Master's Thesis

Distributed Conversion of Public Transport Schedules

Claudio Botta <bottac@ee.ethz.ch>

Supervisors:
ETH:
Benjamin Sigg
Prof. Dr. R. Wattenhofer

Netcetera:
Patrick Bönzli

Contents

Abstract

Back in 2004 Google has published an article 'MapReduce: Simplified Data Processing on Large Clusters'. MapReduce is intended for processing and generating large data sets on a large cluster of commodity machines. Shortly after the MapReduce article appeared, a basic implementation of MapReduce was included in Apache Nutch. Soon that MapReduce implementation matured and was eventually taken out of Nutch into its own project, called Apache Hadoop.

While most software engineers do not typically process terabytes of data, they may still face the problem, that occasionally a batch job does not terminate as fast as desired or needs more memory than a single commodity machine can possibly provide.An example of such a border case is the Transit Feed Converter, a batch processing pipeline used and developed at Netcetera for converting public transport schedules. While the Transit Feed Converter was originally designed for processing smaller amounts of schedule data, the size of the schedule data to process by the Transit Feed Converter has grown relatively fast. In addition, new output formats, which require more complex computations than the original ones, have been added to the Transit Feed Converter.

Having in mind that the data to process by the Transit Feed Converter will still grow and that it is rather likely that further demanding computations will soon be added to the conversion pipeline, we were curious about a scalable solution. Therefore, a prototype port of the Transit Feed Converter from the single-threaded Python implementation to Hadoop/EC2 has been developed.

Acknowledgements

I would like to thank my advisors, Patrick Bönzli and Benjamin Sigg for their great support. Further, I would like to thank Corsin Decurtins (Netcetera) and Jason Brazile (Netcetera), for providing valuable input.

Chapter 1

Introduction

1.0.1 Motivation

Netcetera's Transit Feed Converter is a batch processing application, which has originally been developed to convert public transport schedules from the HAFAS raw data format [haf09] to a format optimized for mobile devices. The HAFAS raw data format is the format, in which transit schedules are provided by most European public transport agencies, whereas the original output format of the Transit Feed Converter is used by Netctera's Wemlin. Wemlin is an iPhone application, which provides transit schedule information to its users. The modular design of the Transit Feed Converter, makes it possible to easily extend the batch processing application, such that it can be used to import or export new formats or to enable it to extract specified information from transit schedules.

Soon the Transit Feed Converter was extended by the ability to export data to the General Transit Feed Specification (GTFS) [Pfe], which is a format suggested and used by Google. With the new export format however, the data that was to process by the Transit Feed Converter has rapidly grown. Further more, the export to GTFS involves more demanding computations than the original exporter did. When it turned out that the amount of data that was to process by the Transit Feed Converter was likely to grow further, it became obvious, that the Transit Feed Converter was about to reach its limits. The main matter was that the processing time started to increase dramatically, such that it was conceivable that the Transit Feed Converter would soon be unable to deliver data within reasonable time. On the other hand, the in-memory data representation was imminent to cause a memory starvation.

In the case of the Transit Feed Converter, the extension became a problem that has lead to the need of a drastic refactoring in spite of the strict modular design. The example shows that in addition to a modular design, a scalable architecture is required in order to get an optimally reusable and versatile software product.

More concretely speaking, the individual building blocks of the software product should be able

to handle growing amounts of work in a graceful manner [Bon], given that the underlying (hardware) platform is able to provide the necessary resources. Such a (hardware) platform, that is able to provide the resources for scalable software is thus an extensible one and/or a dramatically overdimensioned one. Whereas traditionally acquiring such a scalable (hardware) platform is extremely costly and has therefore only been a option for larger businesses, a trend towards building clusters of commodity hardware has enabled small and medium sized businesses to maintain their own scalable platforms. While maintaining a cluster of commodity hardware is an option even for smaller sized businesses, obtaining commodity computing resources from a platform as a service provider, instead of acquiring the hardware, seems even more promising for the following reasons.

- Sporadically occurring load peaks can be handled, as (theoretically) any number of resources may be allocated temporarily.
- Resource planning (e.g. in a growing organization) becomes easier, as new computing resources can be allocated within minutes.

1.0.2 Task Description

This thesis investigates the distribution of an existing transit schedule converter with the goal of enabling the processing of larger data sets in reasonable time. The open-source MapReduce implementation Hadoop was proposed as a distribution framework. However, other frameworks and technologies are considered too.

Special attention needs to be paid to the choice of a suitable data structure and an appropriate storage technology.

The thesis includes the design of a prototype MapReduce application and its data structures to convert, analyze and enrich public transit data from HAFAS to GTFS. Further, the prototype implementation is also presented in the thesis.

This thesis also investigates, how the advantages of the platform as a service may be exploited in order to enable scalable batch processing, such that

- the investment in the development of software may be protected.
- batch processing tasks may be processed within reasonable time, if they would take too long otherwise.

Chapter 2

Problem Description

2.1 Problem Specification

The existing Transit Feed Converter, a single threaded Python application, processes the schedules not fast enough and consumes too much memory. The memory consumption of the Python application causes an additional increase of the processing time or may even prevent the application from terminating successfully. A distributed version should address both issues, while being cost efficient. Cost efficiency is enabled by:

- A solution that is able to be run on low cost **commodity computers**.
- Using a distribution framework that **simplifies** the **development** of a distributed version and reduces the development time.

Further attention has to be paid to the fact that not all parts of the Transit Feed Converter have a linear complexity.

2.2 Requirements for a Viable Distribution Framework

For the given project it is desirable, but not necessary to find a solution that will allow for a reasonable code sharing among the existing implementation and the new distributed version. The framework should allow to reach a reasonable speedup for smaller data sets. On the other hand, it should still be able to handle larger amounts of data, as the size of the input and output data is very different for the various modules (cf. 5.2.6). The extensibility of the existing architecture must be preserved. Maintaining the new implementation should be comparable in costs to the existing version. This includes the ease of deployment. Also should any extension be not much harder to write than before. Ideally the new solution should also be extensible by serially programmed plug-ins.

Further should a viable solution be based on a proven and preferably widespread technology, which will assure the availability of documentation and support. Also should a viable solution be as operating system and hardware independent as possible. The target platform are though commodity computers as they are provided by Amazon EC2 (cf. 6.1). Other pros that a viable framework could have are

- free availability
- source code availability
- teaching at local universities and universities of applied sciences

Preferably, a framework is used, that enforces the use of a programming model or of a design pattern, as this would improve the extensibility and the collaboration among developers of the batch processing application. An appropriate framework should also do the scheduling, handle node outages and provide redundancy.

Chapter 3

Background

3.0.1 Viable Technologies

Processing computationally intensive batch jobs is traditionally a domain of dedicated supercomputers. This section discusses technologies, which are considered to be helpful for developing a high performance batch processing environment based on commodity nodes, such as they are provided by EC2 (see 6.1). The discussion includes traditional and established building blocks of distributed systems, as well as some more recent environments, which are more specialized in distributing batch processes. In High Performance Computing (HPC), a number of architectures are distinguished. Commonly used architectures include Symmetric Multiprocessing (SMP), Massively Parallel Processing (MPP), Constellations and Clusters [DSS]. For the eligibility of a software framework, it matters more than anything, whether or not the architecture has a shared memory. Therefore frameworks which depend on a platform with a shared memory (SMP) are discussed separately from those targeted at systems without one, such as MPPs, Clusters and Constellations.

3.0.2 Shared Memory Distributed System

A distributed system with a shared memory is probably the most comfortable solution from a developer's point of view. Due to the shared memory, such a system presents itself to the programmer as a single system with a large number of CPUs, which it actually is. So all the developer has to care for, is to split up his batch job into enough parallel running processes or threads. Unless the job is embarrassingly parallel, this may not be trivial, though. Therefore, APIs which support parallelization or programming languages specialized in parallelization are important components of such high performance computers. Today, OpenMP is the de facto standard and many platforms come with an implementation of OpenMP. OpenMP is a very userfriendly parallelization framework, since the program written in OpenMP is developed very much like its serial version. Indeed, an existing serial program can often be relatively easily

parallelized using OpenMP [SGM+09]. A very important concept of OpenMP is the use of annotations, through which the developers describe, roughly speaking, where parallelization is possible and where synchronization is required. As an example, an annotated for-loop in C is shown below.

```
void square(int* a, int N) {
    #pragma omp parallel for
    for (i = 0; i < N; i++)
        a[i] = x*x;
}
```

Listing 3.1: OpenMP annotated for loop.

Obviously, this requires both, an OpenMP-capable compiler, as well as a runtime library in order to work correctly. Another proven approach to parallelization is the use of specialized programming languages, namely functional languages such as Haskell, Lisp, Erlang or Scala. The analogon to the above example in Scala [Ode09], a language developed at EPFL, could look as shown in the listing below.

```
def square(a: List[Int]): List[Int] = {
        a.map(i => i*i)
}
```

Listing 3.2: A square function in Scala.

In addition to the pleasant programming models, there exists even a working and free software package, DIPC [KSS], which is able to emulate a shared memory among linux2.2 nodes connected via TCP[1]. The sudden disappearing of SMP-Architectures from top500.org's list (Figure 3.1) of the fastest supercomputers in the early 2000s however, supposes that shared memory based systems are hard to scale.

SMPs and parallelisation technologies have though an important meaning for the following reasons.

- As the Transit Feed Converter has been fast enough for small data sets, it could be sufficient to parallelise the batch processing application, such that it could exploit a multi-core machine.
- The nodes of distributed Systems are often SMPs, such that parallelization technologies as discussed above, are often combined with distribution frameworks (e.g. MPI).

[1] Of course, only the availability of a shared memory among a few nodes, does not yet make the group of nodes an SMP system. In addition, means to launch processes or threads on other nodes, as well as means to choose nodes for the processing of threads or processes are needed.

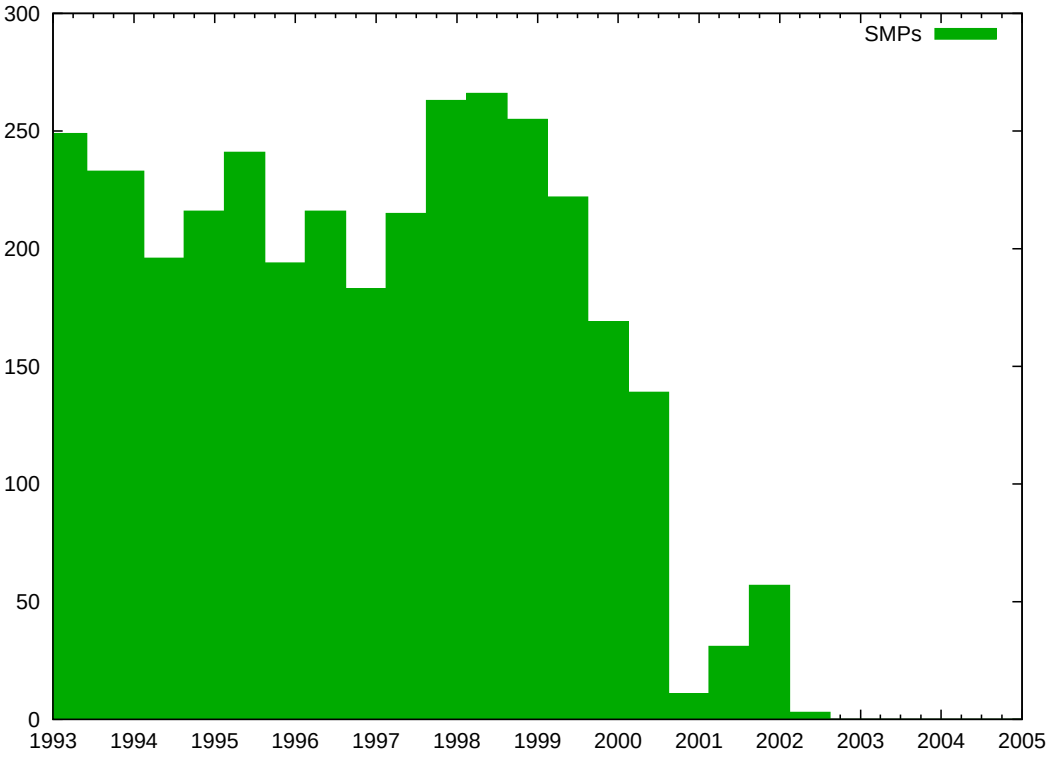


Figure 3.1: The number of SMP-supercomputers from 1993 to 2005 in the top 500 list from top500.org.

- Parallelization may even improve the performance of a program, when it is run on a single processor system, since a parallel program may continue to do computations, even when one part of the program is currently waiting for I/O.

3.0.3 Cluster

The programming models discussed in this section are possible candidates for distributed systems that do not (necessarily) dispose of a shared memory. These systems include Clusters, Constellations, Grids and MPPs. Of course, these programming models could also be used on SMPs.

The most commonly used framework in an HPC context is the Message Passing Interface (MPI). To give an impression of how a program typically exploits parallelism using a message passing infrastructure such as MPI, an equivalent to the square example, based on message passing,

is sketched below. The program will consist of one master and an arbitrary number of slaves. The master's task is to supply its slaves with workload by sending messages to the slaves. In the example the messages consist of a value from an input array and its index in the input array. The master starts with sending such a message to each of the slaves. Then the master waits for any slave to return a result. As soon as the master receives a result, it will insert the result into the output array and provide further work to the slave if any is available. As soon as all work has been submitted to the slaves, the master will just wait for the slaves to return their last result. In pseudo code, the master would thus look like listed below. A working implementation can be found in the appendix B.

```
master(){
        foreach slave{
                index,value = get_next_index_value_pair();
                send((index,value),slave)
        }
        while (work_available)
        {
                result,slave = receive_message_from_any_slave();
                index,value = get_next_index_value_pair();
                send((index,value),slave);
                output[result.index] = result.value;

        }
        foreach slave{
                result,slave = receive_message_from_any_slave();
                halt(slave);
                output[result.index] = result.value;
        }
}
```

Listing 3.3: A Master Process in Messageing.

The slave's purpose is to simply do the calculation. In the given example all slaves simply have to calculate the square of the value received. Finally, it will return the result, which contains the squared of the value received and the index as received. The slave looks thus as follows.

```
slave(){
        until (halted){
                work = receive_message_from_master();
                send((work.index,work.value*work.value),master);
        }
```

```
}
```

Listing 3.4: A Slave Process using Messageing.

In MPI it is up to the user to decide which number of which processes are run on which node. The above program however requires that exactly one master and at least one slave is run. Also it is up to the user to deploy the binaries for the processes to the nodes. However, there exist tools, which can assist the user doing this work. While MPPs offer proprietary tools, to users of a selfmade cluster, the software OSCAR [dLSNG03] might be valuable.

Distributed Erlang

Erlang is different from most programming languages, in the aspect, that it comes with a builtin distribution solution. The actor model on which Erlang is based, is perfectly suitable for distributing. Since actors, or rather processes, as they are called in Erlang, only interact via messaging with other parts of the program, the distributed program looks almost exactly like the parallel one and Erlang handles the transmission of the messages across a network invisibly.

In [SF07] the authors say, '...the step to convert an application running on a single node to a fully distributed (multi-node) application is deceptively simple ...'. However, there are still numerous pitfalls, which have to be taken into consideration. [SF07] [Wik94]

Most importantly for batch processing, for each piece of work, the node on which it should be processed, has to be chosen. For simple batch tasks, this results in an implementation similar to the master-worker model, which has been presented above. The code of an Erlang implementation of the previously used example, which can be used as a template for simple distributed batch processing programs, is listed in the appendix C. In the appendix C it is also shown, how an environment for running such a distributed application, can be set up in incredibly few steps. An advanced solution for scheduling (i.e. for choosing the node for a computation), for the monitoring of tasks or for the rescheduling of failed tasks however, is not provided by Erlang.

Also, Erlang is not the preferred language for writing the tasks which are to be executed on the worker nodes. If these tasks are small enough, it does not matter whether they are optimally parallelized, since several of these tasks can be run in parallel on one worker node, such that the worker node can be utilized. Thus, it is more important that the tasks can be developed easily, rather than that they are optimally parallelized. Therefore these tasks are preferably written in an imperative language, which is of course possible. An example of an Erlang based framework, which expects the user provided code to be written in an imperative language, is the MapReduce implementation disco [dis] by Nokia or RabbitMQ.

RPC

The idea of a service that allows to call a procedure on a remote host, has been around for quite a while, at least since 1975 [Whi75]. Meanwhile many frameworks, which implement such a service, have been developed. Some popular names are Sun RPC, DCOM, CORBA, XML-RPC or SOAP to mention just a few.

With an RPC-like technology, it is relatively easy to implement a master-slave model as discussed previously. Further, RPC-like technologies are interesting, as at least one such framework is available for almost every platform and every language. Though, today the most popular RPC-like technologies like SOAP, XML-RPC or CORBA play an important role rather for making systems interoperable than for batch processing.

MapReduce

MapReduce is a programming model, as well as a programming environment targeted at batch processing of large data sets on large clusters and was first presented in [DG04]. Typically, MapReduce is used to process data sized above 1TB on clusters with dozends, tens or even thousands of nodes.

The MapReduce programming model is inspired by functional programming languages such as LISP. LISP provides two important higher-order functions, one is **map**, the other one is **reduce**. Other functional programming languages have different names for the reduce function, for example fold, accumulate or compress. The purposes of the higher-order functions map and reduce, as they are known from functional languages are the following.

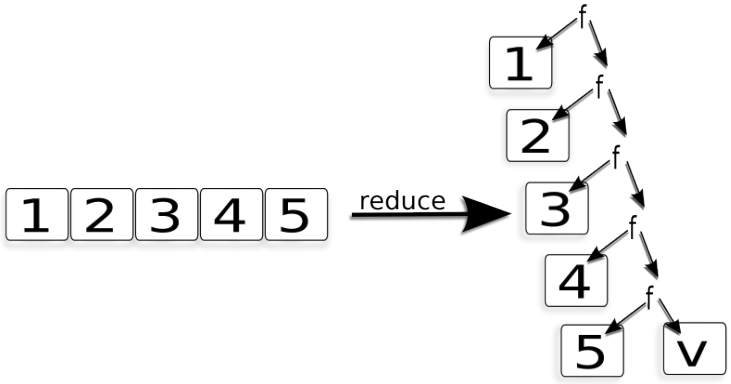


Figure 3.2: A reduce function applied on an array of number, with the initial value v and the combining function f.

- **map**: The map function accepts a **function** f, and a **sequence** s as input arguments. The map function returns a new sequence with the same length as the input sequence s, where the i-th element of the return sequence is the return value of f applied to the i-th

value of the input sequence.

- **reduce**: The reduce function accepts a **combining function** f, a **sequence** s , an optional **initial value** v and optionally a **range**. The range describes, which part of the sequence is to be processed, but for simplicity, the range is neglected below. The combining function, is a function which takes two arguments (a1 and a2) and 'combines' them to a return value. The reduce function will then be applied recursively on the values of the sequence, such that in the i-th recursion, a1 is the i-th value of the sequence and a2 is set to the combine-functions return value from the i+1-th recursion. The innermost combine-function combines the last value in the sequence with the initial value. This is shown in the figure 3.2. There exist also various variations of the reduce-function, which are not discussed here.

Following the denomination of LISP, the MapReduce model is based on two phases, the Map and the Reduce phase. Each job programmed in the MapReduce model is split into those two phases. The processing of the Map phase and of the Reduce phase have similarities to the map function and the reduce function, respectively.

- **Map**: The Map phase is a transformation step, in which individual records are processed independently by a user provided function. The user provided MapReduce Map function looks as listed below.

```
void map(keytype key,valuetype value){
        // custom code
}
```

Listing 3.5: A MapReduce Map function (Pseudo code).

The MapReduce Map function does not have a return value like the user-provided functions passed to the map functions of functional programming languages do. However, a MapReduce Map function can emit any number of key-value pairs.

```
emit(key,value);
```

Listing 3.6: Emitting a key-value pair in MapReduce (Pseudo code).

- **Reduce**: The Reduce phase is an aggregation step, in which values with a common key are aggregated by a user-provided function as listed below.

```
void reduce(keytype key, Iterator<valuetype> it){
        // custom code
}
```

Listing 3.7: A MapReduce Reduce function (Pseudo code).

There also exists an optional Combine phase, which is similar to the Reduce phase and is executed between the Map and the Reduce phase. The Combine phase is however not discussed here for the sake of simplicity.

Now, that the Map and the Reduce phase are introduced, the reminder of this section will discuss how the two phases are put together, where the user-provided functions get their input from and how the model can be exploited to intuitively develop distributed batch processing programs.

[DG04] also characterizes a distributed file system, called GFS, which by default is used to store the input and the output of MapReduce jobs. The idea is to use the MapReduce worker nodes also as the storage nodes of GFS, such that the MapReduce scheduler can try to run MapReduce jobs on nodes of the MapReduce cluster, which are close to the data and ideally already have the data locally. Further, the MapReduce framework saves communication, by assigning not single key-value pairs to the worker nodes, but rather entire batches of key value pair, called partitions. When the Map function spills out intermediate key-value pairs, they are also stored in GFS. Before the processing continues, the intermediate key-value pairs are though sorted by key, in order to assure that the Reducer will receive an iterator containing all values belonging to one key. There may be more than one Reducer, if desired, such that the reduce process is also parallelized. In this case, the MapReduce framework partitions the intermediate key-value pairs, such that all values belonging to a given key are guaranteed to end up in the same partition.

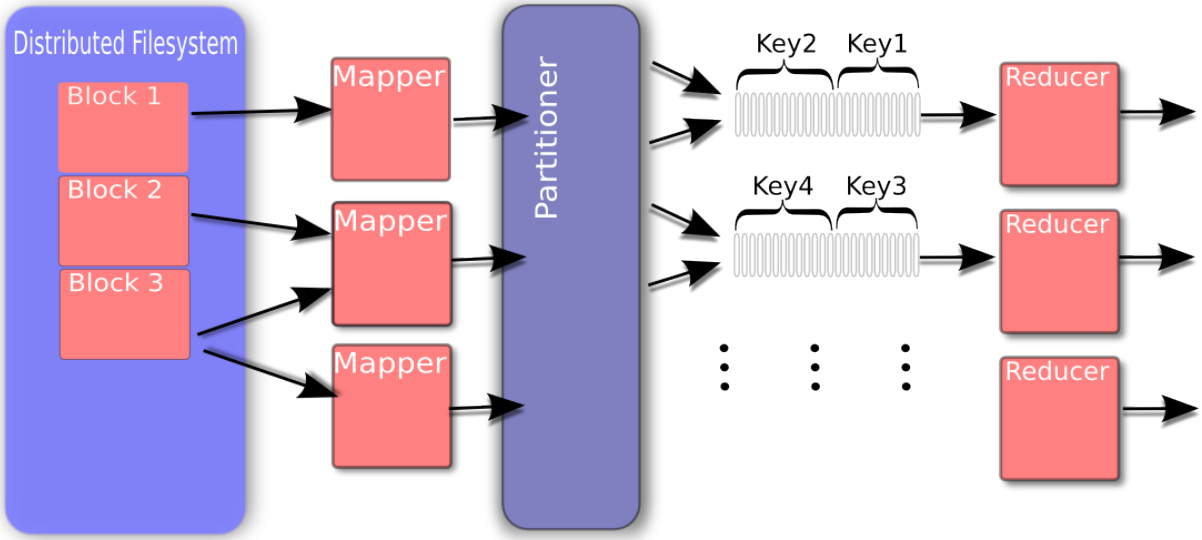


Figure 3.3: The MapReduce model

[DG04] provides many sample MapReduce programs. Programming MapReduce is quite intuitive and higher level abstraction frameworks like Pig [ORS+08] make it even more natural.

Dryad

Dryad is a distribution framework by Microsoft, described in [IBY+07]. It is considered to be Microsoft's response to Google's MapReduce. For instance, in 2006, The New York Times has cited Bill Gates, who was a Microsoft chairman at the time, saying: "They did MapReduce; we have this thing called Dryad that's better," [HM06]. Though, Dryad is definitely not a copy of MapReduce, but it stands for a totally different concept.

In order to run a job in a Dryad environment, a graph has to be supplied to the Dryad cluster. The nodes of this graph are executed on the nodes of the Dryad cluster, whereas the edges of the graph specify the communication between the nodes, i.e. how many items are sent from one node to another. Since this is usually not known initially, the graph can be modified at runtime. Dryad offers a sophisticated framework for manipulating graphs, for example for joining two graphs.

Like most distribution frameworks, e.g. MPI, Dryad allows communication from any node to any node, but the Dryad model forces the developer of a Dryad job to specify the communication and encourages him to do so as early as possible.

Though the article, in which the Dryad model was originally published [IBY+07], does not discuss the impact of that model on scheduling, it can be assumed, that this model may be useful for building an advanced scheduler, which would be able to minimize the traffic across Dryad nodes.

Therefore Dryad is certainly one of the most interesting projects in the area of distributed batch processing at the moment.

According to Microsoft Research, Dryad is already productively used internally at Microsoft. [IBY+07] says, that most users would use LinQ [NBD+03] for programming Dryad jobs.

3.0.4 Comparison

Twiddling with numerous frameworks and technologies during the first approximately 8 weeks of the project, has shown that for most batch processing problems, a simple master-worker model is absolutely sufficient. Most batch processing problems can be split up fine enough that no special scheduling is required. That is to say, for a large number of approximately equally small pieces of work it is a good scheduling strategy, to simply provide the worker nodes with one piece of work any time they complete one, even for a heterogeneous system. Never the less, it is desirable to have a framework which handles the scheduling of splits of work. For example, because a sophisticated framework would also reschedule splits of work that have failed. In accordance with the previously defined requirements, but also considering the lessons learned while evaluating frameworks the following feature-matrix has been elaborated. MapReduce convinced in its feature richness and did not fail in any essential criterion, such that the step

was dared to start the prototype implementation using the framework (Hadoop MapReduce), which has originally been proposed.

	Hadoop	Dryad	RPC	Erlang	Shared Memory (e.g. DIPC)
Free choice of the programming language			x		x
Specialized in batch processing	x				
Framework includes distributed storage	x				
Supports data-local processing	x				
Framework does scheduling	x	x			
Appropriate for high-latency Ethernet	x	x	x	x	
Free implementation available	x		x	x	
Runs on commodity nodes	x	x	x	x	x
Framework splits the input data	x				
Good documentation	x	x	x	x	x
Operating System independent			x	x	

Table 3.1: Feature matrix for the discussed distribution technologies.

Chapter 4

Hadoop

Hadoop is a very comprehensive, Open Source and Java based implementation of the MapReduce programming model as described under 3.0.3. Hadoop includes a distributed file-system HDFS, which is inspired by the Google File System, which is characterized in the same paper as MapReuce [DG04]. Further, Hadoop also includes a NoSQL database, which is designed to behave similar to BigTable as specified in [CDG+08] and depends on the Hadoop Distributed File System as its storage.

The most important characteristics of Hadoop, which have influenced the design of the prototype Distributed Transit Feed Converter, are discussed below.

4.1 Hadoop Common

Hadoop Common has formerly been known as Hadoop Core and contains all essential software needed to build a MapReduce cluster, as well as everything to develop, test and run MapReduce jobs.

4.1.1 JobTracker

The JobTracker is the server, to which the Hadoop jobs are submitted. The node, on which the JobTracker runs, is called the master node. The JobTracker is thus responsible for splitting the submitted job input into smaller parts called splits and for assigning the splits to the worker nodes or slave nodes. Each slave runs a TaskTracker instance. The function of the TaskTracker is discussed later in this chapter. Before a split is assigned to any slave node, the JobTracker sends the user-provided MapReduce program , with which the split is to be processed, to that slave node (cf . figure 4.1). On the slave nodes, the splits are first processed by the user submitted Map method. The split itself however is not sent to the worker node, but rather a

reference to the split. This mechanisms are discussed more detailed below. When the user submitted Map method emits key-value pairs, work for the Reduce method is created. The assignment of those key-value output pairs, also called intermediates, to a so called partition is done by the Partitioner on the slave nodes. The execution time, however is scheduled by the JobTracker.

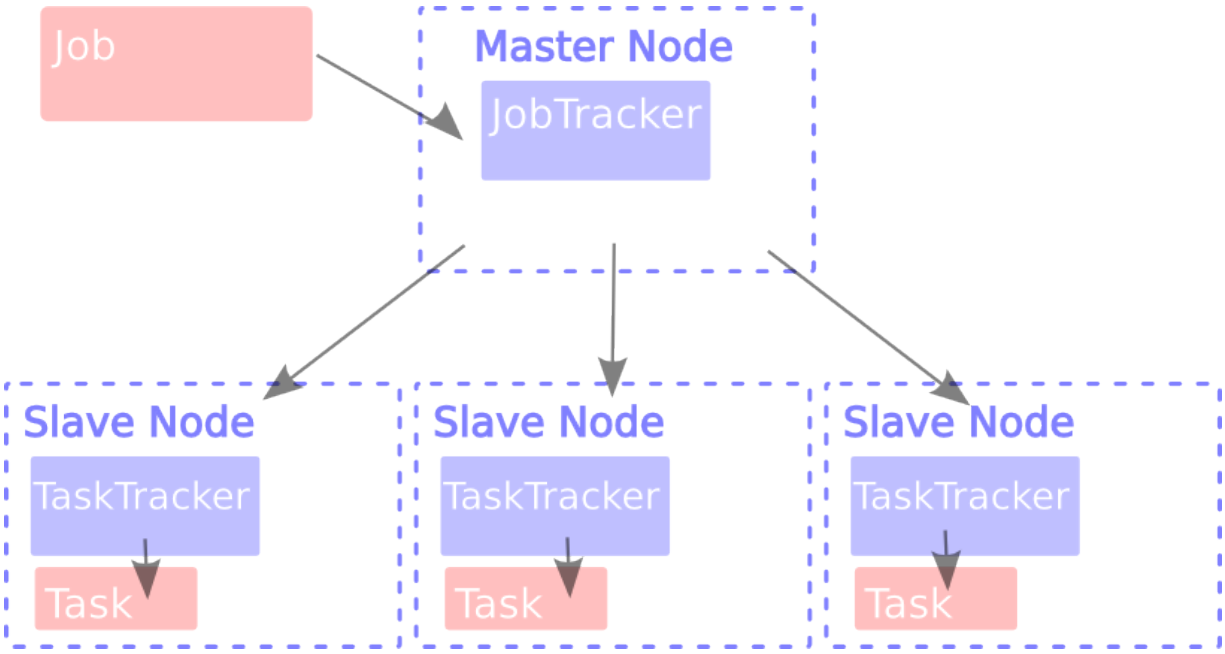


Figure 4.1: High level abstraction of the Hadoop environment.

4.1.2 InputFormat

Input data is specified by InputFormats in Hadoop. The InputFormat is responsible for splitting the input data into InputSplits. Each InputSplit contains a list of the nodes on which the data for this InputSplit is located and informs the JobTracker about the size of the InputSplit. This information is used by the JobTracker for scheduling, trying to run as many computations locally. The size of the splits, respectively the number of splits, into which an input source is divided is determined by the InputFormat, but is configurable for some InputFormats. The InputFormat is also responsible for providing the factory for creating the RecordReader instances on all slaves.

RecordReader

The RecordReader is run on each slave node, where a Map task is scheduled and is resposible for retreiving the actual key-value pairs from the input source. This is illustrated in the figure 4.2.

Most commonly, RecordReaders read from a file, typically stored in the Hadoop Distributed File System. A frequently used RecordReader reads a file from the HDFS line by line and spills out the line as a value and the byte-offset of that line within the file as a key. Since the part of the file, for which a given RecordReader instance is responsible, most likely does not end where a line ends, the convention is that the RecordReader reads on to the first newline symbol after the part it is actually responsible for. As a consequence, all RecordReader Instances, but the one responsible for the beginning of a file, only start parsing the file after the first newline symbol within the part of the file for which they are responsible, as shown in the figure 4.3.

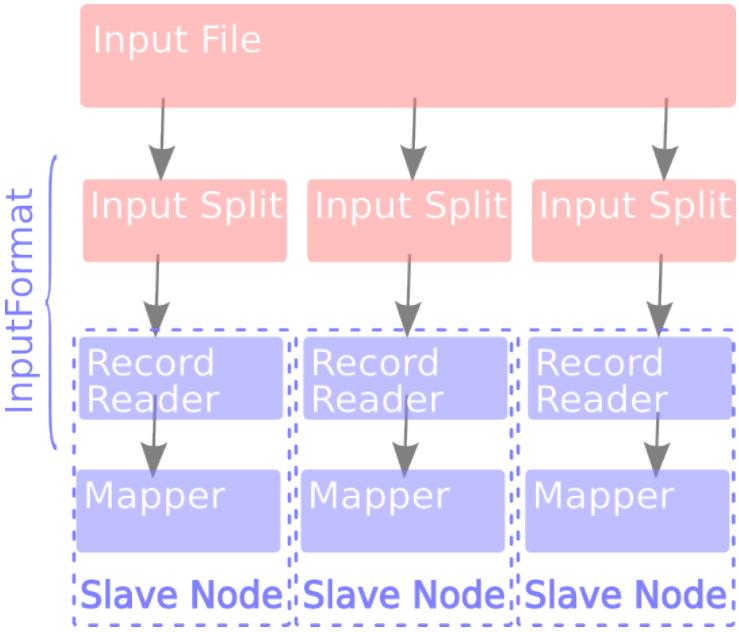


Figure 4.2: Reading input in Hadoop.

Partitioner

The main goal of the Partitioner is to keep the sizes of the partitions as balanced as possible, while assuring that all intermediates with the same key arrive in the same partition. With the knowledge about the expected output of a MapReduce job, a different partitioning strategy than the default one might better reach this goal. Therefore users may provide custom Partitioners.

By default, a Partitioner called HashPartitioner is used, which simply uses a HashCode of the key modulus the number of Partitions to create [Ven09]. The number of reduce tasks is retrieved from the configuration.

Partitions are stored in HDFS, such that the JobTracker can try to execute the reduce jobs locally. When a reduce task is scheduled, the user provided reduce-method is called once for each key in the partition, which belongs to that reduce-task. Thereby the reduce-method also receives a iterator, which iterates through all values of the key, for which the method has been called. It is possible to schedule a reduce-job, before the Map phase has finished. In this case it is unknown, whether the Partition is already complete or not. Therefore, the value-iterators will simply block, after the last available value of a key has been retrieved, until either new values arrive or the Map phase has terminated.

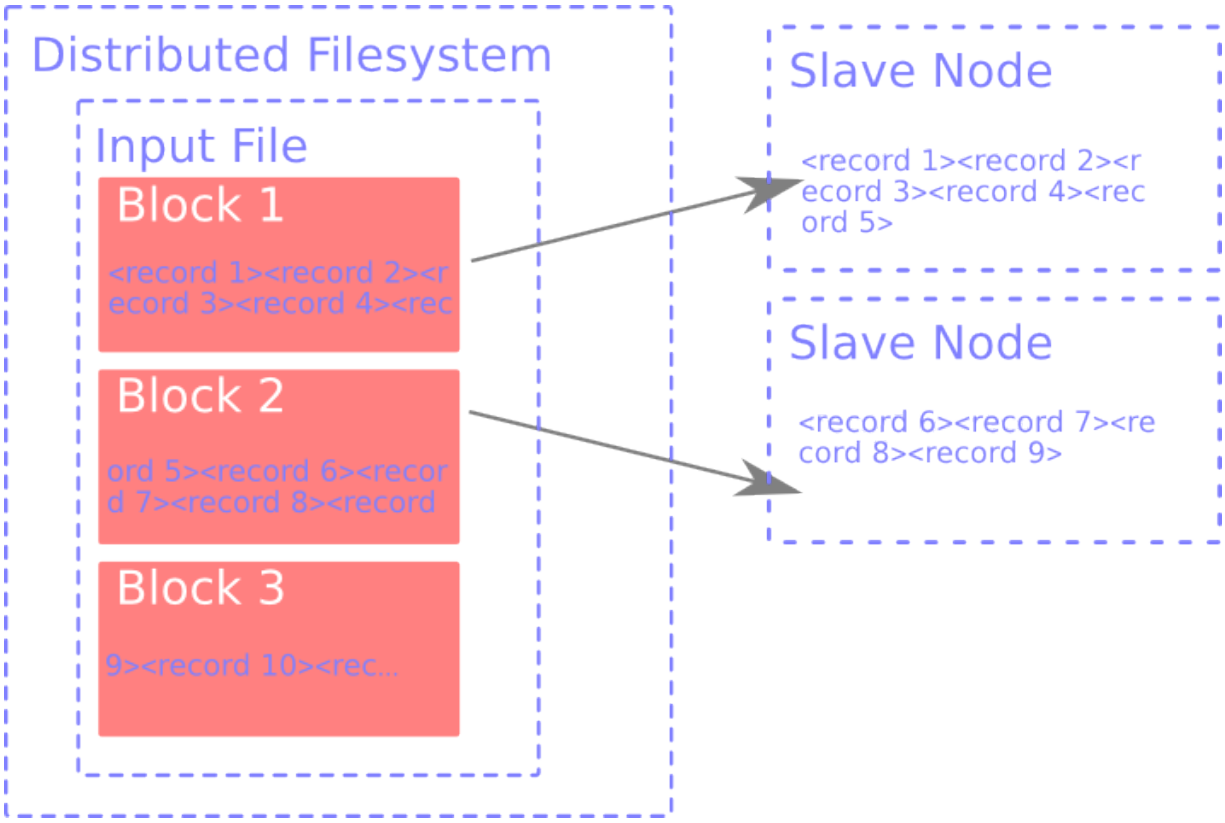


Figure 4.3: Splitting a file into records.

4.1.3 TaskTracker

There is a TaskTracker server running on each slave node. When a MapReduce job is submitted to the JobTracker, it initially distributes the MapReduce program, which is a Java jar-archive and also the job configuration to all TaskTrackers. As soon as the TaskTracker has locally installed and configured the MapReduce program, it is ready to receive tasks from the JobTracker and it will start to process the tasks as soon as it receives any. The TaskTracker is also responsible for reporting the JobTracker about the progress in processing the assigned tasks as well as for reporting failures.

4.1.4 Hadoop Distributed File System HDFS

HDFS is a distributed file system that stores its data in the local file systems of its data nodes. Ideally, the data nodes are also the MapReduce slave nodes, as this allows MapReduce jobs to read input data locally, if the HDFS is used as MapReduce input source. Besides the data nodes, an HDFS cluster also consists of one NameNode, responsible for managing the namespace of the file system and regulating the access to files by clients. [Bor]

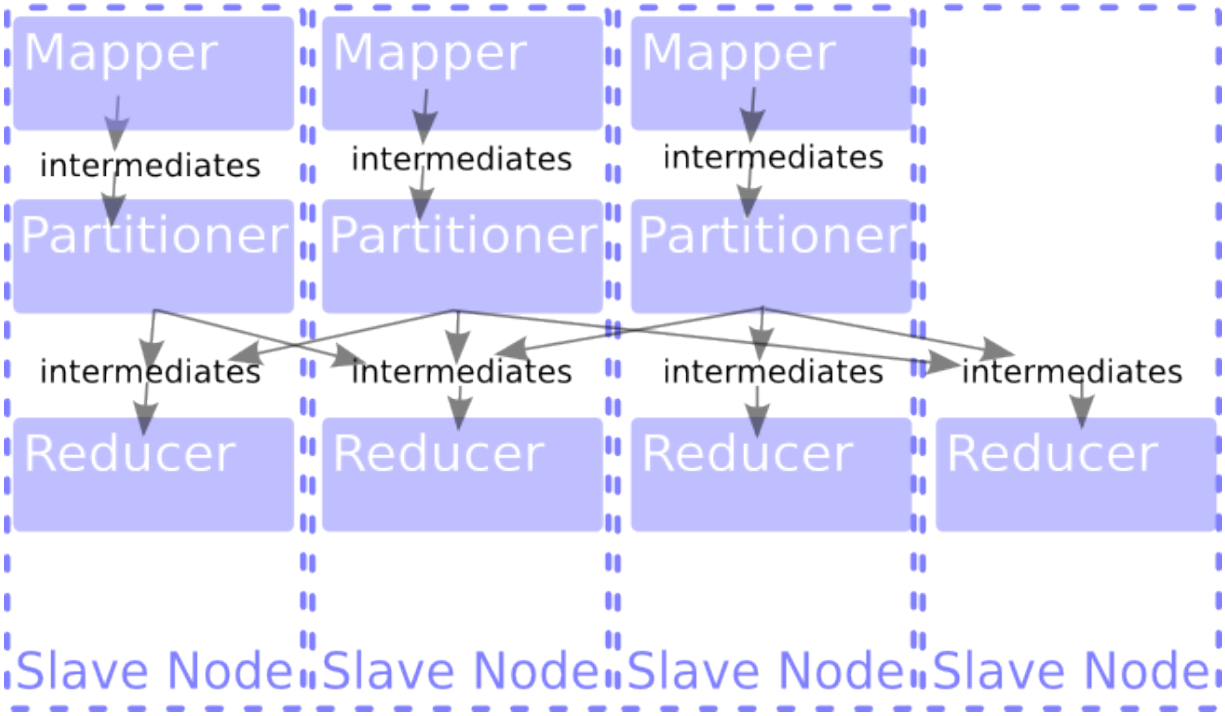


Figure 4.4: The role of the Partitioner.

Qualities

- **Recovery from Hardware Failure**; HDFS is able to automatically recover from hardware failure or from temporary outages, as long as the not accessible data is still replicated in the cluster.
- **Replication** The degree of replication is user configurable. HDFS uses a user configured bandwith for creating replicas of new data.
- **Simple Coherency Model**; HDFS assumes that a file once created, written, and closed needs not be changed in order to simplify the implementation of the coherency and to improve the throughput. This does not restrict MapReduce jobs, but improves their throughput.
- **Streaming Data Access**; HDFS is optimized for streaming data access rather than for random access to data. For example all RecordReaders in Hadoop Common, which read from HDFS, access the HDFS streaming.
- **Extensibility**; It is possible to add data nodes on the fly.

4.2 HBase

HBase is very different from relational databases like MySQL or DB2, not only in the point that HBase does not support SQL or any other querying language, but especially in the fact that in HBase all columns are organized in groups, called column families or just families. When a table is created, only the column families have to be specified, whereas columns may be added and removed dynamically and per row. When a column family of a given row is requested, HBase returns a map, that maps column names to column values. Columns may be added to a family of a given row by altering such a map and submitting it back to HBase [HBaa].

HBase does not manage the type of columns and only allows to store byte arrays in the columns. Using the serialization feature of Java, it is however possible to store even complex types in HBase. Rows are specified by a row key, which is also a byte array. The row key does not belong to any family and other keys than the row key may not be specified. Not allowing secondary[0] or foreign keys, HBase does not need to maintain any indices, but simply organizes its table by sorting them by key, which is very advantageous in order to enable large numbers of rows. First of all, maintaining indices for large numbers of rows may become very time intensive and the indices may grow relatively large too, such that they would no longer fit into a single node's memory and as a consequence, they would no longer allow fast look-ups. The ability of HBase to dynamically add columns to a family within a node may compensate the missing secondary keys to some extend, this is discussed under 5.2.

As mentioned before, the architecture sketched above is inspired by Google's BigTable [CDG+08], HBase is though far away from being a clone of Googles BigTable [Geo]. For example, BigTable enforces access control on a column family level, whereas HBase does not yet have that feature.

By default, HBase uses the Hadoop File System to store its data, which allows HBase to profit from the replication feature of HDFS. More importantly, the distributed file system allows all storage nodes, called HRegionServers [HBab] in HBase, to access all data.

A region of a table is a range of rows, which is stored in one, so called region file, for which one or several HRegionServers are responsible, whereas the HBaseMaster is primarily in charge of assigning regions to the HRegionServers. Further, the HBaseMaster also monitors the health of each HRegionServer and tries to reassigns the Regions of HRegionServers that are no longer reachable.

When the HBase is used as a Hadoop MapReduce source, the regions may be used as splits and Hadoop is aware of the locality of each split. HBase only guarantees eventual consistency [Vog09], using Zookeeper, which is a framework for easily implementing various degrees of consistency. The inconsistency window is configurable, it is typically set to a value between ten seconds and one minute. A too small inconsistency window may lead to a situation, where HRegionServers are considered defective by mistake. Eventual Consistency is fine for working

[0] The support for secondary keys is currently in development, though.

with Hadoop MapReduce, as well as for most batch processing applications. For applications with spontaneous events though, HBase only offers an immature implementation of transactions.

Chapter 5

Design

5.1 System Architecture Overview

The prototype is based on Hadoop MapReduce and uses the HBase NoSQL-database as its main storage. The modular design has been adopted from the original single-threaded Python based implementation. Since in the prototype data is stored in the distributed database rather than passed along from one processing stage to the next, the system is no longer designed as a pipeline. The pipeline stages from the Transit Feed Converter (TFC) are replaced by modules in the prototype of the Distributed Transit Feed Converter (DTFC). The use of the distributed database (HBase), instead of the pipeline allows a more flexible execution planning of the individual modules, i.e. some modules may even be executed simultaneously. Of course, some restrictions to the order of execution still exists. For example, data has to be imported before it can be exported. The handling of those restrictions is discussed later on in this chapter.

Thanks to the use of the distributed database HBase, the planning of the data-structures is straight forward and differs not much from designing a classic object oriented data model. The data model is introduced in the next section and is followed by a a description of the modules of the prototype.

5.2 Data Model

The data model has been designed to be able to fully represent the source data format (HAFAS). On the other hand, the data model is also able to represent aspects of the data which are only extracted during the conversion. An example of such data are the routes, which are also discussed later in this chapter. Even though HBase is not a relational database, the tools and paradigms for designing relational databases are still useful for sketching the data model. The relational model, which has been used as a basis for designing the data model of the DTFC

prototype is depicted in figure 5.1.

The most important difference between the relational data model and the corresponding data model in HBase is caused by the fact that HBase does not allow secondary keys. Looking at the so called junction tables, which are used to build so called many-to-many relationships in relational databases e.g. Stations_Routes in the figure 5.1, it becomes clear, that many-to-many relationships depend on secondary keys in relational databases.

Instead of a secondary key, two junction tables could be used, one linking Routes to Stations and another one linking Stations to Routes. This would not work though with HBase, as the row identifier is required to be unique. Say, one would like to implement a junction table linking Stations to Routes, called Stations_Routes and there were two Routes R1 and R2, which belong to the Station S1. Would an entry S1 $\Rightarrow$ R1 be added to the table Stations_Routes, it would be

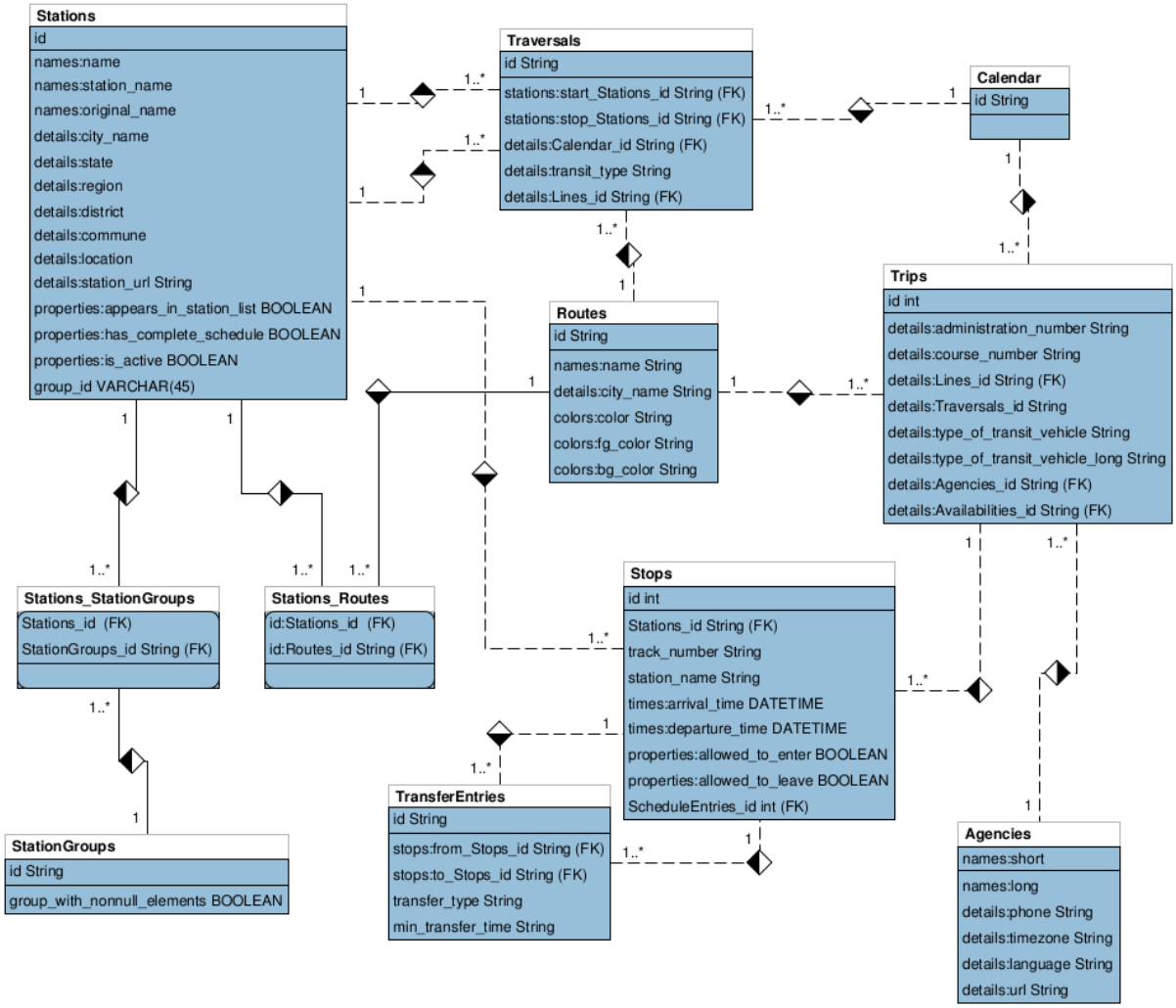


Figure 5.1: Data model

overwritten when the entry S1 ⇒ R2 is added.

However, HBase has another feature, which can be used to model many-to-many relationships, namely the ability to dynamically add columns within a column family. Since each row may have individual columns within a column family, a column can be added for each element the row should be linked to. This is depicted for the above example in the figure 5.2.

Often it is not necessary to be able to look up many-to-many relations in both directions, when it is required, the second table, in the example this is the table Routes, would also need a column entry per relative row from the first table.

The above reflections have lead to an HBase setup with the tables described next.

5.2.1 Stations

The table Stations is used to store the names and further information, such as the geographical location of a public transport station.

5.2.2 Stops

The table Stops holds information about the arrival and the departure time of a certain transit mean at a given station.

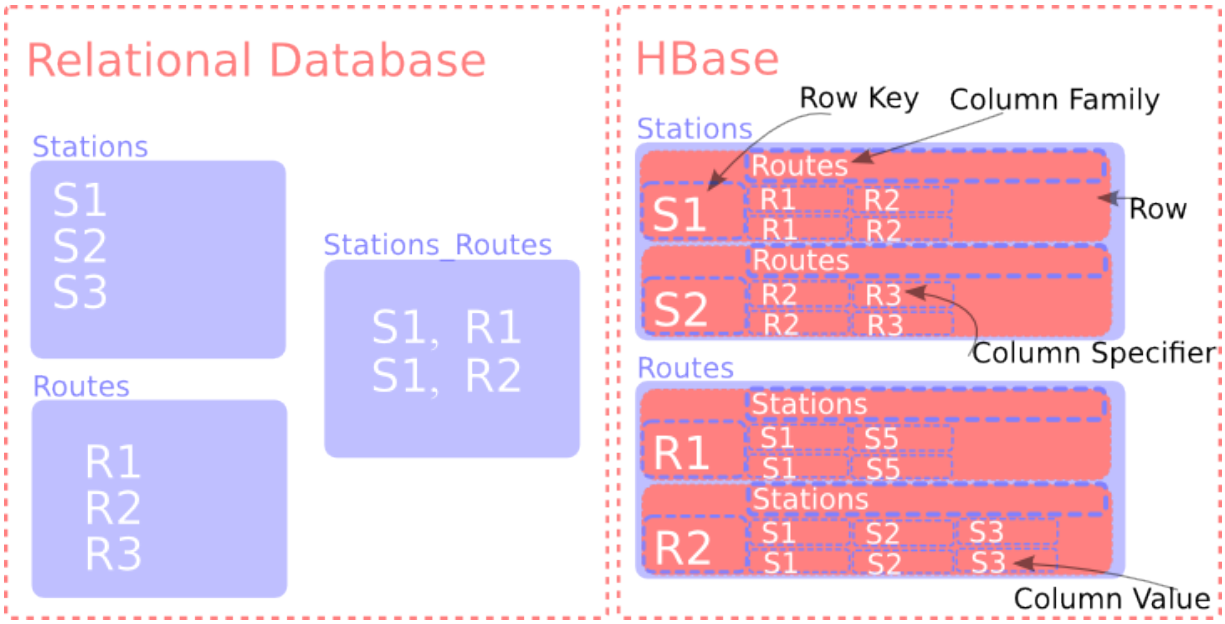


Figure 5.2: Many-to-many relationships in a relational database and in HBase.

5.2.3 Calendar

This table stores a so called bitfield describing the availabilities of services (trips). The bitfield is a 256 bits long bit-string, where a boolean 1 at the i-th position denotes that a service (trip) related to that bitfield is available on the day i days after the day on which the timetable has become valid. The relation between rows of the table Calendar and rows of the table Trips is stored in the table Trips.

5.2.4 Agencies

The table Agencies simply maps transport agency names to agency ids.

5.2.5 Routes

The table Routes lists for each route the trips which belong to that route. A route is a set of trips that is considered as a single service.

5.2.6 Trips

The table Trips relates a trip to its stops and to the trips calendar. A trip is a sequence of at least two stops of a specific transport service.

In addition to the tables described above, a number of temporary tables are used when processing the data.

5.3 Import Modules

Each import job reads from one file only. This approach is the most compatible one with the MapReduce model, as MapReduce jobs may only have one input source. As a workaround, it is sometimes possible to use a multi-file input format or to read files with another API than the MapReduce API, e.g. the HDFS-API. The use of the multi-file input format is rather complicated though and does not work in all cases. Reading files with another API than the MapReduce-API has the disadvantage that these files are not split by the MapReduce framework. In most cases this leads to reading more data than actually needed. In the case, where an importer modules of the prototype DTFC needs to aggregate data from several imput files, all files (but maybe one) are first read into an HBase table. Such, that during a later import job, data can easily be aggregated using the ability to randomly access HBase rows.

5.3.1 Import Agencies

The Import Agencies module simply fills the Agencies table. Since it has very little input data (2000 Bytes), no effort is made to distribute this module. Instead, the import module will run simultaneously with other modules.

5.3.2 Import Bitfields

Import Bitfields is a very simple module consisting of one single Mapper, which receives lines from the bitfield file as input records. The records are parsed and directly written into the Calendar table, such that the Mapper emits no output records. This design follows an example posted in the Hadoop Wiki [Day]. The modules Import Coordinates and Import Stations are designed alike.

5.3.3 Import Trips

The trips form the core of the schedule and cause the largest amount of input data. Therefore it was essential to carefully design this module. Other than in the amount of input data, it also differs in the type of records which the Mapper processes. Unlike all other input files, the file which Import Trips reads from, is not simply line-record based, but one single Trip consists of several lines, therefore a custom record reader is required for the Import Trips module. Further, the Import Trips will not only create the Trips, but also the Stops, as they are implicitly declared within the Trips in the HAFAS raw data format. The Import Trips module will read and write from the Calendar HBase table, since the Trips reference the Calendar entries, but also implicitly define new ones.

5.4 Station Name Mapper

The Station Name Mapper simply changes the name of stations according to a small set of rules. It consists of a single MapReduce Map phase and no Reduce phase. Its MapReduce input source is the Stations table, the outputs are written back to the table rather than spilled out into a MapReduce sink.

A few sample replacement rules are listed in the table 5.1 below.

5.5 Near Stations Merger

The Near Stations Merger module merges all stations, which are geographically located near to each other. The trivial approach to solving this problem is by comparing each Station to all other Stations. This is a typical handshake situation. A way to scalably implement this comparison, is to run several cycles, such that in each cycle all stations are compared to a fraction of the stations. In the illustration 5.3, these fractions are called batches.

Only in the first cycle, all stations are used as input for the Mapper. In the second cycle the stations, which appear in a previously processed batch, are no longer used as the input for the Mapper jobs, as they have already been compared to all other stations.

replace any occurrence of	replace by
ank.	ankomst
afg.	afgang
Byv.	Byvej
Borgm.	Borgmester
Brdr.	Brødrene
Bygd.	Bygaden
Dronn.	Dronning
Gl.	Gammel
g.	
h.	

Table 5.1: Sample replacement, specific to the schedules from the Danish Rejseplanen.

The output of the Mappers are key-value pairs for each pair of stations, which are found to be near to each other. The smaller station (numerically smaller Id) is the key and the larger station is used as the value. The key-value pair can be interpreted as 'The station represented by the key replaces the station represented by the value'. In the Reduce phase all values belonging to one key are simply combined and written to the table 'Replaces'.

Since the resulting table is relatively small, the remaining work is done serially. The resulting table is thus essentially a dictionary, with the key being the station which is meant to replace all stations listed in the dictionary's value-list for that key. In the serial processing part, it is iterated through each such key-value pair. For each pair, with the key S1 it is checked whether any of the stations that are replaced, are listed to replace another Station. If such a Station is meant to replace further stations, those further stations are appended to the value-list of the key S1. This replacement is done recursively.

5.6 Routes Extractor

As there is no such concept as routes (see 5.2.5) in the HAFAS rawdata format, routes have to be guessed before they may be exported to GTFS. Therefore a heuristic has been developed and implemented in the Transit Feed Converter. For the DTFC the very same heuristic is used.

The Routes Extractor starts by creating so called buckets of trips. In a bucket, trips (see 5.2.6)

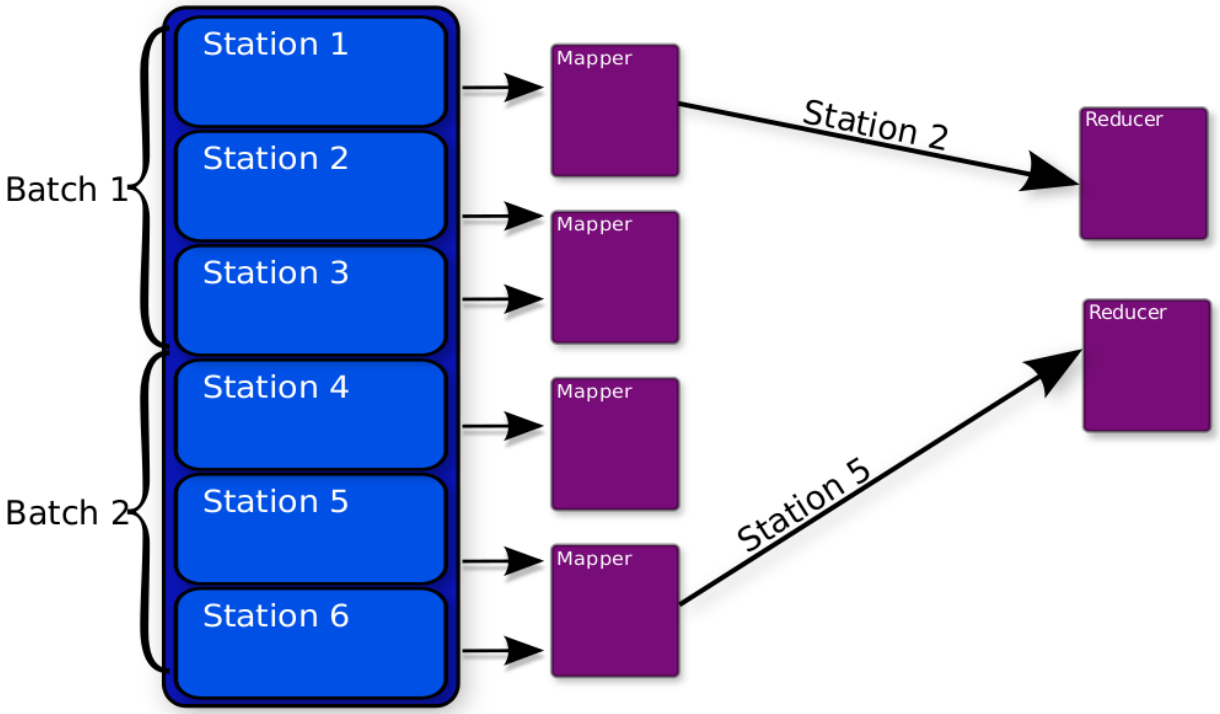


Figure 5.3: Near Stations Merger

are collected that have both, the same line name and the same mean of transport. Grouping items with a common attribute, is a typical task for the Reducer, since HBase is used, though no Reducer is required. Using the attribute, in this case the line name plus the mean of transport as a row-specifier, HBase may be used in order to create those buckets as well. So this job can be implemented as a very simple Map phase. This part is referred to as 'stage 0' of the Routes Extractor.

Next, each bucket is analyzed separately. Each trip in the bucket is compared to each other trip. In a comparison of a $trip_i$ to a $trip_j$, the $percentageSharedStations_{ij}$ is computed as follows $percentageSharedStations_{ij} = \frac{\#commonStations(trip_i, trip_j)}{\#stations(trip_j)}$.

A sample percentageSharedStations matrix is shown in the table 5.3 for the trips listed in the table 5.2.

Trip 1	Station 1 Station 2 Station 4
Trip2	Station 1 Station 5
Trip 3	Station 2 Station 4 Station 5 Station 6

Table 5.2: List of trips.

	Trip 1	**Trip 2**	**Trip 3**
Trip 1	1	0.5	0.5
Trip 2	0.33	1	0.25
Trip 3	0.67	0.5	1

Table 5.3: Sample percentageSharedStations matrix.

Now, a directed graph is initialized with the trips of the current bucket as its vertices and without any edges. Then the percentageSharedStations matrix is processed row by row. For each row i, a dictionary destinationIntensity is created. The destinationIntensity dictionary maps the values of the current row from the percentageSharedStations matrix to the list of column indices at which those values occur. For the first row of the above example the destinationIntensity dictionary is shown in Table 5.4.

The dictionary is then sorted by key and iterated, such that the largest keys are processed first. For each trip j of a given key in the destinationIntensity dictionary, the heuristic tries to add an edge from trip i to trip j and the weight is the dictionary key. As soon as one edge has been added, the iteration is aborted and the next row of the percentageSharedStations matrix is processed. An edge is successfully added, if it does not add loops to the graph. This part is named 'stage 1' of the Routes Extractor and the distributed version of it is implemented as a Map phase.

1	Trip1
0.5	Trip 2 Trip 3

Table 5.4: The destinationIntensity for the first row of the persentageSharedStations matrix in Table 5.3.

In a third Map phase, each graph is processed by one Map method. During this phase, the graph is partitioned into unconnected subgraphs. For each subgraph, a new route will be created.

5.7 Data Flow

The figure 5.7 gives an overview of the source of the incoming and the destination of the outgoing data for each of the above discussed modules.

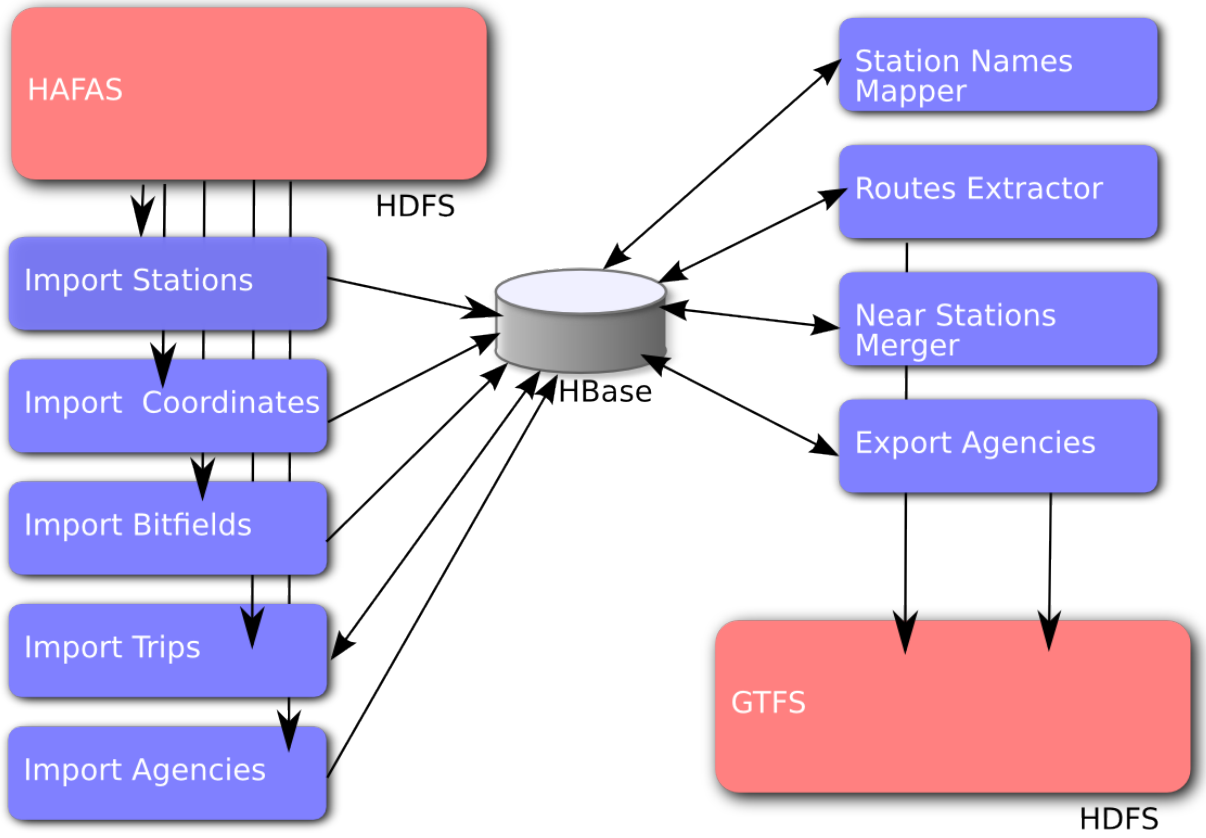


Figure 5.4: Data flow graph of the previously discussed modules.

5.8 Execution Order Dependencies

The figure 5.8 depicts the data dependencies among the module. The dashed lines group modules which are executed simultaneously in the test setup (cf. 7). The arrows (a → b) denote b depends on data computed by a. The lines (a — b) denote a and b may not be executed simultaneously, because at least one module changes data which is processed by the other one, but the actual execution order does not matter.

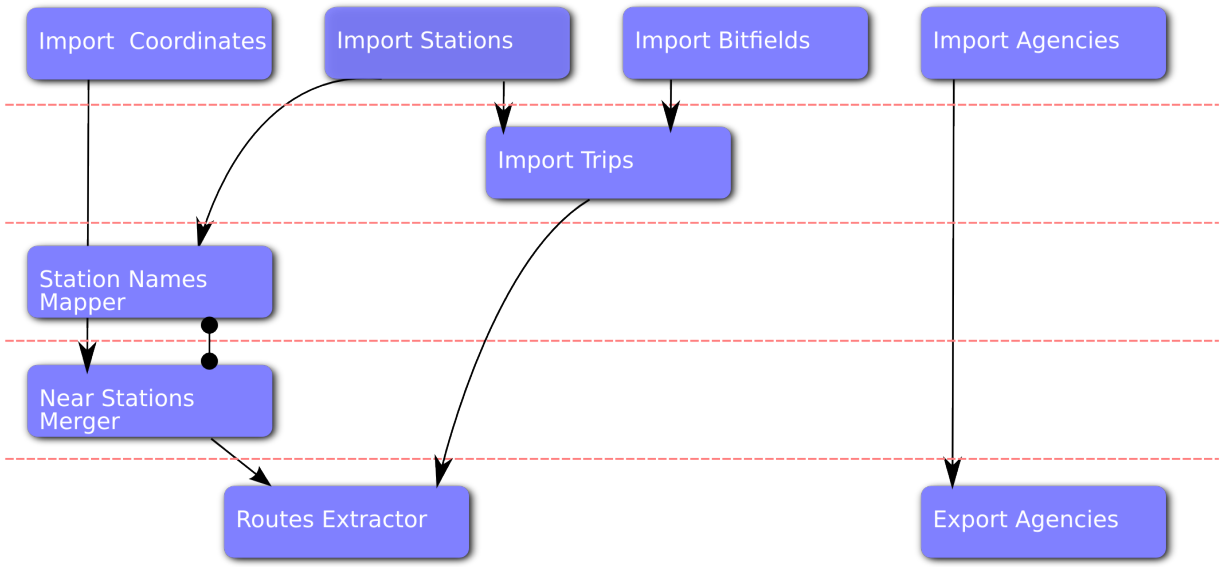


Figure 5.5: Execution order dependencies.

Chapter 6

Implementation

6.1 EC2

The prototype is created for the EC2 platform, which is one of the Amazon Web Services (AWS). AWS customers can allocate and manage EC2 computing nodes using an API or a web interface. Typically, a set of shell scripts provided by Amazon is used to allocate and manage EC2 resources.

Amazon offers a set of EC2 node types differing in the amount of CPU power and in the size of their memory and hard disk. When an instance is allocated, the user selects an instance type and also a disk image, called an AMI in AWS, containing an Operating System compatible to the selected instance type and usually also preinstalled applications.

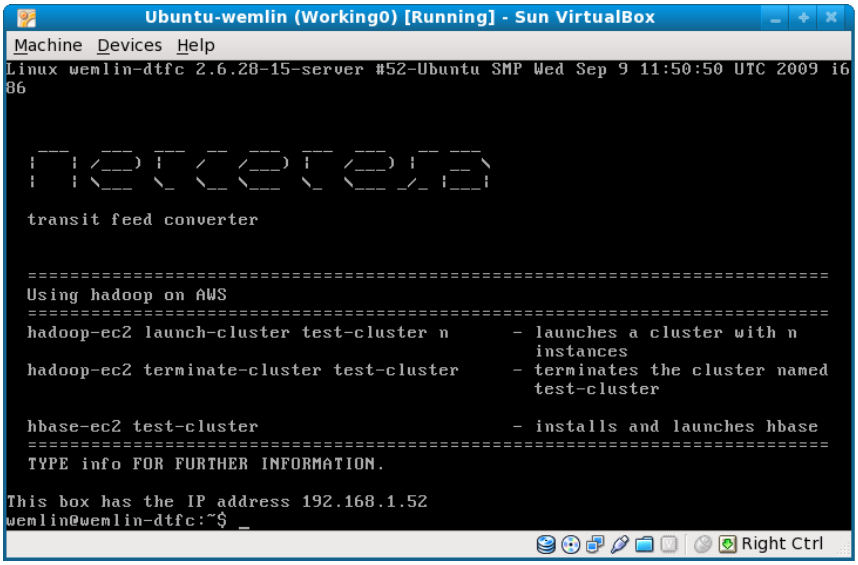


Figure 6.1: The welcome screen of the virtual machine from which a Hadoop/HDFS/HBase cluster can be launched easily.

The pricing of AWS is especially attractive, if the resources are only acquired shortly, while doing a computation. After booting an

EC2 node though, it usually needs first being customized, before any computation may be done. In order to simplify this process, AWS offers the possibility to store a customized AMI [aws].

Some configuration however, is preferably done, whenever the EC2 nodes boot. For example, when a parameter is preferred to be adjustable. In the case of the prototype DTFC, it is the HDFS block size and the number of nodes, that are desired to be adjustable. Adjusting such configuration requires often operations on several, or on all nodes of the Hadoop cluster. Further, they may require administrative operations like stopping and starting services in a defined order. Trying to do this manually, just for running a batch job and loosing everything after terminating the cluster, is a silly idea, not only because it is extremely time-consuming, but also because the chances of making a mistake are extremely high. Therefore, it is unavoidable to automate the setup of an Hadoop/HDFS/HBase cluster on EC2.

As a basis, a set of scripts provided by Cloudera, Inc., plus an AMI also provided by Cloudera, has been useful. For the prototype DTFC, the scripts needed to be slightly modified. Further, scripts for installing and managing HBase, have been added.

In order to make the scripts easier accessible, a set of scripts has been installed on a virtual machine. A screen shot of the running virtual machine is depicted in the figure 6.1. The scripts are briefly described in the appendix A. Using the scripts, as they are installed on the virtual machine, a Hadoop/HDFS/HBase cluster is ready in about seven minutes.

As a very convenient alternative, to using scripts and AMIs from Cloudera and possibly extending them, Aamazon offers another AWS product, called Amazon Elastic MapReduce, which automatically sets up Hadoop clusters and runs Hadoop MapReduce jobs on the cluster. The user of Amazon Elastic MapReduce uses a simple web frontend to submit the Hadoop MapReduce programs and input data. Amazon Elastic MapReduce does not offer support for HBase, though. Thus, the prototype DTFC may not be used with it.

6.2 Datastructure

The HBase API has turned out to be rather inconvenient. Using the HBase API has lead to rather unreadable code and very long lines. So, a wrapper class for the HBase class HTable, as well as a class to represent the HBase rows, have been created. The following features of the HTable wrapper class helped to simplify the working with HBase.

- A table is created by calling the constructor. If the table already exists, the existing table is simply opened.
- Objects are serialized or desirialized when they are stored or written in a column, respectively.
- Supports strings as row, column and family identifiers.

Further, there is an issue with HBase, when an iterator[1] is not touched for a while[2], it is reset, such that it looks, as if the iterator had already walked through the entire table. When the wrapper class is iterated, it automatically handles this issue, such that the iteration continues normally, even in the case of a time-out.

In order to reach a reasonable write speed, it is important that small changes are collected in a buffer until they reach a reasonable size (e.g. 1 MB), before they are sent to the HBase. The wrapper class does this automatically, unless the user forces it to send an item directly.

Though, there remain a few issues that may not be worked around using a simple wrapper class. The largest problem is the enormous memory consumption of HBase, when a large number of changes is submitted to the HBase. Another limitation is that it is seems to be impossible to retrieve a list of cololumn specifiers for a given row and a specified family, without receiving the values of the columns. This is a disadvantage, as each single value may be relatively large (e.g. an integer array with thousands of entries) and is preferably only loaded when definitely needed.

6.3 Modules

6.3.1 Import Modules

In order to be able to fully parallelize and distribute the import process, the input data is loaded into the HDFS, using the HDFS command line tools, before importing data into the HBase. Under EC2 the input data is usually copied from the simple storage (s3), another AWS product. Hadoop would also support reading input directly from s3, but this would require the data to be stored unpacked in s3, further s3 is much slower than HDFS, which uses the slaves' local disks as a storage.

All import modules, but the Import Trips module, simply use the TextInputFormat, which is provided by Hadoop and feeds the text line by line to the Map methods. The Import Trips module requires records which span several lines. A sample record for the Import Trips is listed below.

```
513000301 Østerbakken Adsbøl    00822  00822              % 00388 00015_
533000400 Nybøl v kirken        00827  00827              % 00388 00015_
537001801 Vestermark/Bosager    00835  00835              % 00388 00015_
537001800 Arnkilgade v Sønderb  00835 -00835              % 00388 00015_
537001802 Arnkilgade/Helgoland  00835 -00835              % 00388 00015_
537000101 Løngang/Rebslagergad  00835 -00835              % 00388 00015_
537000100 Sønderborg Busst.     00844                     % 00388 00015_
*Z 00389 00015_                                           % 00389 00015_
*L 10    503004300 537000100  00645  00738               % 00389 00015_
*R 1 537000100 503004300 537000100  00645  00738         % 00389 00015_
*G 013 503004300 537000100  00645  00738                  % 00389 00015_
*A VE 503004300 537000100 000005  00645  00738           % 00389 00015_
```

[1] Table iterators are actually called Scanners in HBase.

[2] The time after which the iteration must continue seems to be the same as the inconsistency window. (see 4.2)

```
503004300 Frøslev-Padborg Skol         00645               % 00389 00015_
503004302 Frøslevvej (Padborg) -00645  00645               % 00389 00015_
503005600 Nørregade v Hasselha -00647  00647               % 00389 00015_
503005400 Padborg St (bus)     -00650  00650               % 00389 00015_
503005404 Jernbanegade 19/24 P -00650  00650               % 00389 00015_
503005403 Jernbanegade 54/57 P -00651  00651               % 00389 00015_
503005402 Padborgvej/Rønshave  -00651  00651               % 00389 00015_
503004700 Bov, Hærvejen v kirk -00652  00652               % 00389 00015_
503005001 Bov, Padborgvej v Øs -00652  00652               % 00389 00015_
503005000 Smedeby, Padborgvej  -00652  00652               % 00389 00015_
513000401 Rinkenæs Syd         -00710  00710               % 00389 00015_
513000400 Rinkenæs             -00710  00710               % 00389 00015_
513001400 Alnor, Egernsund Bro -00712  00712               % 00389 00015_
537001801 Vestermark/Bosager    00733 -00733               % 00389 00015_
537001800 Arnkilgade v Sønderb  00733 -00733               % 00389 00015_
537001802 Arnkilgade/Helgoland  00733 -00733               % 00389 00015_
537000101 Løngang/Rebslagergad  00733 -00733               % 00389 00015_
537000100 Sønderborg Busst.     00738                      % 00389 00015_
*Z 00390 00015_                                            % 00390 00015_
*L 10    503001600 537000100  00849  00944                 % 00390 00015_
*R 1 537000100 503001600 537000100  00849  00944          % 00390 00015_
*G 013 503001600 537000100  00849  00944                   % 00390 00015_
*A VE 503001600 537000100 000002  00849  00944            % 00390 00015_
503001600 Kruså Busst.                 00849               % 00390 00015_
503002100 Kruså, Flensborgvej   00850  00850               % 00390 00015_
```

Listing 6.1: Excerpt from the file fplan, which is the input for the Import Trips module. The excerpt contains one complete record, plus a fraction of its predecessor and successor record. A new record starts with '<newline>*Z'.

Writing a custom Hadoop MapReduce InputFormat is straight forward, however when also a custom RecordReader is required, it may be rather delicate to implement an efficient one. An efficient RecordReader is an essential part of the entire MapReduce process. Some important points to consider when writing a RecordReader are listed below.

- Read reasonable portions from the input stream.
- Offer data to the record consumer as soon as possible.
- Inform the framework as soon as possible when a record is complete.

Further, it is important to thoroughly test a RecordReader before productively using it, as it happens very easily that a RecordReader would skip a Record or insert one twice, because records usually do not end where the split ends. In the case of the Import Trips module, the RecordReader requires two buffers. More concretely speaking, when the RecordReader processes the buffer into which it has read data from the input stream, it will eventually reach the end of the buffer, but it will not always be able to tell whether the currently processed data is still part of the current record, or whether it is already part of a new record. This is for example the case, when a buffer ends with the character-sequence '<newline>*' (A new record starts with

'<newline>*Z'). The fact that this situation occurs relatively rarely, increases that such a situation is overseen in the test-cases, but is still reasonably likely to occur when several hundred megabytes of input data are processed.

6.3.2 Near Stations Merger

During the design phase, the batches (cf. 5.5) were intended to be processed one after another, since each batch would be able to utilize the cluster. This design decision was also encouraged by the design of Hadoop and Hbase. Due to the way scanners[3] work in HBase, it is only known where the successive batch starts, after the current batch has been fully read. This is to say, there is no easy way to figure out, which the key of the n-th, 2n-th, 3n-th etc. row is, other than iterating through the rows. As this iteration is done anyway by the slaves during the Map phase, there is no reason to do an extra iteration on the master, which would unnecessarily load the HBase. According to the principles of MapReduce, side-effects should be avoided. This means in particular, that nothing that has been computed as part of the Map or of the Reduce phase, should be used by another node of the cluster. The fact that Hadoop needs a long time to setup a MapReduce job (ca. 30 seconds), however has lead to a dramatic increase of the processing time as the number of batches grows. So, it was eventually decided that the jobs for batches would be submitted as soon as the preceding batch has been iterated through by any Mapper. For this purpose, an XML-RPC server is installed on the Hadoop master, to which the slave connects as a client and calls a method in order to inform the master about the location of the next batch. The master handles synchronization issues and assures that all batches are only processed once.

6.3.3 Routes Extractor

The implementation of the Routes Extractor was straight forward, there were very few changes necessary to get a distributed version from the serial one. As this fact already seemed apparent at design-time, the port of the Python implementation has been performed in two steps. In the first step, the application was ported to Java using HBase as the main storage. Later, it was very easy to simply replace a few loops by Map jobs in order to fully distribute the application.

There was an issue though, namely the memory consumption caused by committing changes to HBase. The original plan was to store the percentageSharedStations matrix (cf. 5.6) in HBase and later it would only be accessed row by row. This has though lead to a situation, where much more memory has been consumed than by storing the entire matrix in memory.

[3] Scanners are table iterators.

Chapter 7

Test Setup

7.1 Topology of the Cluster

The cluster has one master node and a configurable number of slave nodes. The master serves as the NameNode for HDFS and as the HBase master. Further, it is also the Hadoop JobTracker, but it is neither used to host any data nor to process any of the Map or Reduce tasks. The slave nodes are used as HDFS DataNodes, as HBase RegionServers and as TaskTrackers. The actual network topology remains unknown, as such information is not provided by AWS.

7.2 The Installation Procedure

The cluster installation includes the following steps.

1. The master node is booted up and the Hadoop jobtracker service is started.
2. The slave nodes are booted and Hadoop tasktracker services are started. The slaves know their master and they advertise themselves as tasktracker nodes.
3. Additional configuration is done, e.g. HDFS block size.
4. The Hadoop HDFS NameNode-service is launched on the master.
5. HDFS data nodes are launched on the slaves, they advertise themselves as TaskTracker nodes.
6. The HDFS is formatted.
7. Data is imported into HDFS from s3. (cf. 6.3.1)
8. HBase is installed on all nodes (copied from s3 and unpacked).

9. HBase configuration is adjusted, i.e. the master is informed about which nodes participate as RegionServers in the HBase-cluster.
10. Zookeeper is launched on all nodes
11. HBase master is launched on the master node.
12. RegionServer processes are spawned on all slave nodes.

This procedure takes about seven minutes. Afterwards a cluster with the desired number of slave nodes and the configured block size is ready to launch Hadoop jobs.

7.3 EC2 Nodes

All nodes of the cluster use the default small EC2 node, which according to the AWS website has the following properties.

- 1.7 GB main memory
- 1 EC2 compute unit [1]
- 160 GB hard disk
- 32-bit architecture

7.4 The Test Job

During the test job, all modules are launched as depicted in the figure 5.8. This is to say the jobs are launched in as follows.

1. Import Coordinates, Import Stations, Import Bitfields and Import Agencies are submitted simultaneously to the Hadoop cluster.
2. After they have all terminated, Import Trips is launched.
3. When Import Trip has completed, Station Names Mapper is submitted.
4. After the Station Names Mapper has finished, Near Stations Merger is launched.
5. Finally, Routes Extractor and Export Agencies are launched simultaneously.

[1] The AWS website says, 'One EC2 Compute Unit provides the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor.'

Chapter 8

Evaluation and Results

8.1 Modules

8.1.1 Import Modules

As shown in 5.8, most import modules may run in parallel, which leads to a good utilization of the cluster (figure 8.1)[1]. Not surprisingly, during the import phase, some network traffic is generated (figure 8.2). The traffic is though still relatively low, as HDFS only uses a small, configurable network bandwidth for replicating new data.

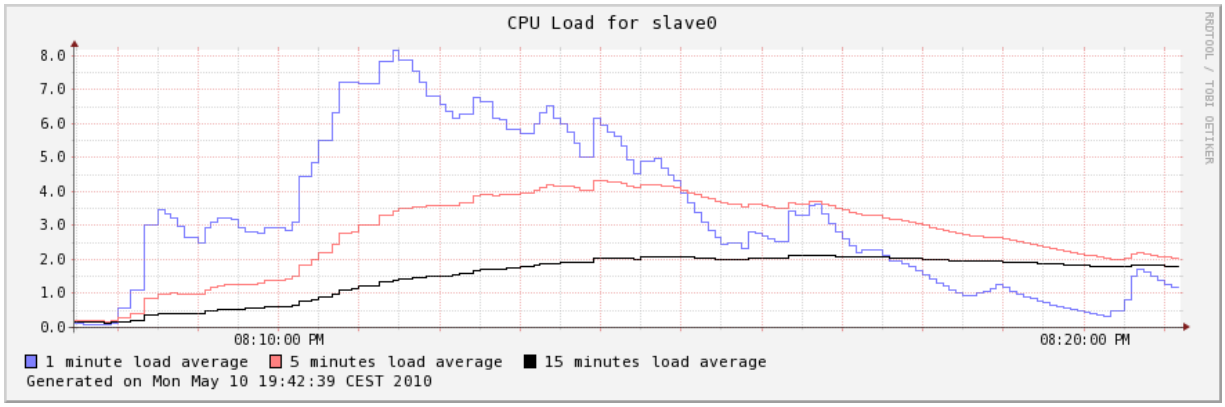


Figure 8.1: The CPU load average, which is a measure for the CPU utilization, of a slave node during the import phase.

In the figure 8.3, the processing time of the entire import process is plotted as a function of the

[1]The figure plots the load average, as it is reported by Linux. [POL02] says about the load average, 'The load average tries to measure the number of active processes at any time. As a measure of CPU utilization, the load average is simplistic, poorly defined, but far from useless.'

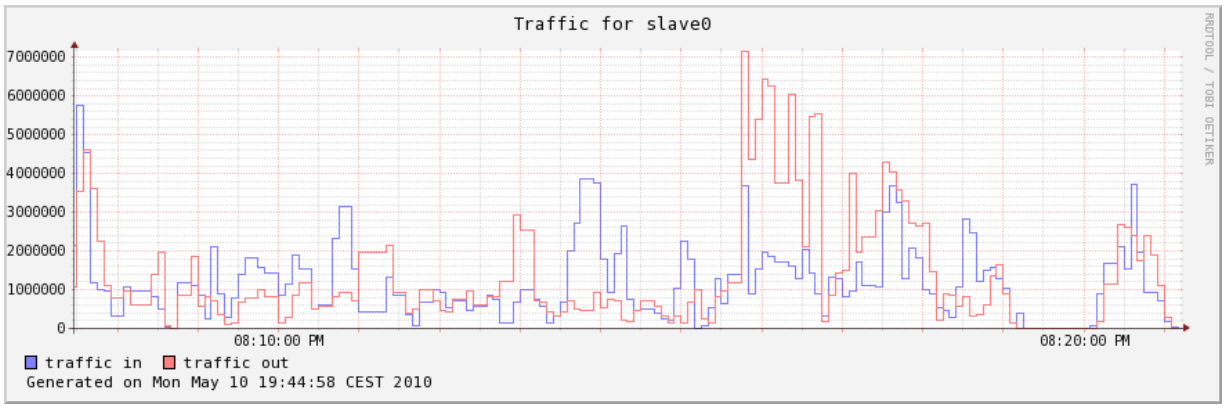


Figure 8.2: The network traffic of a slave node during the import phase in bytes per second.

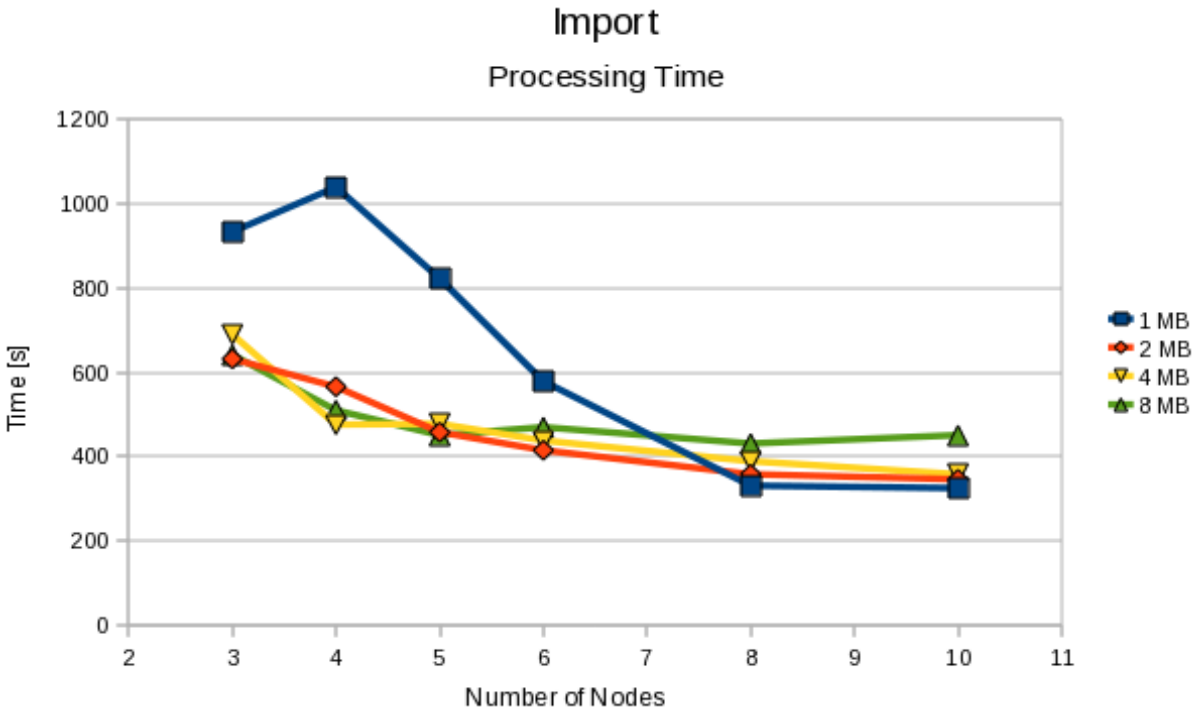


Figure 8.3: Plot of the processing time as a function of the number of nodes for block sizes of 1,2,4 and 8 Megabytes. Each value is based on a single measurement.

number of nodes. The execution time of the distributed prototype import procedure is relatively long in comparison to the execution time of the original Python implementation, which only takes about 4.5 minutes, whereas the execution time of the distributed prototype import may require as long as 10 minutes.

The import time is larger, primarily due to the fact that the distributed implementation imports

data into the distributed database, whereas the original Python implementation simply stores its data in the local main memory, which is of course much faster. Further, the Import Trips module also reads randomly from the previously imported Bitfields table, which is also much faster when done locally.

8.1.2 Map Station Names

The module Map Station Names is the simplest possible module to program. It can be done perfectly in a single Map phase. The running time of the serial Python equivalent is approximately 130 seconds[2]. This processing time may be reasonably reduced (figure 8.4).

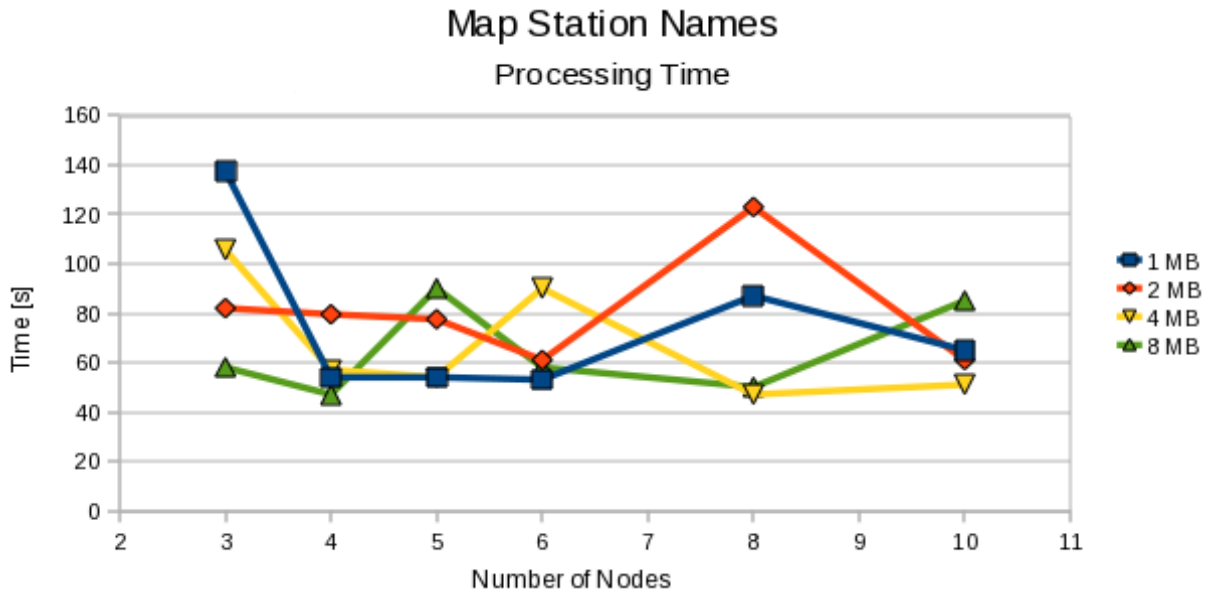


Figure 8.4: Plot of the processing time as a function of the number of nodes for block sizes of 1,2,4 and 8 Megabytes. Each value is based on a single measurement.

In comparison to the running times of the import procedure though, the running times of the Map Station Names seem rather randomly and do not reach a reasonable reduction of the processing time with a growing number of nodes. This has to do with the fact, that the input data for the Map Station Names module is relatively small compared to the entire input data, which is processed during the import process. Further, HBase generates some indeterminacy regarding the execution time. The input data, which is processed by the Map Station Names module is read from an HBase table. As the input data has just been imported shortly before the Map Station Names module is run, the table may not yet have completely split up into regions[3],

[2] On a 1.7 GHZ Intel Dual Core with 2 GB Memory.
[3] There is no way to reliably enforce an immediate split into regions.

such that reasonably less splits are created and not all slave nodes can be loaded with work.

8.1.3 Near Stations Merger

The execution times of the Near Stations Merger module are plotted in the figure 8.5.

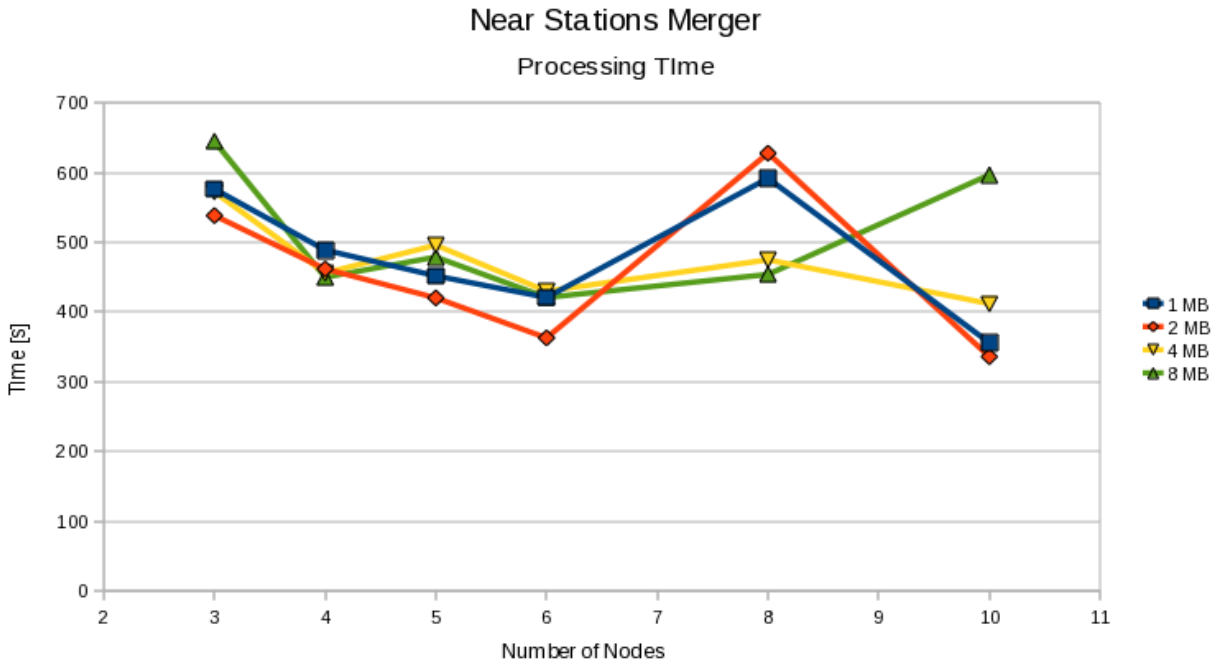


Figure 8.5: Plot of the processing time as a function of the number of nodes for block sizes of 1,2,4 and 8 Megabytes. Each value is based on a single measurement.

The execution times perform extremely bad in comparison with the implementation in the original pipeline, which uses a modification of a divide and conquer approach, which is actually intended to solve the closest-pair problem [ros] [SH]. The algorithm, as it is published in [ros], is modified such that whenever the distance between two points is computed, it is also tested whether the two points have a smaller distance than a previously defined threshold. If so, the pair of points (stations) is listed to be merged. The original implementation runs only about 90 seconds[4]. This algorithm is though not an exact one and may not find some points, which are less distant from each other than the threshold (cf. appendix D.1). Thus, the comparison between the two implementations is not a completely fair one.

The above algorithm seems not to be easily and efficiently distributable using Hadoop. It would though be possible to split the area, in which the points (stations) are found, into equal parts using vertical separation lines. The resulting areas could be used as the Mapper inputs. In

[4]On a 1.7 GHZ Intel Dual Core with 2 GB Memory.

addition to the areas between the separation lines, the areas with a width of twice the threshold, which have the separation lines through their center, would also be use as Mapper inputs.

8.1.4 Distributed Routes Extractor

The Distributed Routes Extractor enables a reasonable reduction of the running time when it is distributed using Hadoop. The results are shown in 8.6. The distribution is however limited due to the fact that especially the stage 1 (cf. 5.6) receives input, which causes extremely unequal computational load per record. Therefore, this records should be split up, such that they could be further distributed. As discussed previously, this caused too much load on the HBase, however.

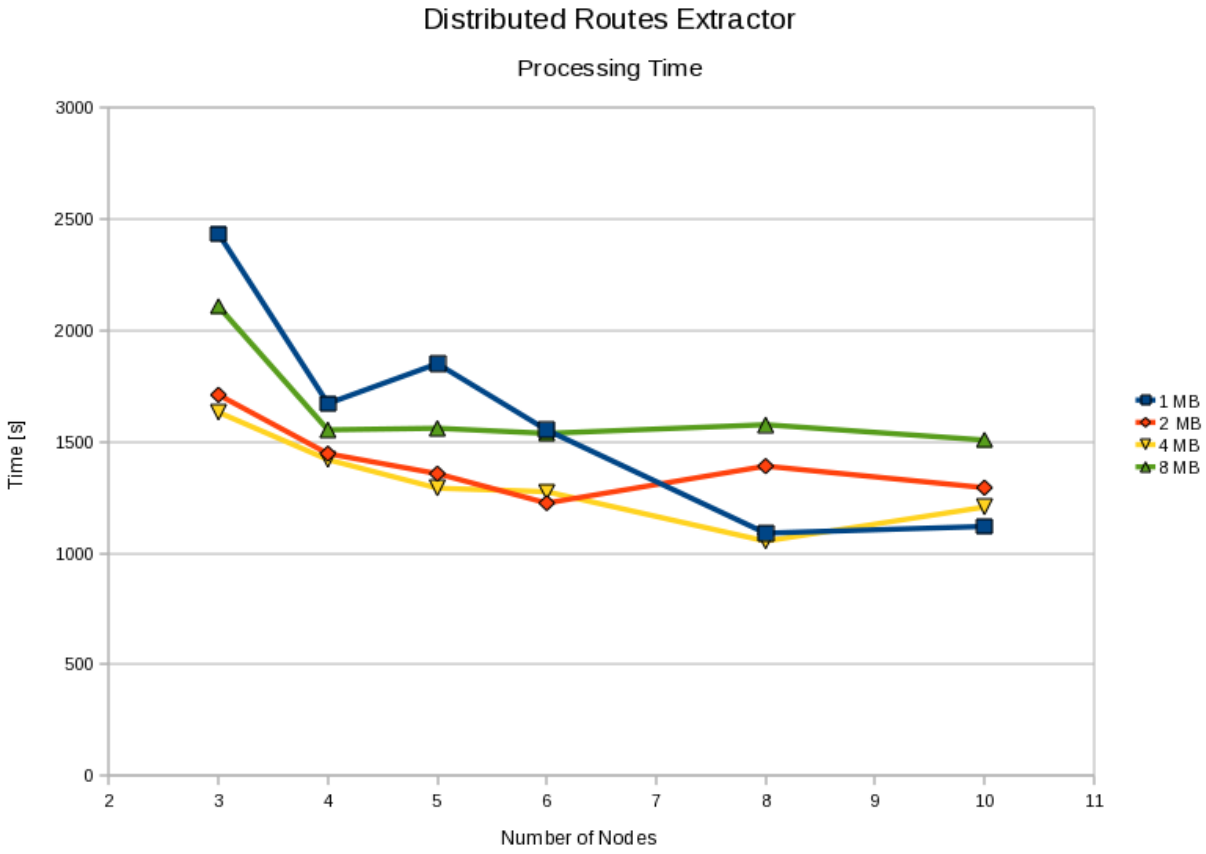


Figure 8.6: Plot of the processing time as a function of the number of nodes for block sizes of 1,2,4 and 8 Megabytes. Each value is based on a single measurement.

The longest record takes about 15 minutes to process[5], further there are a few records, which also take about ten minutes, whereas most of the records take less then one minute to be

[5]On a 1.7 GHZ Intel Dual Core with 2 GB Memory.

processed. Unfortunately it happens rather likely that within one split two of the longest trips appear, such that the shortest possible processing time of the stage 1 never goes below ca. 25 minutes with whichever number of nodes. It would help, if the splits would be made smaller (by downsizing the HDFS blocks), however HBase seems not to work with blocks sized below 1MB. The figure 8.7 shows the memory usage, which is very critical on the slave nodes during the execution of the Distributed Routes Extractor.

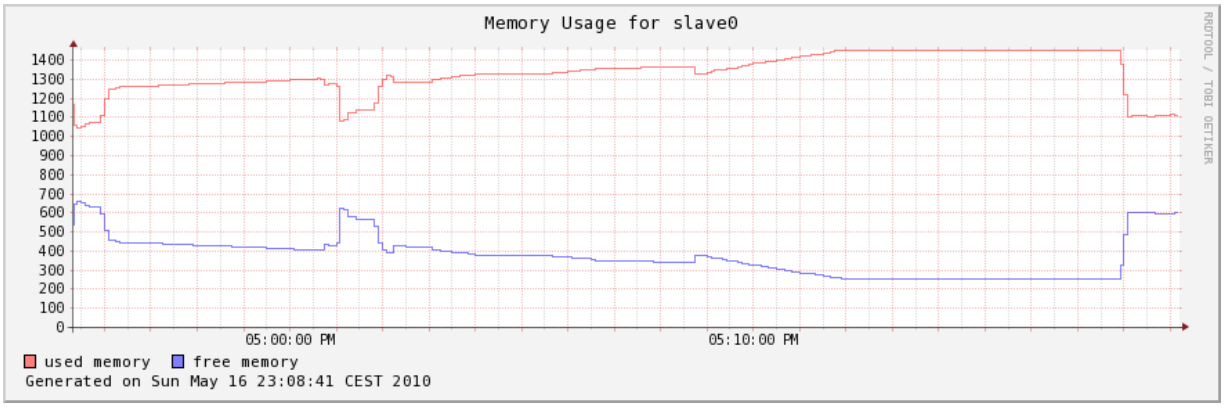


Figure 8.7: The memory usage of a slave node while the Distributed Routes Extractor is running.

Chapter 9

Conclusions

9.1 Conclusions

The performances of the import procedure and the Distributed Routes Extractor module are satisfying, but not compelling. The other modules perform rather disappointingly. Looking at the memory usage on the slave nodes during the execution of the prototype converter, reveals another serious problem; HBase requires more memory than the small EC2 nodes have available in order to run comfortably.

This shows that during the evaluation of a database system, not only the data model should be considered, but also the database performance should be tested under real work load on the target platform. Whereas the data model of HBase convinced, the performance on the target platform was unknown. Knowing that with similarly sized tables (The table Trips may contain several hundred thousand rows, for example.), common relational databases need to be carefully setup in order to deliver reasonable query times, HBase seemed definitely worth a try.

Only after setting up an HBase cluster on EC2 nodes, the difficulties of running HBase on small EC2 nodes became obvious. Before it came to that, another insight has been gained; Setting up a working HBase environment on AWS is totally different from setting up an HBase system on 'real' machines. As every administrative action on AWS must be automated, if it should not be lost at the next reboot, a reasonable amount of unexpected, additional work arose. Though, setting up a Hadoop/HBase cluster has been an enriching experience, as it taught the careful planning of every step of the system maintenance. The result is a reliable and stable Hadoop/HBase cluster, which can be started with the desired number of EC2 nodes in only about seven minutes.

9.2 Outlook

The use of the database HBase has lead to a shortage of memory on the small EC2 nodes. HBase is though said to work better on larger nodes with more memory. This should be verified.

On the other hand, using another database than HBase should also be considered.

It is assumed, that the current prototype is able to handle larger sets of input data than the relatively small data set used in the tests, as a larger data set should allow to distribute the load among a higher number of nodes. The verification of that assumption also remains to be done.

Further, not all of the computationally intensive parts of the Transit Feed Converter have been investigated in the scope of this thesis[1].

Some parts of the Transit Feed Converter may be reasonably improved algorithmically. Those are in particular the module Routes Extractor and the module Near Stations Merger.

[1] The most important module, which has not been discussed, is called Through Coach Extractor.

Appendix A

Scripts for Bringing up Hadoop

Below is a short description of the scripts, which can be used to setup a Hadoop/HBase cluster on AWS. The scripts require a proper installation of Cloudera's Distribution for Hadoop (CDH) and must be located themselves in the path.

- **hbase-ec2 test-cluster** installs an HBase on the running cluster, called test-cluster.
- **setup-cluster #nodes blockSize** installs Hadoop/HBase on a new cluster with #nodes slave instances and one master. The blockSize is specified in bytes.
- **h-ec2-020 #nodes blockSize** creates a new Hadoop-0.20 cluster. Hadoop-0.20 is required in order to use the HBase version, which is installed with 'hbase-ec2'.

Appendix B

A Sample MPI Program

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

#define WORKTAG 1
#define DIETAG 2

/* Local functions */
static void master(void);
static void slave(void);
static int* get_next_work_item(void);
static void process_results(int*);
static int* do_work(int*);

int ind = 0;
int values[] = {1,2,3,4,5,6,7,8,9,10,11,12};
#define n_values  12
int results[n_values];
int result_item[2];
int work_item[2];

int main(int argc, char **argv)
{
  int myrank;

  /* Initialize MPI */
```

```
  MPI_Init(&argc, &argv);

  /* Find out this instances identity */

  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  if (myrank == 0) {
    master();
    printf("[");
    int i = 0;
    for ( ; i < n_values; i++)
        printf("%i,",results[i]);
    printf("]");
  } else {
    lave();
  }

  /* Shut down MPI */

  MPI_Finalize();
  return 0;
}


static void master(void)
{
  int ntasks, rank;
  int* work;
  int result[2];
  MPI_Status status;

  /* Find out how many processes there are  */

  MPI_Comm_size(MPI_COMM_WORLD, &ntasks);

  /* Send one unit of work to each slave. */

  for (rank = 1; rank < ntasks; ++rank) {

    /* Find the next item of work to do */
    work = get_next_work_item();
```

```
    /* Send it to each slave */
    MPI_Send(work,
             2,
             MPI_INT,
             rank,
             WORKTAG,
             MPI_COMM_WORLD);
  }

  /* Get new work until all work is done */

  work = get_next_work_item();
  while (work[0] != -1) {

    /* Receive results from a slave */
    MPI_Recv((void*)result,
             2,
             MPI_INT,
             MPI_ANY_SOURCE,
             MPI_ANY_TAG,
             MPI_COMM_WORLD,
             &status);
        process_results(result);

    /* Send the slave a new work unit */
    MPI_Send(work,
             2,
             MPI_INT,
             status.MPI_SOURCE,
             WORKTAG,
             MPI_COMM_WORLD);

    /* Get the next unit of work to be done */
    work = get_next_work_item();
  }

  /* There's no more work to be done, so receive all the outstanding
     results from the slaves. */
  for (rank = 1; rank < ntasks; ++rank) {
```

```
    MPI_Recv((void*)result, 2, MPI_DOUBLE, MPI_ANY_SOURCE,
             MPI_ANY_TAG, MPI_COMM_WORLD, &status);
    process_results(result);
  }

  /* Tell all the slaves to exit by sending an empty message with the
     DIETAG. */

  for (rank = 1; rank < ntasks; ++rank) {
    MPI_Send(0, 0, MPI_INT, rank, DIETAG, MPI_COMM_WORLD);
  }
}


static void  slave(void)
{
  int work[2];
  int* result;
  MPI_Status status;

  while (1) {
    /* Receive a message from the master */
    MPI_Recv((void*)work, 2, MPI_INT, 0,
             MPI_ANY_TAG, MPI_COMM_WORLD, &status);
    /* Check the tag of the received message. */
    if (status.MPI_TAG == DIETAG) {
      return;
    }

    /* Do the work */
    result = do_work(work);

    /* Send the result back */
    MPI_Send(result, 2, MPI_INT, 0, 0, MPI_COMM_WORLD);
  }
}

int* get_next_work_item(void)
{
```

```
      int* workitem;
      workitem = (int*) malloc( 2*sizeof(int) );

      if(ind < n_values){
                  workitem[0] = ind;
                  workitem[1] = values[ind];
                  ind++;
         }
      else{
                  workitem[0] = -1;
         }
         return workitem;
}


static void process_results(int* result)
{

  if(result[0] != -1)
         results[result[0]] = result[1];
}

int* do_work(int* work)
{
  int* resultitem;
  resultitem = (int*) malloc( 2*sizeof(int) );

  resultitem[0] =  work[0];
  resultitem[1] = work[1] * work[1];
  return resultitem;
}
```

Listing B.1: A sample MPI-Programm, calculating the square of the integers in an array.

Appendix C

A Sample Erlang Program

```
-module(square).
-export([do_work/0, main/0, launch/1]).

do_work() ->
 receive
 {From, {Ind, Val}} ->
 From ! {self(), {Ind, Val*Val}},
 do_work();
 {stop} ->
 io:fwrite("Stopping");
 Other ->
 io:format("Unknown: ~w~n", [Other])
 end.

launch(Pid) ->
        Val = get(get(ctr)),
        if
        Val/=undefined -> Pid ! {self(), {get(ctr), get(get(ctr))}},
        put(running,get(running)+1),
        put(ctr,get(ctr) + 1) ;
        true ->  io:fwrite("No more work to do. \n")
        end
.

main() ->
 receive
 {Id,{Index,Value}} -> put(Index,Value),
```

```
        put(running,get(running)-1),
        square:launch(Id),
        Val = get(running),
        if
        Val>=1     ->  main();
        true -> 0
        end
        ;
 Other ->
 io:format("Unknown: ~w~n", [Other])
end.
```

Listing C.1: A sample Erlang module, which can be used for distributedly calculating squares.

The above module can be used from the shell as follows.

```
c(square).
Hosts = ['name1@host1','name2@host2'].

put(ctr,0).
put(running,0).
put(0,1).
put(1,2).
put(2,6).
put(3,8).

Init = fun(X) -> spawn(X,fun square:do_work/0) end.
Launch = fun(X) -> square:launch(X) end.
Stop = fun(X) -> X ! {stop} end.

Pids = lists:map(Init,Hosts).
lists:foreach(Launch,Pids).
square:main().
lists:map(Stop,Pids).
```

Appendix D

Finding Nearby Points in the TFC

The TFC uses an algorithm for solving the Closest-Pair Problem [SH] in order to find nearby points.

D.1 Algorithm for Solving the Closest-Pair Problem

The goal of the algorithm is to find the distance between the pair of points, which are closer to each other, than any other pair in a given set of points.

A slightly simplified version of the algorithm used for the transit feed converter is listed below.

1. If the problem is small, solve it using a brute-force approach, otherwise continue.
2. Sort points according to their x-coordinate.
3. Split the set of points, such that two subset with the same size are received. The x-coordinate, which is in the middle of the last point in the left subset and the first point of the right subset, is called x-sep.
4. Recursively solve the problem for the right and for the left subset, which will return a minimal distance for the right-hand dLMin subset and one for the left-hand subset dRMin.
5. Compute the minimal distance dLRMin (using brute-force) among the points which are no farther than min(dLMin,dRMin) from the vertical line with the x-coordinate x-sep, which separates the original area into two subareas.
6. Return the minimum of dLMin, dRMin and dLRMin

This algorithm has a complexity of $\mathcal{O}(n \cdot log(n))$ [SH].

D.2 Modification

In order to use the above algorithm to find pairs, which are closer to each other than a given threshold t, the above algorithm is changed, such that whenever distances between two points are computed, the algorithm adds the two points to the global list of close points if the two points are less distant than t. Distances between points are computed, when the brute-force approach is used in step 1, but also in step 5 of the algorithm.

Step 5 is the reason, why some pairs of points may be missed. In step 5, only pairs of points, which are less distant than min(dLMin,dRMin) from the vertical through x-sep are considered. In order to certainly get all points, which are closer to each other than t, the algorithm should compare all points which are not farther than t away from the vertical with the x-coordinate x-sep. This would however require slightly more computational effort. Further, the algorithm would have to be adjusted to assure that none of the subareas become narrower than t.

Appendix E

Test Results

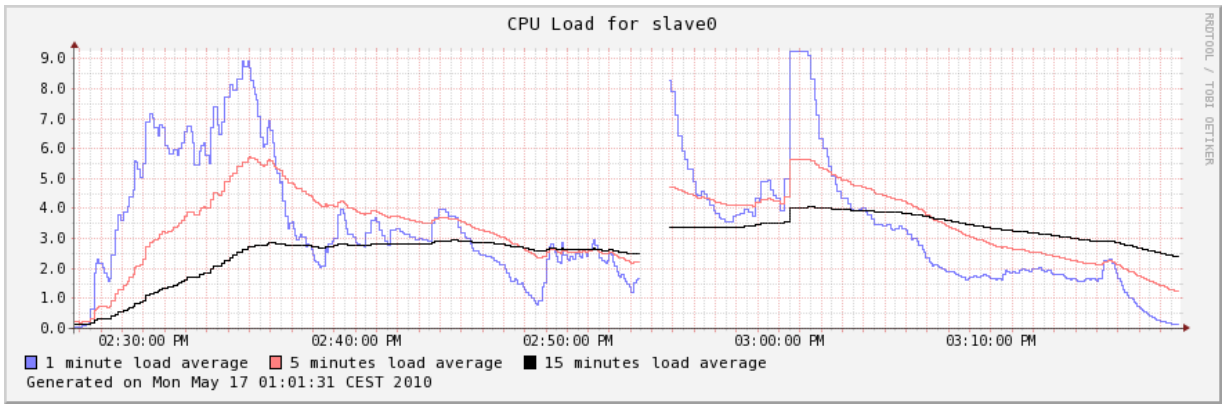
Below, an excerpt of the performance results obtained from the log-files of the test runs is listed. Some measurements depend on a time-measurement on two distinct node.

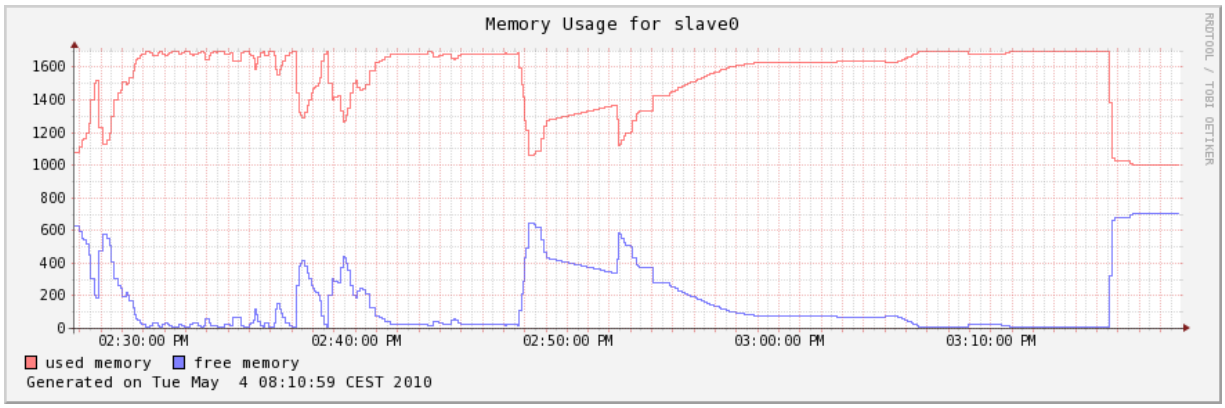
The clock deviation between the slave nodes and the master nodes have been estimated. It can safely be assumed, that the deviation of the clocks is below 1 second.

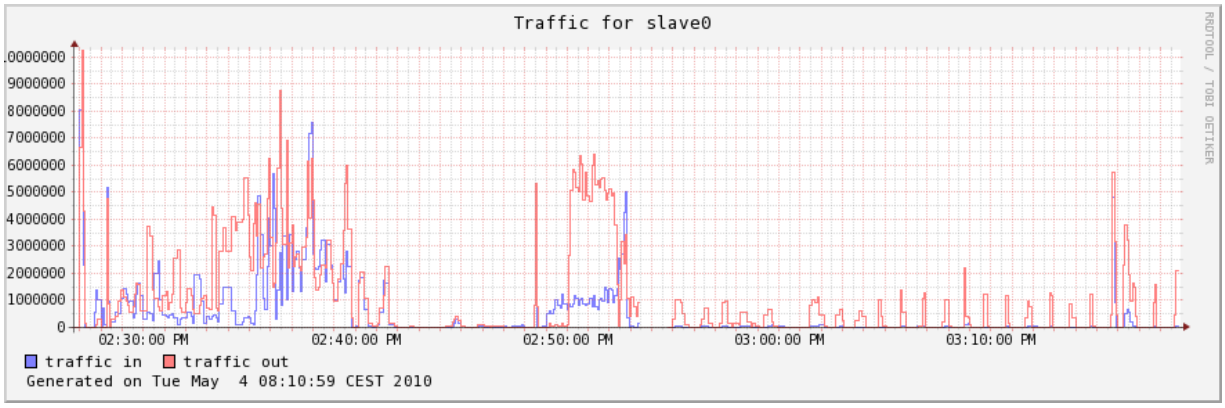
E.1 3 Nodes, Block Size 2097152 Bytes 2010-04-21 17:53:45

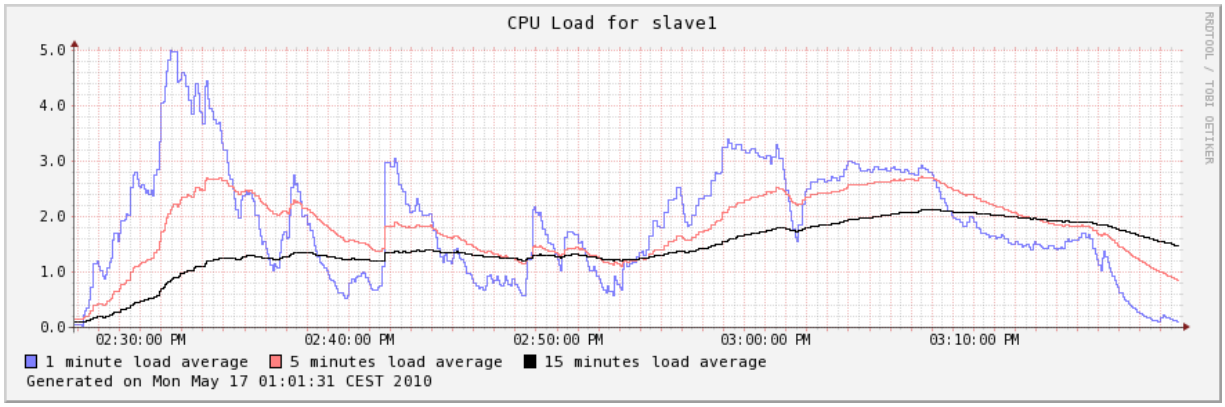
Import time		632.336 seconds
Processing time		2327.969 seconds
Names Mapper	Processing time	91.125 seconds
	Map started after	50.237 seconds
	#tasks	1
	#tasks on slave 2	0
	#tasks on slave 0	1
	#tasks on slave 1	0
Merge Stations by Distance	Processing time	525.17 seconds
	Map started after	39.596 seconds
	#tasks	11
	#tasks on slave 2	4
	#tasks on slave 0	4
	#tasks on slave 1	3
Distributed Routes Extractor	Processing time	1711.693 seconds
	Map started after	44.362 seconds
	#tasks	777
	#tasks on slave 2	170

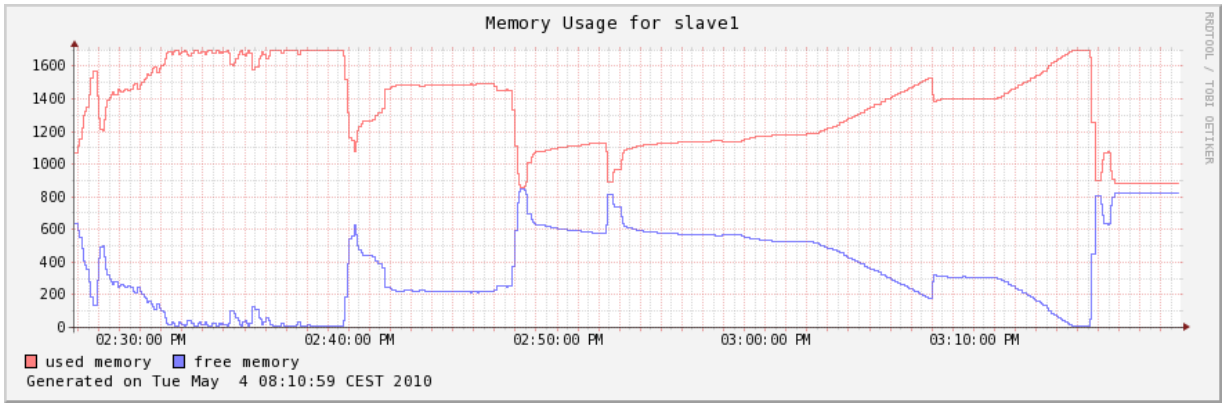
#tasks on slave 0	304
#tasks on slave 1	303
#tasks stage 0	2262
#tasks on slave 2 stage 0	163
#tasks on slave 0 stage 0	298
#tasks on slave 1 stage 0	293
#tasks stage 1	19
#tasks on slave 2 stage 1	7
#tasks on slave 0 stage 1	6
#tasks on slave 1 stage 1	6
#tasks stage 2	4
#tasks on slave 2 stage 2	0
#tasks on slave 0 stage 2	0
#tasks on slave 1 stage 2	4

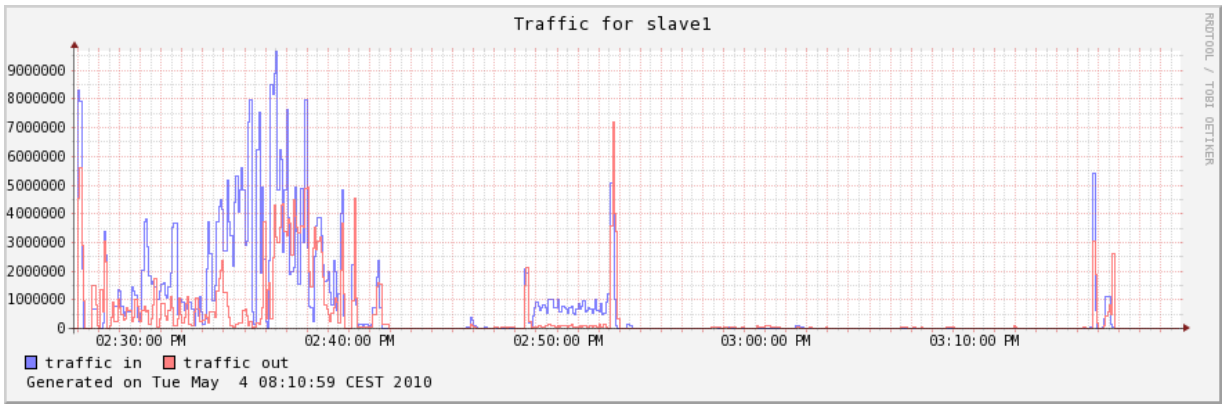


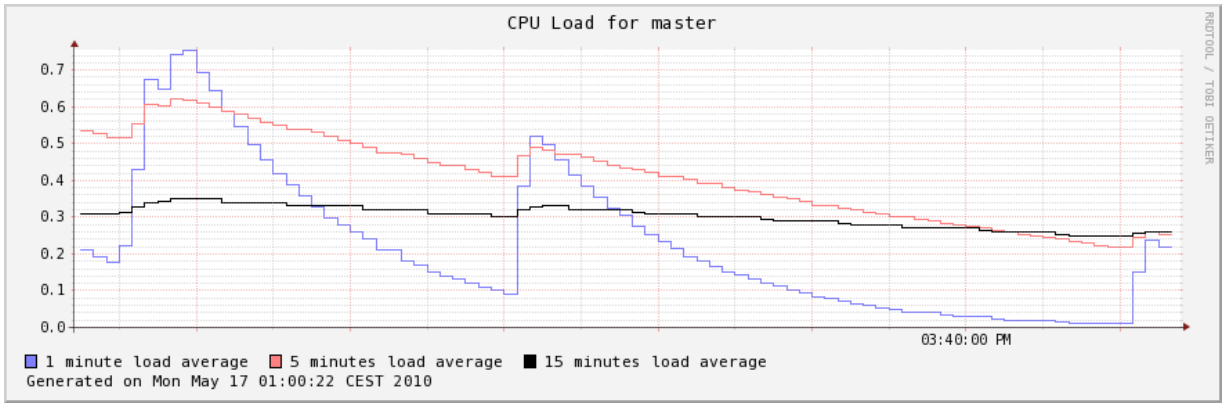


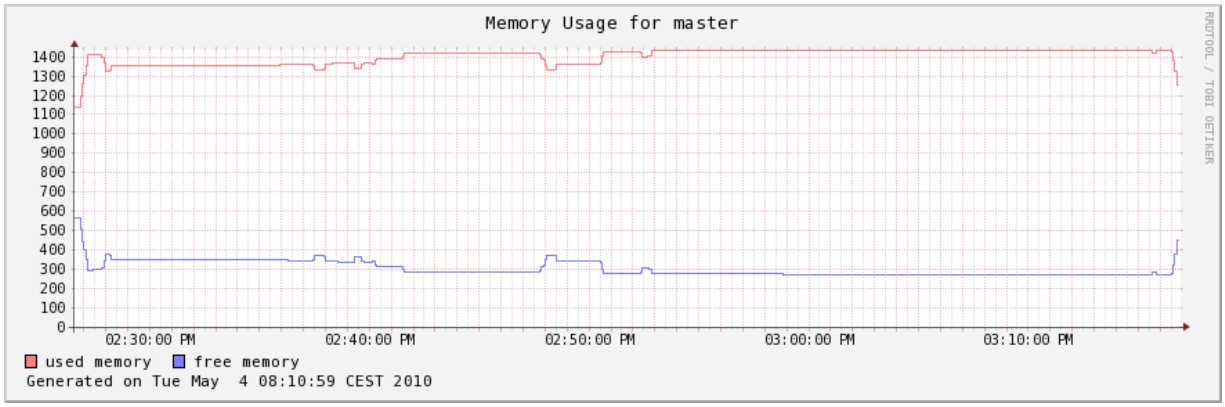

Traffic for slave0
9000000
8000000
7000000
6000000
5000000
4000000
3000000
2000000
1000000
0
02:30:00 PM
02:40:00 PM
02:50:00 PM
03:00:00 PM
03:10:00 PM
traffic in
traffic out
Generated on Tue May 4 08:10:59 CEST 2010
RRDTOOL / TOBI OETIKER

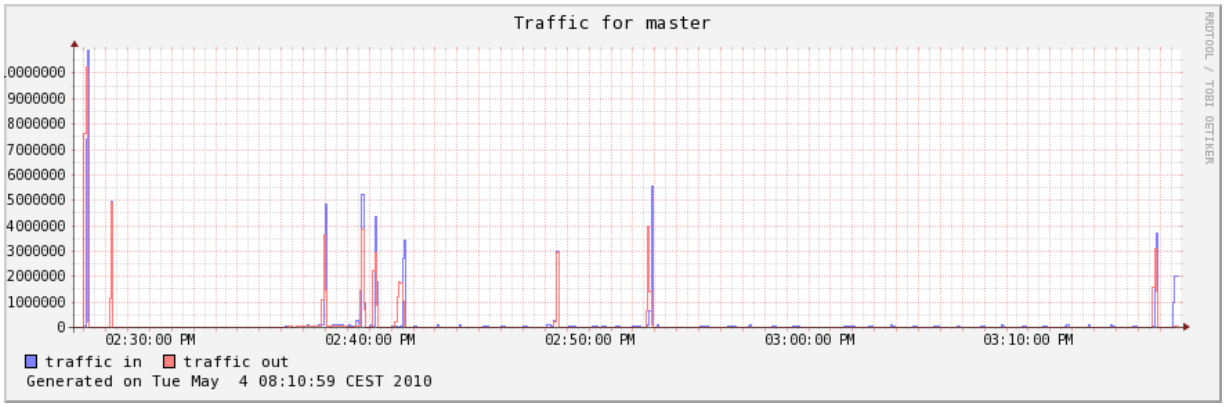

CPU Load for slave1
5.0
4.0
3.0
2.0
1.0
0.0
02:30:00 PM
02:40:00 PM
02:50:00 PM
03:00:00 PM
03:10:00 PM
1 minute load average
5 minutes load average
15 minutes load average
Generated on Mon May 17 01:01:31 CEST 2010
RRDTOOL / TOBI OETIKER

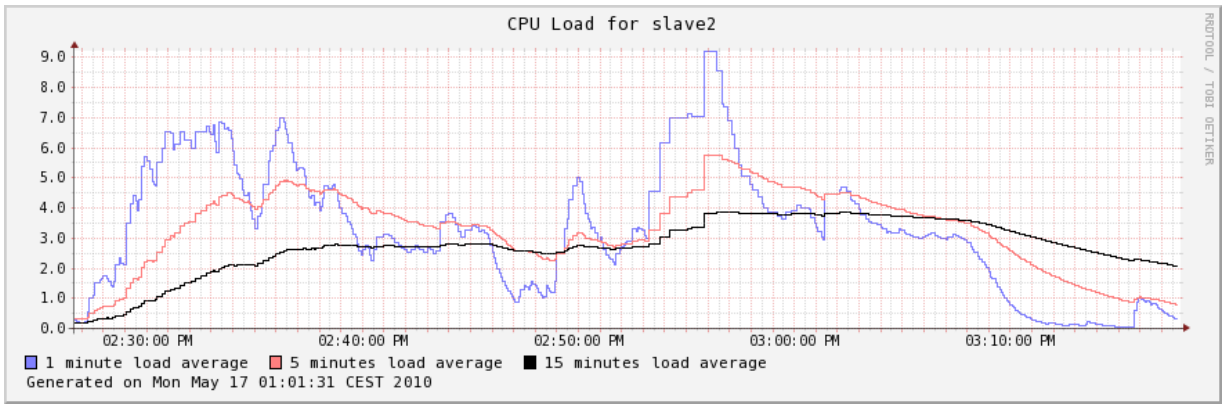

Memory Usage for slave1
1600
1400
1200
1000
800
600
400
200
0
02:30:00 PM
02:40:00 PM
02:50:00 PM
03:00:00 PM
03:10:00 PM
used memory
free memory
Generated on Tue May 4 08:10:59 CEST 2010
RRDTOOL / TOBI OETIKER

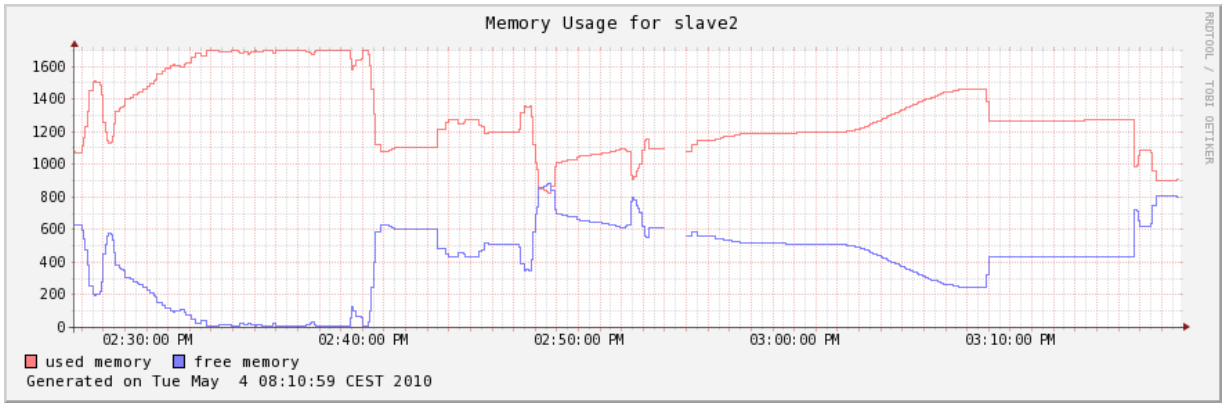

Traffic for slave1
9000000
8000000
7000000
6000000
5000000
4000000
3000000
2000000
1000000
0
02:30:00 PM
02:40:00 PM
02:50:00 PM
03:00:00 PM
03:10:00 PM
traffic in
traffic out
Generated on Tue May 4 08:10:59 CEST 2010
RRDTOOL / TOBI OETIKER

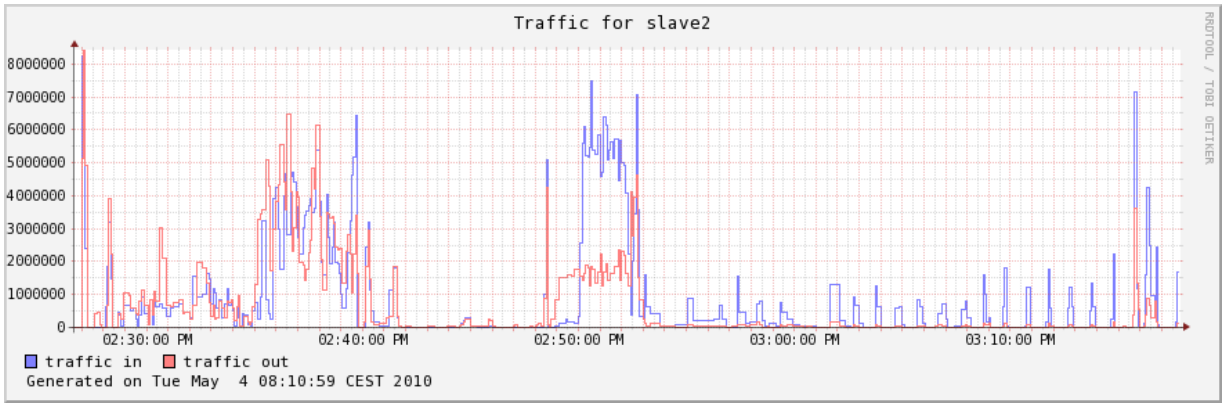

CPU Load for master
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
03:40:00 PM
1 minute load average
5 minutes load average
15 minutes load average
Generated on Mon May 17 01:00:22 CEST 2010
RRDTOOL / TOBI OETIKER


Memory Usage for master
1400
1300
1200
1100
1000
900
800
700
600
500
400
300
200
100
0
02:30:00 PM
02:40:00 PM
02:50:00 PM
03:00:00 PM
03:10:00 PM
used memory
free memory
Generated on Tue May 4 08:10:59 CEST 2010
RRDTOOL / TOBI OETIKER


Traffic for master
9000000
8000000
7000000
6000000
5000000
4000000
3000000
2000000
1000000
0
02:30:00 PM
02:40:00 PM
02:50:00 PM
03:00:00 PM
03:10:00 PM
traffic in
traffic out
Generated on Tue May 4 08:10:59 CEST 2010
RRDTOOL / TOBI OETIKER

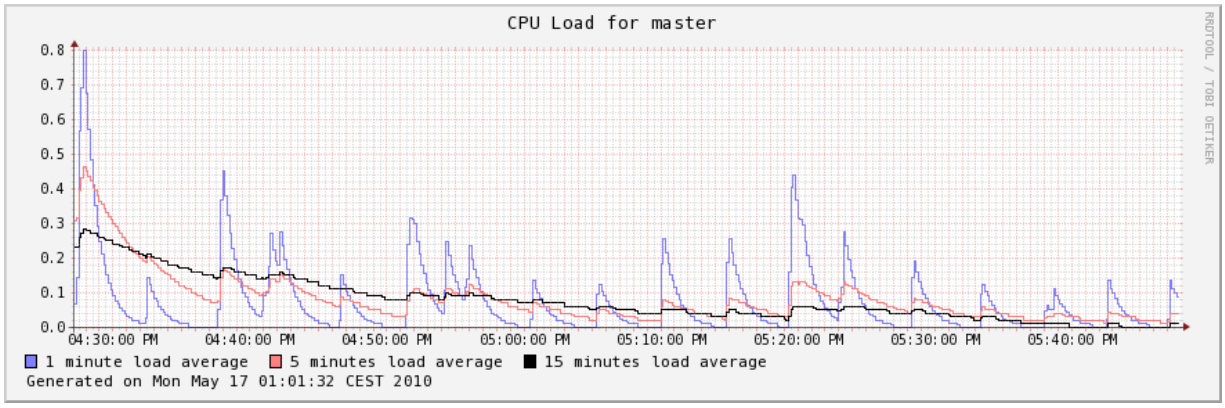

CPU Load for slave2
9.0
8.0
7.0
6.0
5.0
4.0
3.0
2.0
1.0
0.0
02:30:00 PM
02:40:00 PM
02:50:00 PM
03:00:00 PM
03:10:00 PM
1 minute load average
5 minutes load average
15 minutes load average
Generated on Mon May 17 01:01:31 CEST 2010
RRDTOOL / TOBI OETIKER

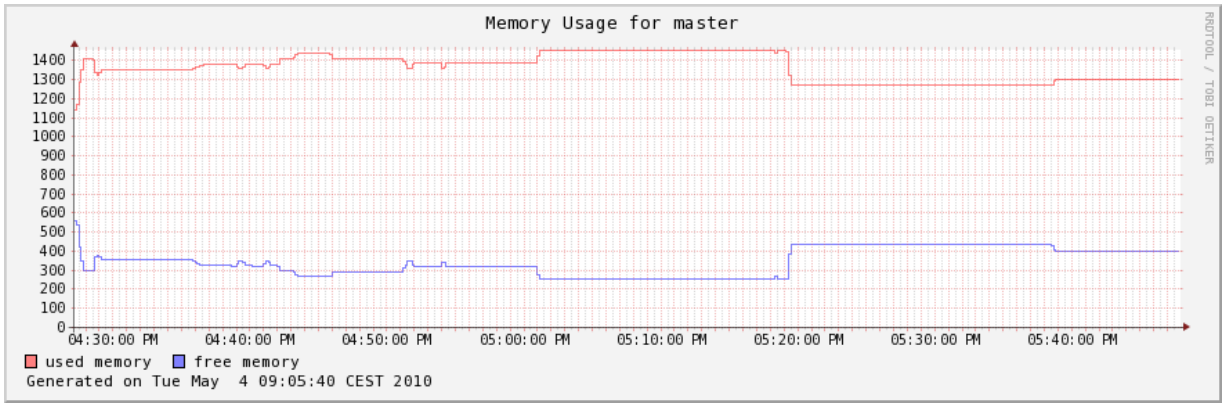

Memory Usage for slave2
1600
1400
1200
1000
800
600
400
200
0
02:30:00 PM
02:40:00 PM
02:50:00 PM
03:00:00 PM
03:10:00 PM
used memory
free memory
Generated on Tue May 4 08:10:59 CEST 2010
RRDTOOL / TOBI OETIKER

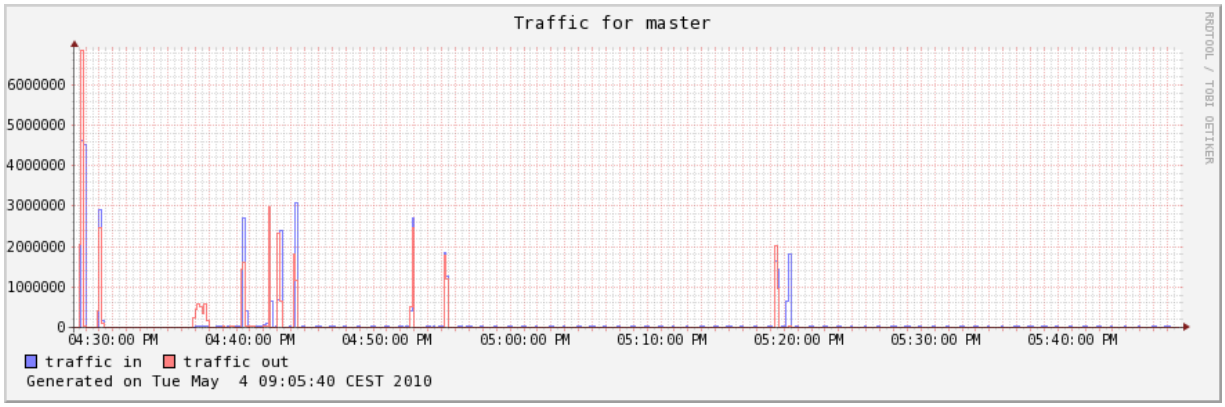

Traffic for slave2
8000000
7000000
6000000
5000000
4000000
3000000
2000000
1000000
0
02:30:00 PM
02:40:00 PM
02:50:00 PM
03:00:00 PM
03:10:00 PM
traffic in
traffic out
Generated on Tue May 4 08:10:59 CEST 2010
RRDTOOL / TOBI OETIKER

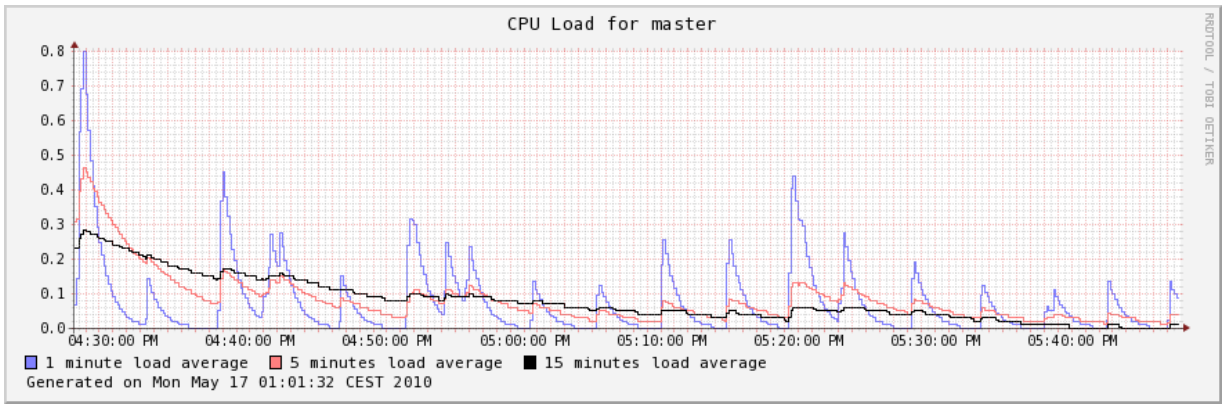
E.2 3 Nodes, Block Size 4194304 Bytes 2010-04-21 14:38:44

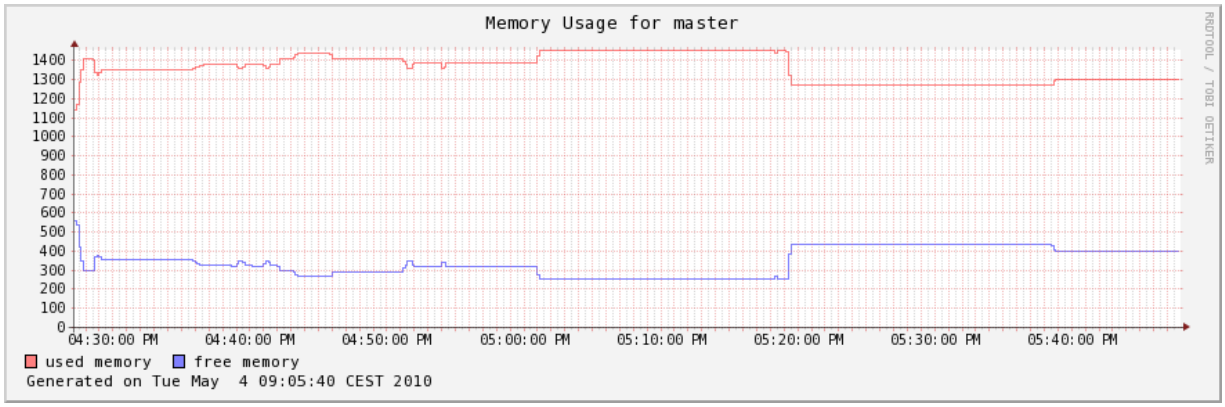
Import time		690.646 seconds
Processing time		2398.092 seconds
Names Mapper	Processing time	102.302 seconds
	Map started after	49.33 seconds
	#tasks	1
	#tasks on slave 2	0
	#tasks on slave 1	0
	#tasks on slave 0	1
Merge Stations by Distance	Processing time	620.486 seconds
	Map started after	44.633 seconds
	#tasks	7
	#tasks on slave 2	2
	#tasks on slave 1	2
	#tasks on slave 0	3
Distributed Routes Extractor	Processing time	1675.368 seconds
	Map started after	46.374 seconds
	#tasks	363
	#tasks on slave 2	132
	#tasks on slave 1	114
	#tasks on slave 0	117
	#tasks stage 0	1041
	#tasks on slave 2 stage 0	125
	#tasks on slave 1 stage 0	109
	#tasks on slave 0 stage 0	113
	#tasks stage 1	12
	#tasks on slave 2 stage 1	3
	#tasks on slave 1 stage 1	5
	#tasks on slave 0 stage 1	4
	#tasks stage 2	4
	#tasks on slave 2 stage 2	4
	#tasks on slave 1 stage 2	0
	#tasks on slave 0 stage 2	0

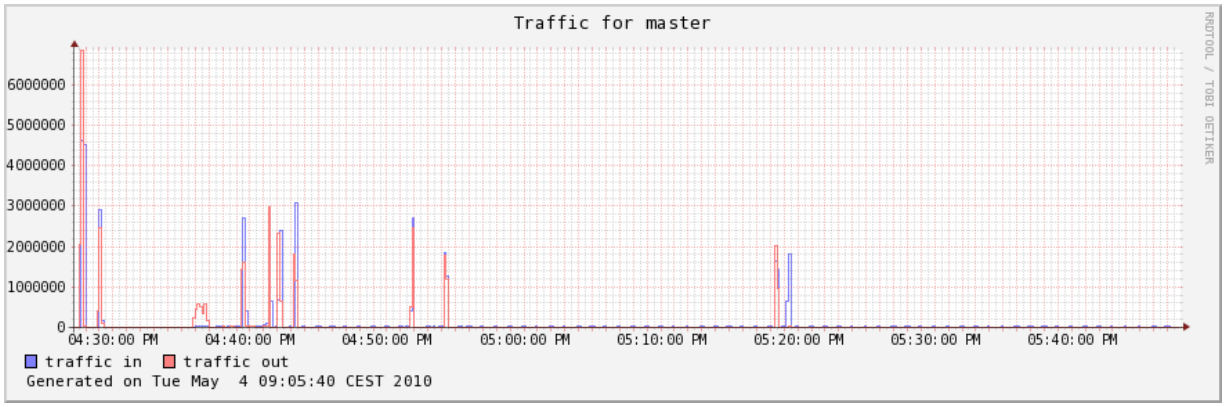

CPU Load for master
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
04:30:00 PM
04:40:00 PM
04:50:00 PM
05:00:00 PM
05:10:00 PM
05:20:00 PM
05:30:00 PM
05:40:00 PM
1 minute load average
5 minutes load average
15 minutes load average
Generated on Mon May 17 01:01:32 CEST 2010
RRDTOOL / TOBI OETIKER

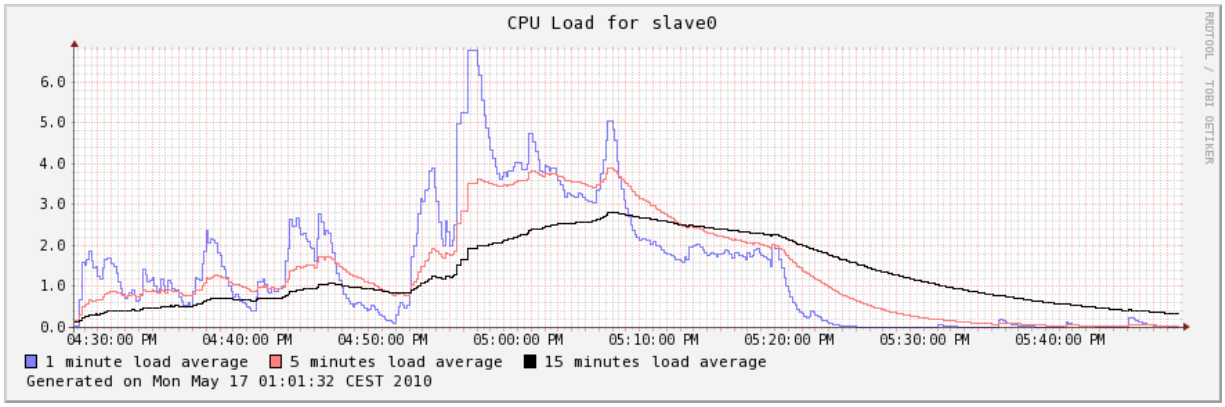

Memory Usage for master
1400
1300
1200
1100
1000
900
800
700
600
500
400
300
200
100
0
04:30:00 PM
04:40:00 PM
04:50:00 PM
05:00:00 PM
05:10:00 PM
05:20:00 PM
05:30:00 PM
05:40:00 PM
used memory
free memory
Generated on Tue May 4 09:05:40 CEST 2010
RRDTOOL / TOBI OETIKER

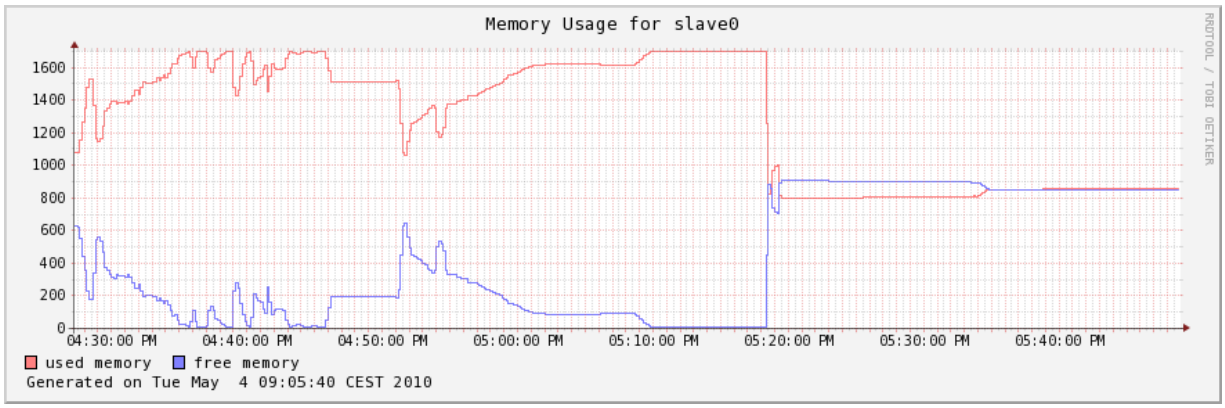

Traffic for master
6000000
5000000
4000000
3000000
2000000
1000000
0
04:30:00 PM
04:40:00 PM
04:50:00 PM
05:00:00 PM
05:10:00 PM
05:20:00 PM
05:30:00 PM
05:40:00 PM
traffic in
traffic out
Generated on Tue May 4 09:05:40 CEST 2010
RRDTOOL / TOBI OETIKER

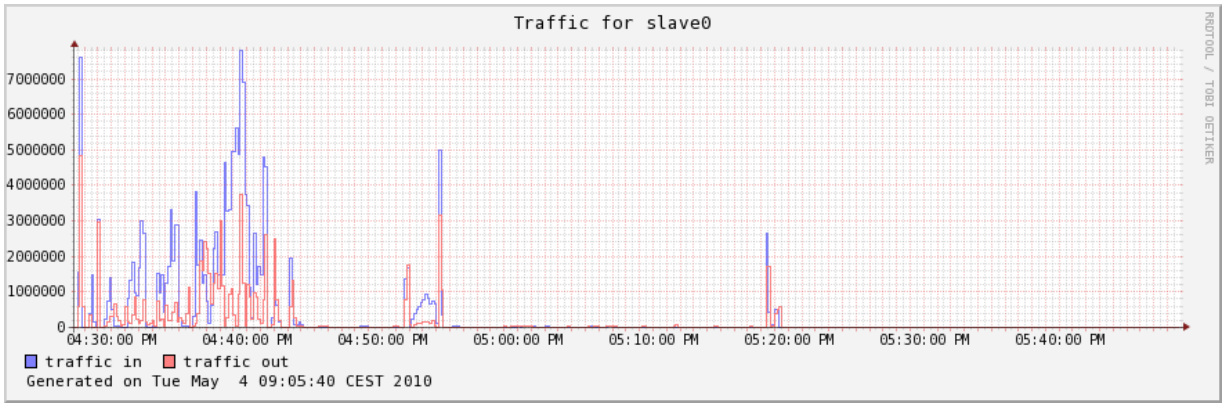

CPU Load for master
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
04:30:00 PM
04:40:00 PM
04:50:00 PM
05:00:00 PM
05:10:00 PM
05:20:00 PM
05:30:00 PM
05:40:00 PM
1 minute load average
5 minutes load average
15 minutes load average
Generated on Mon May 17 01:01:32 CEST 2010
RRDTOOL / TOBI OETIKER

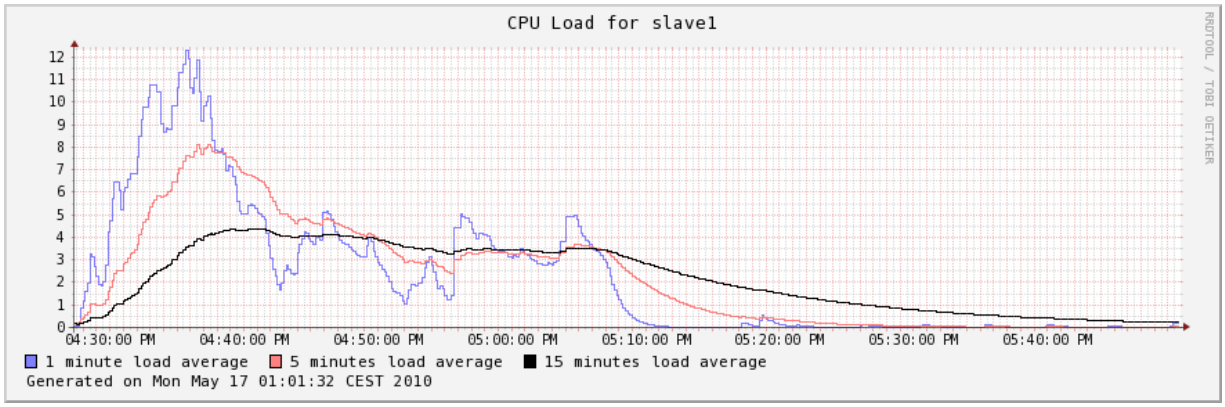

Memory Usage for master
1400
1300
1200
1100
1000
900
800
700
600
500
400
300
200
100
0
04:30:00 PM
04:40:00 PM
04:50:00 PM
05:00:00 PM
05:10:00 PM
05:20:00 PM
05:30:00 PM
05:40:00 PM
used memory
free memory
Generated on Tue May 4 09:05:40 CEST 2010
RRDTOOL / TOBI OETIKER

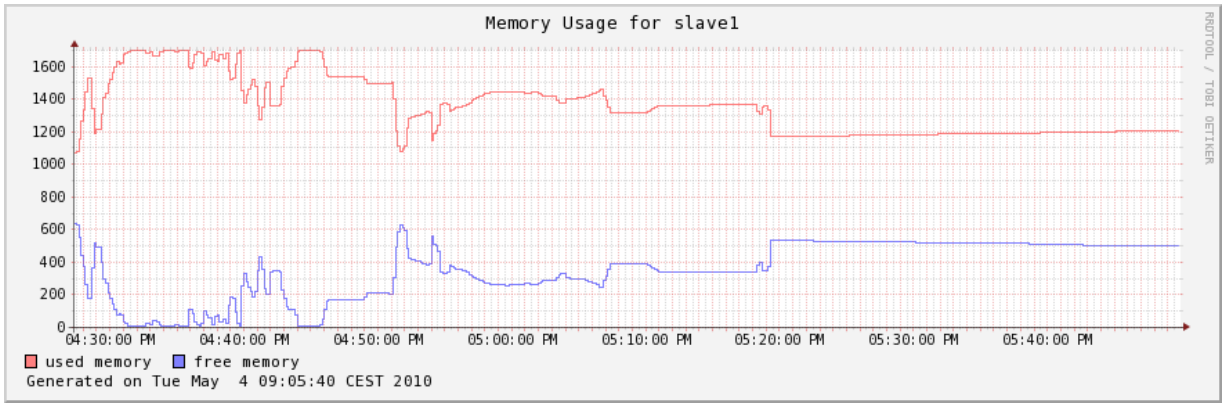

Traffic for master
6000000
5000000
4000000
3000000
2000000
1000000
0
04:30:00 PM
04:40:00 PM
04:50:00 PM
05:00:00 PM
05:10:00 PM
05:20:00 PM
05:30:00 PM
05:40:00 PM
traffic in
traffic out
Generated on Tue May 4 09:05:40 CEST 2010
RRDTOOL / TOBI OETIKER

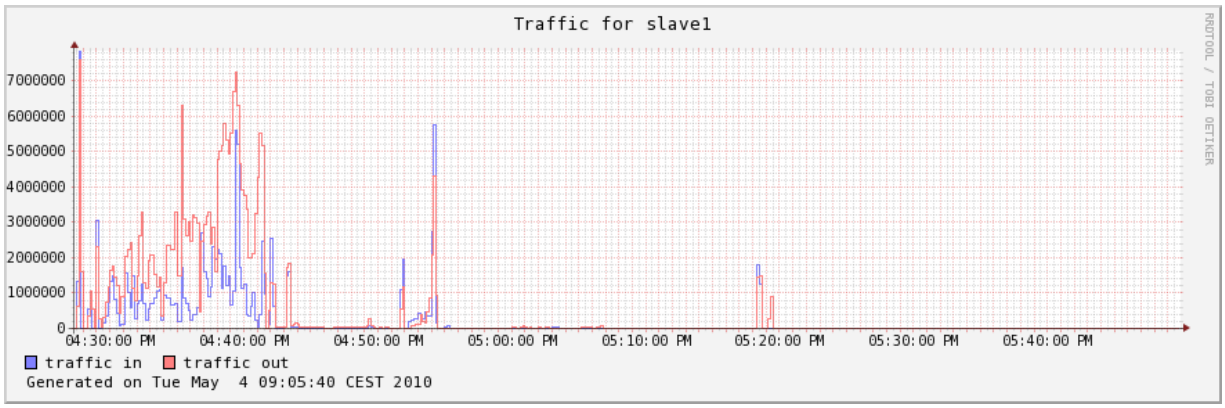

CPU Load for slave0
6.0
5.0
4.0
3.0
2.0
1.0
0.0
04:30:00 PM
04:40:00 PM
04:50:00 PM
05:00:00 PM
05:10:00 PM
05:20:00 PM
05:30:00 PM
05:40:00 PM
1 minute load average
5 minutes load average
15 minutes load average
Generated on Mon May 17 01:01:32 CEST 2010
RRDTOOL / TOBI OETIKER

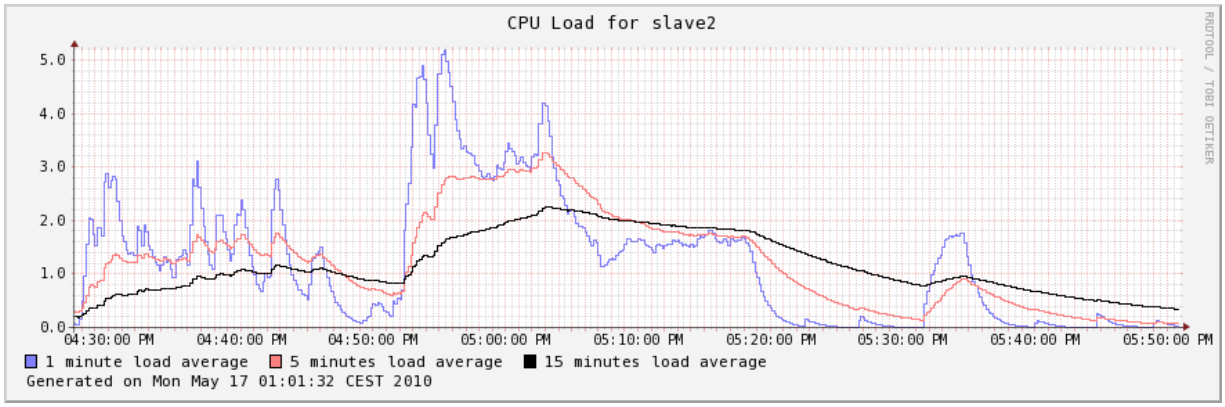

Memory Usage for slave0
1600
1400
1200
1000
800
600
400
200
0
04:30:00 PM
04:40:00 PM
04:50:00 PM
05:00:00 PM
05:10:00 PM
05:20:00 PM
05:30:00 PM
05:40:00 PM
used memory
free memory
Generated on Tue May 4 09:05:40 CEST 2010
RRDTOOL / TOBI OETIKER

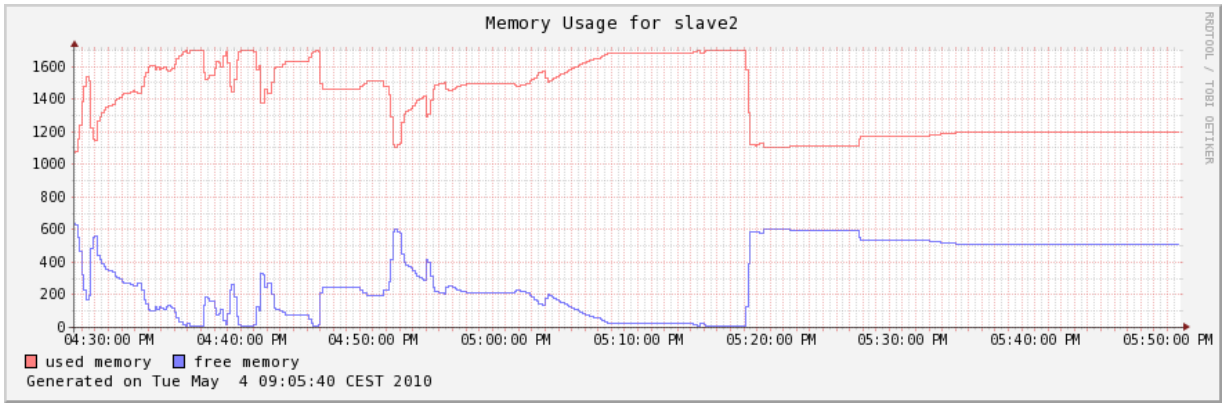

Traffic for slave0
7000000
6000000
5000000
4000000
3000000
2000000
1000000
0
04:30:00 PM
04:40:00 PM
04:50:00 PM
05:00:00 PM
05:10:00 PM
05:20:00 PM
05:30:00 PM
05:40:00 PM
traffic in
traffic out
Generated on Tue May 4 09:05:40 CEST 2010
RRDTOOL / TOBI OETIKER

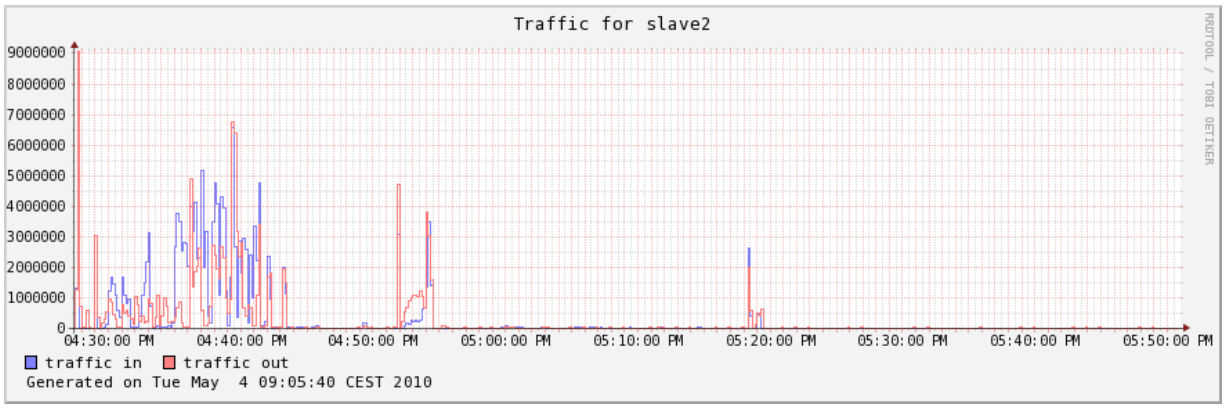

CPU Load for slave1
12
11
10
9
8
7
6
5
4
3
2
1
0
04:30:00 PM
04:40:00 PM
04:50:00 PM
05:00:00 PM
05:10:00 PM
05:20:00 PM
05:30:00 PM
05:40:00 PM
1 minute load average
5 minutes load average
15 minutes load average
Generated on Mon May 17 01:01:32 CEST 2010
RRDTOOL / TOBI OETIKER


Memory Usage for slave1
1600
1400
1200
1000
800
600
400
200
0
04:30:00 PM
04:40:00 PM
04:50:00 PM
05:00:00 PM
05:10:00 PM
05:20:00 PM
05:30:00 PM
05:40:00 PM
used memory
free memory
Generated on Tue May 4 09:05:40 CEST 2010
RRDTOOL / TOBI OETIKER


Traffic for slave1
7000000
6000000
5000000
4000000
3000000
2000000
1000000
0
04:30:00 PM
04:40:00 PM
04:50:00 PM
05:00:00 PM
05:10:00 PM
05:20:00 PM
05:30:00 PM
05:40:00 PM
traffic in
traffic out
Generated on Tue May 4 09:05:40 CEST 2010
RRDTOOL / TOBI OETIKER


CPU Load for slave2
5.0
4.0
3.0
2.0
1.0
0.0
04:30:00 PM
04:40:00 PM
04:50:00 PM
05:00:00 PM
05:10:00 PM
05:20:00 PM
05:30:00 PM
05:40:00 PM
05:50:00 PM
1 minute load average
5 minutes load average
15 minutes load average
Generated on Mon May 17 01:01:32 CEST 2010
RRDTOOL / TOBI OETIKER

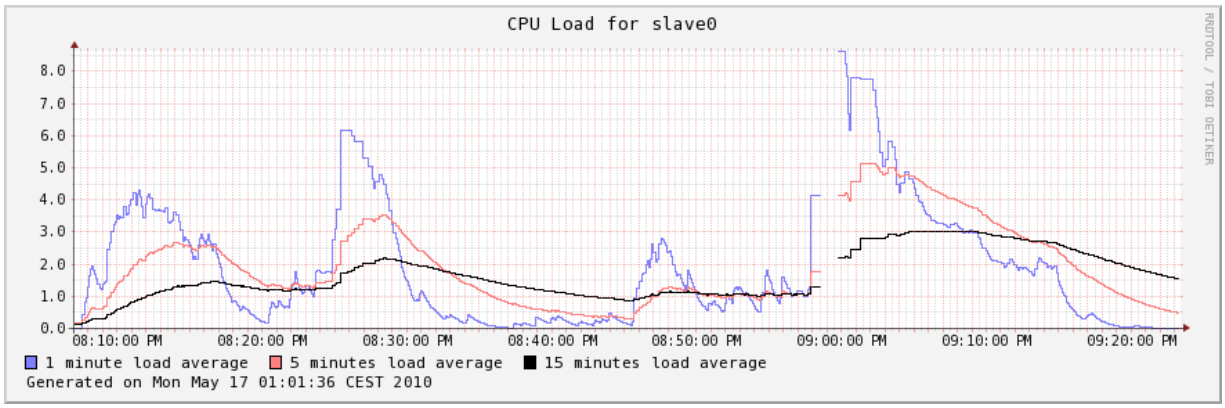

Memory Usage for slave2
1600
1400
1200
1000
800
600
400
200
0
04:30:00 PM
04:40:00 PM
04:50:00 PM
05:00:00 PM
05:10:00 PM
05:20:00 PM
05:30:00 PM
05:40:00 PM
05:50:00 PM
used memory
free memory
Generated on Tue May 4 09:05:40 CEST 2010
RRDTOOL / TOBI OETIKER

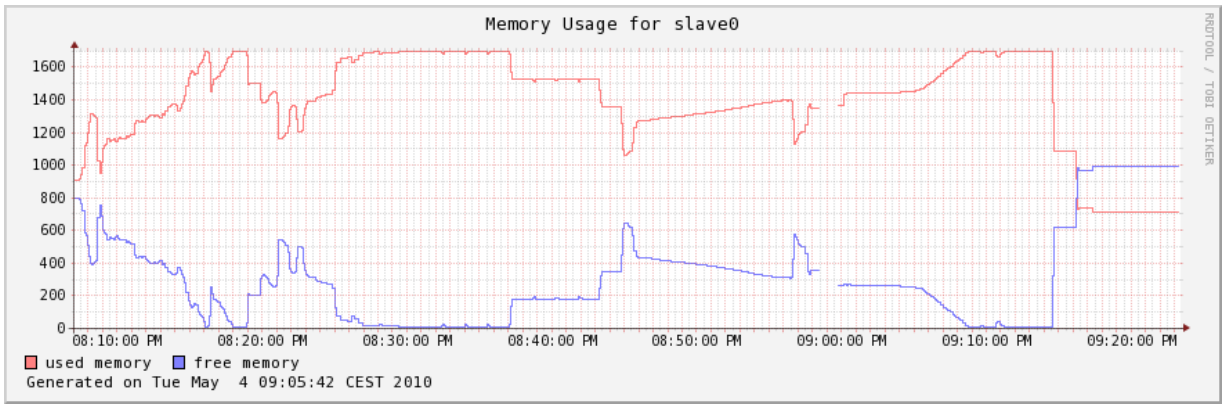

Traffic for slave2
9000000
8000000
7000000
6000000
5000000
4000000
3000000
2000000
1000000
0
04:30:00 PM
04:40:00 PM
04:50:00 PM
05:00:00 PM
05:10:00 PM
05:20:00 PM
05:30:00 PM
05:40:00 PM
05:50:00 PM
traffic in
traffic out
Generated on Tue May 4 09:05:40 CEST 2010
RRDTOOL / TOBI OETIKER

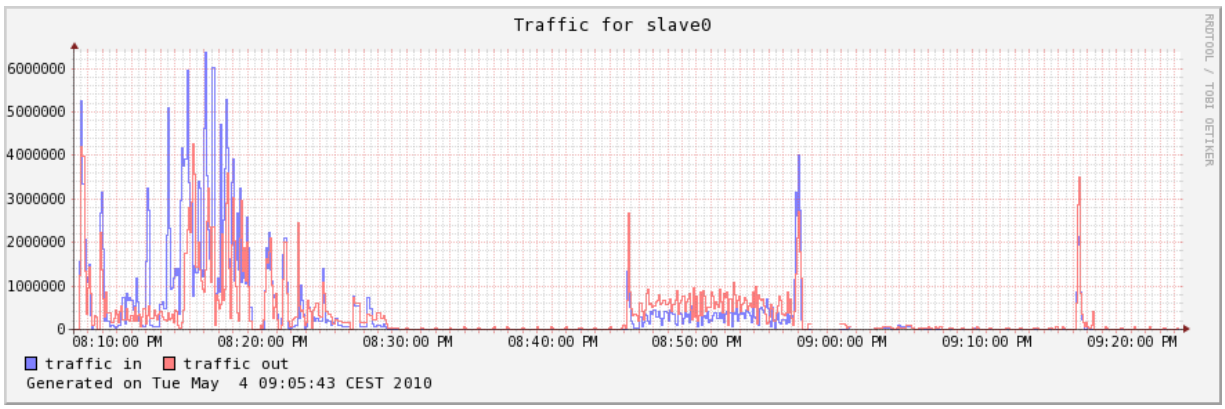
E.3 5 Nodes, Block Size 1048576 Bytes 2010-04-21 19:20:39

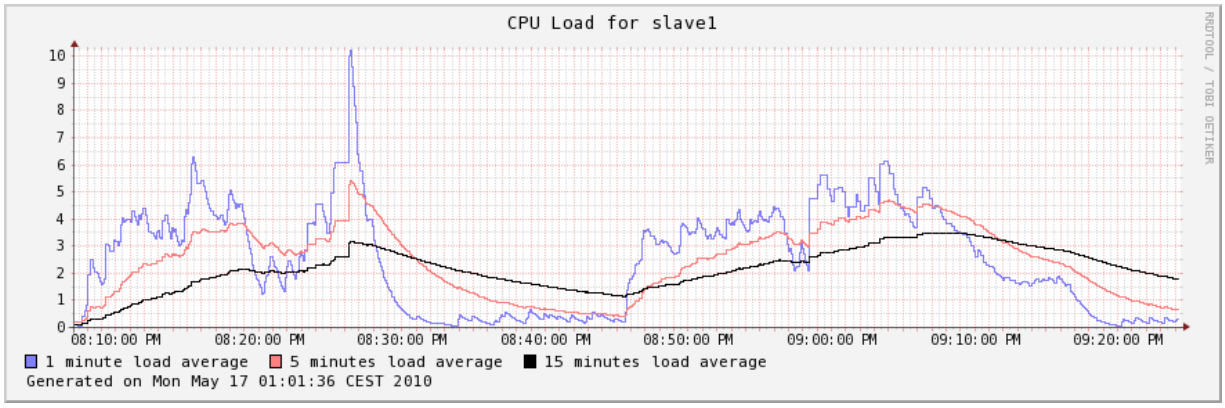
Import time		823.792 seconds
Processing time		3351.429 seconds
Names Mapper	Processing time	50.878 seconds
	Map started after	25.815 seconds
	#tasks	8
	#tasks on slave 2	2
	#tasks on slave 0	1
	#tasks on slave 3	1
	#tasks on slave 1	2
	#tasks on slave 4	2
Merge Stations by Distance	Processing time	1351.755 seconds
	Map started after	30.182 seconds
	#tasks	19
	#tasks on slave 2	3
	#tasks on slave 0	5
	#tasks on slave 3	3
	#tasks on slave 1	4
	#tasks on slave 4	4
Distributed Routes Extractor	Processing time	1948.796 seconds
	Map started after	51.897 seconds
	#tasks	1451
	#tasks on slave 2	342
	#tasks on slave 0	343
	#tasks on slave 3	322
	#tasks on slave 1	347
	#tasks on slave 4	97
	#tasks stage 0	4224
	#tasks on slave 2 stage 0	336
	#tasks on slave 0 stage 0	329
	#tasks on slave 3 stage 0	315
	#tasks on slave 1 stage 0	340
	#tasks on slave 4 stage 0	88
	#tasks stage 1	40
	#tasks on slave 2 stage 1	6
	#tasks on slave 0 stage 1	11
	#tasks on slave 3 stage 1	7
	#tasks on slave 1 stage 1	7

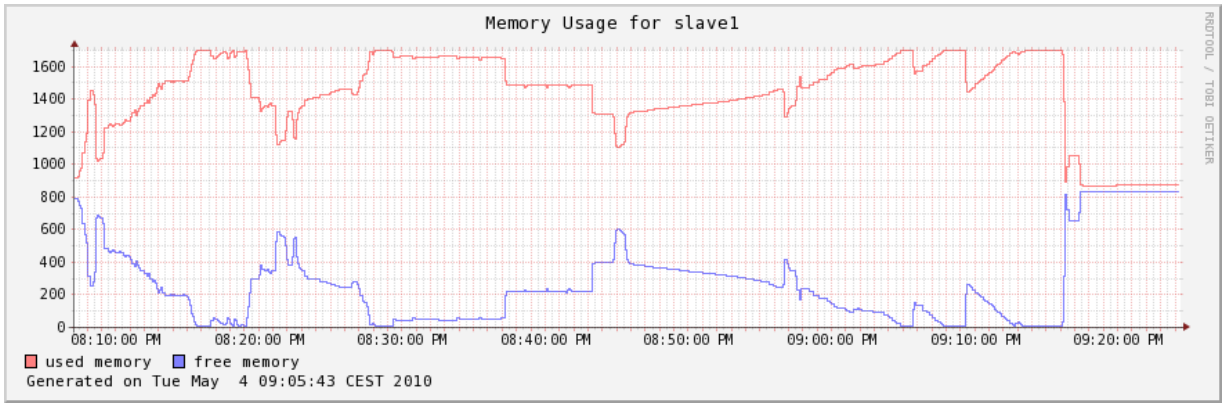
	#tasks on slave 4 stage 1	9
	#tasks stage 2	3
	#tasks on slave 2 stage 2	0
	#tasks on slave 0 stage 2	3
	#tasks on slave 3 stage 2	0
	#tasks on slave 1 stage 2	0
	#tasks on slave 4 stage 2	0

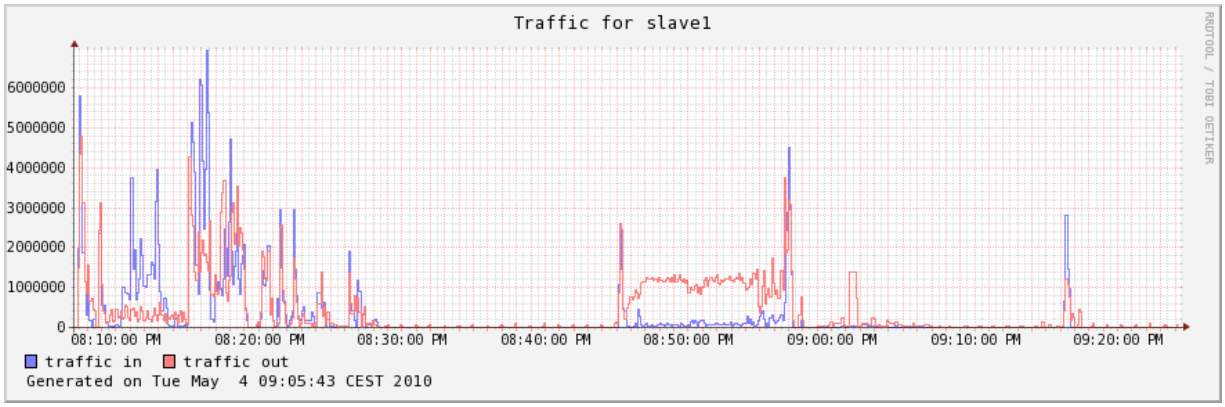


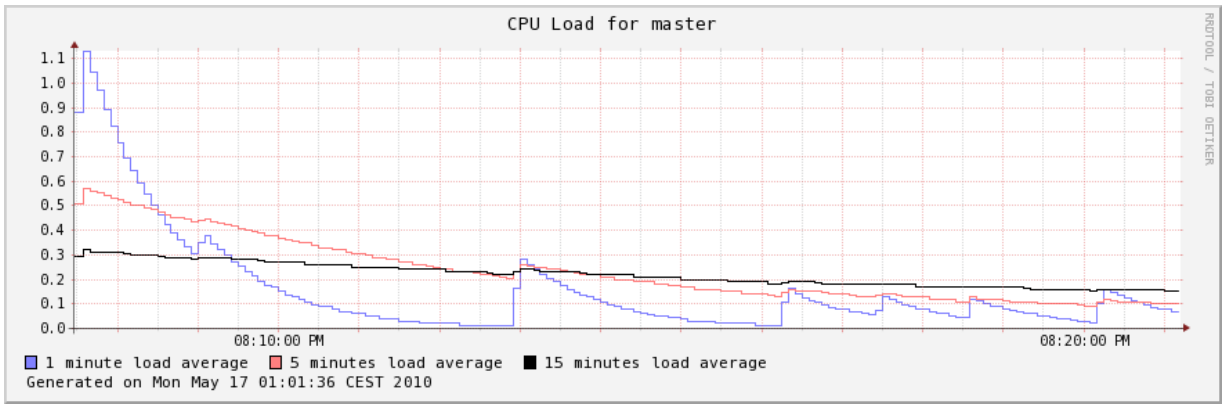


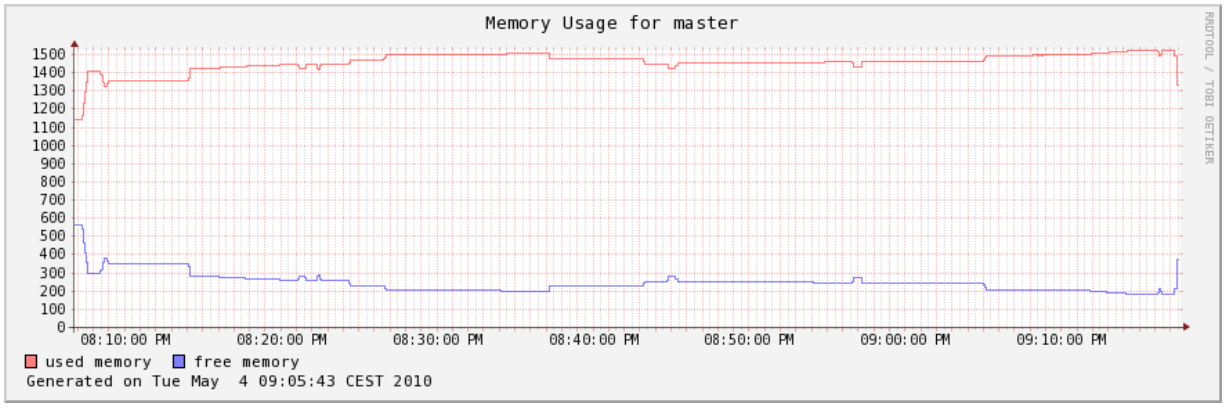


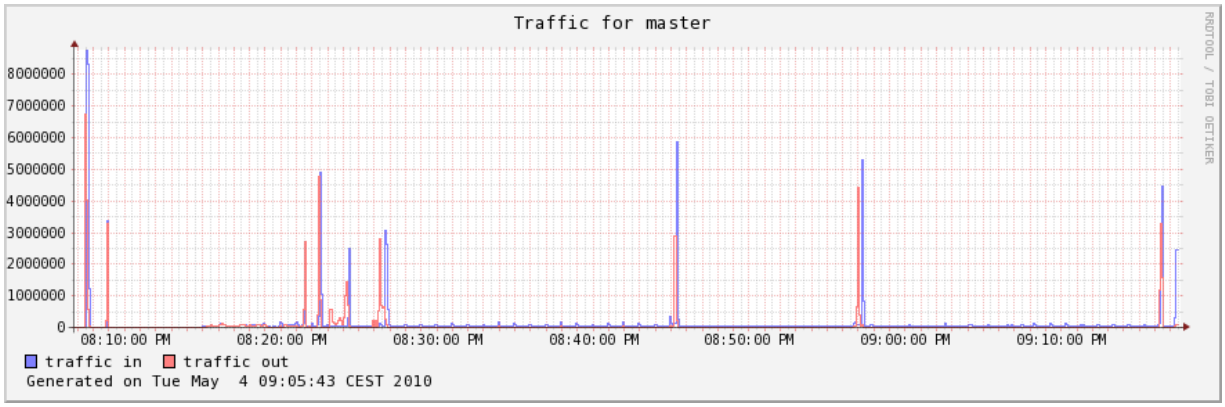

CPU Load for slave1
10
9
8
7
6
5
4
3
2
1
0
08:10:00 PM
08:20:00 PM
08:30:00 PM
08:40:00 PM
08:50:00 PM
09:00:00 PM
09:10:00 PM
09:20:00 PM
1 minute load average
5 minutes load average
15 minutes load average
Generated on Mon May 17 01:01:36 CEST 2010
RRDTOOL / TOBI OETIKER

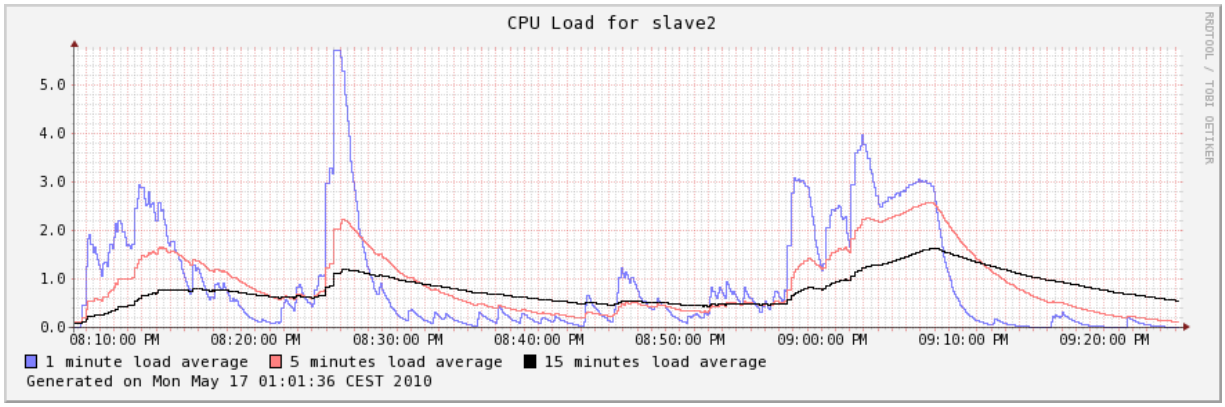

Memory Usage for slave1
1600
1400
1200
1000
800
600
400
200
0
08:10:00 PM
08:20:00 PM
08:30:00 PM
08:40:00 PM
08:50:00 PM
09:00:00 PM
09:10:00 PM
09:20:00 PM
used memory
free memory
Generated on Tue May 4 09:05:43 CEST 2010
RRDTOOL / TOBI OETIKER

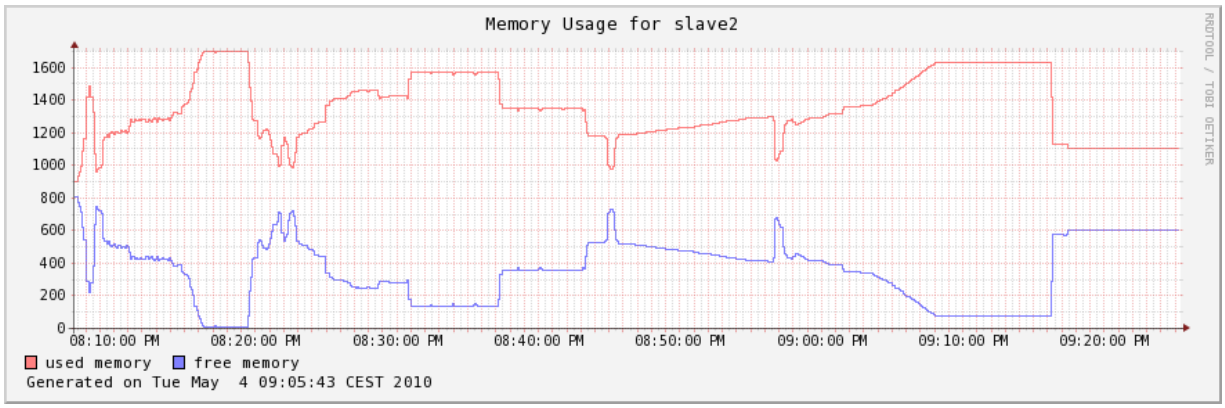

Traffic for slave1
6000000
5000000
4000000
3000000
2000000
1000000
0
08:10:00 PM
08:20:00 PM
08:30:00 PM
08:40:00 PM
08:50:00 PM
09:00:00 PM
09:10:00 PM
09:20:00 PM
traffic in
traffic out
Generated on Tue May 4 09:05:43 CEST 2010
RRDTOOL / TOBI OETIKER

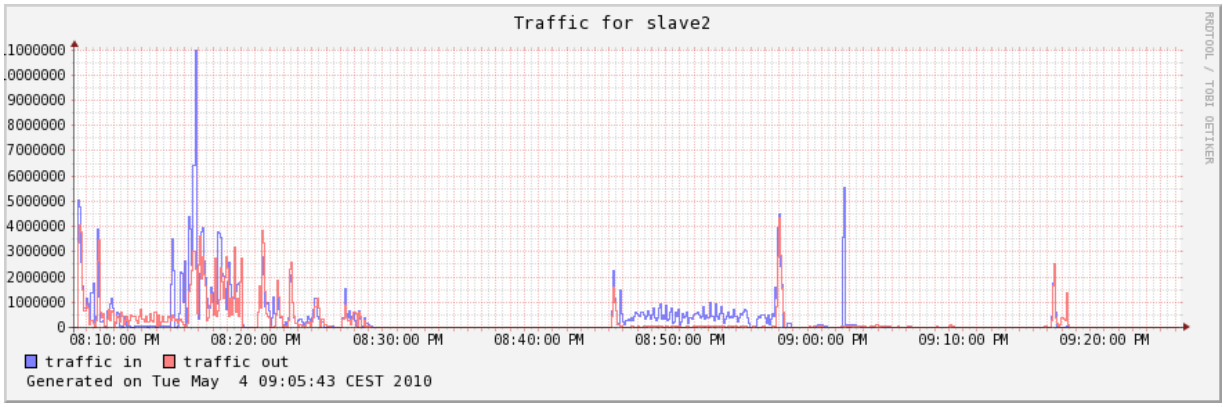

CPU Load for master
1.1
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
08:10:00 PM
08:20:00 PM
1 minute load average
5 minutes load average
15 minutes load average
Generated on Mon May 17 01:01:36 CEST 2010
RRDTOOL / TOBI OETIKER

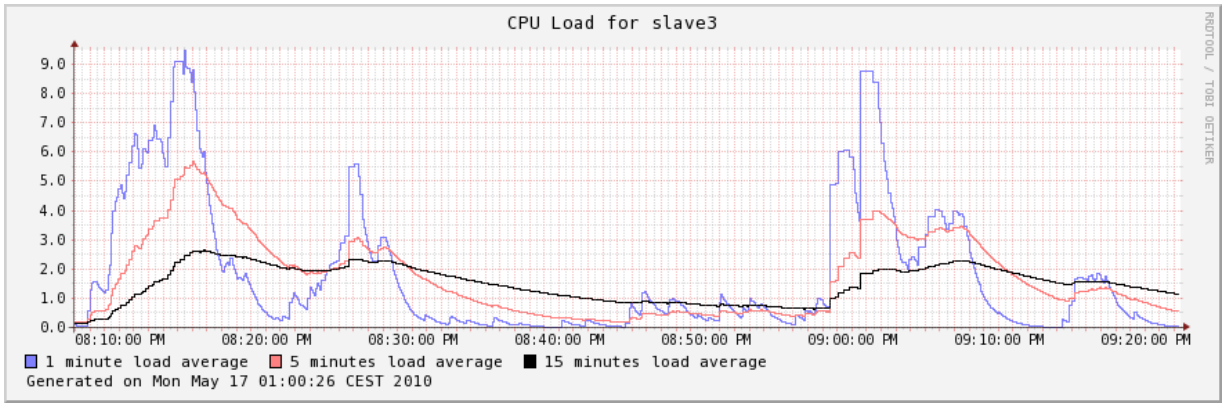

Memory Usage for master
1500
1400
1300
1200
1100
1000
900
800
700
600
500
400
300
200
100
0
08:10:00 PM
08:20:00 PM
08:30:00 PM
08:40:00 PM
08:50:00 PM
09:00:00 PM
09:10:00 PM
used memory
free memory
Generated on Tue May 4 09:05:43 CEST 2010
RRDTOOL / TOBI OETIKER

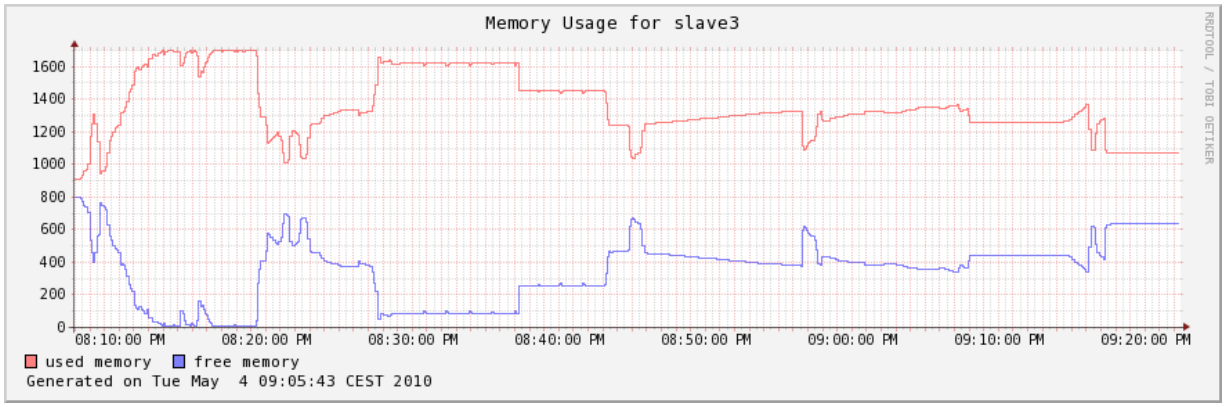

Traffic for master
8000000
7000000
6000000
5000000
4000000
3000000
2000000
1000000
0
08:10:00 PM
08:20:00 PM
08:30:00 PM
08:40:00 PM
08:50:00 PM
09:00:00 PM
09:10:00 PM
traffic in
traffic out
Generated on Tue May 4 09:05:43 CEST 2010
RRDTOOL / TOBI OETIKER

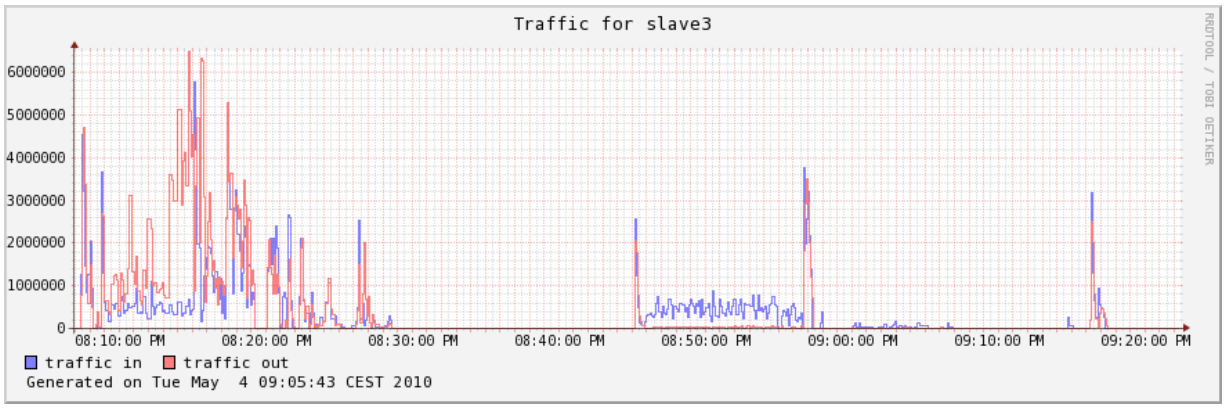

CPU Load for slave2
5.0
4.0
3.0
2.0
1.0
0.0
08:10:00 PM
08:20:00 PM
08:30:00 PM
08:40:00 PM
08:50:00 PM
09:00:00 PM
09:10:00 PM
09:20:00 PM
1 minute load average
5 minutes load average
15 minutes load average
Generated on Mon May 17 01:01:36 CEST 2010
RRDTOOL / TOBI OETIKER

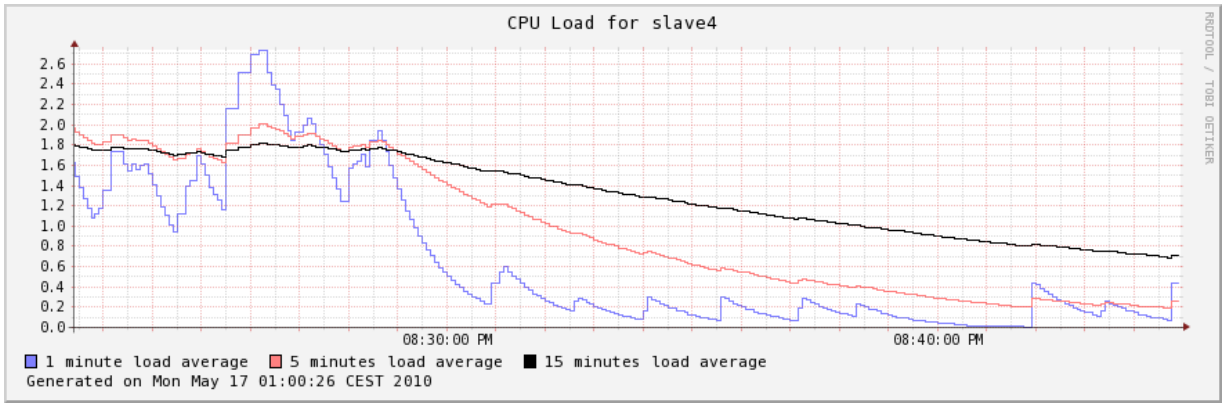

Memory Usage for slave2
1600
1400
1200
1000
800
600
400
200
0
08:10:00 PM
08:20:00 PM
08:30:00 PM
08:40:00 PM
08:50:00 PM
09:00:00 PM
09:10:00 PM
09:20:00 PM
used memory
free memory
Generated on Tue May 4 09:05:43 CEST 2010
RRDTOOL / TOBI OETIKER

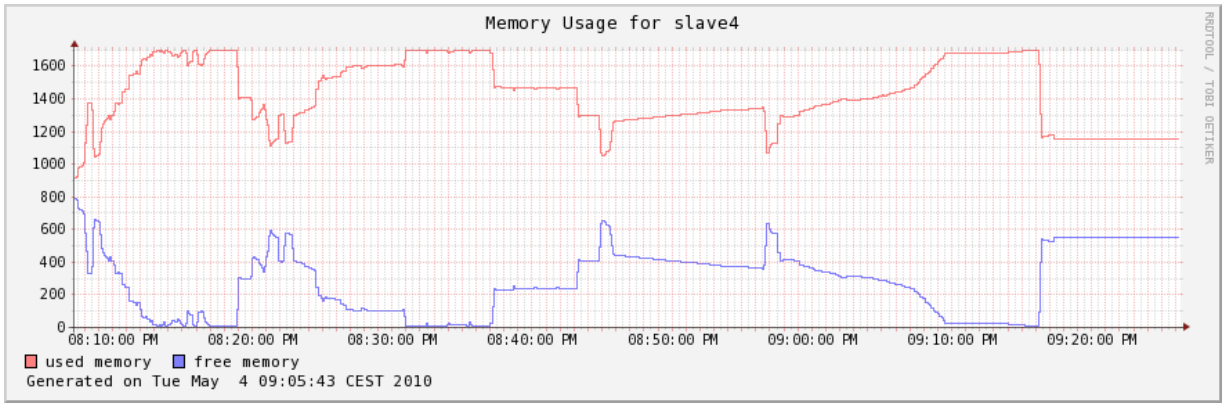

Traffic for slave2
1000000
0000000
9000000
8000000
7000000
6000000
5000000
4000000
3000000
2000000
1000000
0
08:10:00 PM
08:20:00 PM
08:30:00 PM
08:40:00 PM
08:50:00 PM
09:00:00 PM
09:10:00 PM
09:20:00 PM
traffic in
traffic out
Generated on Tue May 4 09:05:43 CEST 2010
RRDTOOL / TOBI OETIKER

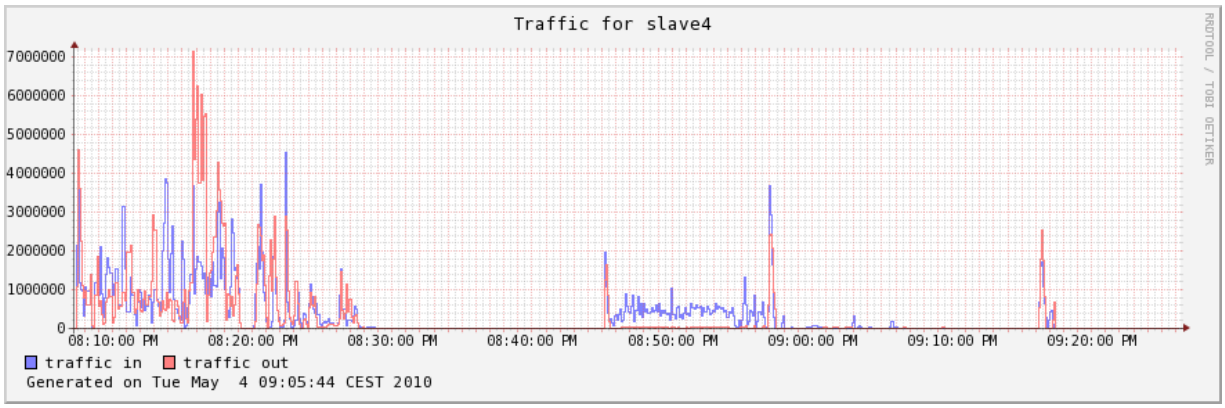

CPU Load for slave3
9.0
8.0
7.0
6.0
5.0
4.0
3.0
2.0
1.0
0.0
08:10:00 PM
08:20:00 PM
08:30:00 PM
08:40:00 PM
08:50:00 PM
09:00:00 PM
09:10:00 PM
09:20:00 PM
1 minute load average
5 minutes load average
15 minutes load average
Generated on Mon May 17 01:00:26 CEST 2010
RRDTOOL / TOBI OETIKER


Memory Usage for slave3
1600
1400
1200
1000
800
600
400
200
0
08:10:00 PM
08:20:00 PM
08:30:00 PM
08:40:00 PM
08:50:00 PM
09:00:00 PM
09:10:00 PM
09:20:00 PM
used memory
free memory
Generated on Tue May 4 09:05:43 CEST 2010
RRDTOOL / TOBI OETIKER


Traffic for slave3
6000000
5000000
4000000
3000000
2000000
1000000
0
08:10:00 PM
08:20:00 PM
08:30:00 PM
08:40:00 PM
08:50:00 PM
09:00:00 PM
09:10:00 PM
09:20:00 PM
traffic in
traffic out
Generated on Tue May 4 09:05:43 CEST 2010
RRDTOOL / TOBI OETIKER


CPU Load for slave4
2.6
2.4
2.2
2.0
1.8
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
08:30:00 PM
08:40:00 PM
1 minute load average
5 minutes load average
15 minutes load average
Generated on Mon May 17 01:00:26 CEST 2010
RRDTOOL / TOBI OETIKER

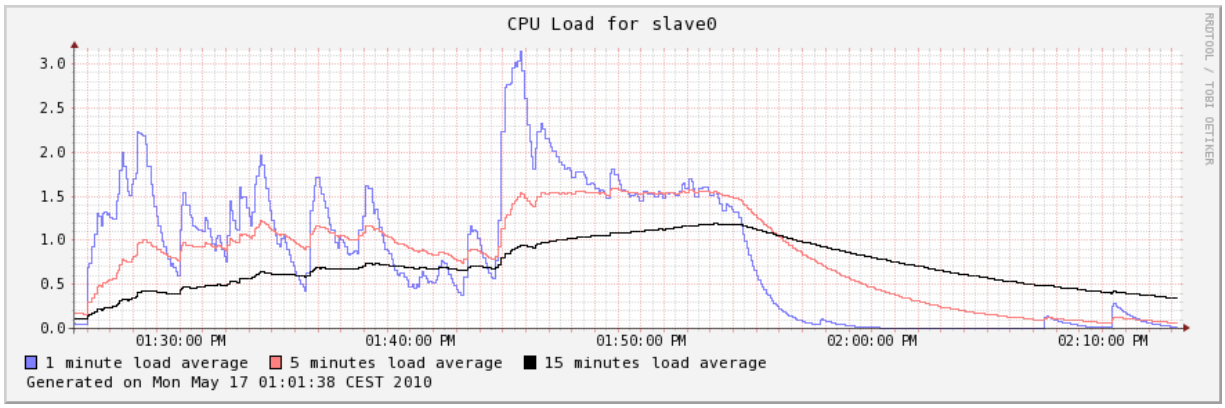

Memory Usage for slave4
1600
1400
1200
1000
800
600
400
200
0
08:10:00 PM
08:20:00 PM
08:30:00 PM
08:40:00 PM
08:50:00 PM
09:00:00 PM
09:10:00 PM
09:20:00 PM
used memory
free memory
Generated on Tue May 4 09:05:43 CEST 2010
RRDTOOL / TOBI OETIKER

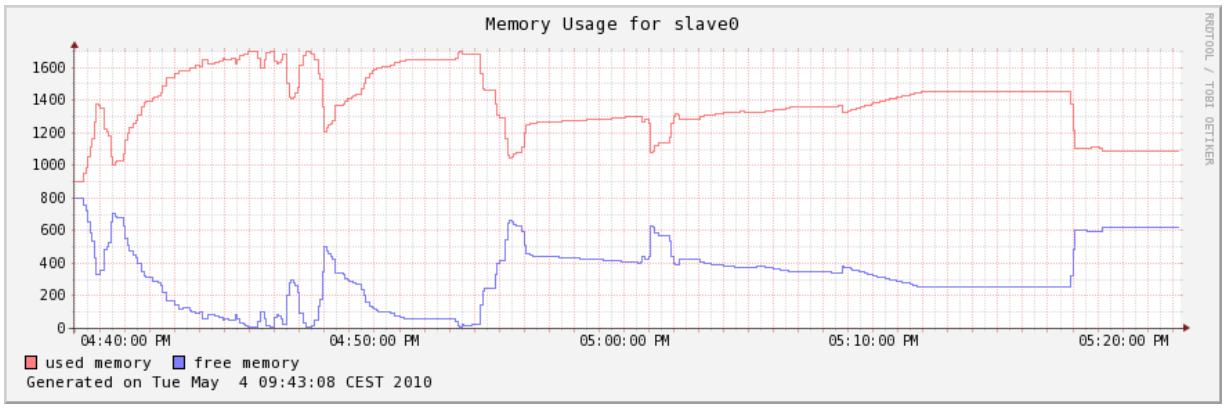

Traffic for slave4
7000000
6000000
5000000
4000000
3000000
2000000
1000000
0
08:10:00 PM
08:20:00 PM
08:30:00 PM
08:40:00 PM
08:50:00 PM
09:00:00 PM
09:10:00 PM
09:20:00 PM
traffic in
traffic out
Generated on Tue May 4 09:05:44 CEST 2010
RRDTOOL / TOBI OETIKER

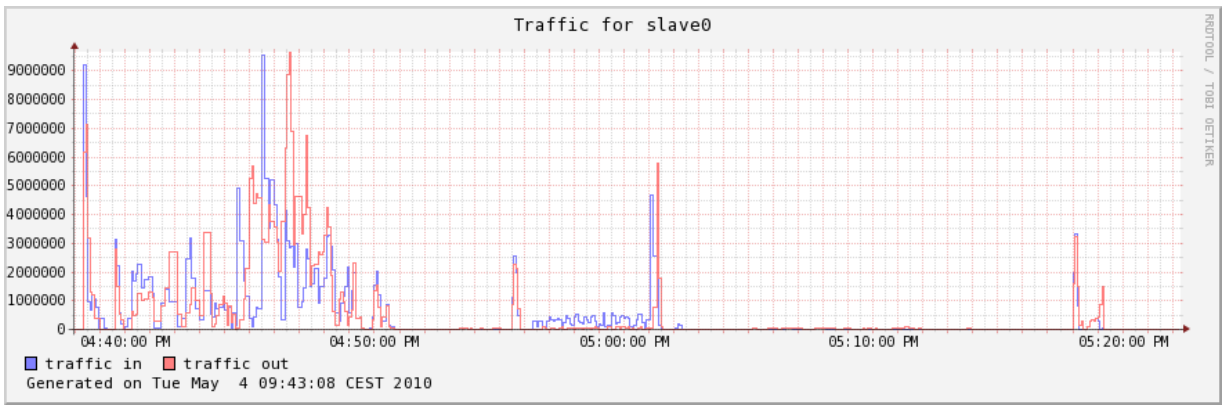
E.4 5 Nodes, Block Size 2097152 Bytes 2010-04-21 20:21:58

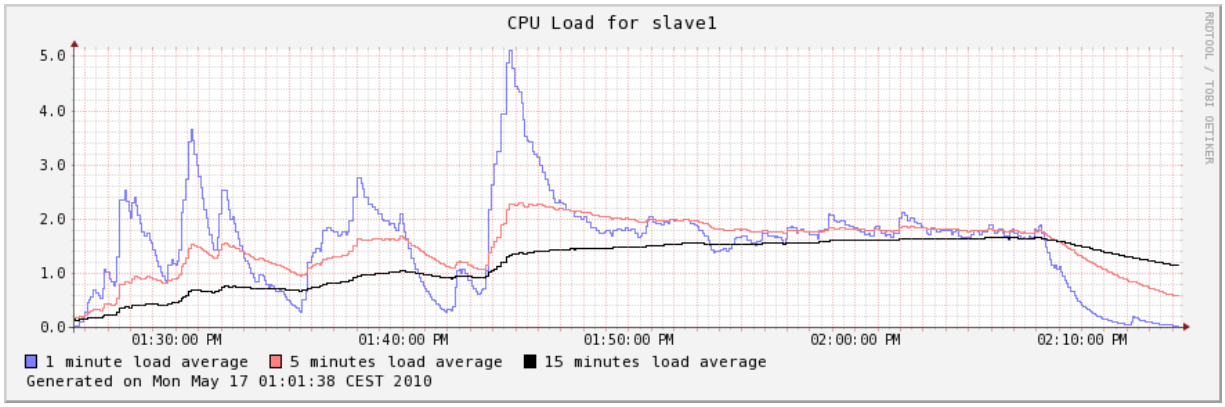
Import time		458.809 seconds
Processing time		1966.95 seconds
Names Mapper	Processing time	81.906 seconds
	Map started after	47.492 seconds
	#tasks	5
	#tasks on slave 0	1
	#tasks on slave 3	1
	#tasks on slave 2	1
	#tasks on slave 1	2
	#tasks on slave 4	0
Merge Stations by Distance	Processing time	453.042 seconds
	Map started after	53.714 seconds
	#tasks	11
	#tasks on slave 0	2
	#tasks on slave 3	2
	#tasks on slave 2	2
	#tasks on slave 1	3
	#tasks on slave 4	2
Distributed Routes Extractor	Processing time	1431.984 seconds
	Map started after	44.972 seconds
	#tasks	779
	#tasks on slave 0	219
	#tasks on slave 3	182
	#tasks on slave 2	183
	#tasks on slave 1	64
	#tasks on slave 4	131
	#tasks stage 0	2265
	#tasks on slave 0 stage 0	215
	#tasks on slave 3 stage 0	177
	#tasks on slave 2 stage 0	179
	#tasks on slave 1 stage 0	56
	#tasks on slave 4 stage 0	128
	#tasks stage 1	20
	#tasks on slave 0 stage 1	4
	#tasks on slave 3 stage 1	5
	#tasks on slave 2 stage 1	4
	#tasks on slave 1 stage 1	4

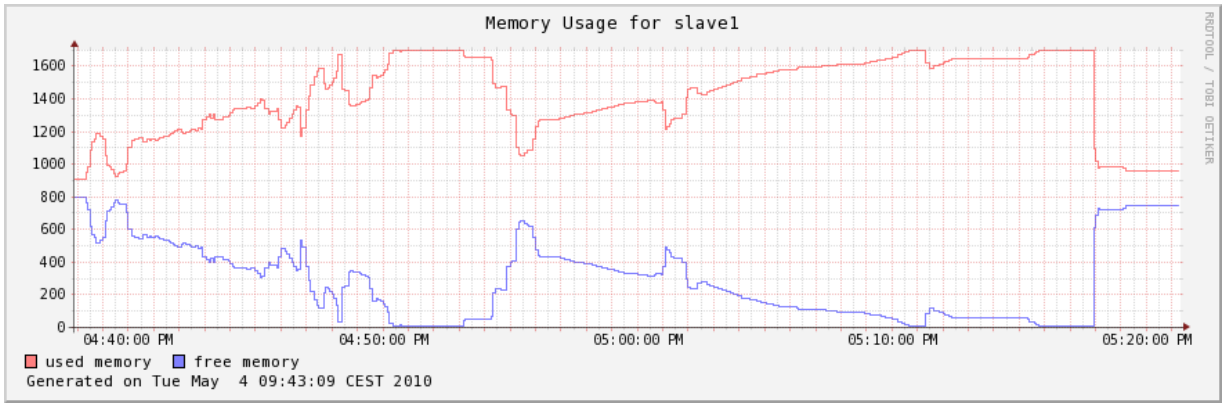
	#tasks on slave 4 stage 1	3
	#tasks stage 2	4
	#tasks on slave 0 stage 2	0
	#tasks on slave 3 stage 2	0
	#tasks on slave 2 stage 2	0
	#tasks on slave 1 stage 2	4
	#tasks on slave 4 stage 2	0

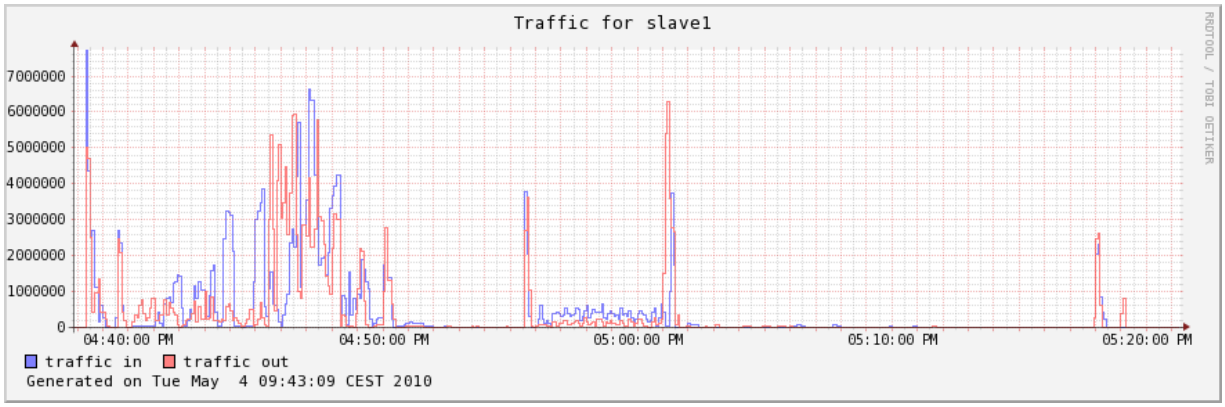


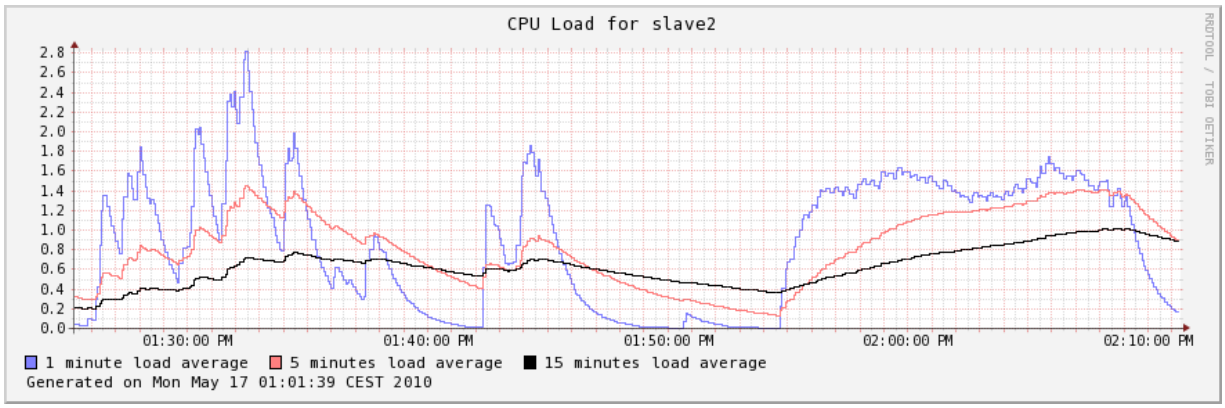


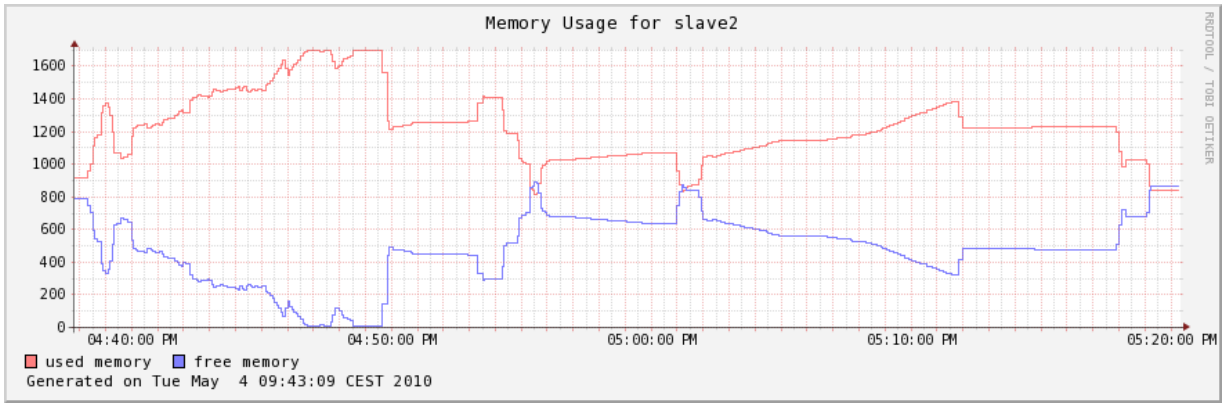


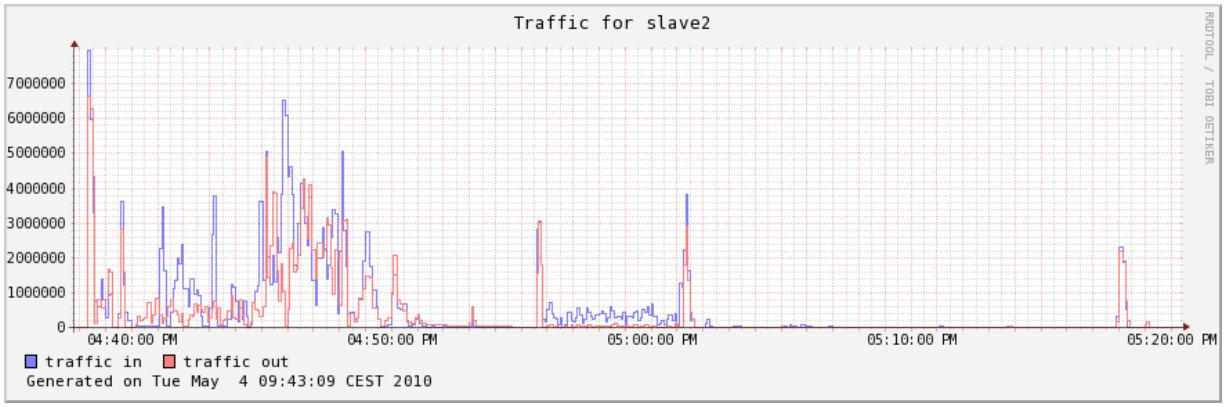

CPU Load for slave1
5.0
4.0
3.0
2.0
1.0
0.0
01:30:00 PM
01:40:00 PM
01:50:00 PM
02:00:00 PM
02:10:00 PM
1 minute load average
5 minutes load average
15 minutes load average
Generated on Mon May 17 01:01:38 CEST 2010
RRDTOOL / TOBI OETIKER

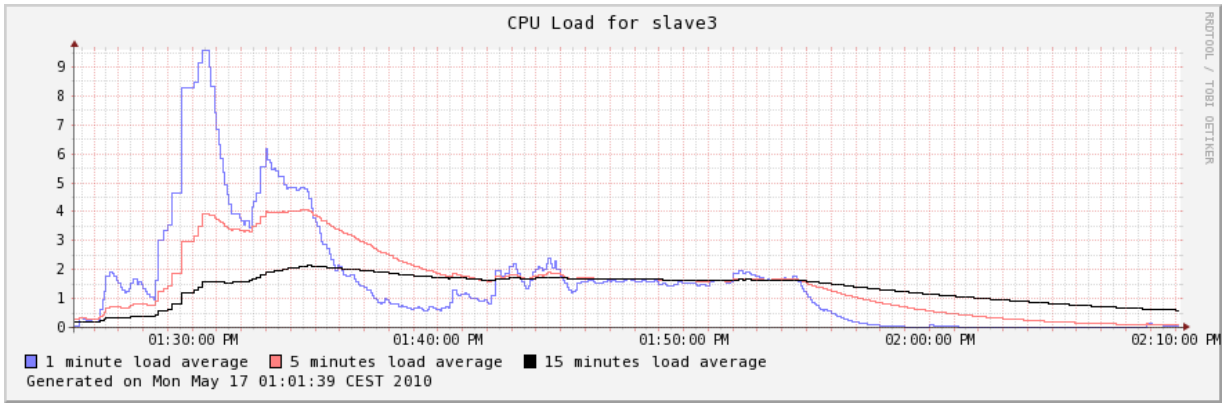

Memory Usage for slave1
1600
1400
1200
1000
800
600
400
200
0
04:40:00 PM
04:50:00 PM
05:00:00 PM
05:10:00 PM
05:20:00 PM
used memory
free memory
Generated on Tue May 4 09:43:09 CEST 2010
RRDTOOL / TOBI OETIKER

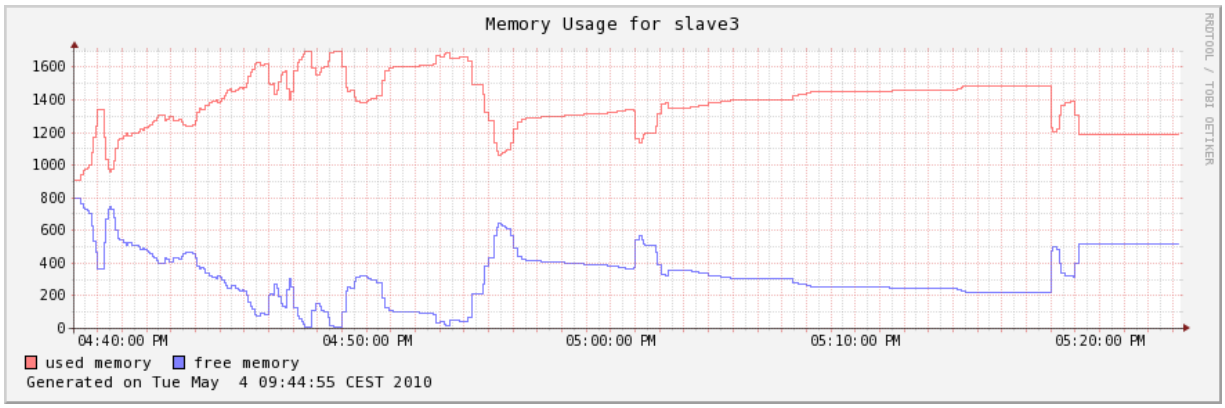

Traffic for slave1
7000000
6000000
5000000
4000000
3000000
2000000
1000000
0
04:40:00 PM
04:50:00 PM
05:00:00 PM
05:10:00 PM
05:20:00 PM
traffic in
traffic out
Generated on Tue May 4 09:43:09 CEST 2010
RRDTOOL / TOBI OETIKER

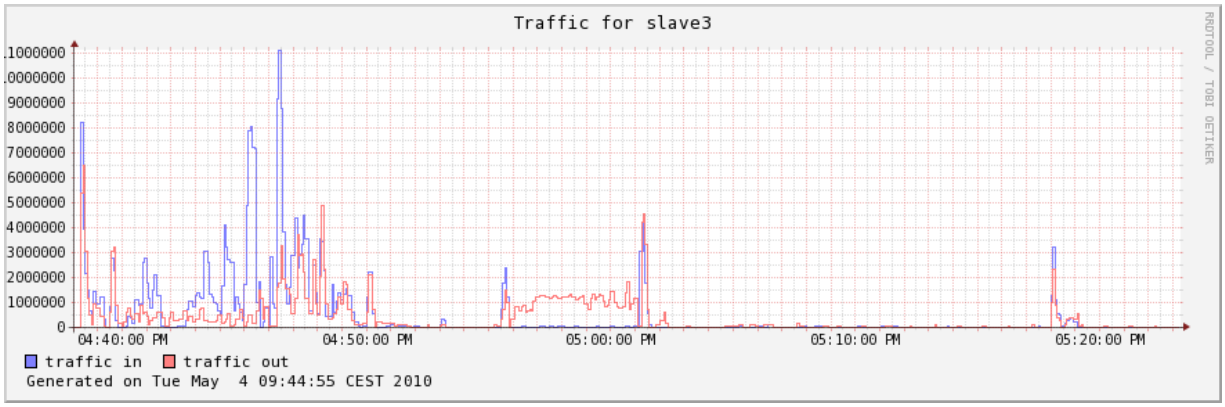

CPU Load for slave2
2.8
2.6
2.4
2.2
2.0
1.8
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
01:30:00 PM
01:40:00 PM
01:50:00 PM
02:00:00 PM
02:10:00 PM
1 minute load average
5 minutes load average
15 minutes load average
Generated on Mon May 17 01:01:39 CEST 2010
RRDTOOL / TOBI OETIKER

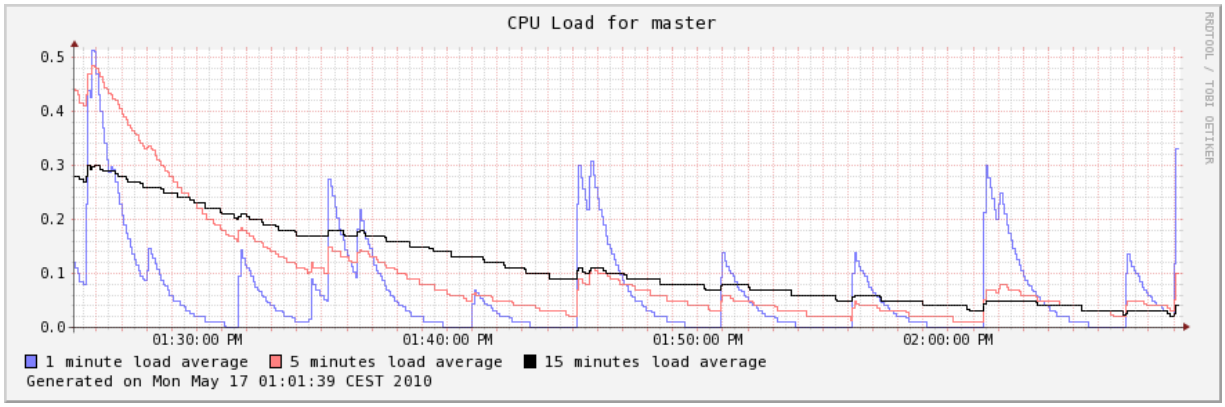

Memory Usage for slave2
1600
1400
1200
1000
800
600
400
200
0
04:40:00 PM
04:50:00 PM
05:00:00 PM
05:10:00 PM
05:20:00 PM
used memory
free memory
Generated on Tue May 4 09:43:09 CEST 2010
RRDTOOL / TOBI OETIKER

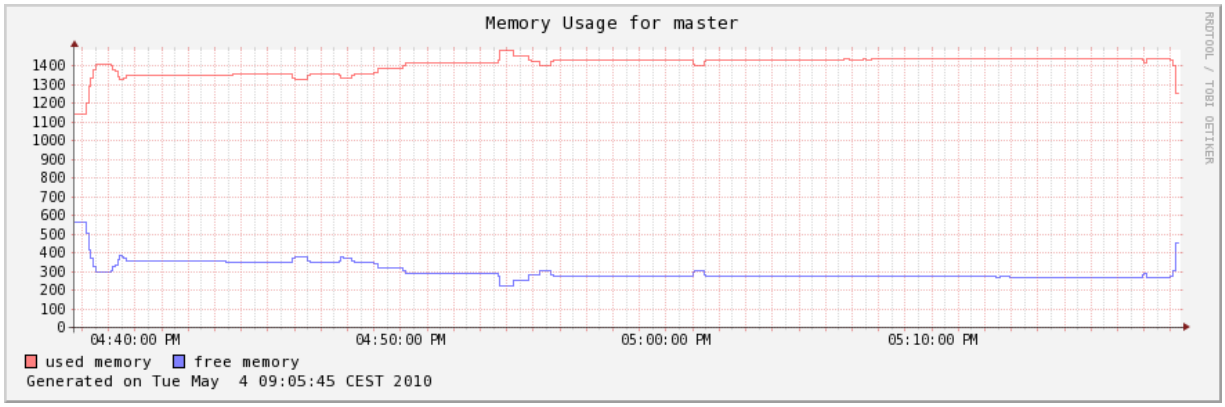

Traffic for slave2
7000000
6000000
5000000
4000000
3000000
2000000
1000000
0
04:40:00 PM
04:50:00 PM
05:00:00 PM
05:10:00 PM
05:20:00 PM
traffic in
traffic out
Generated on Tue May 4 09:43:09 CEST 2010
RRDTOOL / TOBI OETIKER

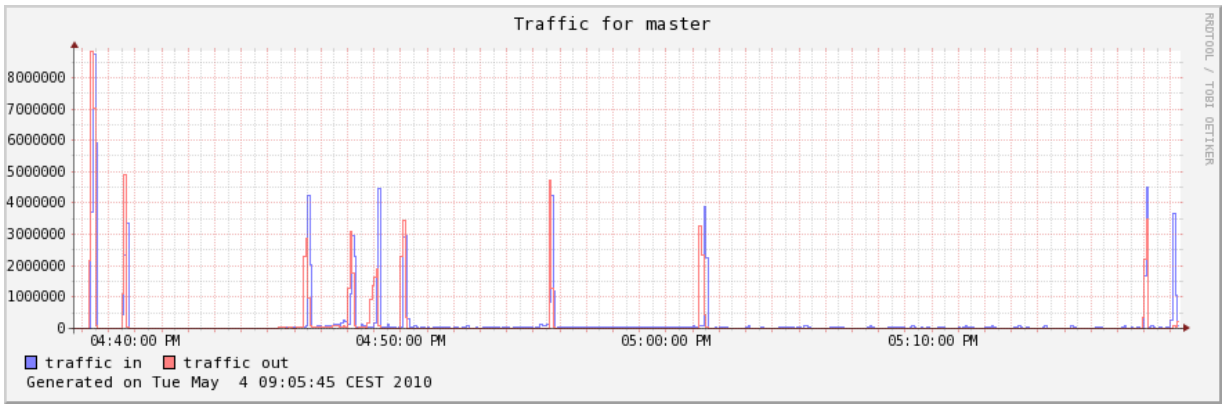

CPU Load for slave3
9
8
7
6
5
4
3
2
1
0
01:30:00 PM
01:40:00 PM
01:50:00 PM
02:00:00 PM
02:10:00 PM
1 minute load average
5 minutes load average
15 minutes load average
Generated on Mon May 17 01:01:39 CEST 2010
RRDTOOL / TOBI OETIKER

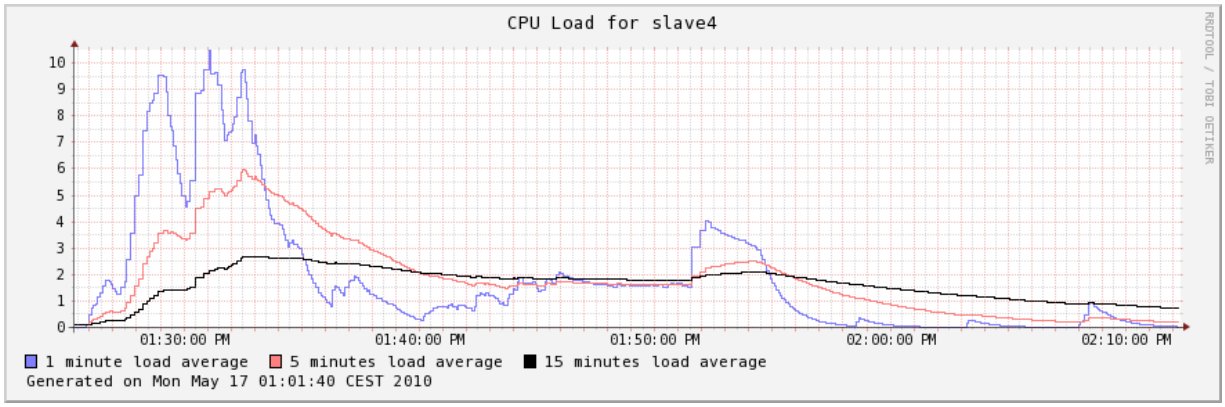

Memory Usage for slave3
1600
1400
1200
1000
800
600
400
200
0
04:40:00 PM
04:50:00 PM
05:00:00 PM
05:10:00 PM
05:20:00 PM
used memory
free memory
Generated on Tue May 4 09:44:55 CEST 2010
RRDTOOL / TOBI OETIKER

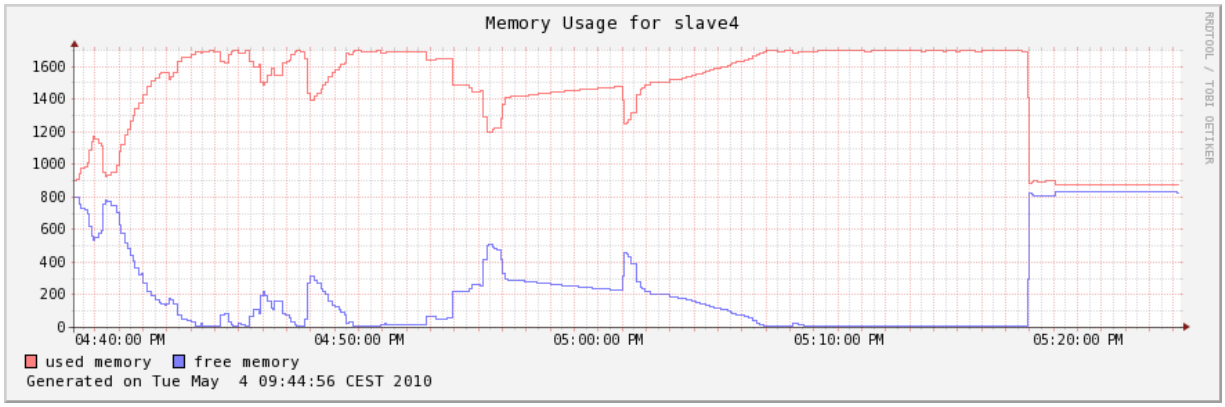

Traffic for slave3
1000000
9000000
8000000
7000000
6000000
5000000
4000000
3000000
2000000
1000000
0
04:40:00 PM
04:50:00 PM
05:00:00 PM
05:10:00 PM
05:20:00 PM
traffic in
traffic out
Generated on Tue May 4 09:44:55 CEST 2010
RRDTOOL / TOBI OETIKER

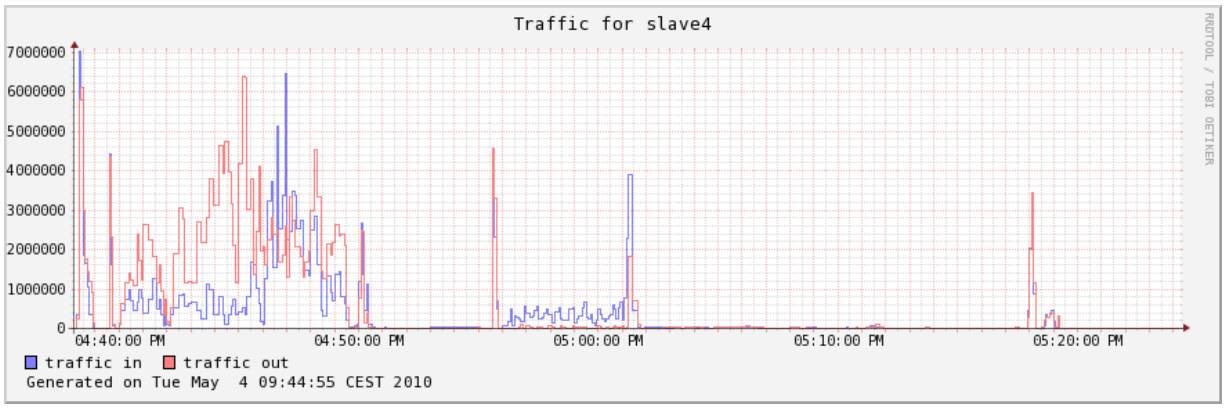

CPU Load for master
0.5
0.4
0.3
0.2
0.1
0.0
01:30:00 PM
01:40:00 PM
01:50:00 PM
02:00:00 PM
1 minute load average
5 minutes load average
15 minutes load average
Generated on Mon May 17 01:01:39 CEST 2010
RRDTOOL / TOBI OETIKER


Memory Usage for master
1400
1300
1200
1100
1000
900
800
700
600
500
400
300
200
100
0
04:40:00 PM
04:50:00 PM
05:00:00 PM
05:10:00 PM
used memory
free memory
Generated on Tue May 4 09:05:45 CEST 2010
RRDTOOL / TOBI OETIKER


Traffic for master
8000000
7000000
6000000
5000000
4000000
3000000
2000000
1000000
0
04:40:00 PM
04:50:00 PM
05:00:00 PM
05:10:00 PM
traffic in
traffic out
Generated on Tue May 4 09:05:45 CEST 2010
RRDTOOL / TOBI OETIKER


CPU Load for slave4
RRDTOOL / TOBI OETIKER
10
9
8
7
6
5
4
3
2
1
0
01:30:00 PM
01:40:00 PM
01:50:00 PM
02:00:00 PM
02:10:00 PM
1 minute load average
5 minutes load average
15 minutes load average
Generated on Mon May 17 01:01:40 CEST 2010

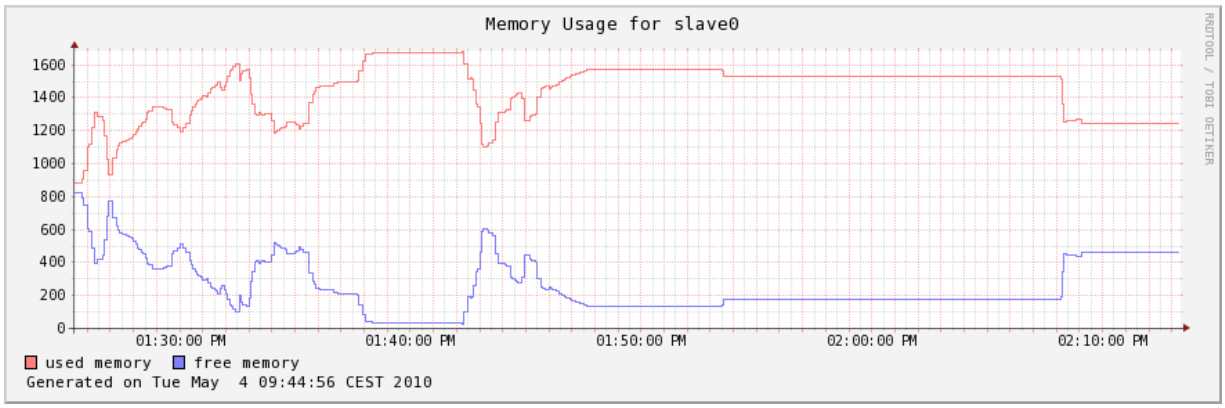

Memory Usage for slave4
RRDTOOL / TOBI OETIKER
1600
1400
1200
1000
800
600
400
200
0
04:40:00 PM
04:50:00 PM
05:00:00 PM
05:10:00 PM
05:20:00 PM
used memory
free memory
Generated on Tue May 4 09:44:56 CEST 2010

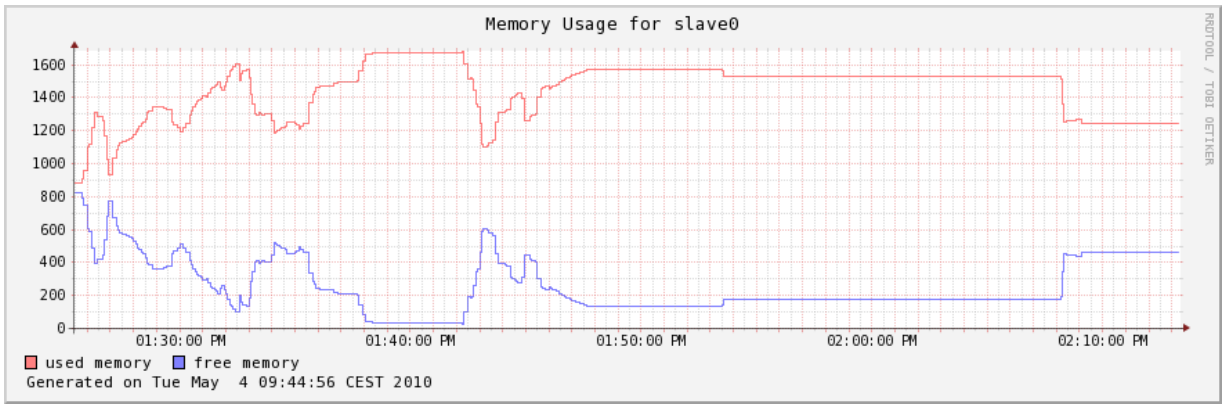

Traffic for slave4
RRDTOOL / TOBI OETIKER
7000000
6000000
5000000
4000000
3000000
2000000
1000000
0
04:40:00 PM
04:50:00 PM
05:00:00 PM
05:10:00 PM
05:20:00 PM
traffic in
traffic out
Generated on Tue May 4 09:44:55 CEST 2010

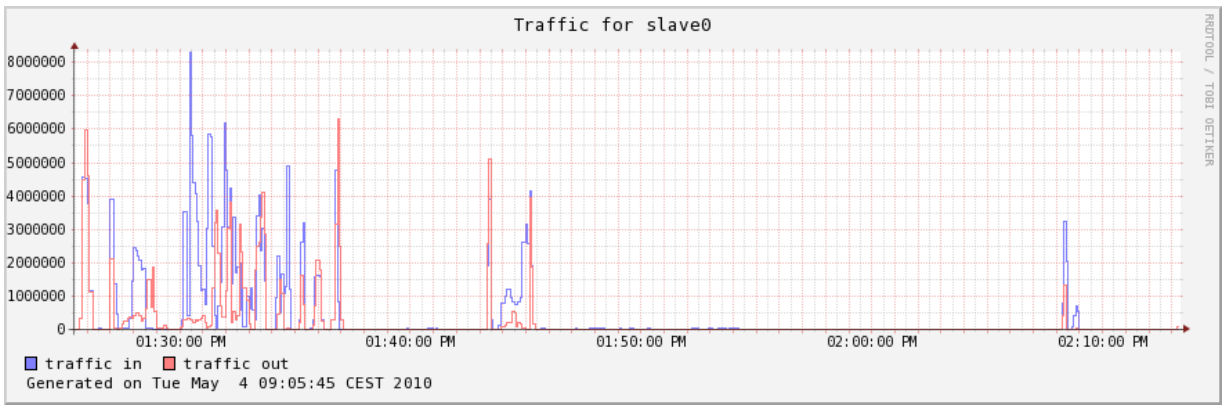
E.5 5 Nodes, Block Size 8388608 Bytes 2010-04-21 16:47:10

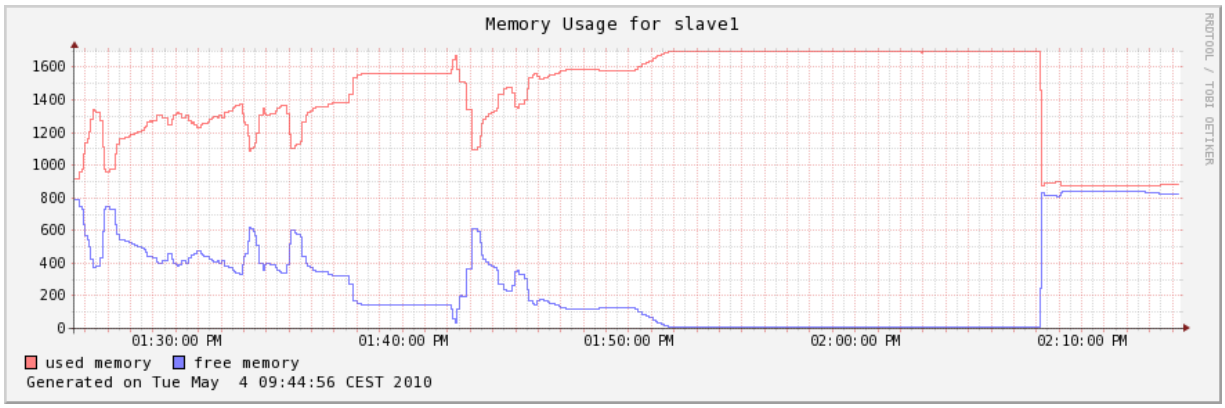
Import time		451.078 seconds
Processing time		2130.14 seconds
Names Mapper	Processing time	90.308 seconds
	Map started after	32.693 seconds
	#tasks	3
	#tasks on slave 3	0
	#tasks on slave 1	1
	#tasks on slave 0	2
	#tasks on slave 4	0
	#tasks on slave 2	0
Merge Stations by Distance	Processing time	479.473 seconds
	Map started after	42.199 seconds
	#tasks	6
	#tasks on slave 3	1
	#tasks on slave 1	2
	#tasks on slave 0	1
	#tasks on slave 4	1
	#tasks on slave 2	1
Distributed Routes Extractor	Processing time	1560.372 seconds
	Map started after	43.015 seconds
	#tasks	275
	#tasks on slave 3	49
	#tasks on slave 1	51
	#tasks on slave 0	51
	#tasks on slave 4	61
	#tasks on slave 2	63
	#tasks stage 0	792
	#tasks on slave 3 stage 0	48
	#tasks on slave 1 stage 0	49
	#tasks on slave 0 stage 0	49
	#tasks on slave 4 stage 0	56
	#tasks on slave 2 stage 0	62
	#tasks stage 1	7
	#tasks on slave 3 stage 1	1
	#tasks on slave 1 stage 1	2
	#tasks on slave 0 stage 1	2
	#tasks on slave 4 stage 1	1

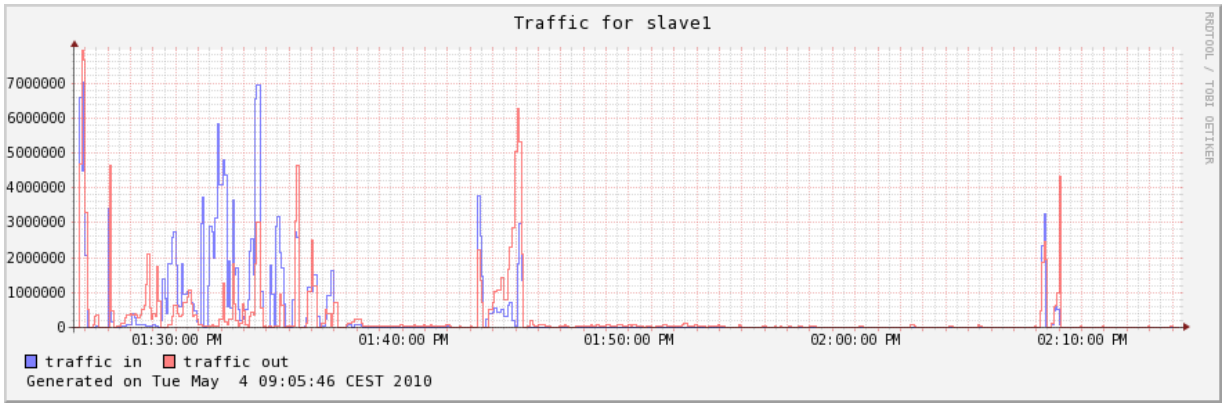
	#tasks on slave 2 stage 1	1
	#tasks stage 2	4
	#tasks on slave 3 stage 2	0
	#tasks on slave 1 stage 2	0
	#tasks on slave 0 stage 2	0
	#tasks on slave 4 stage 2	4
	#tasks on slave 2 stage 2	0

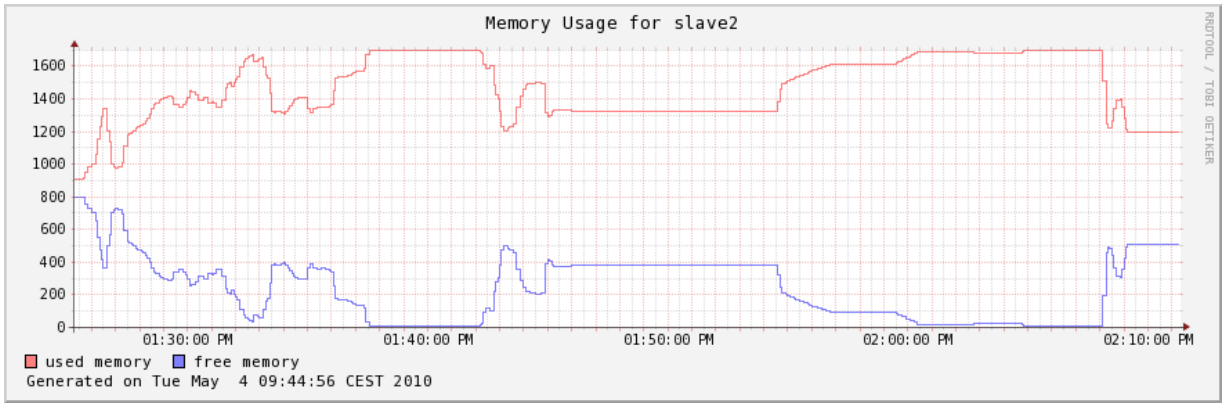


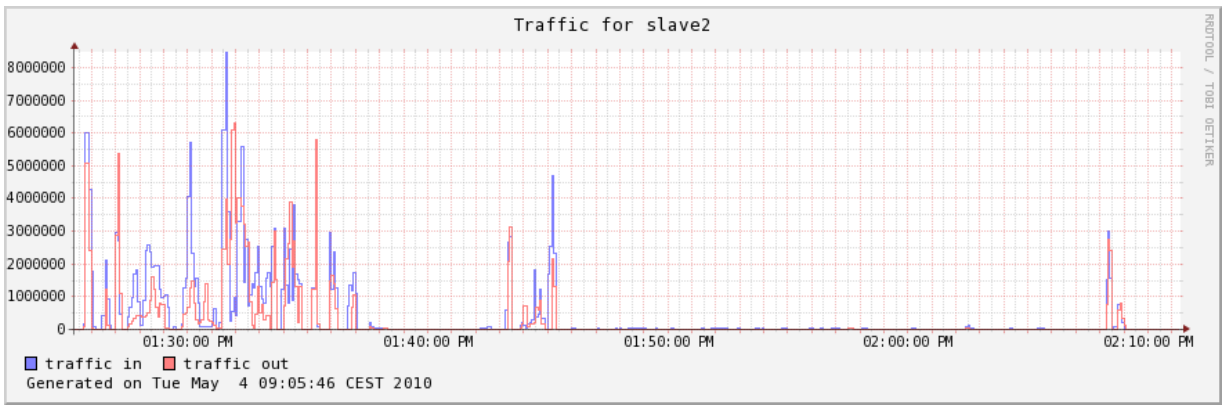


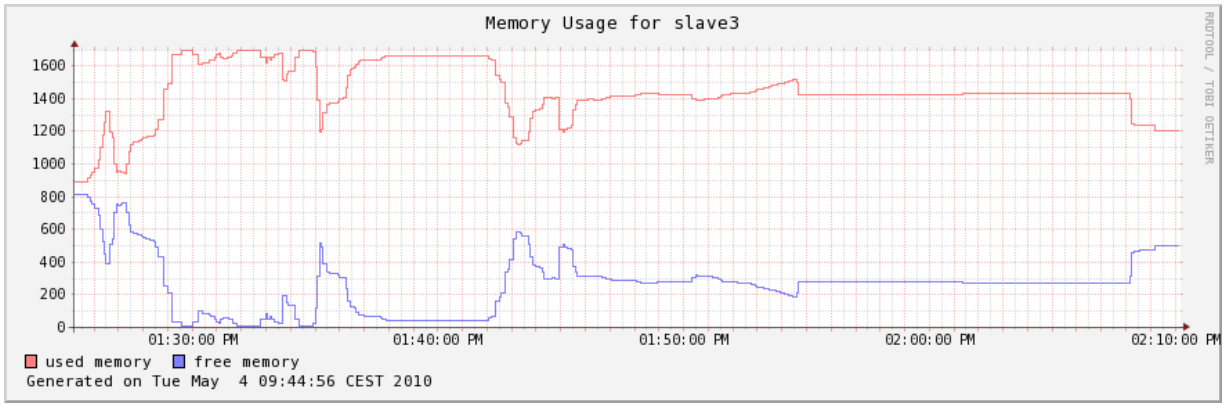


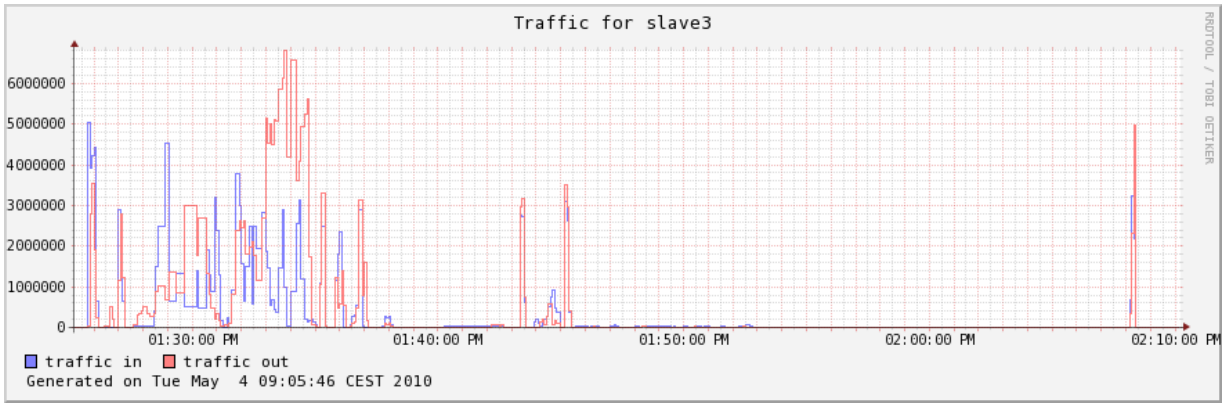

Memory Usage for slave1
1600
1400
1200
1000
800
600
400
200
0
01:30:00 PM
01:40:00 PM
01:50:00 PM
02:00:00 PM
02:10:00 PM
used memory
free memory
Generated on Tue May 4 09:44:56 CEST 2010
RRDTOOL / TOBI OETIKER

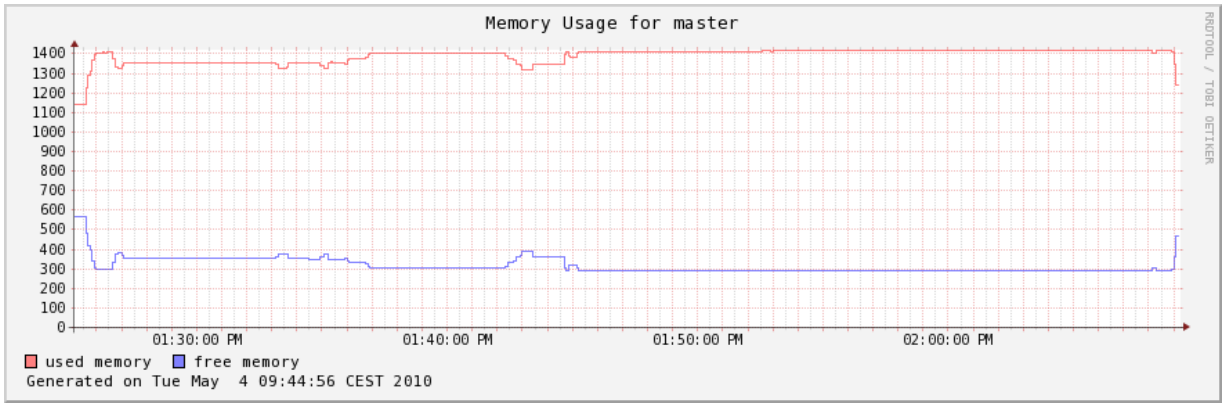

Traffic for slave1
7000000
6000000
5000000
4000000
3000000
2000000
1000000
0
01:30:00 PM
01:40:00 PM
01:50:00 PM
02:00:00 PM
02:10:00 PM
traffic in
traffic out
Generated on Tue May 4 09:05:46 CEST 2010
RRDTOOL / TOBI OETIKER

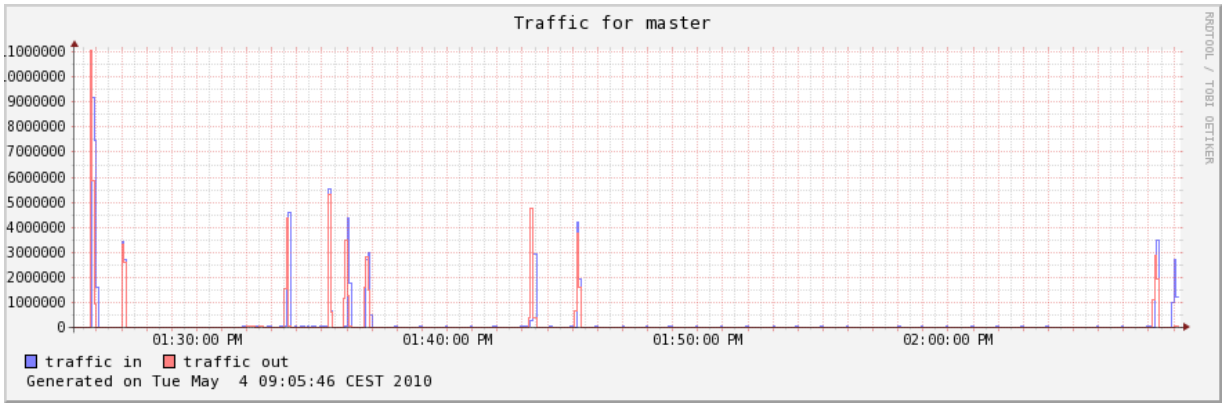

Memory Usage for slave2
1600
1400
1200
1000
800
600
400
200
0
01:30:00 PM
01:40:00 PM
01:50:00 PM
02:00:00 PM
02:10:00 PM
used memory
free memory
Generated on Tue May 4 09:44:56 CEST 2010
RRDTOOL / TOBI OETIKER

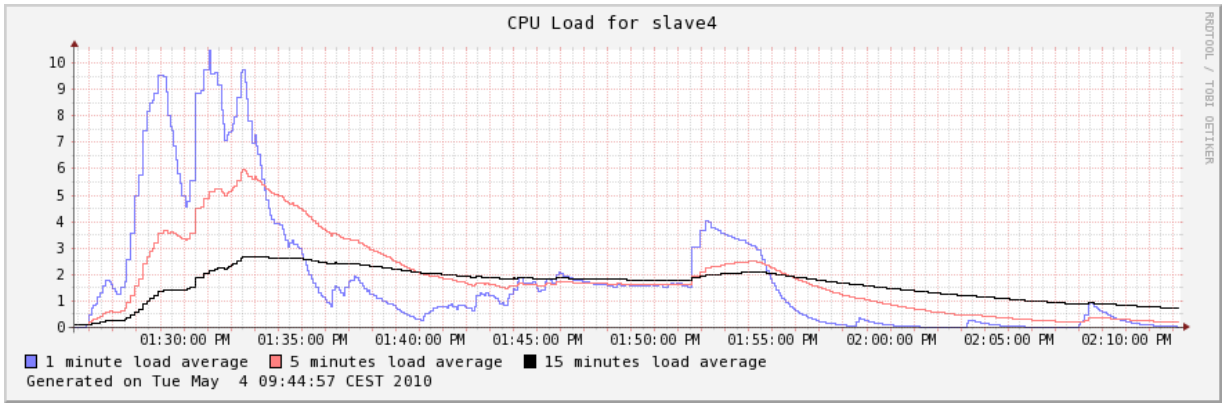

Traffic for slave2
8000000
7000000
6000000
5000000
4000000
3000000
2000000
1000000
0
01:30:00 PM
01:40:00 PM
01:50:00 PM
02:00:00 PM
02:10:00 PM
traffic in
traffic out
Generated on Tue May 4 09:05:46 CEST 2010
RRDTOOL / TOBI OETIKER

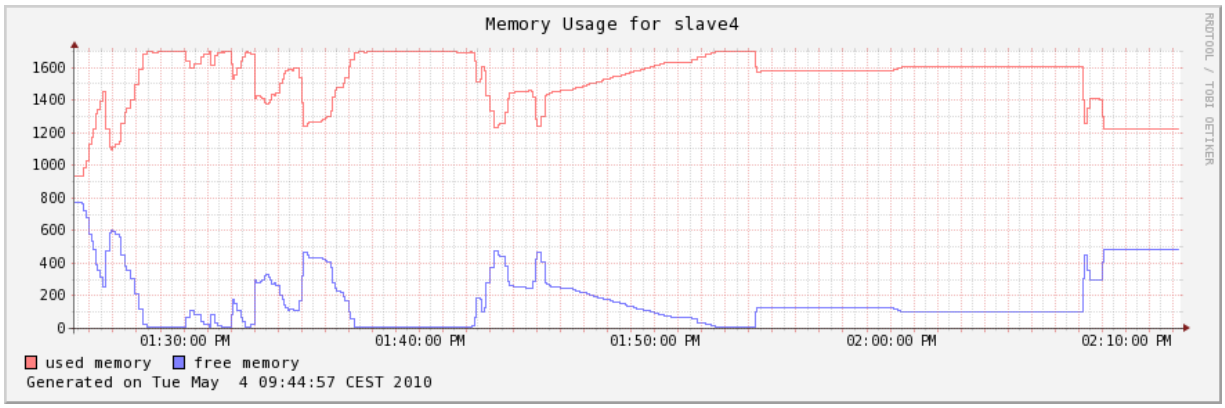

Memory Usage for slave3
1600
1400
1200
1000
800
600
400
200
0
01:30:00 PM
01:40:00 PM
01:50:00 PM
02:00:00 PM
02:10:00 PM
used memory
free memory
Generated on Tue May 4 09:44:56 CEST 2010
RRDTOOL / TOBI OETIKER

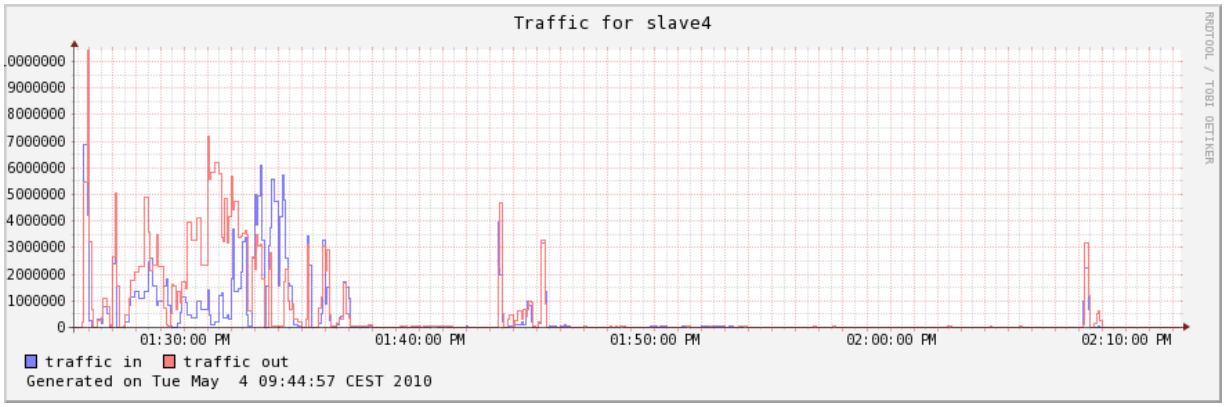

Traffic for slave3
6000000
5000000
4000000
3000000
2000000
1000000
0
01:30:00 PM
01:40:00 PM
01:50:00 PM
02:00:00 PM
02:10:00 PM
traffic in
traffic out
Generated on Tue May 4 09:05:46 CEST 2010
RRDTOOL / TOBI OETIKER


Memory Usage for master
1400
1300
1200
1100
1000
900
800
700
600
500
400
300
200
100
0
01:30:00 PM
01:40:00 PM
01:50:00 PM
02:00:00 PM
used memory
free memory
Generated on Tue May 4 09:44:56 CEST 2010
RRDTOOL / TOBI OETIKER


Traffic for master
1000000
0000000
9000000
8000000
7000000
6000000
5000000
4000000
3000000
2000000
1000000
0
01:30:00 PM
01:40:00 PM
01:50:00 PM
02:00:00 PM
traffic in
traffic out
Generated on Tue May 4 09:05:46 CEST 2010
RRDTOOL / TOBI OETIKER


CPU Load for slave4
10
9
8
7
6
5
4
3
2
1
0
01:30:00 PM
01:35:00 PM
01:40:00 PM
01:45:00 PM
01:50:00 PM
01:55:00 PM
02:00:00 PM
02:05:00 PM
02:10:00 PM
1 minute load average
5 minutes load average
15 minutes load average
Generated on Tue May 4 09:44:57 CEST 2010
RRDTOOL / TOBI OETIKER


Memory Usage for slave4
1600
1400
1200
1000
800
600
400
200
0
01:30:00 PM
01:40:00 PM
01:50:00 PM
02:00:00 PM
02:10:00 PM
used memory
free memory
Generated on Tue May 4 09:44:57 CEST 2010
RRDTOOL / TOBI OETIKER


Traffic for slave4
0000000
9000000
8000000
7000000
6000000
5000000
4000000
3000000
2000000
1000000
0
01:30:00 PM
01:40:00 PM
01:50:00 PM
02:00:00 PM
02:10:00 PM
traffic in
traffic out
Generated on Tue May 4 09:44:57 CEST 2010
MRTOOL / TOBI OETIKER

List of Tables

List of Figures

Bibliography

[aws] Amazon elastic compute cloud user guide. http://awsdocs.s3.amazonaws.com/EC2/latest/ec2-gsg.pdf.

[Bon] André B. Bondi. Characteristics of scalability and their impact on performance. Proceedings of the 2nd international workshop on Software and performance, Ottawa, Ontario, Canada.

[Bor] Dhruba Borthakur. Hdfs architecture. http://tinyurl.com/l62hru.

[CDG+08] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):1–26, June 2008.

[Day] Allan Day. Example to bulk import/load a text file into an htable. http://wiki.apache.org/hadoop/Hbase/MapReduce.

[DG04] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, page 10, Berkeley, CA, USA, 2004. USENIX Association.

[dis] Disco documentation. http://discoproject.org/doc/.

[dLSNG03] Benoît des Ligneris, Stephen Scott, Thomas Naughton, and Neil Gorsuch. Open source cluster application resources (oscar): design, implementation and interest for the [computer] scientific community, 2003.

[DSS] Jack Dongarra, Thomas Sterling, and Horst Simon. High performance computing. clusters, constellations, mpps, and future directions.

[Geo] Lars George. Hbase vs. bigtable comparison. http://www.larsgeorge.com/2009/11/hbase-vs-bigtable-comparison.html.

[haf09] Hafas raw data file format, 2009.

[HBaa] Hbase 0.20.3 api. http://hadoop.apache.org/hbase/docs/current/api/.

[HBab] Hbasearchitecture. `http://wiki.apache.org/hadoop/Hbase/HbaseArchitecture`.

[HM06] SAUL HANSELL and JOHN MARKOFF. Technology; a search engine that's becoming an inventor. *The New York Times*, 2006.

[IBY+07] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 59–72, New York, NY, USA, 2007. ACM.

[KSS] Kamran Karimi, Michael Schmitz, and Mohsen Sharifi. Dipc: A heterogeneous distributed programming system.

[NBD+03] W. A. Najjar, W. Bohm, B. A. Draper, J. Hammes, R. Rinker, J. R. Beveridge, M. Chawathe, and C. Ross. High-level language abstraction for reconfigurable computing. *Computer*, 36(8):63–69, 2003.

[Ode09] Martin Odersky. *The Scala Language Specification Version 2.7*. 2009.

[ORS+08] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110, New York, NY, USA, 2008. ACM.

[Pfe] Joachim Pfeiffer. General transit feed specification.

[POL02] Shelley Powers, Tim O'Reilly, and Mike Loukides. *Unix Power Tools, Third Edition*. O'Reilly, October 2002.

[ros] Closest-pair problem. http://rosettacode.org/wiki/Closest_pair_problem.

[SF07] Hans Svensson and Lars-Aake Fredlund. Programming distributed erlang applications: pitfalls and recipes. In *Erlang '07: Proceedings of the 2007 SIGPLAN workshop on Erlang Workshop*, pages 37–42, New York, NY, USA, 2007. ACM Press.

[SGM+09] Maraike Schellmann, Sergei Gorlatch, Dominik Meilaender, Thomas Kuesters, Klaus Schaefers, Frank Wuebbeling, and Martin Burger. Parallel medical image reconstruction: From graphics processors to grids. In *Parallel Computing Technologies*, volume 5698 of *Lecture Notes in Computer Science*, pages 457–473. Springer Berlin / Heidelberg, 2009.

[SH] M. I. Shamos and D. Hoey. Closest-point problems. Proceedings of the 16th annual IEEE Sympos.

[Ven09] Jason Venner. *Pro Hadoop*. Apress, 1 edition, June 2009.

[Vog09] Werner Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, January 2009.

[Whi75] J.E. White. High-level framework for network-based resource sharing. RFC 707, December 1975.

[Wik94] C. Wikstrom. Distributed programming in erlang, 1994.