat.setInputStream(conf, "localhost", 50002, 10000);
9  StreamOutputFormat.setOutputStream(conf, "localhost", 50001);
10 // Set the query
11 conf.setMapperClass(StreamMapper.class);
12 conf.setReducerClass(StreamReducer.class);
13 try {
14    JobClient.runJob(conf);
15 } catch (Exception e) {
16    e.printStackTrace();
17 }
```

Listing 3.1: Driver method: The StreamServer component listens on a user-defined port. Instead of the usual input format, based on the distributed file system, we introduce a new StreamInputFormat class that handles the provision of key/value pairs to the mappers. As the mappers are dynamically requesting the information, we cannot connect them directly to the input stream. Instead they are connected to the StreamServer proxy, which handles the connection to the client. This is similar to the way in which Hadoop handles input from a database system.

RecordReader gives the mappers access to the input data. In the original implementations, a typical return value of the RecordReader's `next()` method could be a line of output. For each MapTask, we have a separate RecordReader.

StreamRecordReader

Our StreamInputFormat class returns a `StreamRecordReader`. The StreamRecordReader uses a socket to connect to the StreamServer's output proxy. This is done as soon as the StreamRecordReader is initialised. A call to the `next()` function will then return a tuple from the input buffer. If the input buffer is empty, the call blocks. The StreamRecordReader stops providing data once the current window has been exhausted. The MapTask is notified. Hadoop uses a `DataInput` object to wrap around the InputStream. The DataInput class is used to tell the RecordReader about the size of the next input. This is not necessary in our case as we control the size of the output from the StreamServer's output proxy ourselves. Once the MapTask has received the data, the operation is the same as in the stock HOP/Hadoop implementation. The output from the MapTasks is collected by an OutputCollector and distributed for the reduce phase. Once the reducers have finished their task, the data needs to be send back to the StreamServer.

StreamOutputFormat

The `StreamOutputFormat` class is very similar in style to the StreamInputFormat. Instead of a RecordReader, it contains a factory method to create a RecordWriter. Again, no provision is given in the original implementation of Hadoop for writing output to a socket. We have implemented a new class StreamOutputFormat which handles this. The StreamOutputFormat provides a StreamRecordWriter which like the reader connects to a proxy at the StreamServer to deliver its data. The data is then routed via the StreamServer back to the client. The param-

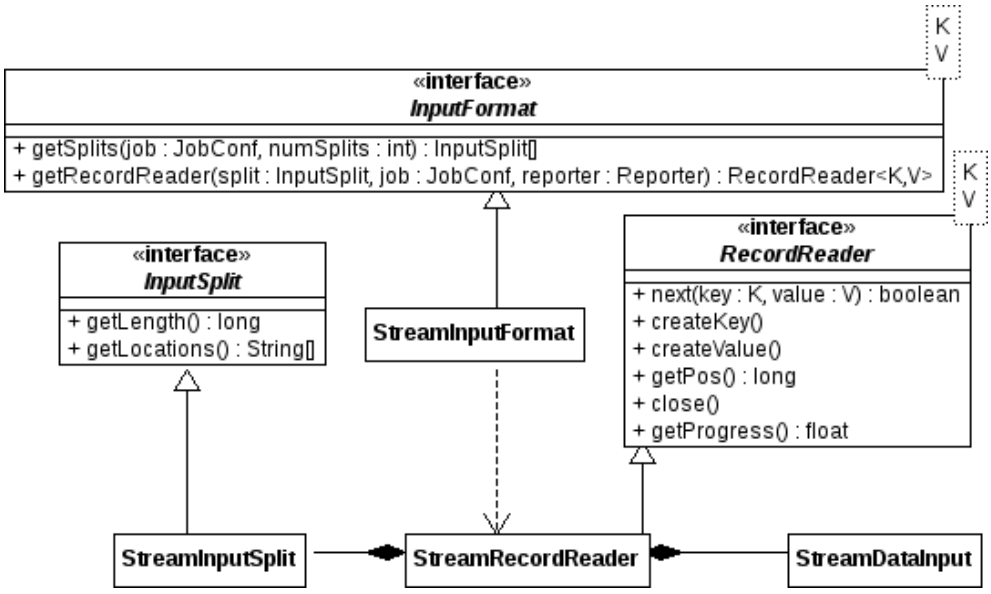


Figure 3.3: UML class diagram depicting the InputFormat hierarchy

eters of the StreamOutputFormat are set via a static method and obtained by the MapReduce framework using reflection.

StreamServer

The `StreamServer` acts as a proxy between the mappers and reduce and the client. We have implemented separate threads to listen to incoming connections from the mappers and reducers. These connections are transient. Every time a RecordReader or RecordWriter is created, the connection has to be re-established. The connection is established from the RecordReader and RecordWriter classes as the StreamServer is oblivious to the current state of the MapReduce process. The StreamServer uses the parameters set by the static methods `setInputStream()` and `setOutputStream()` to choose on which port to listen for incoming windows. The incoming data is buffered and send to the mappers and reduces when requested. In a more low-level implementation, we would use the StreamServer in conjunction with the TCP splicing ideas mentioned in the background chapter (§2.4.3). However, this necessitates some continuity map and reduce tasks. It is not possible to achieve this with the current design which sees the RecordReaders and RecordWriters being re-instantiated for each iteration of the job. In the next section we will discuss our efforts to solve this problem and install persistent queries.

3.3 Persistent queries

In addition to the changes introducing a networked input format format, mentioned above, we must make sure that the jobs we install on the Hadoop cluster are persistent. We envisaged to restart the job as soon as a window has been processed. Re-starting a query is relatively simple - we can easily add another job to the JobTracker, using the same old configuration. In the evaluation section, we will see how this affects the run time of our query. Regardless, we wish to mention a few bottlenecks here as they tie in with the implementation of our stream processor.

3.3.1 Restarting jobs

A persistent query can be constructed by continuously filing new jobs with the JobTracker. This is very straightforwardly done in the `Driver` class of our MapReduce implementation. As our `StreamProcessor` component stays running in the background, the mappers and reducers simply continue querying it using the proxies as before. There are no changes needed in the I/O design to accommodate persistent queries.

However, there is a certain penalty incurred by this operation. Even though we can re-use the old job configuration, it has to go through the entire validation stage of the JobTracker. Furthermore, we cannot permanently install queries on the TaskTrackers. This means that the JobTracker will have to re-install the map and reduce tasks on the TaskTrackers with all the administrative overhead of consulting the distributed file system. The problems associated with this become more clear when we discuss the overhead of the Hadoop infrastructure.

3.3.2 Optimisation attempts

Central to the operation of the Hadoop MapReduce framework is the interface `TaskUmbilicalProtocol`. This interface is implemented by the `TaskTracker` classes and used by the `JobTracker` to establish a remote connection. The protocol describes a daemon which polls a master node for new map and reduce tasks. The TaskTrackers are thus transient in their operation. In the current configuration, they need to ask for new map and reduce tasks using the protocol. This communication is unnecessary in a stream processor, as queries are persistent. In order to optimise the run time of the Hadoop framework for streaming queries, we needed to profile the operation of running queries in a distributed environment. Hadoop offers the possibility to debug queries in a single node environment using a `LocalJobRunner`. However, profiling the single node run would not give us any insight in the distributed operation of the MapReduce framework.

Profiling and structural analysis

In order to obtain information on the bottlenecks, we extended Hadoop's own logging mechanism to show more detailed information on individual run times. Several runs with identical configurations (10,000 identical tuples) showed huge discrepancies in runtime. While the majority of runs resulted in a runtime of approximately 10 seconds, approximately one in ten runs showed a runtime of over 20 seconds. We were unable to exactly pinpoint the source of these problems but we assume they are closely linked to the distributed file system.

With no reliable profiling information, we used code analysis tools to gain more insight into the structure of the Hadoop project. As mentioned above, the task submission is done partly by Java RMI calls. We hoped to identify the extend of the linkage between the different components. The goal was to identify breaking points to separate out some of the book-keeping and to install persistent queries on the TaskTrackers.

Unfortunately, Conventional source code analysis tools like STAN [8] did not help. The problem with the Hadoop source code is that although structural analysis showed little entanglement between classes, this is because of the fact that most communication is done via Java RMI and the distributed file system. This meant that even though almost 11% of the source code is concentrated in four files (Table 3.1), it is near impossible to find the real dependencies. Consequently, any small change to the way in which jobs were handled ended in some other component flagging an error. A lot of these dependencies are linked to the various monitoring

Class	ELOC[1]
hadoop.mapred.TaskTracker	2595
hadoop.mapred.JobTracker	2168
hadoop.mapred.JobInProgress	1940
hadoop.mapred.JobClient	1361
...	...
apache.org.hadoop (whole project)	86181

Table 3.1: Extract from STAN report on Hadoop 0.19.2 (including HOP and Streaming extensions)

and recovery procedures in place. These are clearly overhead and not necessary in lightweight stream processing solution.

In the remainder of this chapter we will focus on the inherent overhead of the Hadoop platform in order to find out what has got to be changed in order for a stream processor to run efficiently on a MapReduce platform.

3.3.3 Hadoop overhead problems

As a batch processing system, Hadoop has a few bottlenecks that might cause problems when we execute our streaming query. In the following section, we will discuss the implementation choices made by the Hadoop project and how they affect our stream processor.

Fault tolerance

Hadoop ensures fault tolerance by using a heartbeat protocol between the JobTracker and the TaskTrackers. If a ping to a TaskTracker fails for a period of time (default is one minute), the JobTracker is going to invoke a recovery procedure. The JobTracker knows which map and reduce task were installed on the faulty TaskTracker and will attempt to restart them on other nodes. If the failed TaskTracker was in the map phase, the failed tasks will be re-executed on another node. If the TaskTracker failed during the reduce phase, all reduce tasks allocated to the faulty node will be re-spawned on other TaskTrackers.

This behaviour is welcome when executing a task running over many hours on a paid-for cluster like EC2, when a failed node could possibly mean a restart of the whole job. However, in a stream processor, the failure to compute the output for a single window is not our main concern. We would rather drop the window in question and go on with the computation of the next one as we would otherwise be out of sync with the stream and introduce unwanted delays. For this reason, the fault tolerance mechanisms of Hadoop are counter-productive for our application.

Monitoring

Hadoop provides excellent resources for monitoring the state of a job. The state is continuously queried and written to log files. Furthermore, the user can browse the distributed file system using a web browser. All these control mechanisms incur computational and communication overhead, that we cannot accept in a stream processing system. As a query is executed in

[1]Estimated lines of code

seconds, or milliseconds, we do not need any information on its completion. This information can be gathered by the client software.

The distributed file system is not envisaged to play a role in our stream processing system. Therefore, any overhead due to its operation has to be deemed unnecessary. We will discuss the impact of the remnants of the distributed file system in the next section. At this point we must conclude that the monitoring as done by the Hadoop framework run against our ideas of a stream processing system.

Interoperability

Hadoop allows running non-Java code instead of the mapper and reducer functions. This feature is accessed using the Hadoop "Streaming" component - which unfortunately does not bear any relation to the concepts discussed in this report. Ensuring compatibility with other languages incurs overhead penalty on the start up of jobs and is likely to interfere with our low latency guarantee.

Another aspect of compatibility is the fact that even though we can remove the distributed file system from the data processing stage, we cannot entirely drop it due to the fact that it is also used as a communication medium between the JobTracker and TaskTrackers for exchanging configuration settings and the MapReduce code. Changes to the distributed file system architecture are therefore only going to be possible with a near complete re-write of the Hadoop code base. However, accepting the overhead of the distributed file system for these is difficult to justify when we want to optimise the system for minimum delay. We must thus conclude that it will be a major obstacle in using the Hadoop framework for stream processing applications.

Automatic scaling

Hadoop's automatic scaling of map and reduce tasks is not build for the data common in stream processing systems. A typical window in our financial data set contains only 10,000 tuples and is worth about 800 kilobytes of data. The scaling process in Hadoop is relatively coarse-grained and will thus result in queries where only a single TaskTracker is utilised. The problem with the standard allocation is that it is exactly what the Hadoop creators have found to deliver the best balance taking into account the overhead from the monitoring and fault tolerance procedures mentioned above. Significantly lowering the split size in our model is likely to result in a slowdown rather than a speedup. This could only be improved by increasing the complexity of the map and reduce tasks. The MapReduce approach is to split more complicated tasks into multiple chained MapReduce jobs. This, however, goes against our idea of decreasing the overhead.

JVM management

Somewhat linked to Hadoop's scaling mechanism is its management of JVMs[2]. With the default setting, Hadoop starts a new JVM for every task. A configuration parameter can be used to enable re-usage of JVMs for multiple tasks. However, JVMs are still spawned with every new job submitted to the JobTracker. This introduces a considerable non-deterministic overhead. In a stream processing system, the execution of a query must usually be completed in a fraction of the time needed to spawn a new JVM. However, as the JVMs are central to the design of the

[2] Java Virtual Machine

Hadoop framework, it is again difficult to remove them without upsetting the other components of the system.

3.4 Lessons learnt

The extension of Hadoop has shown us that it is possible to use a MapReduce system to run streaming queries. However, the legacy of the batch processing framework is to great to efficiently implement a stream processor on top of the Hadoop framework. In order to install persistent queries, that run without the overhead of the distributed file system and Hadoop's monitoring and recovery facilities, we would have to completely re-write the core of the Hadoop framework. As the whole framework is build on the idea of distributing all data by utilising the distributed file system, we have opted for another solution. In the next chapter we will show how we have used the lessons learnt from implementing stream processing on Hadoop to build our own distributed MapReduce stream processor.

Chapter 4

MAPS: An Alternative Streaming MapReduce Framework

A stream processor stands and falls with the latency at which it can process input windows. If the number of incoming tuples exceeds the processing capacity of the stream processor, its internal buffers fill up and data has to be dropped to ensure operation. Furthermore, client processes which depend on the outcome of the installed queries may depend on the timely delivery of results. The result of a query involving stock information becomes worthless as soon as new data indicating a change in the stock price becomes available. These requirements make a stream processor fundamentally different to other information retrieval and storage systems such as web-servers and databases. Although the latter two are also concerned with a speedy fulfilment of requests, the results of queries can be easily cached. Furthermore the data served by a web-server or traditional database is a lot less volatile than a continuous stream of data. Nevertheless, similar design considerations apply. In order to fulfil the low latency constraint, a stream processor must be both lightweight and if possible tightly integrated into the underlying architecture. Especially in a virtualised environment we must ask the question of how much OS support is needed to sufficiently avoid any overheads from the layered architecture. Careful consideration has also to be given to the complexity of the error handling and monitoring procedures. Monitoring the progress of a query on a single window is not relevant as we expect a result within milliseconds or seconds at most. Similarly, a failed query should not be restarted as the input buffers will have filled up with new window data.

In the previous chapter we have discussed the design and implementation of the changes necessary to run a stream processor on top of the Hadoop framework. We have noted that some of the design decisions made for Hadoop run against our concepts of a lightweight stream processor. In this chapter we will use these points to cover the design and implementation of MAPS, a custom MapReduce prototype written in Python. The aim of this chapter is to show a minimal implementation of the MapReduce concept. We will not show how this implementation can be optimised to give a minimum-latency stream processor but rather develop a robust prototype for evaluating the scalability of the MapReduce framework for continuous queries.

We will start by describing the design process of our prototype and explaining the similarities and differences to the Hadoop platform. After that we will show how our design can be implemented in the Python language. The evaluation of our new platform and a numerical comparison to Hadoop is given in the Evaluation chapter (see §6.1.3).

4.1 Motivation for Python as implementation language

Python cannot be regarded as being among the fastest programming languages in use today. It is not even one of the faster interpreted languages. In fact, its performance is quite dismal when compared to code compiled in C or even Java[1]. Nevertheless, it makes a lot of practical sense to implement our MapReduce prototype in Python. An obvious point is the ease of development and the ability to quickly change the structure of the framework to evaluate alternative approaches. It is very straightforward to exchange a module for another without having to worry strict type hierarchies. Of course this might result in a run time error every now and then, but for a system that is merely designed to evaluate a concept, this is acceptable. This is even less of a problem as small modules can be quickly tested and deployed without having to implement the whole application - a reason that is further aided by the fact that no compilation is needed to execute the code. Python gives us a simple way to express the parts of the system that matter. Whereas in Java we would need generics to express the fact that different formats are possible for key/value pairs, the dynamical type system in Python relieves us of this task. We are thus able to focus on the control flow among the modules on the master and slave nodes without having to worry much about the format of the data exchanged. Above all other reasons, however, we have chosen Python for its proximity to the functional programming paradigm behind MapReduce. It makes sense to operate on lists rather than arrays when we discuss map and reduce functions. In the background section, we have discussed how languages like Pig and Sawzall (§2.2.2) were introduced to simply the developer's task of writing MapReduce jobs. We feel that there is no need for any of these if queries can be expressed in functional languages whose concepts lie at the heart of MapReduce. Last but not least we wish to conclude this discussion by highlighting a quote from Google's Sawzall paper [34]:

> It may seem paradoxical to use an interpreted language in a high-throughput environment, but we have found that the CPU time is rarely the limiting factor; the expressiveness of the language means that most programs are small and spend most of their time in I/O and native run-time code.

As the paper suggests, the complexity of our MapReduce framework lies not within its implementation but rather with the choices made with regards to data distribution, load balancing and inter-process communication. Most time-critical libraries exhibited by the Python language are written in C and do therefore merely involve a small marshaling overhead.

4.2 Design decisions

When designing our MAPS prototype we will focus on rebuilding the essential parts of the Hadoop MapReduce system while leaving the non-essential overhead out. We will be re-implementing the core MapReduce logic and forgoing any sort of reporting and advanced error handling. For a discussion on the overheads of the Hadoop MapReduce implementation with regards to stream processing consult the previous chapter.

To honour Hadoop's division into JobTrackers and TaskTrackers, we will split the design into two parts. First we will discuss the requirements for our master node, containing the StreamServer and the job logic. Then we will go on to describe the implementation of our slave or worker nodes. The master node will handle the incoming data and the load balancing over the worker

[1] http://shootout.alioth.debian.org/u32/benchmark.php?test=nbody&lang=all

nodes. It will further deal with the addition and removal of slave nodes. There will be no provision for re-starting failed jobs. All the communication with the client is done through the master node. The worker nodes contain merely the logic needed to run the map and reduce functions and to participate in the distributed sort in between the two phases.

4.2.1 Role of the JobTracker

Polling vs central allocation of input splits

In Hadoop, the role of the JobTracker is merely to accept incoming jobs, to distribute them over the TaskTrackers and to provide accounting information to the calling process. Even though the JobTracker is still somewhat involved in a mediating role, it is not directly involved in the computation of the results. The job is set up using the JobClient and then send to the JobTracker for distribution. Neither class is directly involved in the gathering of input data, the processing of intermediate data and the output of the final result. In Hadoop, the data source is completely decoupled from the actual computation. This has allowed us to quickly extend the existing paradigm to allow streaming queries. However, it has also introduced additional overhead due to the introduction of additional proxies to connect the InputReaders and OutputReaders to the StreamServer. The upside of this distribution of control is an increased parallelism in the data allocation to the TaskTrackers. The downside is additional control logic in both the StreamServer component and the TaskTrackers. For our design, we have chosen to slightly modify the role of the JobTracker. As shown in Figure 4.2, the JobTracker now deals with the splitting of the input stream. The QueryHandler sub-module of the JobTracker then allocates the splits to the TaskTrackers and calls their map and reduce functions.

Query installation

Instead of the TaskTrackers polling the JobTracker for new tasks, we have chosen to push the query to all available TaskTrackers from the JobTracker. We have chosen this option as we are dealing with continuous queries. It is unlikely that a query is changed during the run time of our stream processor.

When we spawn a new TaskTracker, it remains idle until it receives a request from a JobTracker. The first request by the JobTracker is a call to assert that the TaskTracker is live. It will then attempt to move the code for the map and reduce functions over the network and install them within the TaskTracker. This setup must take place before the first window arrives on the stream processor. We do not at this stage plan to introduce functionality that enables changing the queries in a running system. However, adding a new vanilla TaskTracker to the running system should cause the JobTracker to install the query on this node to make it available for processing as the next window comes in.

Query validation

Ideally, the JobTracker should validate the query before installing it on the TaskTrackers. The map and reduce function must fit the interface with the TaskTracker (i.e. accept key/value pairs) and each other (i.e. provide compatible key/value pairs). This concept is shown in Figure 4.1. As we have chosen not to implement any error handling during the execution of queries, it is paramount to make sure that only valid queries are installed on the TaskTrackers in the first

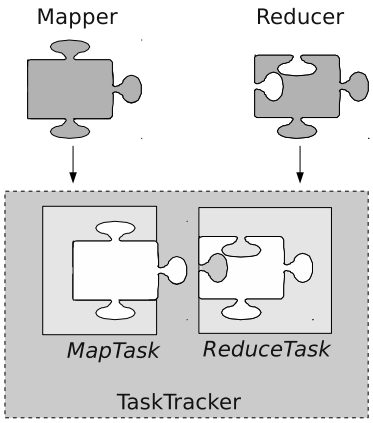


Figure 4.1: Query validation. A functions have to fit to their partner and the framework.

place. To achieve this, the JobTracker checks if the supplied code correctly defines the mapper and reduce functions as required by the TaskTrackers.

4.2.2 Role of the TaskTrackers

A TaskTracker's main responsibility in our architecture is to compute the results for the application of the map and reduce on its input split. In addition, it is involved in the partitioning of the results for the reduce phase by sorting the output of its map function. However, the TaskTracker is always passive in its behaviour. The query and its invocations are pushed from tasks in the JobTracker. The TaskTrackers contain no error handling and expect the query from the JobTracker to have been validated. As their map and reduce tasks are invoked from the JobTracker, they are also oblivious to the structure of the map reduce process. They are merely work horses, designed to quickly return a result to the master for processing.

4.2.3 Possible extensions

The system is designed to easily cope with additions such as a variation in which the master node handles the map tasks as well. This could be interesting when there is a big discrepancy between the work done during the map and reduce phases. This shall, however, not be part of our original design as distributing the input data helps us to conduct the necessary sort in a distributed merge-sort fashion. Currently, we have only installed a single query on the TaskTrackers. The architecture can be easily extended to allow for multiple, pipelined queries. This can be interesting when a single MapReduce task is not expressive enough anymore. We have ignored this for now due to the added overhead in communication. In Chapter 6, we will discuss the parallel overhead in detail.

4.3 Components to be implemented

In this section we will describe the components which need to be implemented for our MAPS framework. Figure 4.2 gives an overview of the components involved in a complete MapReduce cycle. The JobTracker handles the distribution of the query. The QueryHandler is a subordinate of the JobTracker which deals with creating the input splits and connecting to the TaskTrackers to spawn the map functions. The OutputCollector takes the partial lists from the mappers and

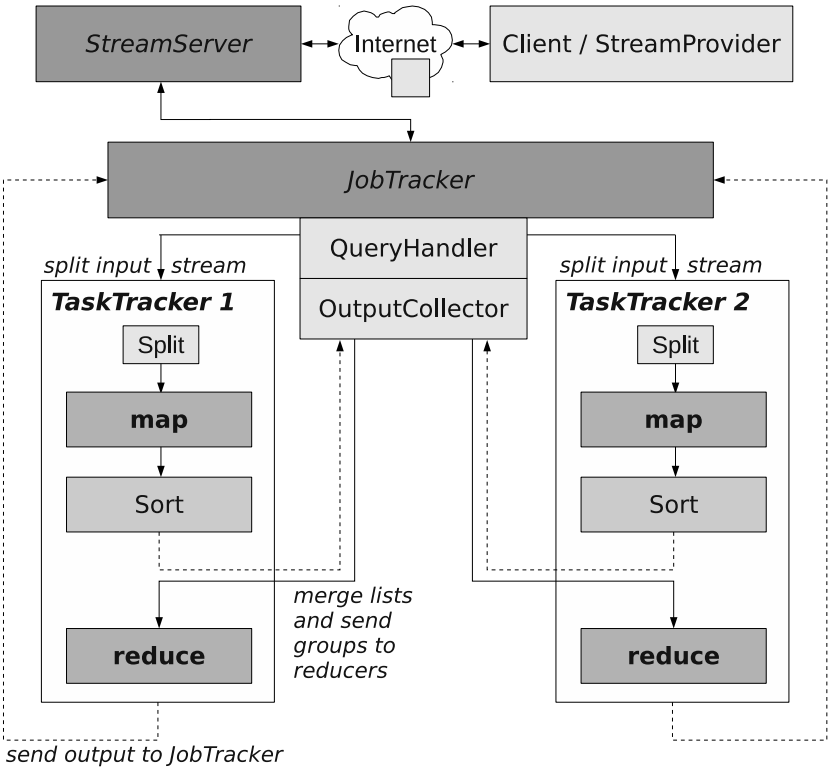


Figure 4.2: Overview of our MAPS prototype

produces the grouped `(key, [value])` pairs for the reducers. The final output is passed back to the StreamServer via the JobTracker.

4.3.1 Master node

StreamServer Central to the design of the master node is the StreamServer, which handles the incoming stream and provides windows for the mappers and reducers to process. The StreamServer should listen on a TCP port, ready for any incoming connections. Once a connection is established and a window worth of data received, the StreamServer should forward the data to the job on the JobTracker.

JobTracker The JobTracker is the focal point of our MapReduce implementation. It should run independently of the stream server and ping the TaskTrackers at regular intervals. The JobTracker must obtain information about live TaskTrackers from a name node. All communication between the components of our framework is done using remote method invocation (RMI). A process registers its methods with the name server to make them accessible to other processes on the network. It is due to the invoker to make sure that the information on the name node is current. The JobTracker must therefore keep a list of all active TaskTrackers.

QueryHandler *(Module of JobTracker)* Once a job is submitted from the StreamServer to the JobTracker, the JobTracker hands control to the QueryHandler. During the map phase, the QueryHandler is responsible for splitting the window into equally sized splits. The number of splits is determined by the number of currently active TaskTrackers. To make sure that the number of TaskTrackers does not change during this calculation, we do not allow adding

new nodes during the execution of a query. We assume that the chances of a node failing are negligible. If a node does fail, the current window will be dropped. Once the window has been split into equally sized chunks, the QueryHandler will spawn worker threads to handle the communication with the map functions on the TaskTrackers. These threads will remotely invoke the map function and receive the sorted result.

OutputCollector *(Module of JobTracker)* When the result has been received, we must merge the output lists to be able to generate the grouping for the reducers. In our implementation, this is done by the OutputCollector. The OutputCollector merges the lists, groups them by key and allocates a new set of splits based on the (key, [value]) pairs. These splits are then again distributed to the TaskTrackers for the reduce phase. Once the reduce phase is completed, the data is send via the JobTracker back to the StreamServer to be communicated to the client node.

4.3.2 Slave nodes

TaskTracker During both the map and reduce frame, a TaskTracker called by the QueryHandler will spawn a worker thread to run the requested function on the supplied input data. As the TaskTrackers are invoked by an RPC call (see §2.2.4), the result is passed straight back to the QueryHandler. Once the result has been computed, the TaskTrackers return to an idle state and wait for further requests.

4.4 Implementation

In this section we will discuss the implementation of our prototype. First, we will the discuss specifics regarding the threading model. We will then go on to explain how we deal with remote object communication and how the MapReduce query is installed on the TaskTrackers

4.4.1 Helper threads

Figure 4.3 shows the class diagram of our MAPS implementation. Most of its components have already been discussed in the previous section. In addition, we have introduced the `MapRunner` and `ReduceRunner` classes on the JobTracker side and the `MapTask` and `ReduceTask` classes on the TaskTracker side. These classes are threads associated with running the query on an input split. On the JobTracker, these are communicating with the TaskTrackers. On the TaskTracker they can be used to achieve further parallelism by splitting the input split a second time. The MapTask is further responsible for handling the sorting of its input split prior to returning the result back to the OutputCollector.

4.4.2 Inter-node communication - Python Remote Objects

We have chosen to communicate data and command information between the master node and the slave nodes by remote procedure calls (RPC see §2.2.4). However, since Python is an object oriented language, we are not registering functions but whole objects. The operation is the same. We have decided to use the Pyro (Python Remote Objects) library to provide the framework for these calls.

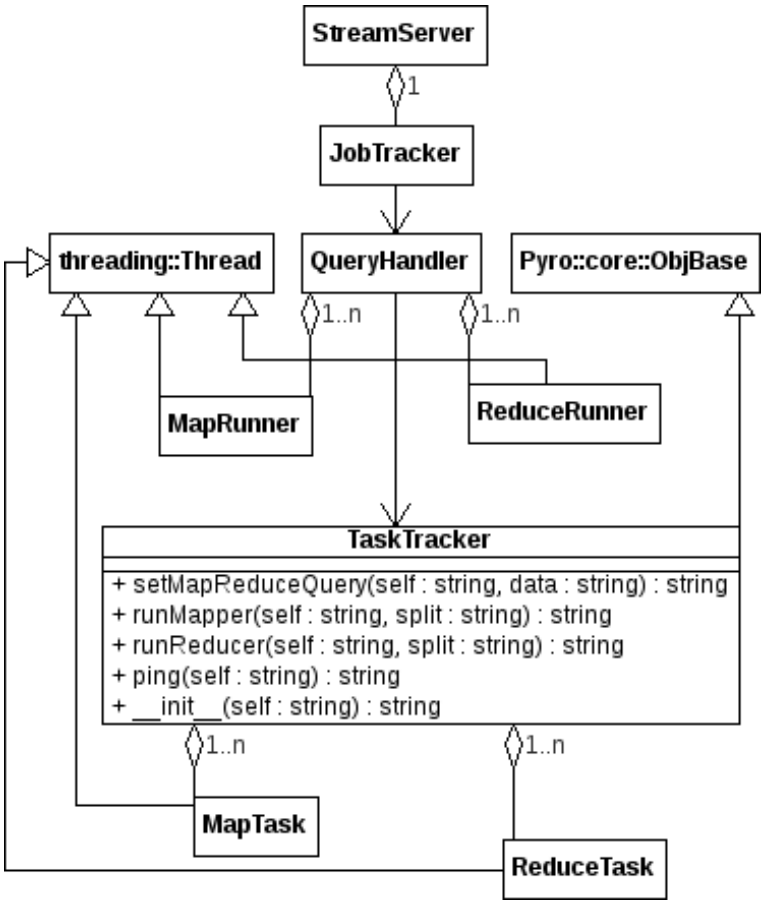


Figure 4.3: UML class diagram depicting the high level organisation of our MAPS prototype

Pyro handles the pickling and unpickling (serialising and de-serialising) of data automatically, which means that unlike in Java, no extra code is required. Another advantage of Pyro is that it already delivers a name server. The name server broadcasts its information on the network which makes the connection setup straightforward on simple networks. In order for it to reliably work on a cloud infrastructure, we have to make sure that the network is such that the clients can discover the name server. If this is not possible, we must set the details of the nameserver per hand like in the Hadoop implementation and distribute configuration files with the TaskTracker and JobTracker executables. All TaskTrackers register at the namenode using a unique identifier preceded by `tasktracker`. No two TaskTrackers are allowed to register with the same Uniform Resource Identifier (URI). The sub-string "tasktracker" makes sure that the JobTracker recognises the tasktrackers as such.

The TaskTracker creates a daemon similar to the `TaskUmbilicalProtocol` in Hadoop (see line 8 in Listing 4.1). This daemon is connected to the TaskTracker implementation (line 11) and registered with the name server (line 10). Once the request loop is started, the TaskTracker is ready to serve any requests. In Listing 4.2 we show how a simple client can now connect to the TaskTracker and execute its `ping()` method. Note, that the actual implementation is somewhat more involved as we do not know the URI of the TaskTracker in advance.

4.4.3 Dynamic loading of modules

The distribution of the map and reduce functions is done by sending the source code over the network from the JobTracker to the TaskTrackers. The query is then installed by a simple

```
import Pyro.core
import Pyro.naming

class TaskTracker(Pyro.core.ObjBase):
    def ping(self):
        return True

daemon=Pyro.core.Daemon()
ns=Pyro.naming.NameServerLocator().getNS()
daemon.useNameServer(ns)
uri=daemon.connect(TaskTracker(),"tasktracker")
daemon.requestLoop()
```

Listing 4.1: Pyro Server (Example)

```
import Pyro.core

tasktracker = Pyro.core.getProxyForURI("PYRONAME://tasktracker")

if tasktracker.ping():
  print "Success!"
```

Listing 4.2: Pyro Client (Example)

import statement (see Listing 4.3). The overhead of the transfer is negligible as the MapReduce class is installed before the query is executed.

4.4.4 Query validation

As discussed above, the MapReduce query will be written as a Python class and loaded and evaluated at runtime. In the absence of any checks, the MapReduce program will fail during the execution of the query if the class does not contain the necessary methods (i.e. the mapper and reducer functions). To remedy this, we have included a few sanity checks in our design which make sure that the aforementioned methods are actually contained within the query implementation (see Listing 4.4). This is necessary as Python is a dynamically typed language which only checks types at run time. We therefore have to rely on our own checks to make sure that the user has supplied the right function. At the moment we do not include any more sophisticated checks which means that the distribution of MapReduce queries is a possible attack vector for a malicious user. As this is merely a prototype implementation, we will leave proper query validation for future work.

```
mapred = __import__("mapred").MapReduce()
methods = [method for method in dir(mapred) if callable(getattr(mapred, method))]
if ("mapper" in methods and "reducer" in methods):
  print "[JobTracker] MapReduce function validated."
else:
  print "[JobTracker] Error, missing MapReduce implementation."
```

Listing 4.4: Dynamic loading of MapReduce implementation module during run time

```
mapred = __import__("mapred")
self.mapredimpl = mapred.MapReduce()
self.maptask = MapTask(self.mapredimpl)
```

Listing 4.3: Dynamic loading of MapReduce module and initialisation of MapTask during runtime

4.5 Discussion

When we designed MAPS, the goal was to remove some of the overhead introduced by the Hadoop framework (§3.3.3). Our design is not based on a distributed file system while still enabling the user to distribute the map and reduce functions through a single point of entry. We have compromised on speed by using remote procedure calls, staying true to the Hadoop framework. For a more efficient implementation we would consider use a technology like MPI [26] for distributing the data.

Besides the communication overhead, we acknowledge that the choice of the implementation language has reduced the efficiency of our stream processor. The significance of this overhead is set into perspective when we compare the run time of our custom implementation to Hadoop in Chapter 6.

The evaluation must further show if our decision to centralise the distribution of input splits has worked. We deem it to be unnecessary to have the TaskTrackers polling for new data. In Hadoop, one of the reasons why this is done is because idle TaskTrackers can do speculative execution of tasks, thereby avoiding the situation in which all TaskTrackers wait for the output of a single one. The MapReduce algorithm has to synchronise the TaskTrackers in between the map and reduce phases to do the grouping. With speculative execution, the impact of an unresponsive TaskTracker could be remedied. In our implementation we will immediately see if a TaskTracker is unresponsive. The window is dropped and the TaskTracker removed from the pool for the next iteration. Speculative execution only introduces an additional overhead that we cannot be prepared to take if we want to achieve a low latency response.

Chapter 5

Load Balancing Into The Cloud

The second part of our project focuses on a solution to enable a local stream processor to utilise the cloud infrastructure to guarantee a latency bound on the computation time. In this chapter we will first describe the design and implementation of simple local MapReduce style stream processor. We will then go on to design two load balancing strategies. First we will consider an **always-on** approach in which the load is balanced between the local node and the cloud in order to achieve a higher throughput overall. The always-on approach is not economical. In fact, it raises the question why we would not move all processing to the cloud in the first place, thereby eliminating the need for both the load balancer and the local processor. However, the always-on load balancer can be used to find the best split between cloud processor and the local node with respect to network bandwidth and the scalability of the query. A higher bandwidth between the load balancer and the cloud or a more complex query will result in more windows being processed by the cloud.

Building on these ideas we will introduce an alternative approach. In **adaptive**-load balancing, the cloud is used as a backup system to deal with peaks in the load. We expect the local stream processor to deal with the stream most of the time. However, when the incoming data exceeds a certain rate, it has to begin to drop windows. At this point the adaptive load balancing algorithm should begin to move computation to the cloud, thereby lowering the number of windows dropped while ensuring the latency constraints are met. By only utilising the cloud infrastructure to support the local processor on-demand, we can keep costs at a minimum.

5.1 Local stream processor

In the last chapter, we have described a distributed MapReduce framework written in Python. This framework may be simplified to run on a single node. It is clear that any of the other stream processors described in the background section are equally applicable here. However, an implementation similar to our cloud architecture will result in more seamless load balancing as we can standardise the input and output formats. In fact, we could simply re-use MAPS and deploy it on a single node. Performance figures for this approach are given in the next chapter. We have chosen to simplify our existing Python solution to accommodate the need for the load balancer to run on the same node. We shall now describe how we simplified the design and the effects of these changes on the processing of a single MapReduce query.

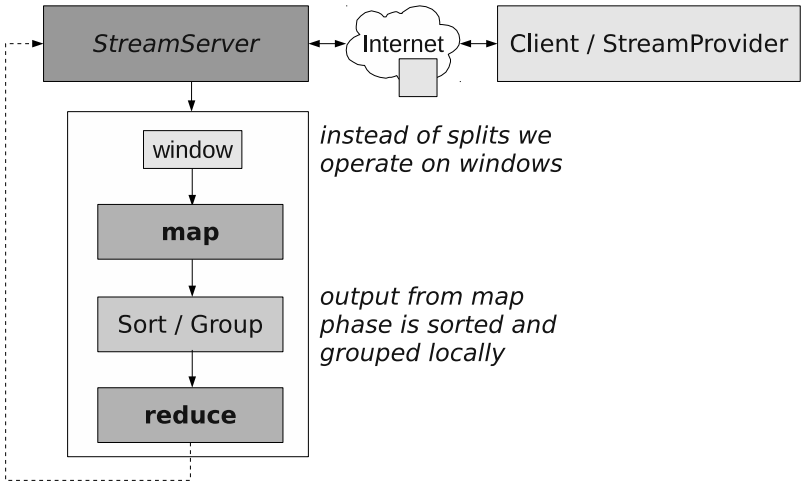


Figure 5.1: Simplified local stream processor

5.1.1 Simplifications

As we do not need to communicate data with other nodes, there is no need for a remote method invocation architecture. Furthermore, we do not need to distribute the query over a set of nodes. We therefore drop the concepts of a JobTracker and its associated TaskTrackers. This makes it possible to simplify handling of a request to a simple call to the MapReduce logic. Figure 5.1 shows the simplified stream processor. Like in the distributed framework, the MapReduce code is contained in a single class file. The only difference is that this file is now part of the stream processor architecture. This is acceptable since the overall design of the local stream processor has become simple enough for the end user to manipulate. In fact, we dropped any hints of load balancing within the local stream processor for this exercise. The threaded reducers as implemented in the last chapter are deliberately omitted to deliver a bare minimum configuration.

5.1.2 Query invocation

The process of starting a query on an incoming window remains the same. The StreamServer component is still present in our single node implementation. The client will establish a connection with the StreamServer and send a window worth of data. Instead of calling the JobTracker it will then directly invoke the MapReduce function. There is no need for an intermediate handler due to the simplicity of the MapReduce framework. The only work done outside the map and reduce functions are the sorting and grouping after the map phase. As we only have a single node, the sorting mechanism designed in the last chapter is still applicable as a sort on a single node results in the whole output to be ready for the reducer. We do not have to merge any splits after the map phase. Once the query has been executed, the final result is passed back to the StreamServer and send over the network back to the client.

5.2 Design

In this section we will show the design and implementation of our two load balancing solutions. As the always-on solution contains only a subset of the features of our full adaptive load balancer,

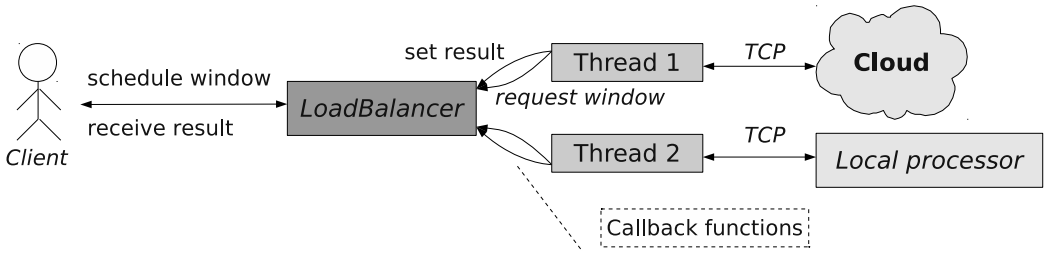


Figure 5.2: Diagram depicting the load balancer as a standardised stream processing interface. The interface provided by the LoadBalancer to the client is the same as a stream processor's. When the load balancer is initialised, its threads connect to their respective stream processors. They request windows as soon as these connections have been established. Once a result has been received, they notify the LoadBalancer and ask for new data.

we will consider its design first. After that we will go on to explain in detail the extended version which can switch between *local-only* and *cloud-supported* processing.

5.2.1 Always-on load balancing

This approach is the most similar to the LARD design discussed in §2.4.2. Instead of connecting straight to either the local or the cloud stream processor, we connect via a proxy load balancer. This load balancer sends windows alternatively to the local or the cloud stream processor[1]. While sending the data, it also records the response time of the implementations. The ratio between the response times, determines the internal buffer size (eg. if the cloud needs twice as much time to return a result, the number of windows processed by the local node is doubled). By using an internal buffer, the load balancer ensures that we do not receive earlier windows after more recent ones as this would defeat the purpose of computing them in the first place.

The always-on load balancer starts dropping windows if even with full utilisation of both the local processor and the cloud processor there is not enough computational power to ensure that windows are processed in time. The input queue would have to be infinite. This is similar to the load shedding discussed in the background chapter. Like above, the load balancing algorithm recognises this case by monitoring the rate at which input tuples arrive and the occupation of the job queue. If tuples arrive at a rate too great for the combined system to manage, they are dropped to make sure that our (fixed size) queue is not exceeded. Information about dropped tuples is written to a log file for later examination.

The algorithm

The load balancer is passive and does not send windows to the stream processors without receiving a request first (see Figure 5.2). The exact implementation is discussed in the next section. In Listing 5.1 we present a pseudo-code representation of the algorithm used to service a request for new data.

In the always-on case, the input for the stream processors is never taken of the input queue. Instead it is taken from the split buffer (lines 13 and 18). As mentioned above, the split buffer reflects the ratio between the processing times of the cloud and the local stream processor. We are using the buffer to make sure that we are processing windows in the correct order.

[1]One may argue that returning old windows at all makes no sense if more recent data is available. In this report we assume that a client is interested in a certain range of fresh windows. In high frequency trading, this could give a trader an idea of where the price of an asset is going.

```
1  procedure get_window(source):
2    synchronized(split_buffer):
3      if split_buffer is empty do:
4        if not initialised do:
5          add two windows to the split_buffer
6          initialised := true
7        else do:
8          local_latency := local_timer.previous()
9          buffer_size := (cloud_timer.previous() / local_latency) + 1
10         add buffer_size windows to the split_buffer
11
12   if source == local do:
13     window := split_buffer.take() {Blocking call}
14     start local_timer
15     return window
16
17   else if source == cloud do:
18     window := split_buffer.takeNewest() {Blocking call}
19     start cloud_timer
20     return window
```

Listing 5.1: Pseudo code showing the handling of an incoming request for data in the always-on approach

When the system is booted, the split buffer is empty. A request for input data then fills it with two elements only (line 5). We do not know the ratio between the cloud's and the local processor's processing times, yet. Any subsequent calls, will have either one or both of these values. The timers' `previous()` functions make sure that if no times were recorded, we do get values which allow us to eventually boot-strap a successful splitting.

At the moment, the algorithm only allows for two stream processors to take part in the load balancing. Depending on the source field in the function call, either the `local_timer` (line 14) or the `cloud_timer` (line 19) are started. This system can be extended to a larger number of participants by using indices into an array of counters. The split can then be calculated by normalising the array of run times such that all elements are greater than one.

Safety

The system is free of deadlocks. The only way for a thread to block is to wait on the split buffer. By doing so, it releases the lock, so another process could fill the buffer. The two processes servicing the stream processors are continuously calling `get_window()` to ask for new data. Therefore in order for both to block they would both have to reach line 12 with an empty split buffer. Since neither has removed an element from the buffer since the beginning of the method call, at least one of them must have seen the empty queue and filled it. But the queue is empty - contradiction. As long as we do not have another thread accessing the queue, the design is deadlock free.

5.2.2 Adaptive load balancing

The structure of the adaptive load balancing algorithm is similar to the always-on approach. As described in the previous section, we monitor the size of the queue at the front end node to decide how to make use of the cloud's resources. An overview of the proposed architecture is shown in Figure 5.3. Input windows are received by the StreamReader and appended to the input queue in the QueueManager. Depending on the state of the load balancer, these windows

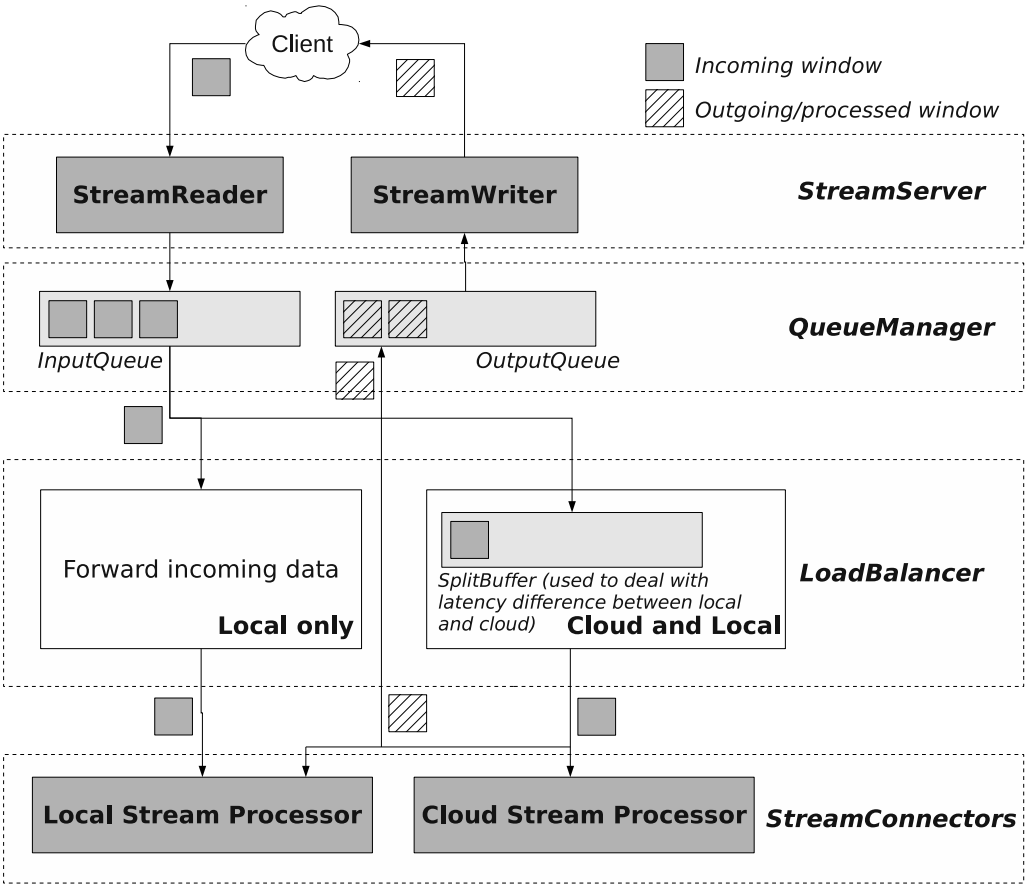


Figure 5.3: Diagram depicting the proposed architecture of our adaptive load balancer.

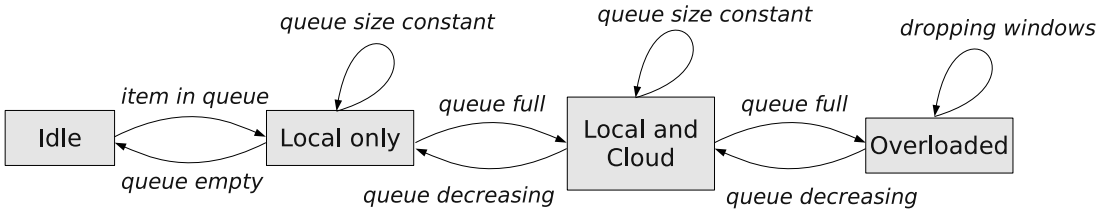


Figure 5.4: State diagram depicting our on-demand load balancer

will then be allocated straight to the local stream processor or allocated via the split buffer to the cloud and the local stream processor. The output from both stream processors is send to the output queue in the QueueManager. The StreamWriter drains the output queue and sends the data over the network back to a client.

States

The state transition diagram of our on-demand load balancing algorithm is shown in Figure 5.4. It contains four basic states. In the initial state, the load balancer is *idle*. Once a window worth of data is received, it is appended to the queue and a state transition to *local-only* stream processing occurs. As long as the queue size remains constant, the local node can cope with the demand and no state change occurs. If the queue becomes empty, we fall back into the idle state.

If the queue becomes full (ie tuples arrive at a rate too high for the single node to cope), we move to the *cloud-and-local* state in which both, the local node and the cloud processor handle the input. Again, as long as the queue size remains constant, we stay in this state. A decreasing queue means that we have superfluous processing power. Therefore this event causes us to return to the local-only state.

In the event that the queue should fill up even though we use all available resources, we move to the *overloaded* state. In this state, the load balancer drops windows in order to cope with the load. Only when no more windows are dropped and the queue size stabilises, do we return to the previous state.

The algorithm

Listing 5.2 shows the `get_window()` function for the adaptive load balancer. It is very similar to the one of the always-on approach. In order to change the state we obtain a lock on the split buffer (line 2). The split buffer is only filled if we are in the *cloudAndLocal* state (line 3). If no previous measurements for the performance of the cloud have been obtained (i.e. `initialised == false`), the buffer is filled with two elements only (line 5). Otherwise we compute the input split (lines 8-10). The input split is equal to the number of times the local processor can process a window before the result from the cloud is returned[2].

A request from the cloud is then serviced from the split buffer or send to sleep if no data is availabe (i.e. cloud processing is not enabled). A request from the local node needs to check the state (line 15) to find out if its request is serviced by the main input queue or the split buffer.

The state of the load balancer is changed from the QueueManager via the two callback functions shown in Listing 5.3.

Safety

The algorithm is free of deadlocks. To prove this, we show that it is impossible for both the cloud and the local processor to wait on the split buffer at the same time. Let us consider the case in which the split buffer is empty and we are in the *localOnly* state. The cloud is blocked on the split buffer. We must ensure that the call from the local processor does not block. The call can only block if we are in the *cloudAndLocal* state as the request would otherwise be served directly from the input queue (else branch at line 20). Due to the `synchronized` statement, there are only two possible places during the execution of `get_window()` where a state change could occur. This is either before the first line of the function or at any time after line 23. Note that it is possible (albeit unlikely) for multiple state changes to occur. The code after line 23 is very similar to the always-on implementation and behaves in the same way. Note that the `synchronized` is not needed as the `takeNewest()` method needs to obtain this lock anyways.

Now let us show that the `take()` call on line 16 can never block. For it to block, we must be in the *cloudAndLocal* state and the split buffer must be empty. As we have the lock on the split buffer since line 1, no other process could have changed the state or the contents of the buffer (compare Figure 5.3). Therefore we must have seen the empty buffer and the *cloudAndLocal* state at line 3. We would have acted and filled the buffer, but the buffer is empty. Contradiction. Therefore, we cannot block on line 16.

[2]Due to constraints in the network bandwidth, the cloud currently takes longer to process a window - see §6.2.5

```
1  procedure get_window(source):
2    synchronized(split_buffer):
3      if split_buffer is empty and state = cloudAndLocal do:
4        if not initialised do:
5          add two windows to the split_buffer
6          initialised := true
7        else do:
8          local_latency := local_timer.previous()
9          buffer_size := (cloud_timer.previous() / local_latency) + 1
10         add buffer_size windows to the split_buffer
11
12
13     if source == local do:
14       {Combined processing, need to access buffer like in the always-on case}
15       if state == cloudAndLocal do:
16         window := split_buffer.take() {Blocking call}
17         start local_timer
18         return window
19       {Local-only processing, we can access the main queue}
20       else do:
21         start local_timer
22         return window from input queue
23
24   if source == cloud do:
25     window := split_buffer.takeNewest() {Blocking call}
26     start cloud_timer
27     return window
```

Listing 5.2: Pseudo code showing the handling of an incoming request for data in the adaptive approach. Note how the cloud accesses the newest element in the buffer, whereas the local nodes process the queue in regular FIFO order.

```
1  procedure notify_window_dropped():
2    synchronized(split_buffer):
3      state := cloudAndLocal
4
5  procedure notify_queue_empty():
6    synchronized(split_buffer):
7      if split_buffer is empty and state = cloudAndLocal do:
8        state := localOnly
```

Listing 5.3: Pseudo code showing the callback functions used by the QueueManager in the adaptive approach

As long as more than a single element is added to the split buffer, fairness is also given as the call from the cloud will eventually wake up to a filled buffer.

Note Deadlocks could still occur if the communication between the QueueManager and the callback methods is not properly coordinated. If the QueueMAnager obtains a lock on the (main) input queue and then called either `notify_window_dropped()` or `notify_queue_empty()` we will deadlock if we are in the *localOnly* state and a simultaneous call to `get_window()` has progressed past the synchronized statement. Therefore we must not lock the input queue during the callbacks.

5.3 Implementation

In this section we will discuss details specific to our implementation of the design discussed above. We will explain our choice of data structures as well as give some more detail on the components of our system.

The load balancer acts as a proxy to our actual stream processors. It is therefore vital, that its overhead is kept at a minimum. If the local stream processor is too slow, we can utilise the cloud. However, if the load balancer cannot keep up with the data stream, we must drop windows. For this reason, we have decided to deviate from our pattern and to implement this component in Java. All the communication between the nodes is done using TCP sockets and no remote method invocation is needed as part of the load balancer. As the interface for the cloud and local stream processors is the same, the load balancer can essentially treat them in the same manner. Furthermore, it is possible to extend the existing system to handle multiple local processors as well as multiple clouds.

As the components and algorithms used in the always-on approach are a subset of the adaptive load balancer, we will only discuss the implementation of the latter in this section. The always-on case can be easily obtained by modifying the `AdaptiveLoadBalancingAlgorithm` component to include the algorithm shown in Listing 5.1.

5.3.1 Concurrent data structures

In the design process we have referred to blocking queues. In our implementation we have chosen to use the `ArrayBlockingQueue<E>` and `LinkedBlockingQueue<E>` from *java.util.concurrent* for this task. These classes give use bounded and unbounded queues respectively. Both handle thread safe addition and removal of elements. Most importantly, the queues behave in a FIFO (first in - first out) fashion and block if no elements are available.

5.3.2 Logging

We used the Java Logger from *java.util.logging.Logger* to log progress within our load balancer. The output was redirected to a file. In Chapter 6 we will examine this file using standard UNIX utilities like grep and awk. To prepare for the analysis, we therefore made sure that any logged messages contained their origin as well as a time stamp.

5.3.3 Components to be implemented

In Figure 5.3 we have shown a high level view of our solution. Figure 5.5 shows the main classes participating in the process. We will now examine these in more detail. The solution is held together by a main class which takes as input the size of the input queue and constructs the other parts of our load balancer. In our description of the components involved, we will try to go through the aforementioned diagrams from top to bottom, starting with the interface to the client (StreamServer) and finishing with the interface to the stream processors.

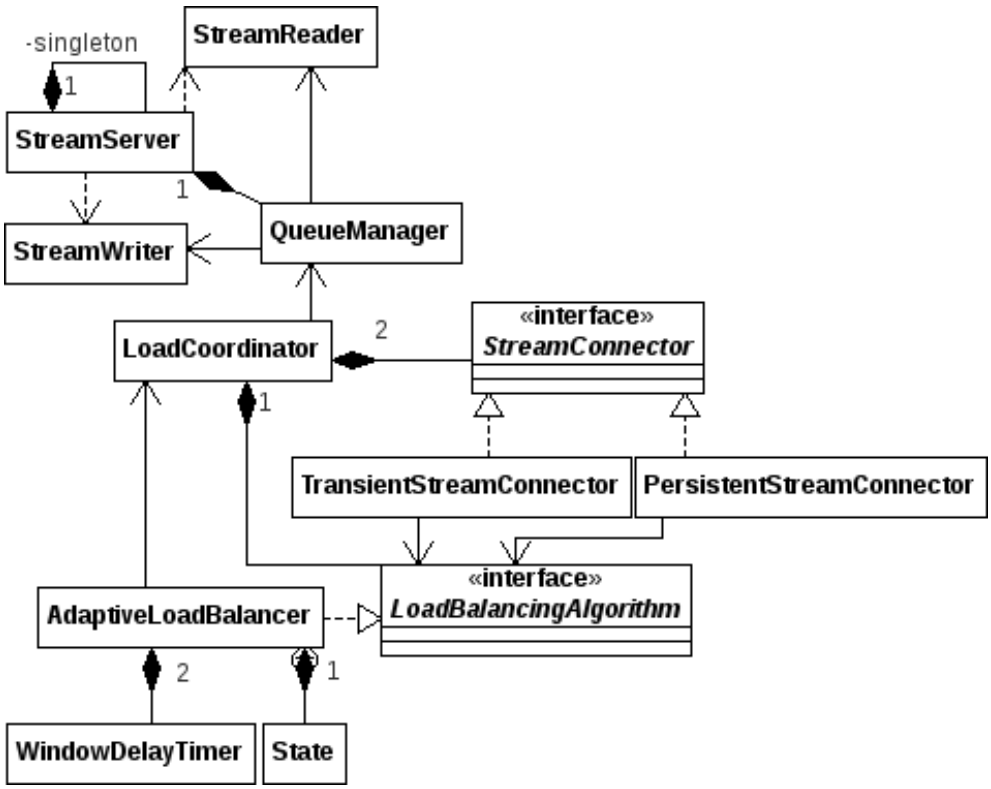


Figure 5.5: Class diagram depicting the organisation of our load balancing solution

StreamServer

The StreamServer component can be compared to its namesake in the Python and Hadoop stream processor implementations. It is responsible for managing the connection to the client. In order to fulfil this task, it creates two threads - StreamReader and StreamWriter - to serve the input and output streams respectively.

StreamReader The StreamReader is a thread which listens for incoming tuples. Once a window of data has been collected, it is send to the QueueManager. It is the responsibility of the QueueManager to decide how to deal with it. The StreamReader does not wait on the QueueManager and returns immediately to servicing the input stream.

StreamWriter The StreamWriter is a thread which allows the client to connect on a different port. We have separated the handling of input and output for the load balancer such that it becomes possible for the output of the loadbalancer to be forwarded to a different host. We assume that the originator of the stream is not necessarily the same as the subscriber (eg. a stock exchange and a bank respectively). The StreamWriter calls the QueueManager and blocks on the OutputQueue if no data is available.

QueueManager

The QueueManager has jurisdiction about the input and output queues. It handles the dropping of windows if the size of the input queue is about to be exceeded. The oldest window is dropped from the queue and the new window appended. The QueueManager uses the callback function

`notify_window_dropped()` defined in the `QueueEventsListener` interface (see below) to make the algorithm aware of this event. Likewise, the `notify_queue_empty()` callback is used when the input queue has been exhausted. This enables the load balancing algorithm to evaluate the possibility of scaling back to local-only processing. The QueueManager gives the algorithm the possibility to obtain a single window from the input queue. Alternatively, it is possible to request a list of n windows. In the latter case, the QueueManager will return $min(size(inputQueue), n)$ windows. This call is used when the algorithm tries to fill its split buffer.

The QueueManager also deals with the output queue. The `collect()` method from the load balancing algorithm is re-directed here. As mentioned above, the output queue is read by the StreamWriter and forwarded back to the client.

LoadBalancingAlgorithm

LoadBalancingAlgorithm is the interface describing the responsibilities of our algorithm. It extends QueueEventsListener, OutputCollector and StreamProvider. The `QueueEventsListener` describes callback methods which can be used by the QueueManager to inform the algorithm if windows have been dropped or the queue has been drained empty. This information is needed by the LoadBalancingAlgorithm to make decisions about the next state. The `OutputCollector` and `StreamProvider` interfaces are linked to the callbacks from the classes implementing the StreamConnector interface (see below discussion). They allow the stream server to query the algorithm for windows to process and to contact it with the output data.

LoadCoordinator The LoadCoordinator class connects to the QueueManager and creates the instance of the load balancing algorithm to deal with the callbacks from the stream processors. It connects the callback from the QueueManager to the LoadBalancingAlgorithm.

AdaptiveLoadBalancingAlgorithm This is a concrete implementation of the `LoadBalancing Algorithm` interface and the algorithm shown in Listing 5.2 and the callbacks of Listing 5.3. The state is implemented using an enum type. The internal buffer is constructed from a `ArrayBlockingQueue<E>` as mentioned previously. It is using the `WindowDelayTimer` custom timer implementation to compute the size of the internal buffer.

WindowDelayTimer The WindowDelayTimer class is used by the load balancing algorithm to get information about the latency of our stream server implementations. When we start processing on either the cloud or the local node, we start the timer using its `start()` method. A call to the `stop()` method returns the time spent processing. The time is obtained using the System.nanoseconds() method. The class further delivers the time previously recorded using the `previous()` method. An average of the last 5 (a different number can be specified in the constructor) times can be obtained by the `average()` method. We prefer this to the previous() implementation as it gives us a better estimate on the latency of the next window.

Stream connectors

The StreamConnector interface describes a thread to handle the connection to a stream server. We have two concrete implementations: PersistentStreamConnector and TransientStreamConnector. As mentioned previously, these threads connect to the stream server and then call upon the load balancing algorithm to provide input data.

PersistentStreamConnector This class maintains a single connection to the StreamServer of our stream processors (cloud or local). This connection is kept alive for as long as the load balancer is running. This is the most efficient way to connect the load balancer to the stream processors in a connection-oriented protocol.

TransientStreamConnector This class can be used to connect to a stream processor which ends the connection after the computation of a single window. It automatically re-connects to be ready for the transmission of the next window.

5.4 Discussion

In this chapter we have shown how to design an adaptive load balancing algorithm to handle the distribution of work between a local stream processor and the cloud implementations described in the previous chapters. We have chosen Java as our implementation language to make sure that the overhead from the load balancer is as small as possible. We have deliberately moved the control flow of load balancing to the stream connectors. Their calls to the `get_window()` function drive the computation of results. This allows us to extend the algorithm to run on more than a single cloud and local node.

We have shown how our always-on and adaptive algorithms are safe with respect to deadlocks. It remains to be seen if the coarse locking had an impact on the performance. As we noted during the discussion of the safety of the adaptive algorithm, we must add at least 2 windows to the split buffer in order for the cloud to have the chance of getting a window to process. It will be interesting to see how the sizes of both the input queue and the split buffer behave.

In the next chapter we will assess the performance of our load balancing solution in regard to its ability to reduce the number of windows dropped as well as its impact on the latency of the combined system.

Chapter 6

Evaluation

In this chapter, we will start by evaluating our MapReduce implementations for stream processing on a cloud infrastructure. We will be looking at the theoretical foundations of parallel algorithm design and show measurements for both Hadoop and MAPS.

We will evaluate our two load balancing solutions. First, we will try to understand how we can utilise the cloud's resources to give an always-on enhancement to the local stream processor. We will then go on to test our dynamic load-balancer and its capabilities to amortise peaks in the load on the local machine.

Notation

In our evaluation of the algorithms, we will make use of the big-oh notation to provide an upper and lower bound on their sequential and parallel run time. We shall use $\Theta(g(n))$ such that for any $f(n) \in \Theta(g(n))$, f is bounded both above and below by g asymptotically. Furthermore, we shall use the following formula to calculate the speedup of a parallel computation, where S_p is the speedup of the parallel implementation over the sequential one. T_1 denotes the time spend by the sequential algorithm. T_2 denotes the time spend by the parallel algorithm on p machines.

$$S_p = \frac{T_1}{T_p}$$

From the above equation, it becomes obvious, that the best possible speedup is $S_p = p/(1 + p\delta)$, where δ stands for the the communication overhead per processor. We will assume super-linear speedup is not possible in our implementation. Following from this assumption, we will try to ignore where possible, the impact the memory hierachy has on our simulations.

We will measure the efficiency of our parallel/MapReduce implementation by the following function, where p denotes the number of processors:

$$E_p = \frac{S_p}{p} = \frac{T_1}{pT_p}$$

The efficiency of the parallel algorithm shows how well we utilise the extra resources. Normally, we are not able to obtain a speedup equal to p when using p processors due to δ, the communication overhead per processor. The efficiency measures the fraction of time a processing unit is usefully employed [26].

6.1 Stream processing in the cloud

Before we analyse the results of our simulations, we will establish the theoretical foundations for the analysis of the scalability of our MapReduce solution. This should give us an indication of how well our implementation should perform and where to expect bottlenecks in performance. We are mostly interested into the amount of work necessary to keep the MapReduce nodes busy.

6.1.1 The Put/Call parity problem

In our analysis, we will refer to the complexity of the sequential put/call parity algorithm. The exact operation of this algorithm is described below. The meaning of the financial vocabulary is explained in the background chapter.

The algorithm

The query used over our data set finds pairs of exchanges with the same strike price and expiry data for put and call options. All data refers to a single stock. The data is stored as a plain text file of tuples. Each tuple contains 16 comma-separated values of which only five are of interest to us.

1. Strike price
2. Expiration day
3. Expiration year
4. Exchange identifier
5. Expiration month code[1]

The first step is to re-order the input tuples to obtain the `(key, value)` pairs, where

```
key = (StrikePrice, Expiration Day, Expiration Year)
value = (ExchangeIdentifier, ExpirationMonthCode)
```

The list `[(key, value)]` of key value pairs is then grouped by key to give the the `(key, [value])` tuples for the reduce function. The reduce function operates on one of these tuples.

Thus, we will ignore the key and focus on finding pairs of exchanges with corresponding put and call options. For this we start with the first value in the list and compare it to all other values. If the value was a put option, we are looking for any matching call options and vice versa. Once the first entry has been compared to the remainder of the list, we continue with the next entry. Duplicates are ignored. The complexity of this operation is of the order $\Theta(n^2)$. This operation is repeated for every key. The paired exchanges are attached to the keys and returned as a list of `(key, [parity])` pairs, where

```
parity = [((ExchangeIdentifier_1, ExpirationMonthCode_1),
           (ExchangeIdentifier_2, ExpirationMonthCode_2))]
```

The output of the map function is concatenated to give the final output of the algorithm.

[1] also tells us if option is put or call

Sequential complexity

The time complexity of the above algorithm is $\Theta(n^2)$. This can be found by breaking the algorithm down into three phases:

1. **Map phase** The algorithm iterates over the input window and reorders the individual key/value pairs. Each of these permutations takes a fixed number of steps m. The upper and lower bound on the run time of the map phase is therefore $\Theta(n)$.

2. **Sort phase** The sorting algorithm is an adapted version of CPython's build-in sort functionality. The only extension is a customised comparison function which takes a fixed amount of time. The run time of the sort phase is therefore bound by the CPython implementation[2] - $\Theta(nlog(n))$.

3. **Reduce phase** The grouping by key takes n steps, as the whole array has to be traversed. However, the run time is dominated by looping over the groups and finding the parities. There is a maximum of n groups, namely if no key occurs more than once. In this case, the reduction takes n steps. The other extreme case is if there is only a single key. This time, the computation of the parities takes two nested loops and therefore n^2 steps. The upper bound on the complexity of the reduce phase is $\Theta(n^2)$. However, looking at the format of the key, this case is unlikely. We expect to see options with different strike prices in our data set, for example. Taking this into account, the complexity can be revised. Assuming that we have m unique keys. With n tuples, this leaves an average of n/m tuples per group. Traversing all groups and computing the parities gives us a complexity of $m(n/m)^2 \Rightarrow \Theta(n^2/m)$.

From the description of the phases we have $\Theta(n) + \Theta(nlog(n)) + \Theta(n^2/m)$ which leads to an asymptotic run time of $\max(\Theta(nlog(n)), \Theta(n^2/m))$. The asymptotic run time is $\Theta(nlog(n))$ only if $m > n/log(n)$.

6.1.2 Theoretical analysis of the parallel algorithm

With the above sequential algorithm already being available in MapReduce form, it is easy to find a parallel equivalent.

Map phase From the definition of the sequential algorithm, we know that it the map phase takes n/p steps on p processors. Assuming that the whole input window is only available at a single front-end node, we will need to transfer n/p tuples to each mapper. After this step, we need to sort the data prior to the reduce phase.

Instead of transmitting the output of the map phase to a single node and sort it there, we will use the fact that the data is already distributed over the mappers to do a distributed sort. For this we sort the n/p elements locally, using the build-in sort functionality. This takes $\Theta((n/p)\log(n/p))$ steps.

[2] http://wiki.python.org/moin/TimeComplexity

Single node merge After the local sort, the data is transferred to a single node for merging. Note, that we could further minimise the number of steps by joining input splits in parallel, but we opted for the centralised version in our Python implementation for simplicity. The merging is done in $p-1$ rounds and takes a maximum number of $\sum_{k=1}^{p-1} \frac{n}{k}$ steps. $\sum_{k=1}^{p-1} \frac{1}{k}$ is the sum is a harmonic number and equates to roughly $\ln(p)$. The complexity of the merging process is therefore $n \ln(p)$.

Reduce phase Finally, after grouping the sorted data (n steps like in the sequential algorithm), we partition the groups over the available nodes for the reduce phase. The scalability of this operation is highly dependent on the input data. In the worst case, we have a single group, induced by a single key. In this case, the parallel run time is n^2 as in the sequential case. For the average case, we assume again, that the number of groups is greater or equal to the number of processing nodes and that groups are roughly the same size. This requirement is not strictly necessary, as we can balance the load based on the length of the groups. Assuming that the load balancing prior to the reduce phase gives us a perfect balance leads to $\Theta(n^2/mp)$ steps for our reduce implementation.

Without evaluating the communication overhead T_o yet, we have the following expression for the parallel runtime:

$$T_p = T_o + n/p + (n/p)\log(n/p) + n\ln(p) + n^2/mp$$

We now define the cost of our parallel algorithm to be:

$$C_p = pT_p$$

In order for our algorithm to be cost-optimal, we need the cost to show the same asymptotic growth in terms of the input as the sequential version. Substituting our parallel run time gives an expression for the cost:

$$C_p = pT_o + n + n\log(n) - n\log(p) + pn\ln(p) + n^2/m$$

Since $n \gg p$ we have $n^2/p > n\log(n) > pn\ln(p)$ and thus the asymptotically largest term of this equation is $\max(pT_o, n^2/m)$. Note that if the difference between p and n becomes too small, the algorithm will not be cost optimal anymore. We will return to this problem when we evaluate the run time of our MapReduce stream processors. As n^2/m is equal to the cost of the sequential algorithm, we must evaluate the T_o term now. Normally, we would assume that we are operating over a normal TCP store-and-forward network. The communication overhead is thus given by $T_o = t_s + (mt_w + t_h)l$. For simplicity let us however, adopt the simple cost model as given on page 59 in *Introduction to Parallel programming* [26], which gives the communication cost as:

$$T_{comm} = t_s + t_w m$$

In this formula, t_w denotes the time to transmit a word, m is the number of words and t_s is the start up time. As this expression does not take the connectedness of the network into account, it can be applied to any network infrastructure. We chose this expression since the network topology of a cloud cluster is not always known to the client. Our data contains on average 80 characters per tuple in ASCII encoding. Each tuple contains 20 words. For a window size of 10,000 tuples, this makes 200,000 words or 781.25 kilobytes. In a random sample of 10,000

Operation	From	To	Time
Scatter	JobTracker	TaskTrackers/Mappers	$t_s + t_w \times 10,000 \times 20$
Gather	TaskTrackers/Mappers	JobTracker	$t_s + t_w \times 10,000 \times 3$
Scatter	JobTracker	TaskTrackers/Reducers	$t_s + t_w \times 10,000 \times 3$
Gather	TaskTrackers/Reducers	JobTracker	$t_s + t_w \times 750$
Total			$t_s \times 4 + t_w \times 260,750$

Table 6.1: Summary of communication times for our data set/MapReduce implementation on an idealised, fully connected network

tuples, we have 30 distinct keys and with an average of 6 associated distinct quotes. The map phase does not add or remove tuples but it does reduce their size to 3 words by ignoring the most of the fields. We will assume that the total number of tuples is only reduced during the *reduce* phase. It is difficult to give an exact number to the number of tuples (and their lengths) after this phase. Our experiments have showed a lot of variation. However, a reasonable average is around 3000 characters or 750 words. In any case the final communication step after the reduce phase is unlikely to have much impact as its run time is several orders of magnitude lower than the previous scatter and gather operations. The communication costs for each individual step under the simplified cost model mentioned above is shown in Table 6.1.

We will assume that any machine can only send and receive data sequentially. This means that regardless of the number of processors, a scatter operation over p processors is only completed when n elements have been transmitted. Assuming the start up time t_s is negligible, the total time taken for communication in our algorithm is 260,750 times the time to transmit a single word. If the computation time ($\Theta(n^2/m)$ as shown previously) on a single node is not significantly higher than this value, the algorithm is not cost efficient and not scalable.

Commentary We have made a few assumptions in this section. In reality, the communication overhead is likely to be greater than in the simple communication model as the cloud infrastructure is not fully connected. Furthermore, we might not be able to achieve perfect load-balancing over the reduce nodes. Our analysis of the communication overhead gives us a good indication about possible weaknesses of the MapReduce implementation when dealing with streaming data.

6.1.3 Hadoop vs Python MapReduce

Our very first experiment is to show that MAPS outperforms the Hadoop framework on streaming tasks. Thereby, we aim to establish which one of the two stream processors shall be used during the rest of this evaluation. In order to measure the response time of both the Hadoop and Python implementation, we have written a simple Python script (StreamTester) which reads a single predefined window from disk and relays it over the network to our stream processors. In addition to sending the window, the script measures the time it takes from sending the first tuple to the reception of the *STOP* signal from the stream processor. We will record multiple runs with the same window data and choose the one with the lowest value for our evaluation. The lowest value is chosen as it reflects the actual time needed for our computation. As our input data and algorithm do not change during the experiment, any inconsistencies in the run time must be attributed to external factors such as network congestion or other processes requesting resources on our stream processors. The window data will be read from disk at the local node and send over the network to the dedicated machines in the college data centre.

	Local Desktop (Development machine)	Cloud machines
Processor	Intel Core i7 920 @ 2.67GHz	AMD Opteron 2346 HE @ 1.81GHz
RAM	3GB	4GB
Network	Down 17Mb/s, Up 1Mb/s [3]	100Mbit Fast Ethernet
OS	OpenSuse 11.2	Ubuntu Jaunty (9.04)
Kernel	2.6.31.12-0.2-desktop	2.6.28-17-server

Table 6.2: Specifications of our test machines

Window size	Time in seconds
100	5.387
1000	6.225
10000	9.241

Table 6.3: Delays for varying window sizes processed by our streaming version of Hadoop, excluding start up overhead. Measurements varied a lot. Figures given are minimum values of 10 independent measurements over an **Internet** connection.

	Window size		
Tasktrackers	100	1000	10000
1	0.175s	0.753s	6.310s
2	0.151s	0.665s	6.170s
3	0.200s	0.626s	5.951s
4	0.220s	0.610s	5.931s

Table 6.4: Delays for processing several different window sizes on MAPS with varying number of TaskTrackers. Figures given are minimum values of 5 independent measurements over an **Internet** connection.

Setup

For our tests, we have constructed a *small* cloud of four Quad-Core AMD Opteron machines (see Table 6.2). These machines run at 1.8Ghz and have access to 4GB RAM. We will first test the latency implied by sending the query over a standard Internet connection.

> **Henceforth, the term *cloud* shall refer to our cluster of machines in the college data centre.**

Results

Table 6.3 shows the performance of our extended Hadoop framework. Each measurement was repeated several times with no significant variation. For this test, we have omitted the time it takes for the job to be loaded on the TaskTracker. Nevertheless, the custom version (Table 6.4) is still up to 30 times faster for 100 tuples and 32% faster for 10,000 tuples. Moreover, when we take the start up into account, all three queries run at an average of 11 seconds on the Hadoop implementation. This is due to the fact that the stream server can receive tuples whilst setting up the MapReduce job. Notwithstanding these measurements, it is impossible to quantify the run time of our Hadoop streaming queries as they fluctuate heavily. We have observed the same query on the same data to deviate by as much as 10 seconds! The reasons for these problems are described in §3.3.3. The unpredictability of these fluctuations, the drawbacks in the Hadoop design and the considerably worse performance of its implementation has led us to adopt MAPS for the rest of this evaluation.

Communication overhead

In §6.1.3, we have shown how important the communication overhead is for determining the scalability of our cloud implementation. In attempt to minimise the latency between the

	Window size		
Tasktrackers	100	1000	10000
1	0.142s	0.179s	1.602s
2	0.118s	0.148s	1.401s
3	0.150s	0.183s	1.366s
4	0.166s	0.195s	1.337s

Table 6.5: Delays for processing several different window sizes with varying number of TaskTrackers. Figures given are minimum values of 5 independent measurements over a **LAN** connection.

StreamTester and the cloud, we will from now on send the query from within the same network in which the cloud nodes are located. To evaluate the effects of this change, we will discuss the implications of the round-trip-time (RTT) and show how the locality of the StreamTester influences our results.

Round-trip time We have measured the round-trip time using the standard UNIX ping utility. As some routers may drop ICMP packets or delay them, a ping does not always give an accurate reflection of the RTT. However, it can give us a good indication of where the overhead in our query comes from. Our experiments have given us an average round trip time of 15ms. This means that any packet send from the local node to the stream server on the same network is only acknowledged 15ms later. Sending an 8KB burst of data over a 100Mbps link gives us a bandwidth delay product [38] of 1.5×10^6 bits. 8KB only utilise around 4.4 percent of our channel's capacity. This efficiency is too low for exploiting the network infrastructure. As can be seen in Table 6.5, the round trip time is sufficiently high for having a real impact on the latency of our response. Increasing the number of tuples by two orders of magnitude, does indeed help as the results show.

Incidentally, the average round-trip time for a ping request from our local node over the Internet to the nodes in the college data centre is 30ms. The difference to the round-trip time for the local area network as discussed above is similar to the difference between the respective measurements for 100 tuples in both cases.

Bandwidth The migration of the StreamTester to the same network as the cloud also shows the impact of the channel bandwith on the response time. Since we have established that our distributed MapReduce algorithm is only cost-optimal and sufficiently scalable if $n \gg p$ in , we cannot work on windows smaller than 10,000 tuples. A comparison of the response time over the Internet and the local area network yields the fact that at least 75% of the time is spend on communication when receiving the input window over the Internet. This severely cripples any chances of the cloud infrastructure to shine with scalability. The MapReduce implementation can only be successful if we have a high bandwidth, low latency connection between the cloud and the provider of the stream.

Scalability

Assuming that we have the network architecture as described in the previous section, we will now evaluate the scalability of our MapReduce framework. Table 6.6 shows the speedups obtained in from adding more TaskTrackers. For a window size smaller than 10,000 tuples, using more than three nodes to compute the result causes a drop in performance. This is due to the communication overhead mentioned above and the further addition of inter-process communication.

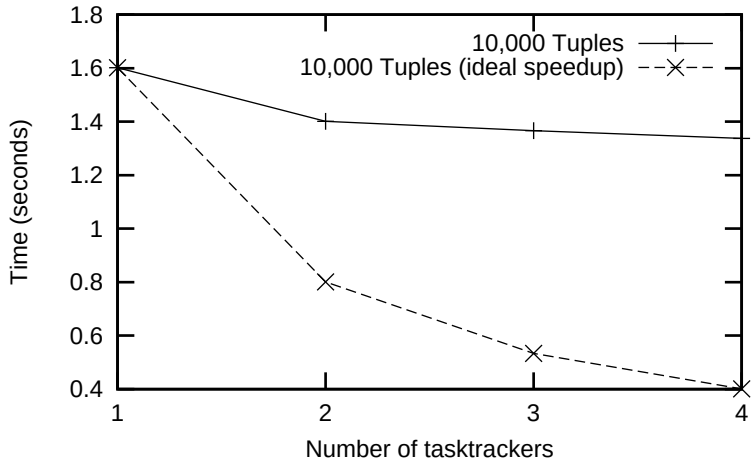


Figure 6.1: Diminishing returns when adding more TaskTrackers at a constant window size

	Window size		
Tasktrackers	100	1000	10000
1	1.000	1.000	1.000
2	1.203	1.209	1.143
3	0.947	0.978	1.173
4	0.855	0.918	1.198

Table 6.6: Speedups obtained by our distributed MapReduce implementation

	Window size		
Tasktrackers	100	1000	10000
1	100%	100%	100%
2	60%	60%	57%
3	-	-	39%
4	-	-	30%

Table 6.7: Efficiency of our distributed MapReduce implementation

In these cases the $n \log n$ and $np \ln p$ terms become more dominant. In addition to the constant overhead such as the communication start up times, these eclipse the actual computation time and cause the response time to increase.

Figure 6.1 shows diminishing returns when we add more TaskTrackers but keep the window size constant. An increase in the number of TaskTrackers causes the efficiency of the parallel implementation to drop from 57% to 30% (Table 6.7).

Resource utilisation

We have monitored the resource utilisation using the `top` program. The following data is for the second configuration (Table 6.5). The input stream is send and received on the same network. When running two TaskTrackers, we have observed that the machine running the JobTracker and a TaskTracker is loaded between 25% and 50%. This is expected as we have not enabled the threading of map and reduce tasks. Only 2 cores are utilised. The machine running a single TaskTracker only experiences between 10% and 25% load. This is due to the fact that much of the time is spend doing I/O to send and receive the input window. When we increase the number of TaskTrackers, their hosts' CPU utilisation drops below 10%. This confirms our earlier observations that the communication cost becomes a bottleneck.

	Window size		
Tasktrackers	100	1000	10000
1	1.42ms	0.179ms	0.160ms
2	1.18ms	0.148ms	0.140ms
3	1.50ms	0.183ms	0.137ms
4	1.66ms	0.195ms	0.134ms

Table 6.8: Tuple latencies for processing several different window sizes with varying number of TaskTrackers (**LAN**)

6.1.4 Conclusion

Our simulations have proved the suspicion that Hadoop's overhead is too great for streaming applications. However, we have showed that it is possible to scale MapReduce tasks over multiple machines using our custom MAPS implementation. More available TaskTrackers lead to a decrease in tuple latency (see Table 6.8). For an input window of 10,000 tuples, we could decrease the tuple latency from 0.160ms to 0.134ms. This is a good value as the average latency required for our data set is 1.29ms (6.2.2).

The calculations at the beginning of this chapter have showed that there is likely to be a bottleneck due to the communication cost. Careful consideration has to be given to the number of compute nodes in relation to the window size. The experiments have confirmed this. Using more than 2 nodes to compute an input window consisting of 10,000 tuples yields no considerable benefits. For input windows smaller than 10,000 tuples, MapReduce is not feasible. The communication and coordination overhead outweighs the benefits from parallelisation.

Increasing the window size to 100,000 tuples causes problems of its own. 100,000 tuples are approximately 8MB. A transfer over a slow Internet connection would take too long to complete. If the data is available on the same network, a larger input window might be an option. However, this leads us to a big limitation of the MapReduce paradigm. Although we could pipeline the incoming data straight into the MapTasks, we need to wait until the completion of the map phase to be able to start the reduction. In our case and we assume that this is not an exception, the map function is simple. Essentially, we are waiting for the network transfer to complete. This limits our capabilities to efficiently overlay communication and computation.

In summary, the MapReduce approach is used best for large window sizes and computationally intensive queries. The minimum requirement is for the computation complexity of the reduce query to exceed the communication complexity. The second requirement is for the map function to be complex enough to be efficiently pipelined with the incoming data. If these requirements are not fulfilled, traditional stream processing systems are better suited to the task. Future work should investigate how the MapReduce paradigm can be used to enhance existing stream processors in these special cases.

6.2 Loadbalancing into the cloud

In this section we will evaluate how the two load balancing schemes introduced in Chapter 5 perform in assisting the local stream processor through the added capacity of the cloud.

Our main goal is to find out how effective the load balancer is in smoothing out peaks in the input stream. First, taking the network latency into account, we will find the optimal split between the local stream processor and the cloud load balancer. This will be done using the *always-on* approach discussed in §5.2.1. This split constitutes an upper bound on the extra work which can be done in the combined stream processor. We will then measure and compare the performance of the *local-only* stream processor to the *adaptive* load balancer which uses the cloud's resources to assist the local node when windows are being dropped.

6.2.1 Experimental setup

Figure 6.2 shows the setup of our experiment. The specifications of the nodes are given in Table 6.2. The test stream is injected into the load balancer on the same node using the StreamTester component. The load balancer connects to the local stream processor and to the stream processor in the cloud. The output is send back to the StreamTester component. The StreamTester is a combination of Python and shell scripts. It monitors the resource utilisation of the local stream processor and the load balancer using the `top` utility. The output from `top` is read every second. It is then piped to a python script which outputs an average value for every 10 minutes worth of tuples[4].

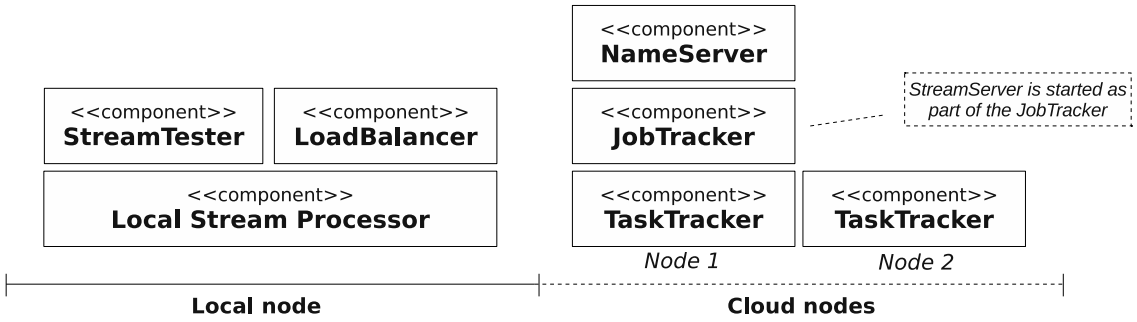


Figure 6.2: Our experimental setup. The local node is located at a different site to the cloud nodes.

As discussed in the previous chapter, the load balancer keeps a log about incoming and outgoing data as well as dropped and processed windows. It records state changes and the latencies of the local and cloud stream processors. This log file is post-processed using standard UNIX utilities like `grep` and `awk`.

6.2.2 Analysis of the data set

Our dataset contains quotes for options on the Apple stock for a single day from 8am to 4pm. The total size of the data set is 1.6GB. This is equivalent to 22,372,560 tuples. Each tuple is timestamped. If we take the average tuple inter-arrival time as a measure, our stream processor would have to process 777 tuples per second. This gives an average tuple latency constraint

[4]ie. independent of the playback speed of the component injecting the stream, we always have the same number of values for CPU and memory utilisation.

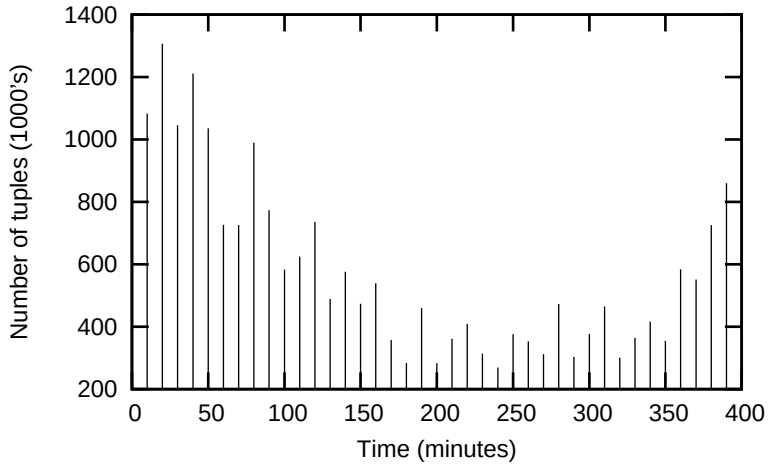


Figure 6.3: Arrival rate (Sampled every 10 minutes)

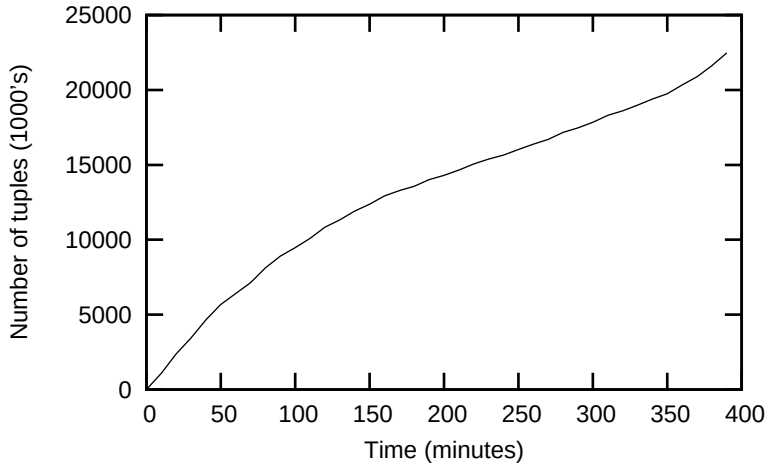


Figure 6.4: Cumulative arrival rate (Sampled every 10 minutes)

of 1.29 milliseconds per tuple. The average is not characteristic of the actual distribution of inter-arrival times. Figure 6.3 shows how the inter-arrival times are distributed over the data set. The trading during the morning hours is more intensive. The reason for this could be trades carried from the previous day; more likely, however, is an impact of the daily news. The inter-arrival time drops constantly until it reaches a trough around midday. This is emphasised by the gradient in Figure 6.4. The cumulative arrival rate shows almost a horizontal gradient after 200 minutes. The afternoon trading is relatively moderate compared to the morning. Only during the last 45 minutes does the trading pick up a bit. This has implications for our load-balancer. It is likely that it must immediately become active to deal with the surge in morning traffic. As the day passes, we expect the local stream processor to take over and to deal with the stream on its own.

6.2.3 Input parameters

The behaviour of our adaptive load balancing system is defined by four parameters. *Input queue* and *cloud on/off* are variables of the load balancer. *Stream synchronisation rate* and *playback speed* are parameters of the StreamTester component.

Maximum size of input queue Specifies the size of the input queue at the load balancer. If the queue has been filled up, we start dropping tuples and the cloud starts processing. If it has been drained, processing switches back to local-only.

Cloud on/off Switches the cloud stream processor on and off.

Stream synchronisation rate Specifies at which intervals the elapsed CPU time is compared to the current position in the stream. A higher value will lead to a burstier stream.

Playback speed Allows to adjust the rate at which the original stream is played back at. Honours the timestamps of the tuples.

6.2.4 Measured properties

Below, we list the properties we measured for different combinations of the above variables. All measurements were taken at the local node.

Tuples dropped/Percentage dropped The percentage of dropped tuples with respect to the total number of tuples gives us an indication of the efficiency of the load balancer.

Window latency The combined stream processor should result in more tuples being processed. We want to see how tuple/window latency is affected.

CPU/Memory requirements The cloud's CPU utilisation has been discussed in the previous section. For the load balancer, we are interested in the CPU and memory utilisation at the local node.

Average size of input queue The average size of the input queue. This value should get close to the maximum size if the local node is overloaded.

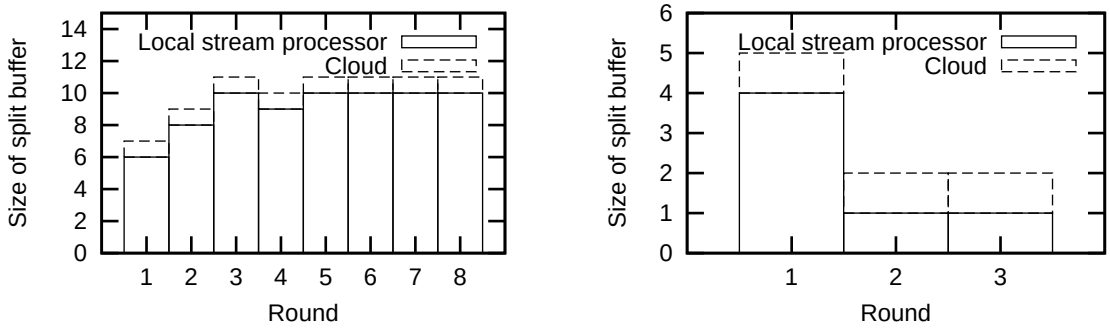
Average size of split buffer The average size of the split buffer.

6.2.5 Always-on: Finding the best split

In this experiment, we wish to evaluate the maximum number of windows which can be processed when the load balancer is operated in *always-on* mode. The input window is fixed to 10,000 tuples and replayed, in order for the computational load to remain the same.

We expect the load balancer to quickly find a reasonable split. When interpreting the results, we must take into account that it is running on the same machine as the local processor. The cloud implementation will be slower than the local stream processor due to the communication overhead for the transmission of the input window. We would therefore like to find out to which extend the additional resources can be utilised when the cloud is part of a combined stream processing system.

Figure 6.5(a) shows how the most efficient split is found after just 3-5 iterations. MAPS can process a single window in roughly 6 seconds. The local node needs 0.6 seconds. This means that for every 10 windows processed on the local node, 1 window is processed in the cloud. The cloud helps the local node to process 9% more windows. This split can be somewhat improved by using a higher bandwidth connection. As shown in §6.1.3, the cloud is able to process the same window in 1.4 seconds. The new split would be 3-7. For every 7 windows processed on the local node, we can process 3 windows in the cloud. The split could be further improved by



(a) Local node connecting to the cloud via an **internet** connection

(b) Local node connecting to the cloud via a **LAN** connection

Figure 6.5: Histograms depicting the load distribution between the local stream processor and the cloud stream processor

using more powerful machines in the cloud. At the moment the result is skewed towards the local node as its specifications are slightly better than those of the cluster nodes.

For Figure 6.5(b), we have moved the whole experiment to the cloud. The stream tester, load balancer and local stream processor run on a dedicated node in the college data centre. The cloud is left unchanged. The high bandwidth, low latency connection between the nodes on the same network guarantees a 1-1 split. For the rest of the evaluation, we shall be using the first setup (i.e. 10-1 input split).

6.2.6 Adaptive load balancing

We will now explore the performance of our adaptive load balancer. Figure 6.6 shows how enabling the cloud stream processor helps the load balancer to reduce the number of dropped windows by around 8% over the local-only solution. The benefits of the cloud's capacity become more clear when the synchronisation rate is increased. The increased synchronisation rate eliminates bursts which were not present in the original stream. When the StreamTester component only synchronises with the stream every 10,000 elements, it is likely to very quickly fill the load balancer's input buffer. Furthermore, it often waits for quite a substantial time after a burst and before it sends the next data. By synchronising more often, we can eliminate the effect of both the bursts and the idle periods. In this section we will discuss the effect of the 4 input parameters on the performance of the adaptive load balancer. We will start by conducting a few preliminary experiments to fix suitable values for the *input queue* and *synchronisation rate*. Building on those values, we will show how the cloud's resources can help to improve on the performance of the local stream processor.

Preliminary experiments

We consider our main variable to be the *playback speed*. Therefore we wish to start by measuring the effect of the *input queue* and the *synchronisation rate*. The following measurements were taken with the full adaptive load balancer. The playback speed was set to 13 times the original speed. This reduces the original runtime of 6.5 hours to a run time of approximately 30 minutes. At this speed, the load balancer will drop tuples.

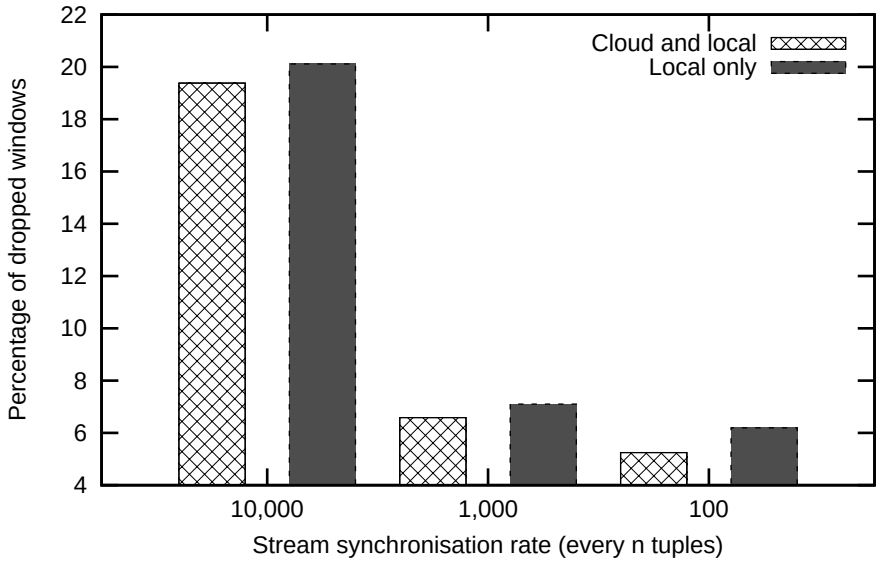
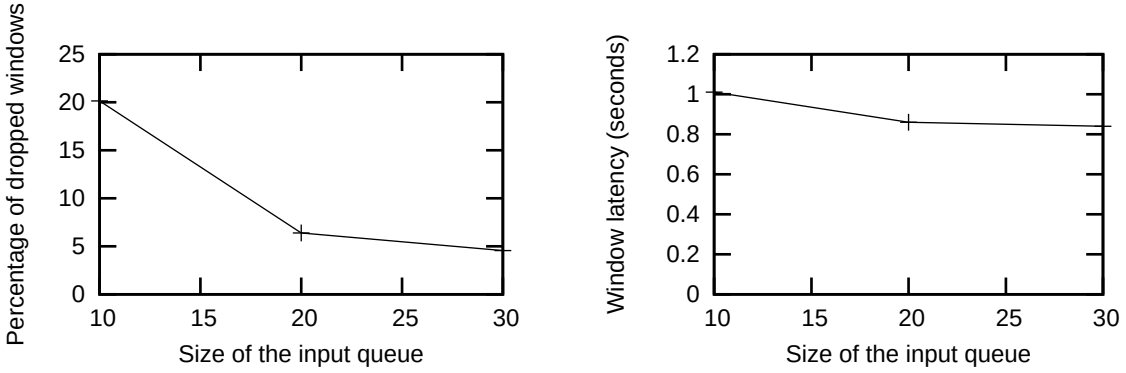


Figure 6.6: Chart depicting the effect of enabling the cloud stream processor for different synchronisation rates (Playback speed: 13x, Synchronisation rate: every 1,000 tuples)



(a) Graph depicting the effect of the queue size on the percentage of tuples dropped

(b) Graph depicting the effect of the queue size on the average window latency

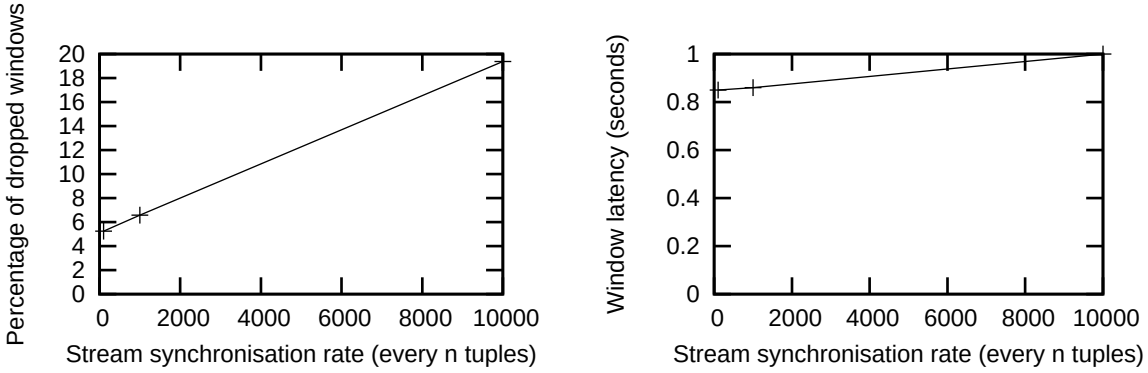
Figure 6.7: Graphs showing the effect of the queue size (Playback speed: 13x, Synchronisation rate: every 1,000 tuples)

Maximum queue size As can be seen in Figure 6.7(a), the percentage of dropped tuples falls sharply when increasing the size of the input queue from 10 to 20. Increasing the maximum length of the queue from 20 to 30 does not give us much added value. From Table 6.9 we can see why the effect of increasing the maximum size of the input queue beyond 20 is small. The average size of the input queue is around 10 elements. Thus in the average case, 20 elements are more than enough. In order to adequately serve the peaks in the load, the queue must be substantially bigger. We have chosen to fix the maximum queue size to 20 for the rest of our experiments.

Figure 6.7(b) shows the effect on average latency is very small. This is because of the massive communication overhead incurred from our low-bandwith link. In order for the cloud's added resources to have a more positive effect on window latency, we need a more even split between windows processed locally and windows processed on the cloud. Since the average window latency does not give us any meaningful information in the current setting, we will forgo its calculation and focus on the percentage of dropped windows after these preliminary experiments. We can, however, already conclude that any window which is not dropped has a minimum latency of around 0.6 seconds (with empty input queue) and a maximum latency of $size(inputqueue) \times \frac{6}{11}$

Max. size of queue	Avg. size of input queue	Avg. size of split buffer
10	9.36	5.21
20	8.15	10.81
30	10.77	10.69

Table 6.9: Table depicting the effect of the input queue (Playback speed: 13x, Synchronisation rate: every 1,000 tuples)



(a) Graph depicting the effect of the synchronisation rate on the percentage of tuples dropped

(b) Graph depicting the effect of the synchronisation rate on the average window latency

Figure 6.8: Graphs showing the effect of the synchronisation rate (Playback speed: 13x, Size of input queue: 20)

seconds (we can process 11 elements every 6 seconds).

Stream synchronisation rate In this experiment we are trying to find a suitable value for the synchronisation rate. The goal is to eliminate the bursts and waiting periods not inherent in the original stream. Synchronising the stream at a very fine granularity (ie. every tuple) is not feasible as it would introduce a lot of unnecessary computation. We have run three experiments at different synchronisation rates - every 10,000, 1,000 and 100 tuples. Increasing the synchronisation rate beyond 10,000 does not make sense as our window size is set to 10,000 tuples.

Figure 6.8(a) shows the impact of the change in the synchronisation rate. Synchronising every 1,000 tuples gives us a drop rate of just over 6%. Increasing the synchronisation rate to every 100 tuples gives a drop rate of 5%. In order to minimise the overhead from synchronisation we have chosen to synchronise every 1,000 tuples in the next experiments. Again, like with the queue size above, Figure 6.8(b) shows that the latency stays between 0.8 seconds and 1 second per window. We will consider latency no further after this.

Varying the playback speed

4x In the previous two experiments we have established sensible values for the *maximum queue size* and the *synchronisation rate*. In the upcoming experiments these shall be set to 20 and 1,000, respectively. In the final part of this evaluation, we will vary the inter-arrival time of the original data using our *speed* variable in order to measure its effect in terms of dropped windows and CPU utilisation at the local node. Playing the available data at up to four times its original speed does not put any strain on the adaptive load balancer. No windows are dropped and the cloud's resources are never requested. Figure 6.9 shows the CPU usage over the whole length of the stream. The spikes in CPU usage can be explained with the spikes in the inter-arrival

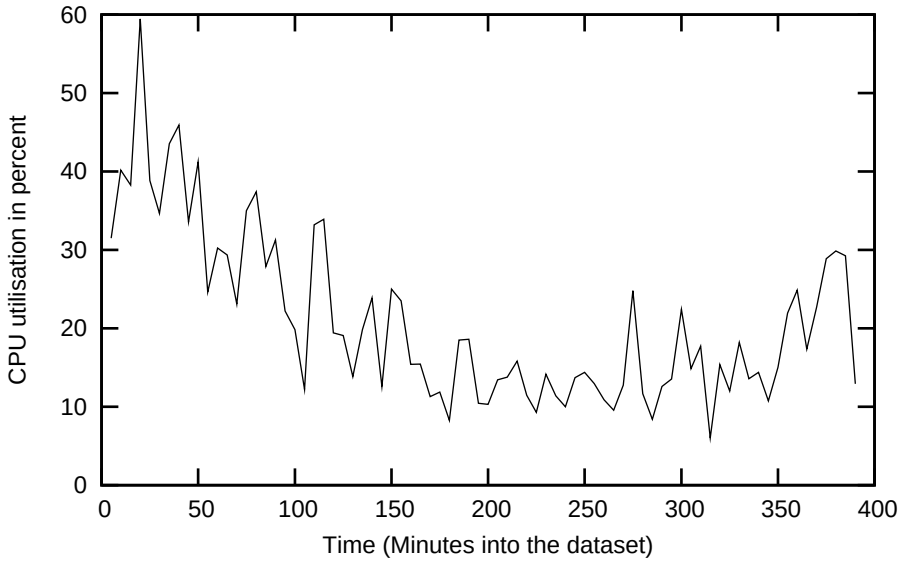


Figure 6.9: Graph showing the CPU utilisation at the local node when playing the stream at 4 times its original speed

times shown in Figure 6.3. The average CPU utilisation never exceeds 60% which explains why no windows were dropped.

16x When the stream is played at 16 times its original speed, 12% of the incoming windows are dropped when using the adaptive load balancer and 14% when using only the local stream processor. Figure 6.10 shows the CPU utilisation for the first 100 minutes into the stream. As shown in Figure 6.3 and discussed previously, the tuple inter-arrival rate is highest in the morning, so this part of the graph shows best how well our load balancer works.

The downward spikes occur due to the load balancer switching states from *local-only* to *local-and-cloud*. The input queue is drained and processing is shared with the cloud. The CPU utilisation goes up again when the right split has been found and the local stream processor is fully occupied.

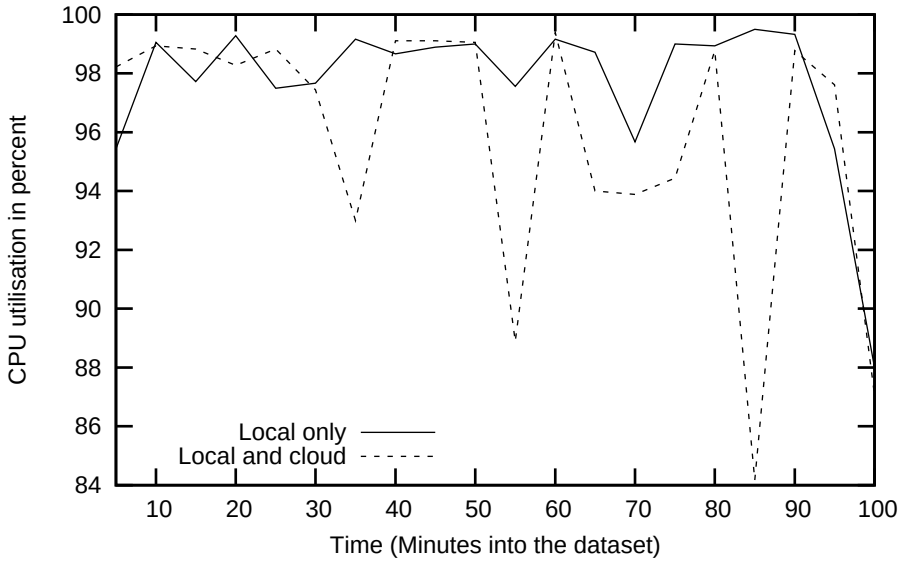


Figure 6.10: Graph showing the CPU utilisation at the local node when playing the stream at 16 times its original speed

Increased network bandwidth

For our final experiment, we have moved both stream processors and the load balancer to the cluster nodes. This move is to simulate a high bandwidth, low latency connection between the load balancer and the cloud. As discussed in §6.2.5, the best split between the local stream processor and our MAPS framework for a window size of 10,000 tuples is 1-1. First we tested the load balancer with only the local stream processor enabled. When we played the input stream at 16x the original speed, the load balancer dropped 53% of the windows. A run with the MAPS stream processor enabled, halved this value to 25%. Figure 6.11 shows the CPU utilisation at all three participating nodes. The node running the local stream processor (LP) and the load balancer (LB) is running at full capacity almost throughout the simulation. To avoid dropping windows during the morning rush hour, it needs to continuously use the cloud nodes. Like in the previous experiments, we have used two nodes in the cloud. The first is running the JobTracker and a single TaskTracker. The second node is running a single TaskTracker only. From the graph, we can see how the first node is fully utilised when the local node is overloaded. The downward spikes occur as the input queue runs empty and the load balancer switches to local only processing.

As a side note, Figure 6.11 also re-iterates the points made in §6.1.3 about the resource utilisation of the MAPS framework. The node running the single TaskTracker is currently only utilising around 10% of its capacity. Clearly, the JobTracker introduces an additional parallel overhead which impairs the scalability of the framework.

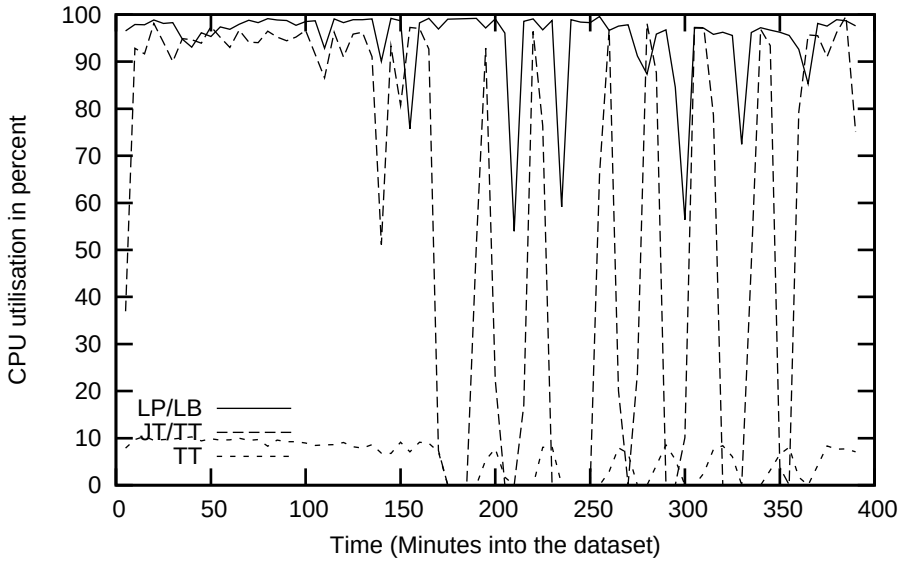


Figure 6.11: Graph showing the CPU utilisation at all three participating nodes when playing the stream at 16 times its original speed. JT = JobTracker, TT = TaskTracker, LP = Local Processor and LB = LoadBalancer denote the processes running on the three nodes.

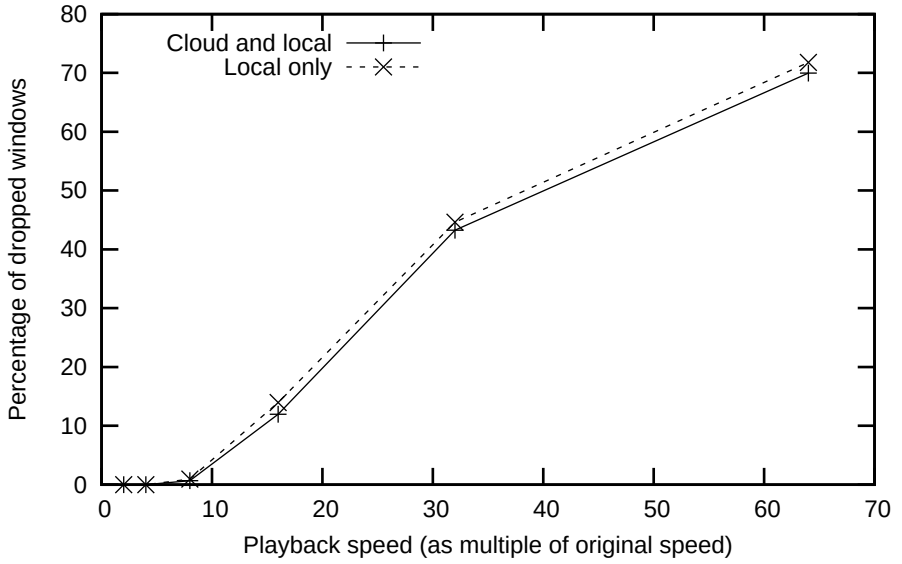


Figure 6.12: Graph depicting the relationship between dropped windows and playback speed

6.2.7 Conclusion

Our experiments have shown that the load balancer is able to utilise nearly the full potential of the MAPS framework. The split between the local processor and the cloud processor directly reflects the different latencies. Figure 6.12 compares the percentage of dropped tuples for playback speeds ranging from 2x to 64x on our first setup using a low bandwidth internet connection. It shows that whatever the speed at which we play the input, the adaptive load balancer drops less windows when the processing on the cloud is enabled. The small difference is solely due to the bandwidth constraints of our Internet connection. In §6.2.5 we have showed that a higher bandwidth will result in a better split between the local and the cloud stream processor. §6.2.6 showed that this has a direct impact on the number of tuples dropped by the load balancer.

The memory usage in all our experiments was very modest. When the local node is overloaded, the input queue occupies roughly 30 megabytes of memory (all 20 slots are filled). However, as Table 6.9 shows, the average size of the queue is smaller. 10 windows constitute roughly 15 megabytes of data. Likewise, the size of the split buffer will be around 10 as discussed previously. It accounts for another 15 megabytes. These values were confirmed in our measurements with an average memory utilisation of 11% for the load balancer process[5].

In order to fully utilise the cloud's resources we will still have to optimise our MAPS framework or use a different stream processor. Further research into the average complexity of streaming queries and window sizes has to be conducted if MAPS were to be chosen. Only if the query is complex and the window sufficiently large, MAPS is able to scale in the cloud.

[5]Local node has 3 gigabytes of main memory

Chapter 7

Conclusion

In this report have used a cloud-based stream processor to investigate load balancing strategies to efficiently handle bursts in the input stream. We claim that utilising the cloud to handle peak load situations can significantly reduce the costs of stream processing.

Previous stream processing systems have already discussed the parallelisation of queries [13] [29]. Mortar uses an overlay network and distributes the operators over a tree [29]. This helps to deal with failures but does not make a distinction between normal and peak load. Cayuga's publish subscribe system allows queries to be pipelined [13]. Pipelining is dependent on the query and not very scalable. The row/column scaling approach also mentioned in Logothetis and Yocum [13] uses available resources in a round-robin fashion. This scales very well when the queries are stateless. However, when queries are stateful, the input stream has to be split and routed to the correct node. In this report we have chosen to adopt the MapReduce programming model for our cloud stream processor to investigate an alternative approach which scales the query over a number of machines based on the size of the input.

7.1 Project review

In the first part of this report we have shown how a MapReduce stream processing system can be implemented on a homogeneous cluster. In the second part we have shown how a local stream processor can utilise the cloud's resources to deal with bursts in the input stream.

7.1.1 Contributions

Streaming extension for the Hadoop framework In §3 we have discussed the implementation of the extensions necessary to receive input from a stream in the Hadoop framework. We have measured the performance of this solution and explained why its overheads currently prohibit efficient stream processing.

MAPS: A Lightweight MapReduce framework Building on the experience from extending the Hadoop framework, we have designed and implemented MAPS, a lightweight MapReduce framework written in Python (§4). We have showed how the elimination of the distributed file system significantly reduces the parallel overhead and leads to a more stable run time. In our evaluation of the system we have showed how the scalability of a query is closely linked to the

complexity of its map and reduce functions as well as the size of the input window. These observations confirm previous work on distributed stream processing as operator placement (Mortar) and pipelining (Cayuga) both depend on the query being split up for parallelisation.

Loadbalancing strategies In §5 we have discussed the design and implementation of two related load balancing solutions which make use of cloud resources to relieve a local stream processor.

By utilising our MAPS prototype on a cluster infrastructure, we have showed how our *always-on* loadbalancer can increase the total number of windows processed by combining the cloud and local stream processors. In the *adaptive* setting, our load balancer is able to utilise cloud resouces on-demand to reduce the number of windows dropped.

In §6.2.5 we have showed how the efficiency of the MAPS cloud processor varies with the bandwidth of the connection between the local node and the cloud. We have showed that the bandwidth between the local node and the cloud is a more decisive factor for the number of windows dropped than the latency of the connection.

7.2 Future work

7.2.1 Improving the efficiency of the MapReduce framework

In Chapter 4 we have motivated our choice for Python as implementation language. As our MapReduce framework was merely a prototype, this choice was acceptable. In order to fully utilise the potential of the cloud's resources, we would develop a more efficient version in Java or C. Subject to the technology made available by the chosen cloud provider, it would be sensible to use an efficient message-passing framework like MPI in order to minimise the communication cost. This is especially important as computation time on the cloud has monetary implications.

7.2.2 Pipelined MapReduce jobs

Cayuga's pipelining approach can be easily applied to our MapReduce framework by allowing for multiple, chained MapReduce jobs. Chaining multiple MapReduce jobs does not only potentially help with scaling the query; it also allows us to write more expressive queries. However, careful attention must be paid to the complexity of the individual queries in order to make sure it is not dropping below a threshold where communication becomes a bottleneck.

7.2.3 MAPS scaling

The best number of TaskTrackers to use for a certain window size should be found automatically. Currently, the MAPS framework uses as many TaskTrackers as are registered to the name server. A future version of this framework must take the cost of computing data on the cloud as well as the scalability of the query into account and decide an optimum value for the number of nodes participating in the MapReduce process. MAPS supports threaded reduce tasks. Due to time constraints, we have left their evaluation for future work. The MapReduce framework should ideally not only scale over nodes but utilise all available processing power on each one of them in an effort to reduce inter-node communication.

7.2.4 Adapting Cayuga to scale on a cloud infrastructure

The evaluation has showed that MapReduce does not efficiently scale when small windows and computationally less intensive queries are used. In order to make the best use of the cloud's capacity, we suggest to adapt the distributed Cayuga implementation for a cloud infrastructure. Row/column scaling can be used to balance less complex queries. For more complex queries and larger window sizes it may be considered to compile the CQL queries to MapReduce jobs. Fusing Cayuga and MapReduce technology could possibly help to overcome the performance bottlenecks arising from handling stateful queries.

7.2.5 Eliminating communication bottlenecks

Splitting the input stream Concerning the load balancer, further research into splitting the stream closer to the source or subscribing both the local node and the cloud to the same input stream should be conducted. If we can find a way to forgo the copying of data between the local node and the cloud, we can significantly reduce the latency. An interesting aspect of this will be the control flow between the local node and the cloud to coordinate which windows should be processed on the two stream processors.

Query-aware load balancing The communication between the local node and the cloud could be reduced by a query-aware loadbalancer which only send the elements of the tuples to the cloud which are necessary for the computation of the MapReduce query. Essentially, the load-balancer operates as a first instance map function to prepare the data for the cloud. In our case the data is reduced from 20 to 3 words per tuple after the initial map function. If this had been done by the load balancer, we could have significantly reduced the overhead from the communication.

Window compression In addition, we could have further reduced the communication overhead by compressing the input window. An input window may contain duplicates. More importantly, however, we might be able to take advantage of the fact that most of the data is numeric and rather short. Sending a 2 digit number over the network using 2 chars is not efficient.

7.2.6 Parallel programming with language support - F#

Our MAPS implementation has showed that it is possible to write a distributed MapReduce framework in a functional language. Microsoft's F# programming language has parallel constructs already built in. Further work could look directly at distributing functional programs in MapReduce style over a set of nodes. Tying in with the previous point, this could mean that the low level communication is handled by fast native code while at the same time retaining the benefits of an expressive functional programming language for designing the query.

7.2.7 Cost-benefit analysis

It is very important to conduct a proper cost-benefit analysis before any computation is moved to the cloud. As mentioned in the introduction, it makes sense to use a cloud provider for stream processing tasks when the servers are only needed a maximum of 6 hours a day and are not used on weekends. In addition, when trying to achieve low-latency responses to market conditions,

the location of the cloud nodes becomes important. EC2 nodes in North America will not be suited to place trades at the HKEx in Hong Kong. However, if nodes in close proximity to the stock exchanges are available, the cloud based solution becomes very attractive. While building a new data centre in the vincinity of the market is infeasible, a cloud-based solution can be implemented at a very small cost. Further research into the latency properties of existing cloud solutions has to be conducted to quantify these benefits.

Bibliography

[1] Hadoop FAQ. URL http://wiki.apache.org/hadoop/FAQ. Accessed on 27/01/10.

[2] Hadoop MapReduce tutorial. URL http://hadoop.apache.org/common/docs/current/mapred_tutorial.html. Accessed on 27/01/10.

[3] ZooKeeper. URL http://hadoop.apache.org/zookeeper. Accessed on 27/01/10.

[4] More chip cores can mean slower supercomputing, Sandia simulation shows, January 12, 2009. URL http://www.physorg.com/news151158992.html. Accessed on 01/06/10.

[5] SalesForce (Cloud Provider). URL http://www.salesforce.com. Accessed on 05/06/10.

[6] Joyent (Cloud Provider). URL http://www.joyent.com/. Accessed on 05/06/10.

[7] RightScale Cloud Management Platform. URL http://www.rightscale.com. Accessed on 05/06/10.

[8] STAN - an eclipse based structural analysis tool. URL http://www.stan4j.com. Accessed on 15/03/10.

[9] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R Motwani, U. Srivastava, and J. Widom. Stream: The stanford data stream management system. Technical Report 2004-20, Stanford InfoLab, 2004. URL http://ilpubs.stanford.edu:8090/641/.

[10] Matt Asay. Q&A: Visa dips a toe into the Hadoop pool. *Cnet.com - The Open Road*, September 17, 2009. URL http://news.cnet.com/openroad/?keyword=Hadoop.

[11] Jeff Barr. Amazon EC2 Beta, August 2006. URL http://aws.typepad.com/aws/2006/08/amazon_ec2_beta.html. Accessed on 13/01/10.

[12] Chris Bemis. Put-Call Parity, 2006. URL http://www.math.umn.edu/finmath/seminar/bemisslidesB.pdf. Accessed on 20/12/09.

[13] Lars Brenna, Johannes Gehrke, Mingsheng Hong, and Dag Johansen. Distributed event stream processing with non-deterministic finite automata. In *DEBS '09: Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, pages 1–12, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-665-6. doi: http://doi.acm.org/10.1145/1619258.1619263.

[14] H. Bryhni, E. Klovning, and O. Kure. A comparison of load balancing techniques for scalable web servers. *IEEE Network*, 14(4):58–64, 2000.

[15] Valeria Cardellini, Emiliano Casalicchio, Michele Colajanni, and Philip S. Yu. The state of the art in locally distributed web-server systems. *ACM Comput. Surv.*, 34(2):263–311, 2002. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/508352.508355.

[16] Constantin Christmann, Juergen Falkner, Dietmar Kopperger, and Anette Weisbecker. Schein oder Sein: Kosten und Nutzen von Cloud Computing. *iX Special 2/2010 - Cloud, Grid, Virtualisierung*, pages 6–9, February 2010.

[17] Ariel Cohen, Sampath Rangarajan, and Hamilton Slye. On the performance of tcp splicing for url-aware redirection. In *USITS'99: Proceedings of the 2nd conference on USENIX Symposium on Internet Technologies and Systems*, pages 11–11, Berkeley, CA, USA, 1999. USENIX Association.

[18] T. Condie et al. MapReduceOnline. Technical Report UCB/EECS-2009-136, UCB, 2009.

[19] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/1327452.1327492.

[20] Alan Demers, Johannes Gehrke, Mingsheng Hong, Mirek Riedewald, and Walker White. Towards expressive publish/subscribe systems. In *Advances in Database Technology - EDBT 2006*, volume 3896 of *Lecture Notes in Computer Science*, pages 627–644. Springer Berlin / Heidelberg, 2006. ISBN 978-3-540-32960-2. doi: 10.1007/11687238_38. URL `http://www.springerlink.com/content/y684305339173080/`.

[21] Mara Der Hovanesian. Cracking The Street's New Math. *BusinessWeek*, April 18, 2005. URL `http://www.businessweek.com/magazine/content/05_16/b3929113_mz020.htm`.

[22] David DeWitt and Michael Stonebraker. MapReduce: A major step backwards. URL `http://databasecolumn.vertica.com/database-innovation/mapreduce-a-major-step-backwards/`. Accessed on 14/12/09.

[23] Domenico Ferrari and Songnian Zhou. An empirical investigation of load indices for load balancing applications. In *Performance '87: Proceedings of the 12th IFIP WG 7.3 International Symposium on Computer Performance Modelling, Measurement and Evaluation*, pages 515–528, Amsterdam, The Netherlands, The Netherlands, 1988. North-Holland Publishing Co. ISBN 0-444-70347-0.

[24] Ian Foster, Carl Kesselman, and Steven Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *Int. J. High Perform. Comput. Appl.*, 15(3):200–222, 2001. ISSN 1094-3420. doi: http://dx.doi.org/10.1177/109434200101500302.

[25] Charles Hayden. Announcing the Map/Reduce Toolkit. *New York Times - Open Blog*, May 11, 2009. URL `http://open.blogs.nytimes.com/tag/hadoop`.

[26] Vipin Kumar. *Introduction to Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. ISBN 0201648652.

[27] Marcel Kunze and Christian Baun. Wolkige Zeiten, Cloud Computing: Hype vor der Konsolidierung. *iX Special 1/2010 - Programmieren heute*, pages 111–116, January 2010.

[28] S. Loebman, D. Nunley, Y.C. Kwon, B. Howe, M. Balazinska, and J.P. Gardner. Analyzing Massive Astrophysical Datasets: Can Pig/Hadoop or a Relational DBMS Help?

[29] Dionysios Logothetis and Kenneth Yocum. Wide-scale data stream management. In *ATC'08: USENIX 2008 Annual Technical Conference on Annual Technical Conference*, pages 405–418, Berkeley, CA, USA, 2008. USENIX Association.

[30] Sun Microsystems. RPC: Remote Procedure Call Protocol specification: Version 2. RFC 1057 (Informational), June 1988. URL `http://www.ietf.org/rfc/rfc1057.txt`.

[31] Nasdaq. Market Share Statistics. URL http://www.nasdaqtrader.com/trader.aspx?id=marketshare. Accessed on 10/01/10.

[32] Vivek S. Pai, Mohit Aron, Gaurov Banga, Michael Svendsen, Peter Druschel, Willy Zwaenepoel, and Erich Nahum. Locality-aware request distribution in cluster-based network servers. *SIGPLAN Not.*, 33(11):205–216, 1998. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/291006.291048.

[33] Aaron Pan. Algo-trading is a strategic focus, says Nomura. *Asiamoney*, 21, April 2010.

[34] Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. Interpreting the data: Parallel analysis with sawzall. *Sci. Program.*, 13(4):277–298, 2005. ISSN 1058-9244.

[35] Justin Smith. Facebook Using Hadoop for Large Scale Internal Analytics. *Inside Facebook*, June 6, 2008. URL http://www.insidefacebook.com/2008/06/06/facebook-using-hadoop-for-large-scale-internal-analytics.

[36] Michael Stonebraker, Daniel Abadi, David J. DeWitt, Sam Madden, Erik Paulson, Andrew Pavlo, and Alexander Rasin. Mapreduce and parallel dbmss: friends or foes? *Commun. ACM*, 53(1):64–71, 2010. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/1629175.1629197.

[37] Sun Microsystems. Remote Method Invocation (RMI). URL http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp. Accessed on 17/05/10.

[38] Andrew S. Tanenbaum. *Computer networks: 2nd edition*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988. ISBN 0-13-162959-X.

[39] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms (2nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006. ISBN 0132392275.

[40] Heather Timmons. A London Hedge Fund That Opts for Engineers, Not M.B.A.'s. *New York Times*, August 18, 2006. URL http://www.nytimes.com/2006/08/18/business/worldbusiness/18man.html?_r=1&ex=1313553600.