D5.2

Development of Best Practices for Large-scale Data Management Infrastructure

Coordinator: Steffen Stadtmüller (KIT)
With contributions from: Hannes Mühleisen (FUB), Chris Bizer (FUB), Martin Kersten (CWI), Arjen de Rijke (CWI), Fabian Groffen (CWI), Ying Zhang (CWI), Günter Ladwig (KIT), Andreas Harth (KIT), Mitja Trampus (JSI)

1st Quality reviewer: Lyndon Nixon
2nd Quality reviewer: Alexandra Moraru

Deliverable nature:	R
Dissemination level: (Confidentiality)	PU
Contractual delivery date:	
Actual delivery date:	
Version:	1.0
Total number of pages:	86
Keywords:	Hadoop, MapReduce, large-scale data analysis, Web Data Commons, data extraction, SciLens, Scientific DBMS, MonetDB, local-area cluster, RDF, triple store, nested key/value store, news aggregation, high volume data streams

Abstract

The amount of available data for processing is constantly increasing and becomes more diverse. We collect our experiences on deploying large-scale data management tools on local-area clusters or cloud infrastructures and provide guidance to use these computing and storage infrastructures.

In particular we describe Apache Hadoop, one of the most widely used software libraries to perform large-scale data analysis tasks on clusters of computers in parallel and provide guidance on how to achieve optimal execution time when performing analysis over large-scale data.

Furthermore we report on our experiences with projects, that provide valuable insights in the deployment and use of large- scale data management tools:

The **Web Data Commons** project for which we extracted all Microformat, Microdata and RDFa data from the Common Crawl web corpus, the largest and most up-to-date web corpus that is currently available to the public.

SciLens, a local-area cluster machine that has been built to facilitate research on data management issues for data-intensive scientific applications.

CumulusRDF, an RDF store on cloud-based architectures. We investigate the feasibility of using a distributed nested key/value store as an underlying storage component for a Linked Data server, which provides functionality for serving large quantities of Linked Data.

Finally we describe the **News Aggregator Pipeline**, which is a piece of software to perform the acquisition of high volume textual streams, it's processing into a suitable form for further analysis and the distribution of the data.

Executive summary

In this deliverable, we describe our experiences on deploying large-scale data management tools on local area clusters and provide guidance to use these tools and computing infrastructures.

We address Apache Hadoop, one of the most widely used software libraries to perform large-scale data analysis tasks on clusters of computers in parallel. Hadoop is a software library that provides a reliable shared storage and analysis system for distributed computing, with storage provided by HDFS and analysis by MapReduce. However, designing MapReduce jobs can be intricate and relying on the distributed system architecture of Hadoop does not guarantee desired processing speeds for large-scale data. We describe how MapReduce jobs can be designed to achieve performance goals in general. Furthermore we provide guidance on how to optimise MapReduce jobs. We detail the different attributes of a MapReduce job or an Hadoop cluster itself that have to be addressed to optimise the execution time of a task and to make the processing of large-scale data feasible.

We than report on four projects concretely, that provide valuable insights in the deployment and use of large-scale data management tools:

First we describe the ***Web Data Commons*** project, in which we extracted all Microformat, Microdata and RDFa data from the Common Crawl web corpus. The common crawl web corpus is the largest and most up-to-date web corpus that is currently available to the public. We offer the extracted data for download in the form of RDF-quads and CSV tables. Here we give an overview of the project and present extracted statistics about the popularity of the different encoding standards. We describe in detail our experiences when extracting the data in a distributed manner on the Amazon EC2 cloud and processing the results for publication with Hadoop.

Secondly, we report on our experiences with the design and utilisation of ***SciLens***, a local-area cluster machine that has been built to facilitate research on data management issues for data-intensive scientific applications. *The main contribution of the SciLens cluster is that it provides a platform on which we can carry out experiments to help develop, examine, compare and improve new DBMS techniques for the scientific applications. SciLens enables various possibilities to run experiments on large clusters using real-world large datasets, obtained directly from different scientific disciplines.* We give background information about the research on DBMS support for data-intensive sciences. We will briefly describe the data management problems in the data-intensive sciences, the mismatch between what the scientists need and what the current DBMSs provide, our plans to advance the state-of-the-art of the scientific data management and analysis, and the need of a SciLens cluster to help realising the plans. Then, we discuss the design issues of SciLens. Furthermore, we describe the initial and the final architecture of SciLens. We discuss two main lessons we have learnt during the building of SciLens and describe the tools we have developed to deploy the relational DBMS MonetDB on SciLens. Finally, we conclude with an outlook to future work.

Thirdly, we report on ***CumulusRDF***. Publishers of Linked Data require scalable storage and retrieval infrastructure due to the size of datasets and potentially high rate of lookups on popular sites. We investigate the feasibility of using a distributed nested key/value store as an underlying storage component for a Linked Data server, which provides functionality for serving large quantities of Linked Data via HTTP lookups and in addition offers single triple pattern lookups. We devise two storage schemes for our CumulusRDF system implemented on Apache Cassandra, an open-source nested key/value store. We compare the schemes on a subset of DBpedia and both synthetic workloads and workloads obtained from DBpedia's access logs. Results on a cluster of up to 8 machines indicate that CumulusRDF is competitive to state-of-the-art distributed RDF stores.

Finally, we describe the **News Aggregator Pipeline**, which is a piece of software to perform the first stage in processing and analysing high volume textual streams. The News Aggregator Pipeline handles the acquisition of large amounts of data and it's processing into a suitable form for further analysis: RSS feeds and Google news are crawled to obtain links to news articles, which are downloaded. The articles are parsed to obtain further links to RSS sources and a clear text version of the article body. Furthermore the News Aggregator pipeline provides an efficient mechanism for the distribution of the data, by exposing a stream of articles to end-users in clear text as well as a stream of semantically annotated articles.

Document Information

Version Log			
Issue Date	**Rev. No.**	**Author**	**Change**
2012-06-20	0.1	Steffen Stadtmüller	Initial draft of TOC
2012-07-04	0.2	Günter Ladwig, Andreas Harth	Report on CumulusRDF
2012-07-10	0.3	Hannes Mühleisen, Chris Bizer	Report on Web Data Commons
2012-08-09	0.4	Steffen Stadtmüller	Hadoop Best Practices
2012-08-21	0.5	Martin Kersten, Arjen de Rijke, Fabian Groffen, Ying Zhang	Report on SciLens
2012-08-29	0.6	Mitja Trampus	Report on News Aggregator Pipeline

Table of Contents

List of Figures

List of Tables

Abbreviations

COTS	Commercially available Off-The-Shelf
DBMS	Database Management System
DIC	Data Intensive Computing
EC2	Amazon Elastic Compute Cloud
FITS	Flexible Image Transport System
GDK	Goblin Database Kernel
HDD	Hard Disc Drive
HDFS	Hadoop Distributed File System
HPC	High Performance Computing
I/O	Input/Output
JBOD	Just a Bunch Of Drives
JVM	Java Virtual Machine
LAN	Local-Area Network
LOFAR	Low-Frequency Array
MAL	MonetDB Assembler Language
NetCDF	Network Common Data Form
ORFEUS	Observatories and Research Facilities for European Seismology
QDR	Quad Data Rate
QoS	Quality Of Service
Pan-STARRS	Panoramic Survey Telescope and Rapid Response System
RAID	Redundant Array of Independent Disks
RDBMS	Relational Database Management System
RDF	Resource Description Framework
RFC	Request for Comments
S3	Amazon Simple Storage Service
SPARQL	SPARQL Protocol and RDF Query Language
SQS	Amazon Simple Queue Service
SSD	Solid State Drive
UDF	User Defined Function
VLAN	Virtual Local Area Network, Virtual LAN

1 Introduction

In this deliverable we report on the collected and synthesized experiences made by PlanetData members on deploying large-scale data management tools on local-area clusters or cloud infrastructures and provide guidance to use these computing and storage infrastructures to conduct research.

For this purpose we describe projects and provide guidance regarding four aspects of large-scale data management:

- Data analysis on large datasets.
- Infrastructure for large-scale data management
- Storage and provisioning of large datasets
- Large-scale data acquisition

With regard to data analysis, we provide an introduction into Apache Hadoop in Chapter 2, one of the most widely used software libraries to perform large-scale data analysis tasks on clusters of computers in parallel. We focus on the challenges when analysing data with Hadoop and provide a guide to improve the performance of analysis tasks. In Chapter 3 we report on the analysis of two large Web corpora for the Web Data Commons project, which was preformed with Hadoop as well as a queuing approach on Amazon cloud infrastructure. With Web Data Commons we show exemplarily how large analysis tasks can be optimised with regard to execution time and cost reduction.

In Chapter 4 we describe our experience with the design and utilization of a local-area cluster, called SciLens, as infrastructure for large-scale data management. Derived from the building of SciLens we give guidance on how to select the hardware to achieve the best trade-off between price and performance and how to manage a cluster that is intended for experimental use.

We address data storage and provisioning in Chapter 5 with CumulusRDF, a scalable storage and retrieval infrastructure that is able to handle a potentially high rate of lookups. We investigate the feasibility of using a distributed nested key/value store as an underlying storage component for a Linked Data server, which provides functionality for serving large quantities of Data.

For large-scale data acquisition we report on the News Aggregator Pipeline in Chapter 6. The News Aggregator Pipeline handles the acquisition of large amounts of data and it's processing into a suitable form for further analysis. Furthermore the News Aggregator pipeline provides an efficient mechanism for the distribution of the data, by exposing a stream of articles to end-users in clear text as well as a stream of semantically annotated articles.

We conclude in Chapter 7.

2 Large Scale Data Processing with Hadoop

In this section we will give an introduction and guidance for large-scale data processing with the open-source Apache Hadoop software library. Hadoop is a framework that allows for the distributed processing of large datasets on clusters of computers and is in wide use in industry and science[1]. The data processing is handled on the Hadoop Distributed File System (HDFS) in a two-stage process called MapReduce, consisting of the parallel execution of computation tasks (Map) and the integration of results (Reduce).

The amount of available data for processing is constantly increasing and becomes more diverse (Ganz et al., 2008). The motivation behind distributed processing of data is that storage capacities of hard drives have increased massively over the years, while access speeds have not kept up. To reduce the necessary time for read and write operations multiple disks can be employed at once. However, the parallel use of multiple hard disks results in new problems for data processing:

- Hardware failure: As soon more pieces of hardware are employed, the chances on one to fail increases. The standard way - for which Hadoop provides facilities in its file system HDFS - to mitigate this problem and to avoid data loss is through replication: Redundant copies of data blocks are kept by the system so that, in the event of failure, another copy is available.
- Result integration: Most analysis tasks need to be able to combine the data in some way. Doing this correctly is notoriously challenging. MapReduce provides a programming model that abstracts the problem from disk reads and writes, transforming it into a computation over sets of keys and values.

Altogether, Hadoop provides a reliable shared storage and analysis system, with storage provided by HDFS and analysis by MapReduce.

In this chapter we give an overview on how an HDFS storage system works in section 2.1, particularly with respect to failure handling. Furthermore we describe the advantages of distributed data analysis with MapReduce in section 2.2. We describe how efficient MapReduce jobs can be designed especially with respect to joins. In section 2.3 we provide a guide how Hadoop jobs can be optimised with respect to execution time.

2.1 Hadoop Distributed File System

For the storage of data partitioned over a number of machines across a network, Hadoop provides the *Hadoop Distributed File System (HDFS)*. HDFS is designed to store large files in a cluster of commodity hardware. This means that a particular focus of the design of HDFS is handling of hardware failures, which are more likely when commodity hardware is employed.

Files in HDFS are broken into block-sized chunks. The blocks of an HDFS are usually significantly larger (default value: 64 MB) than it would be expected from a single disk system. However the block size can be configured manually. The reason for the large block size is the reduction of necessary time for seeks. This allows the transfer time of large files, consisting of several data blocks, to be close to the actual disk transfer rate, by minimizing the time necessary to seek the start of the blocks.

The block abstraction for a distributed file system allows files to be larger than any individual disk in the cluster: Since the blocks from one file are not required to be stored on the same disk, they can take advantage of any of the disks in the cluster.

Furthermore, the concept of blocks allows for an easy mechanism to provide fault tolerance and availability. Every block in the HDFS can be replicated to a number of physically separate machines (default value: three). If a block becomes corrupted or a machine fails, a copy of the affected block can be read from another location. Hadoop performs this failure management transparent for the user. Furthermore, Hadoop automatically replicates affected blocks from their alternative location to other live machines to maintain the replication level.

Figure 1 shows a conceptual overview of the blocks of an HDFS and how a file is stored and replicated over the cluster of machines.

[1] See: http://wiki.apache.org/hadoop/PoweredBy

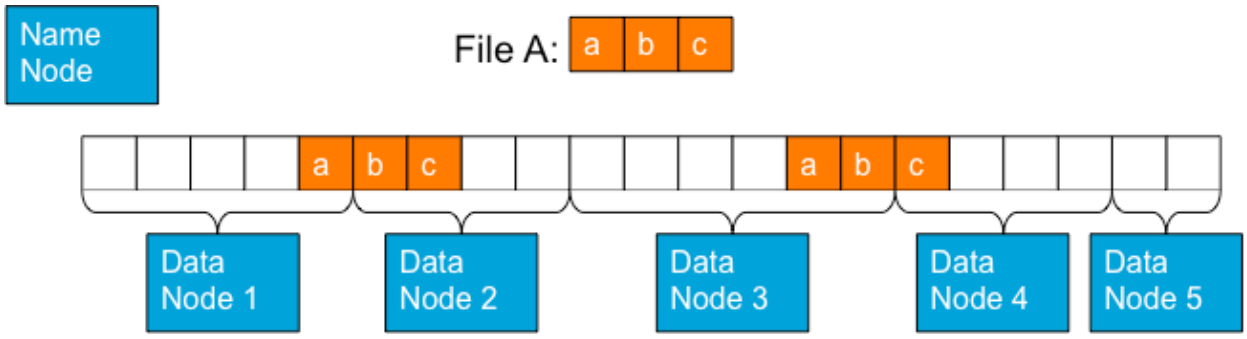

Figure 1: Desgin of an Hadoop Distributed File System

To deploy an HDFS means to run a set of daemons (i.e., resident programs) on different servers in a network, where every daemon performs a specific role in a master/slave architecture (White 2010):

2.1.1 NameNode daemon

The NameNode daemon is the master in the HDFS master/slave architecture and thus directs the slave DataNodes. The NameNode manages the file system namespace. It maintains the file system tree and the metadata for all the files and directories in the tree. This information is stored persistently on the local disk. The NameNode also knows the DataNodes on which all the blocks for a given file are located. Clients access the file system by communicating with the NameNode.

The function of the NameNode is memory and I/O intensive. As such, the server hosting the NameNode typically doesn't store any user data or perform any computations for a MapReduce program to lower the workload on the machine. This means that the NameNode server doesn't double as a DataNode.

Due to its role as master in the HDFS, the NameNode forms a single point of failure for the HDFS: With destruction of the NameNode all files on the file system would be lost, since there would be no way of reconstructing the files from the blocks stored across the network. To mitigate this problem the NameNode stores the state of the file system persistently. It is recommended to do this persistent backup not only on the local file system of the NameNode server, but additionally on a physically separated disk.

Another way of making the HDFS more resilient against NameNode failure is the use of a *Secondary NameNode*.

2.1.2 Secondary NameNode daemon

The Secondary NameNode serves as an assistant daemon in the HDFS cluster. Similar to the actual NameNode the secondary NameNode is supposed to run on a separate machine, with no other daemons deployed on the same server.

The secondary NameNode is not in contact with the slaves in the cluster, but communicates only with the NameNode to monitor the state of the HDFS by taking snapshots of the system at defined intervals. This helps to minimize data loss when the NameNode fails: The Secondary NameNode can be reconfigured to act as the actual NameNode with the last snapshot as starting configuration.

2.1.3 DataNode daemon

The slaves in the HDFS that perform the low level I/O operations (i.e., reading and writing of HDFS blocks on the local filesystems) are called DataNodes. A client interacting with the HDFS is informed by the NameNode about the DataNodes on which the blocks for a specific file reside. The client then interacts directly with the DataNodes to process the file. Furthermore, if data blocks have to be replicated, the DataNodes communicate directly with each other (data transfer is not tunneled through the NameNode).

DataNodes are constantly reporting to the NameNode. Upon initialization, each of the DataNodes informs the NameNode of the blocks it's currently storing. After this mapping is complete, the DataNodes

continually poll the NameNode to provide information regarding local changes as well as receive instructions to create, move, or delete blocks from the local disk.

2.2 Data Analysis with MapReduce

MapReduce is a programming model for data processing. The open source implementation by Apache for Hadoop is based on technology published by Google (Dean et al. 2008, which they used to scale their system.

Hadoop can run MapReduce programms written in various programming languages (e.g., Java, Ruby, Python, C++). Map reduce programs are inherently parallel, thus allowing to run large-scale data analysis tasks on clusters of computers.

MapReduce works by breaking the processing into two phases: the map phase and the reduce phase. Each phase is defined by a data processing function, and these functions are called mapper and reducer, respectively. In the mapping phase, MapReduce takes the input data and feeds each data element to the mapper. In the reducing phase, the reducer processes all the outputs from the mapper and arrives at a final result. In simple terms, the mapper is meant to filter and transform the input into something that the reducer can aggregate over.

MapReduce uses *lists* and *(key/value)* pairs as its main data primitives. The keys and values are often integers or strings but can also be dummy values to be ignored or complex object types. In a MapReduce framework, applications are written by specifying the mapper and reducer. In the following we describe the data flow of a MapReduce application in general (Lam 2011):

1. The input to an application must be structured as a list of (key/value) pairs, `list(<k1, v1>)`. The input format for processing multiple files is usually `list(<String filename, String file_content>)` . The input format for processing one large file, such as a log file, is `list(<Integer line_number, String log_event>)`.
2. The list of (key/value) pairs is broken up and each individual (key/value) pair, `<k1, v1>`, is processed by calling the map function of the mapper. In practice, the key `k1` is often ignored by the mapper. The mapper transforms each `<k1, v1>` pair into a list of `<k2, v2>` pairs. The details of this transformation largely determine what the MapReduce program does. Note that the (key/value) pairs are processed in arbitrary order. The transformation must be self-contained in that its output is dependent only on one single (key/value) pair.
3. The output of all the mappers are (conceptually) aggregated into one giant list of `<k2, v2>` pairs. All pairs sharing the same `k2` are grouped together into a new (key/value) pair, `<k2, list(v2)>`. The framework asks the reducer to process each one of these aggregated (key/value) pairs individually.

Table 1: Overview of input and output in MapReduce data flow

	Input	Output
map	<k1, v1>	list(<k2, v2>)
reduce	<k2, list(v2)>	list(<k3,v3>)

Table 1 shows an overview of Input and Output of mapper and reducer in a MapReduce application.

Hadoop does not allow the keys and values to be of any arbitrary type (e.g., standard Java classes). This is because the MapReduce framework has a certain way of serializing keys and values to move them across a clusters network. All classes used for keys and values have to support this type of serialization. However, Hadoop provides a number of predefined classes, including wrapper classes for all basic data types that can be employed as types for keys and values. Table 2 shows a list of these predefined classes.

Table 2: Predefined Data Types in Hadoop MapReduce

Class	Description
`NullWritable`	Null-Value when key of value is not needed
`BooleanWritable`	Standard Boolean (Wrapper)
`Text`	String in UTF8 format (Wrapper)
`ByteWritable`	Single Byte (Wrapper)
`DoubleWritable`	Double (Wrapper)
`FloatWritable`	Float (Wrapper)
`IntWritable`	Integer (Wrapper)
`LongWritable`	Long (Wrapper)

2.2.1 Parallel Processing

A MapReduce *job* is a unit of work that a client wants to be performed (i.e., a specific analysis task); it consists of the input data, the MapReduce program and configuration information. Hadoop runs a job by dividing into *tasks*. Here two kinds of tasks can be distinguished: map-tasks and reduce-tasks.

The initial input to a MapReduce job is divided into fixed-size pieces called *input splits,* each consisting of a list of *records*. A record is essentially a key/value pair `<k1, v1>`, that can be consumed by the specified mapper for the given job. Hadoop creates one map task for each input split, which runs the user-defined map function for each record in the input split.

The map tasks are performed in parallel on different machines in the cluster. Therefore the map tasks have to be self-contained, i.e., the processing of on key/value pair has to be performed independently from all other tasks of a job.

Hadoop tries to assign map tasks to individual machines on the cluster in such a way, that the map tasks run "close" to the data. This means that a map task is usually performed on a machine, where all (or most) of the records of the corresponding input split are stored. Figure 2 illustrates this process, which is referred to as *data locality optimization*. Note that required records that do not reside on the machine to which a map task is assigned, have to be communicated over the network, which results in an inefficient job performance (White 2010).

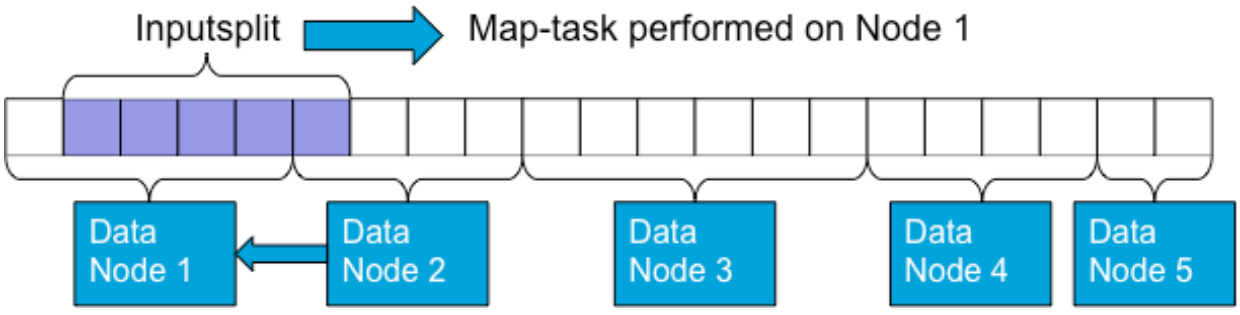

Figure 2: Map task assignment to DataNodes

Here another positive aspect of data replication becomes apparent (cf. section 2.1): If data blocks are replicated over several machines, it is more likely to identify a singular machine that contains all blocks with the necessary record in an input split.

However, a more reliable way to ensure optimal data locality is to set the maximum size of an input split to the size of one block in the HDFS, thus ensuring, that every input split is contained in one block and therefore can be processed locally on one machine.

The output of the map tasks is intermediate data, which can be deleted after completion of the job (i.e., after the reduce tasks are completed). Therefore they do not need to be stored on the HDFS (with replication) and are written to local disk. If a map task fails, than Hadoop will automatically rerun the map task on another node.

The process in which the output from the map tasks is sorted by its key and transported to the reduce tasks is known as *shuffle*. An understanding of this process is crucial for the optimisation of MapReduce jobs.

Each map task has a circular memory buffer, where output data is stored (default: 100 MB). If a defined threshold size of the buffer is filled (default: 80%) a background thread will start to *spill* the output to the local disk. Map output will continue to be written to the buffer while the spill is performed, however if the buffer is completely filled the map task will block until the spill is complete.

During the spill process the background thread divides the output data into partitions corresponding to the reducers that will use individual partitions as input. Furthermore, the entries in every partition are sorted with their key during the spill.

The reduce tasks are performed on machines in the cluster similar to the map tasks. Since every reduce task has to process all `<k2,v2>` key/value pairs with the same key `k2`, all necessary partitions with map output data have to be gathered across the cluster for the reduce task. Therefore the reduce task does not profit from data locality. Since the map tasks may not finish at the same time, the reduce tasks start copying their required partitions as soon as each map task finishes.

The gathered outputs from the map tasks are stored in the memory of the machine that performs a reduce task, if the output is small enough. If the in-memory buffer reaches a threshold size, the gathered data is merged (maintaining the sort order) and again spilled to disk.

When all the output data from the map tasks is gathered, the reduce task executes the user defined reduce function for each key in the gathered data. The output of this phase is usually written to HDFS, since it is the overall output of the MapReduce Job. Every reducer generates an output file.

Figure 3 shows an overview of a complete MapReduce Job.

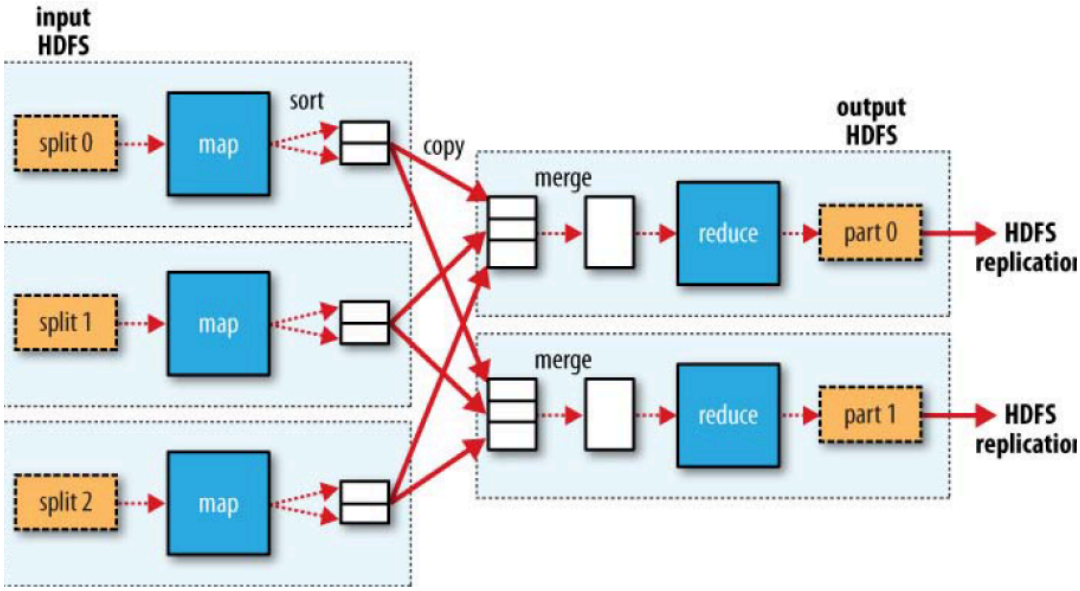

Figure 3: Overview of a MapReduce job (White, 2010)

Similar to the storage component of Hadoop is the analysis component of a MapReduce job performed by daemons running on the machines in the cluster in a master/slave architecture (White 2010):

2.2.1.1 JobTracker daemon

The JobTracker is the master in the MapReduce analysis master/slave architecture of Hadoop. The JobTracker is often run on the same server as the NameNode daemon of the HDFS component. Once the code for a specific MapReduce Job is submitted to the cluster, the JobTracker determines an execution plan

by identifying the files that are to be processed and assigning tasks to other machines in the cluster. The JobTracker also monitors the tasks as they are running. If a task fails, the JobTracker will automatically relaunch it, possibly on a different machine in the cluster, up to a predefined limit of retries.

2.2.1.2 TaskTracker daemon

The slaves in the MapReduce architecture are called TaskTrackers. They manage the execution of individual tasks on the machine they are running on. Note, that a TaskTracker is able to launch map tasks as well as a reduce tasks.

Each TaskTracker is responsible for executing the individual tasks that the JobTracker assigns. Although there is a single TaskTracker per slave node, each TaskTracker can spawn multiple Java Virtual Machines to handle many map or reduce tasks in parallel.

Every TaskTracker constantly communicates with the JobTracker and reports the progress of the assigned task. If the JobTracker fails to receive a signal from a TaskTracker within a specified amount of time, it will assume the TaskTracker has crashed and will resubmit the currently assigned task to another TaskTracker.

2.2.2 Join operations in MapReduce jobs

For specific data analysis tasks often two or more different dataset have to be considered. The process to reconcile the different datasets for a MapReduce job is called *join*. Joins can be performed in different ways, depending on the size and partitioning of the input datasets. The decision how to implement a join for a MapReduce job can have significant influence on the execution time of the job. Therefore we give an overview of the different approaches to joins in MapReduce jobs in this section, and provide guidance how to choose the optimal approach for a given job.

2.2.2.1 Side Data Distribution

If only one of the input datasets constitutes "big data" (i.e., a large dataset), while the other input datasets are fairly small (e.g., metadata about the large dataset), it is a valid option to copy the small datasets on all machines in the cluster that run tasks. This process of distributing data to all TaskTrackers is called *side data distribution* (White, 2012). It allows the individual TaskTrackers to effect the join within the performed map function (or reduce function). Figure 4 illustrates the side data distribution for two input datasets.

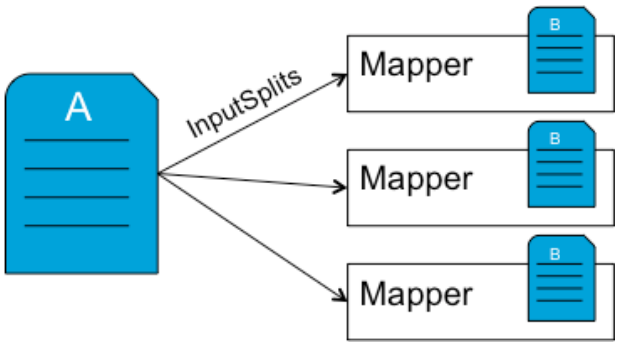

Figure 4: Side data distribution for a large dataset A and a small side data B. Join is performed in the map function.

Side data can be seen as extra read-only data needed by a job to process the main dataset. To distribute the side data on all machines in the cluster two approaches are possible:

1. *Job configuration:* In the configuration files of a Hadoop system arbitrary key/value pairs can be defined. These defined key/value pairs can be read from within a map or reduce function, since every TaskTracker reads the configuration files before a job is launched.

   However this approach is only feasible if the side data is only a few kilobytes, since the configuration files are read into the memory of every machine that runs a TaskTracker or the JobTracker, even if they are not needed. Note that the memory size of the machines should not be seen as the upper limit for this approach, since the calculations performed on the machines with a

TaskTracker are also memory intensive and therefore the overall performance of a job depends on the available memory during execution.

2. *Distributed cache:* Files containing side data can also be specified to be used with the distributed cache of Hadoop. Such files can reside on an arbitrary file system connected to the Hadoop cluster.

   When a job is launched the specified files are copied to the file system of the machine that runs the JobTracker (usually HDFS). Before the individual tasks are started the TaskTrackers copy the files to their local disk (i.e., the cache), allowing the files to be used within their tasks.

   The TaskTracker also monitors if the files are needed for additional tasks from the same job. If the files are no longer needed they are deleted to make room for new files when the cache exceeds a defined size (default: 10 GB).

   This approach can handle larger side datasets than the distribution with the configuration files, since it does not use the memory of the machines. At the same time the required disk read and write operations can make this approach slower that using the configuration files.

The resulting network traffic from the distribution of the side data can have a negative impact on the job performance. If side data distribution can and should be used, depends to large extend on the employed Hadoop cluster. More precisely the following characteristics of the cluster have to be considered:

- *Size of the cluster*: More available machines in the cluster require the data to be copied more often. Therefore side data distribution is more suited for smaller clusters.
- *Network bandwidth*: The higher the available bandwidth in a network is, the faster can the data be distributed. Therefore side data distribution is more suited in networks with high bandwidth.

2.2.2.2 Map-side Joins

A map-side join works by performing the join before the data reaches the map function as depicted in Figure 5. For this to work, though, the inputs to each map must be partitioned and sorted in a particular way. Each input dataset must be divided into the same number of partitions, and it must be sorted by the same key (the join key) in each source. All the records for a particular key must reside in the same partition.

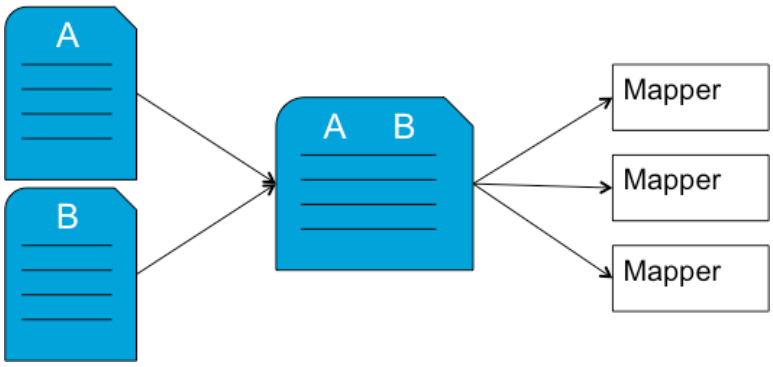

Figure 5: Map-side join of two large datasets A and B

Hadoop provides a special Class (`CompositeInputFormat`) that can be used to determine the input splits for map-side joins: The intermediate records `<k1, v1A>` and `<k1, v1B>` of two partitions that are to be joined (due to the same join key) are combined to a key/value pair `<k1, v1A+v1B>`, with is the record of the actual input split used in one of the map tasks.

The downside of this approach is that an optimal data locality for the execution of the map tasks cannot be guaranteed: since the intermediate records that are combined to a record in the actual input split might not reside on the same machine, Hadoop has to transfer parts of the data to perform a map task.

Due to the very strict requirements on the input datasets (divided in the same number of partitions sorted by the same key) map-side joins are only suited for a very specific scenario: output data of other MapReduce jobs with the same number of reducers and the same key fulfil the requirements of map-side joins. Therefore map-side joins can be used to further process data resulting from a series of such previous MapReduce jobs.

2.2.2.3 Reduce-side Joins

Reduce-side joins are the general solution to join datasets with Hadoop. Here the datasets do not have to be structured in any particular way, however this approach is less efficient since the complete MapReduce framework is employed to combine the datasets. This means that a complete MapReduce job is performed to generate a combined input dataset for further processing. Therefore this approach should only be used if none of the other described approaches can be employed.

We describe how reduce-side joins work as illustrated in Figure 6.

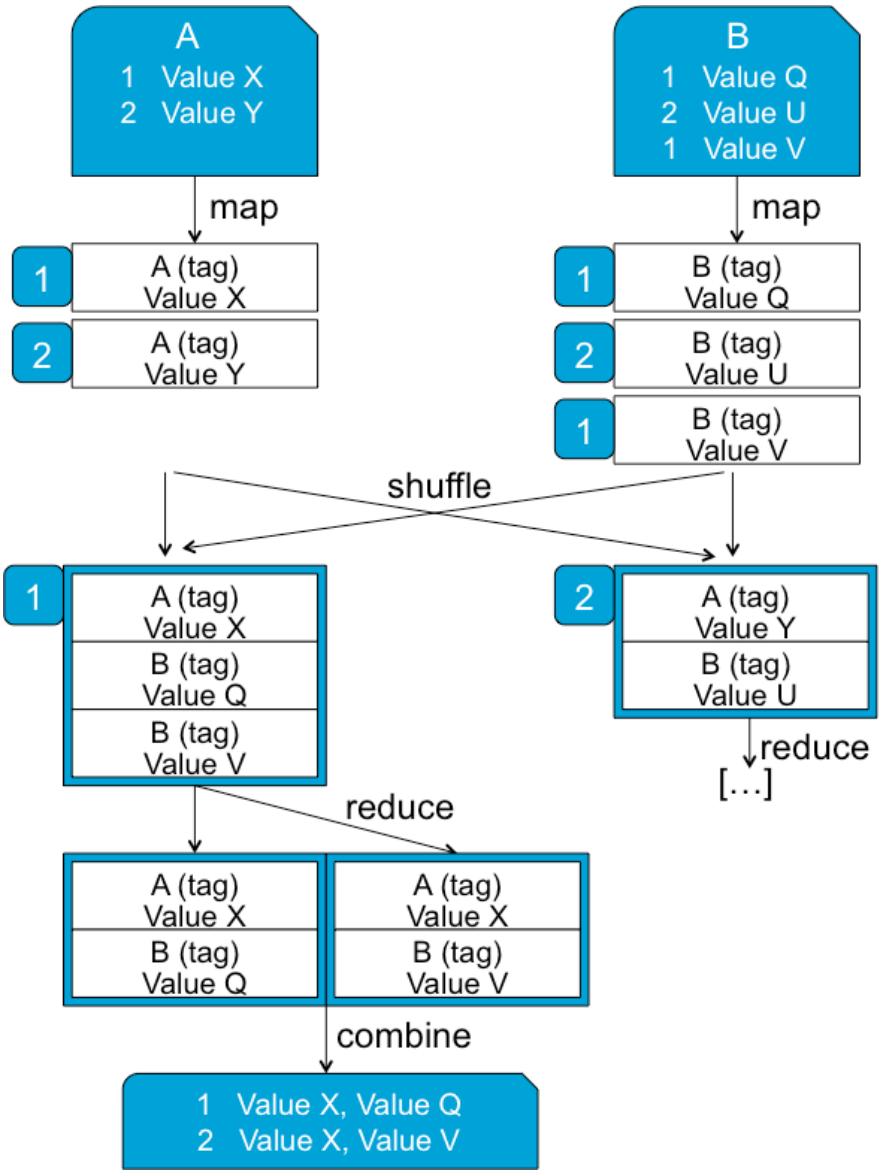

Figure 6: Data flow of a reduce-side join

In a reduce-side join, the map is to read the (potentially heterogeneous) records of the different datasets and package them to key value pairs, where the key is the join key and the value is the original complete record, tagged with the data source (e.g., filename). This implies that the key with which the records are supposed to be joined has to be extracted by the map function from the records of all datasets.

As described in section 2.2.1, in the shuffle phase all key/value pairs are sorted according to their keys and passed to the reducers, so that one reducer processes all pairs with the same key. In the reduce phase the full

cross-product of the values is calculated according to the contained data source tag. The resulting value pairs are finally combined to the records of the joined dataset (Lam 2011).

2.3 Hadoop Job Optimisation

The time required to perform tasks with large-scale data is usually the most critical factor when deciding if a task is feasible. In this section we provide guidelines and describe techniques to increase the performance of data analysis tasks performed with Hadoop in particular in terms of execution time.

2.3.1 Input Data

To optimise the execution time of Hadoop jobs not only the necessary time necessary for the calculation in the actual task must be considered, but also the time required to read the initial input data can be an important factor. To adjust MapReduce programs or even the employed Hadoop cluster specifically for the input data that is to be consumed by a job can result in a significant decrease of execution time. The same holds true if the input data itself can be easily pre-processed to reduce the necessary workload in a job.

In the initial step of a MapReduce job the input data is divided into input splits to which the map tasks are assigned. This process can be an initial bottleneck for the performance of a MapReduce job, since it is performed by the JobTracker on a single machine. To achieve an optimal performance when determining the input splits, the individual records in a split should be identifiable with as little computational effort as possible. Good examples for records that are easy to identify include:

- *Line-by-line records*: The key of a record is the byte offset of the beginning of a line in the input data file or the line number. The value in a record is the content of the line.
- *File-by-file records:* The key of a record is the filename. The value is the content of the file.

In every case it should be avoided to sort the input data or group them together to input splits according to some characteristics within the data. This would lead to a significant increase in execution time, since it would be executed on the single machine of the JobTracker. Furthermore it would lead to input splits that are fragmented over the cluster, which implies the loss of optimal data locality. Grouping and sorting of input data are operations that have to be performed in the map or shuffle phase of a MapReduce job.

Problems arise when the input data is in a structured format (e.g., XML) that requires a more elaborate parsing of the data to identify records. In such cases (if file-by-file records can also not be employed) the simplest pattern possible to identify the desired records. E.g., searching for XML elements by the name of the XML-tags, rather than by the structure.

Often MapReduce jobs do not use all of the accessible input data. Reducing the number of bytes to read can enhance the overall throughput of the job. E.g., if the structure of the input data is known, and the necessary data to perform a MapReduce job can be found in the first 20 characters of every line in the input dataset, the value of line-by-line records should not be set to the complete content of the line, but only to the first 20 characters.

Finally another value that can be adjusted to optimize the job performance is the size of the input splits and in turn the number of splits created out of the input data. Having many splits means the time taken to process each split is small compared to the time to process the whole input. So if the splits are processed in parallel, the processing is better load-balanced if the splits are small, since a faster machine will be able to process proportionally more splits over the course of the job than a slower machine. Even if the machines are identical, failed processes or other jobs running concurrently make load balancing desirable, and the quality of the load balancing increases as the splits become more fine-grained.

On the other hand, if splits are too small, then the overhead of managing the splits and of map task creation begins to dominate the total job execution time. For most jobs, a good split size tends to be the size of a HDFS block (White 2012), 64 MB by default, although this can be changed for the cluster (for all newly created files), or specified when each file is created.

RDF Data

The considerations for input data of MapReduce jobs hold true for input data in RDF format: here especially relevant is that different serializations of RDF data are more suited than others: serialisations that allow to read triples line by line, independently of other triples should always be preferred (e.g., N-Tuple), while more structured serializations are to be avoided if possible (e.g., RDF/XML, N3).

Recent results (Huang et al. 2011) have shown that some of the standard configurations of Hadoop should be changed to achieve a considerable performance increase when handling graph structured data like RDF:

- By default Hadoop uses hash functions to decide where on a cluster a data block is stored. For graph structured data this usually results in vertices being randomly distributed across the cluster, due to the result of the hash function applied to the identifier of the vertices. This can be a significant disadvantage for graph operations (e.g., graph pattern matching). For such operations the graph is traversed by passing through neighbours of vertices, however data that is close to each other in the graph can end up spread out across the cluster on different physical machines. This leads to a significant increase in network traffic.

  To achieve a better data locality, neighbours in a graph should be stored close to each other. Therefore, to store graph data a clustering algorithm should be employed rather than hash functions.

- As described in section 2.2.1 the replication of data serves not only for fault tolerance but also helps to achieve better data locality, since it increases the possibilities to assign map tasks to machines. The default approach of Hadoop is to replicate every data block a fixed number of times. However for graph data it is beneficial to replicate data that is on the border of a partition of the graph more often than data that already has all its neighbours stored locally in the same partition:

  Vertices that are stored on the border of a partition are likely to have neighbours on other machines in the cluster. If these vertices are replicated more often it is likely that neighbours are found on the same physical machine, thus reducing network traffic.

2.3.2 Shuffle Optimisation

The phase in a MapReduce job where the output data of the map tasks is transferred to the reduce tasks is a point where a lot of job specific optimisations are possible to decrease execution time. Table 3 shows an overview of the most important properties that can be changed to optimise the shuffle process.

Memory

The output of the map tasks is written to a memory buffer and spilled to disk if it is filled up to a defined threshold. If the buffer fills up completely during a spill, the map task block until the spill is completed. This should be avoided if possible, either by lowering the threshold or by increasing the size of the memory buffer for jobs with very fast map tasks.

It should also be avoided to have multiple spills to disk per map task. To achieve one spill per map task for optimal performance the size of the memory buffer has to be adjusted, however this is only possible if the size of the map output (per task) can be estimated ex ante.

On the reduce side, the best performance is obtained when the intermediate data can reside entirely in memory. By default, this does not happen, since for the general case all the memory is reserved for the reduce function. The proportion of heap size, that can be used to retain map output data in memory is by default 0, but can be changed if the memory requirements of the reduce function are not too high. Note that there is a tradeoff between the time saved due to less disk I/O operations and the performance decrease of the reduce tasks due to less available memory.

The general principle is to give the shuffle as much memory as possible. The amount of memory given to the Java virtual machines (JVM) in which the map and reduce tasks should be made as large as possible for the amount of memory on the machines. By default Hadoop allocates 1GB of memory for each daemon on a machine (TaskTracker and DataNode). In addition, the TaskTracker launches separate child JVMs to run map and reduce tasks in, which need be factored into the total memory footprint of a worker machine.

The number of tasks that can be launched simultaneously (default 2 map tasks and 2 reduce tasks) should be governed by the number of available processors on the machines. Because MapReduce jobs are normally I/O-bound, it makes sense to have more tasks than processors to get better utilization. The amount of oversubscription depends on the CPU utilization of each job, but a good rule of thumb is to have a factor of between one and two more tasks (counting both map and reduce tasks) than processors. The amount of memory allocated to each child JVM can also be set (default: 200 MB).

The overall memory allocation should fit into the available memory of the machine with other processes running on the machine taken into account. It has to be avoided that processes are swapped out of the memory as this it leads to severe performance degradation. The precise memory settings are necessarily very cluster-dependent, and can be optimized over time with experience gained from monitoring the memory usage across the cluster (White, 2010).

To further optimize the use of JVMs the number of tasks a single JVM can perform can be configured (default: 1 task): using separate JVM for every map and reduce task incurs start-up costs for each task. If the map tasks do their own initialization, such as reading into memory a large data structure that initialization is part of the start-up cost as well. If each task runs only briefly then the start-up cost can be a significant portion of a task's total run time. However Hadoop allows the reuse of already started JVMs, thus amortizing the start-up costs across multiple tasks of the same job.

Compression

Map output is spilled to disk and then copied to the machine performing a reduce task. Since disk I/O operations are very crucial in terms of job execution time – especially for large-scale data – it can be beneficial to compress these intermediate data. File compression brings two major benefits: it reduces the space needed to store files, and it speeds up data transfer across the network, or to or from disk.
However the reduce tasks need to decompress the data to process them, for which they need additional time. This means that there is a tradeoff to be considered. The employed compression algorithm can influence this tradeoff: Different compression formats have different characteristics and exhibit a space/time trade-off: faster compression and decompression speeds usually come at the expense of smaller space savings. ZIP can be recommended as a general-purpose compressor, since it is in the middle of the space/time trade-off. Further ZIP supports splitting (at file boundaries), this means that is possible to seek at any point in the stream and start reading at any point further on. Splittable compression formats are especially suitable for MapReduce jobs.

Table 3: Configuration properties that can be used to optimise the job performance

Property	Type	Default value	Description
io.sort.mb	int	100	Size of the memory buffer to use when sorting map output.
io.sort.spill.percent	float	0.80	Threshold usage proportion for the memory buffer to start spilling to disk.
mapred.compress.map.output	boolean	false	Compress map output.
mapred.map.output.compression.codec	Class name	org.apache.hadoop.io.compress.DefaultCodec	The compression codec to use for map outputs.
mapred.reduce.copy.backoff	int	300	The maximum amount of time, in seconds, to spend retrieving one map output for a reducer before declaring it as failed.
mapred.inmem.merge.threshold	int	1000	The threshold number of map outputs for starting the process of merging the outputs and spilling to disk.
mapred.job.reduce.input.buffer.percent	float	0.0	Proportion of total heap size to be usedfor retaining map outputs in memory during the reduce. For the reduce phase to begin,

			the size of map outputs in memory must be no more than this size.
mapred.tasktracker. map.tasks.maximum	int	2	Maximum number of map tasks that will be run on a TaskTracker simultaneously.
mapred.tasktracker. reduce.tasks.maximum	int	2	Maximum number of reduce tasks that will be run on a TaskTracker simultaneously.
mapred.child.java.opts	String	-Xmx200m	The JVM options used to launch the TaskTracker child process that runs map and reduce tasks.
mapred.job.reuse.jvm. num.tasks	int	1	Maximum number of tasks a JVM can run. A value of -1 means no limit.

2.3.3 Combiner

Combiners can be employed to further improve MapReduce jobs. Combiners add an additional combine phase between map and reduce. They are supposed to condense the output of the map tasks. Therefore Combiners can reduce the amount of data shuffled between the map and reduce phases, and lower network traffic improves execution time.

Combiner can be seen as local reducers, who perform a reduce task over the output of a map job locally on the same TaskTracker the map task was performed. This means combiner do not aggregate the output of multiple map tasks like it is done the actual reduce phase.

Since the combiner function is an optimization, Hadoop does not provide a guarantee of how many times it will call it for a particular map output record. This implies that for a combiner to work, it must be an equivalent transformation of the data with respect to the reducer (Lam 2011). If we take out the combiner, the reducer's output will remain the same. Furthermore, the equivalent transformation property must hold when the combiner is applied to arbitrary subsets of the intermediate data.

As an example consider a MapReduce job where the map tasks extract a list of numerical values from the input data and the reduce task determines the maximum value of the extracted data. Here a combiner can be employed to directly identify the maximum value from the subset of result values generated by the individual map tasks. Thus, only a small portion of the initial map output has to be transferred to the reducer, where less data has to be processed (potentially allowing to keep it in memory for further optimization as described in section 2.3.2). This is illustrated in Figure 7.

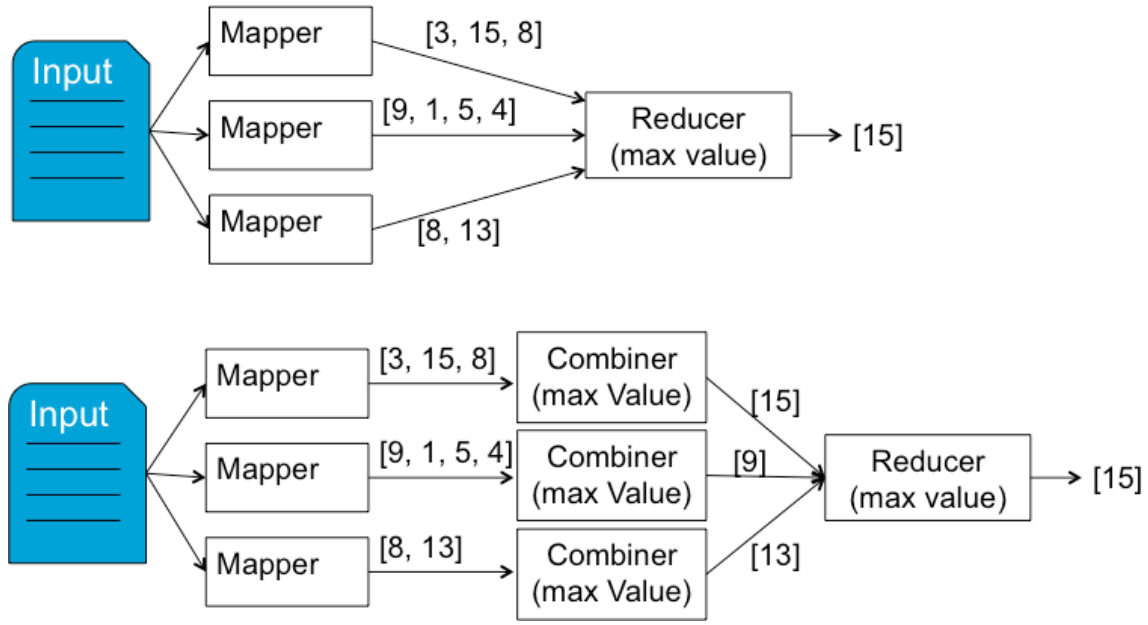

Figure 7: MapReduce to determine maximum value with combiner (bottom) and without combiner (top)

2.3.4 Speculative Execution

The MapReduce framework specifies the map tasks and the reduce tasks to be idempotent. This means that when a task fails, Hadoop can restart that task and the overall job will end up with the same result. Hadoop can monitor the health of running nodes and restart tasks on failed nodes automatically. This makes fault tolerance transparent to the developer.

However this behavior can also be exploited to increase the performance of the jobs: Tasks often run slow because of temporary congestion. This doesn't affect the correctness of the running job but certainly affects its performance. Even one slow-running task will delay the completion of a MapReduce job. Until all mappers have finished, none of the reducers will start running. Similarly, a job is not considered finished until all the reducers have finished.

Hadoop uses the idempotency property again to mitigate the slow-task problem. Instead of restarting a task only after it has failed, Hadoop will notice a slow-running task and schedule the same task to be run in another node in parallel. Idempotency guarantees the parallel task will generate the same output. Hadoop will monitor the parallel tasks. As soon as one finishes successfully, Hadoop will use its output and kill the other parallel tasks. This entire process is called *speculative execution* (Lam, 2011). Speculative execution is turned on by default and should only be turned off if the tasks of a job have side effects and are therefore not idempotent (e.g., a task writes to external files).

2.4 Conclusion

Hadoop is a software library that provides a reliable shared storage and analysis system, with storage provided by HDFS and analysis by MapReduce. However, designing MapReduce jobs can be intricate and rely on the distributed system architecture of Hadoop does not guarantee desired processing speeds for large-scale data.

We described how MapReduce jobs can be designed to achieve performance goals in general. Furthermore we provided guidance on how to optimise MapReduce jobs by detailing the different attributes of a MapReduce job or the Hadoop cluster itself that have to be addressed to optimise the execution time of a task and to make the processing of large-scale data feasible.

3 Web Data Commons – Structured Data from Large Web Corpora

In this section we describe our experience with the Web Data Commons project[2] (Mühleisen et al. 2012), for which we extracted all Microformat, Microdata and RDFa data from the Common Crawl[3] web corpus. The common crawl web corpus is the largest and most up-to-date web corpus that is currently available to the public. Furthermore, we provide the extracted data for download in the form of RDF-quads and CSV tables.

Here we give an overview of the project and present extracted statistics about the popularity of the different encoding standards as well as the kinds of data that are published using each format. We describe in detail our experiences when extracting the data and processing the results for publication.

3.1 Introduction

In recent years, much work has been invested in transforming the so-called "eyeball" web, where information is presented for visual human perception towards a "Web of Data", where data is produced, consumed and recombined in a more or less formal way. A part of this transformation is the increasing number of websites, which embed structured data into their HTML pages using different encoding formats. The most prevalent formats for embedding structured data are: Microformats, which use style definitions to annotate HTML text with terms from a fixed set of vocabularies; RDFa, which is used to embed any kind of RDF data into HTML pages; and Microdata, a recent format developed in the context of HTML5.

The embedded data is crawled together with the HTML pages by Google, Microsoft and Yahoo!, which use the data to enrich their search results. These companies have so far been the only ones capable of providing insights into the amount as well as the types of data that are currently published on the Web using Microformats, RDFa and Microdata. While a previously published study by Yahoo! Research (Mika, 2011) provided many insights; the analyzed web corpus is not publicly available. This prohibits further analysis and the figures provided in the study have to be taken at face value.

However, the situation has changed with the advent of the Common Crawl. Common Crawl is a non-profit foundation that collects data from web pages using crawler software and publishes this data. So far, the Common Crawl foundation has published three Web corpora, one dating 2009/2010, one dating February 2012 and recently a corpus dating July 2012 (which is not yet part of the Web Data Commons project). Together the first two corpora contain over 4.5 Billion web pages. Pages are included into the crawls based on their PageRank score, making these corpora snapshots of the popular part of the web.

The Web Data Commons project has extracted all Microformat, Microdata and RDFa data from the Common Crawl web corpora and provides the extracted data for download in the form of RDF-quads and CSV tables.

3.2 Embedding Structured Data

This section summarizes the basics about Microformats, RDFa and Microdata and provides references for further reading.

3.2.1 Microformats

An early approach for adding structure to HTML pages was Microformats[4]. Microformats define a number of fixed vocabularies to annotate specific things such as people, calendar entries, products etc. within HTML pages. Well known Microformats include hCalendar for calendar entries according to RFC2445, hCard for people, organizations and places according to RFC2426, geo for geographic coordinates, hListing for classified ads, hResume for resume information, hReview for product reviews, hRecipe for cooking recipes, Species for taxonomic names of species and XFN for modeling relationships between humans.

[2] See: http://www.webdatacommons.org
[3] See: http://commoncrawl.org
[4] See: http://microformats.org

For example, to represent a person within a HTML page using the hCard Microformat, one could use the following markup:

```
<span class ="vcard">
<span class ="fn">Jane Doe</span>
</span>
```

In this example, two inert elements are used to first create a person description and then define the name of the person described. The main disadvantages of Microformats are their case-by-case syntax and their restriction to a specific set of vocabulary terms.

To improve the situation, the newer formats, RDFa and Microdata, provide vocabulary-independent syntaxes and allow terms from arbitrary vocabularies to be used.

3.2.2 RDFa

RDFa defines a serialization format for embedding RDF data (Klyne et al. 2004) within (X)HTML pages. RDFa provides a vocabulary-agnostic syntax to describe resources, annotate them with literal values, and create links to other resources on other pages using custom HTML attributes. By also providing a reference to the used vocabulary, consuming applications are able to discern annotations. To express information about a person in RDFa, one could write the following markup:

```
<span xmlns:foaf=”http://xmlns.com/foaf/0.1/” typeof="foaf:Person">
<span property="foaf:name">Jane Doe</span>
</span>
```

Using RDFa markup, we refer to an external, commonly-used vocabulary and define an URI for the thing we are describing. Using terms from the vocabulary, we then select the “Person” type for the described thing, and annotate it with a name. While requiring more markup than the hCard example above, considerable flexibility is gained. A mayor supporter of RDFa is Facebook, which has based its Open Graph Protocol[5] on the RDFa standard.

3.2.3 Microdata

While impressive, the graph model underlying RDF was thought to represent entrance barriers for web authors. Therefore, the competing Microdata format emerged as part of the HTML5 standardization effort. In many ways, Microdata is very similar to RDFa, it defines a set of new HTML attributes and allows the use of arbitrary vocabularies to create structured data. However, Microdata uses key/value pairs as its underlying data model, which lacks much of the expressiveness of RDF, but at the same time also simplifies usage and processing. Again, our running example of embedded structured data to describe a person is given below:

```
<span itemscope itemtype="http://schema.org/Person">
<span itemprop="name">Jane Doe</span>
</span>
```

We see how the reference to both type and vocabulary document found in RDFa is compressed into a single type definition. Apart from that, the annotations are similar, but without the ability to mix vocabularies as it is the case in RDFa. The Microdata standard gained attention as it was selected as preferred syntax by the Schema.org initiative, a joint effort of Google, Bing and Yahoo, which defines a number of vocabularies for common items and carries the promise that data that is represented using these vocabularies will be used within the applications of the founding organizations.

3.3 Format Usage

For the Web Data Commons project we used two web corpora released by the Common Crawl foundation. The first corpus contains web resources (pages, images...) that have been crawled between September 2009 and September 2010. The second corpus contains resources dating February 2012, thereby yielding two distinct data points for which we can compare the usage of structured data within the pages. As a first step, we have filtered the web resources contained in the corpora to only include HTML pages, thus reducing the

[5] See: http://ogp.me

size of the input data of our extraction jobs. Table 4 shows a comparison of the two corpora. We can see how HTML pages represent the bulk of the corpora. The newer crawl contains fewer web pages. 148 million HTML pages within the 2009/2010 crawl contained structured data, while 189 million pages within the 2012 crawl contained structured data. Taking the different size of the crawl into account, we can see that the fraction of web pages that contain structured data has increased from 6% in 2010 to 12% in 2012. The absolute numbers of web pages that used the different formats are given in Table 5. The datasets that we extracted from the corpora consist of 3.2 billion RDF quads (2012 corpus) and 5.2 billion RDF quads (2009/2010 corpus).

Table 4: Comparison of the Common Crawl corpora used for Web Data Commons

	2009/2010 corpus	**2012 corpus**
Crawl Dates	09/2009-09/2011	02/2012
Total URLs	2.8 Billion	1.7 Billion
HTML Pages	2.5 Billion	1.5 Billion
Pages with Data	148 Million	189 Million

Table 5: URLs using the different formats

Format	**2009/2010 corpus**	**2012 corpus**
RDFa	14,314,036	67,901,246
Microdata	56,964	26,929,865
geo	5,051,622	2,491,933
hcalendar	2,747,276	1,506,379
hcard	83,583,167	61,360,686
hlisting	1,227,574	197,027
hresume	387,364	20,762
hreview	2,836,701	1,971,870
species	25,158	14,033
hrecipe	115,345	422,289
xfn	37,526,630	26,004,925

Figure 8 shows the distribution of the different formats as a percentage of the number of URLs with the respective format to the total number of URLs within the corpora and compares the fractions for the 2009/2010 corpus and the 2012 corpus. We see that RDFa and Microformats gain popularity, while the usage of the single-purpose Microformats remain more or less constant. The reason for the explosive adoption of the Microdata syntax between 2010 and 2012 might be the announcement in 2011 that Microdata is the preferred syntax of the Schema.org initiative.

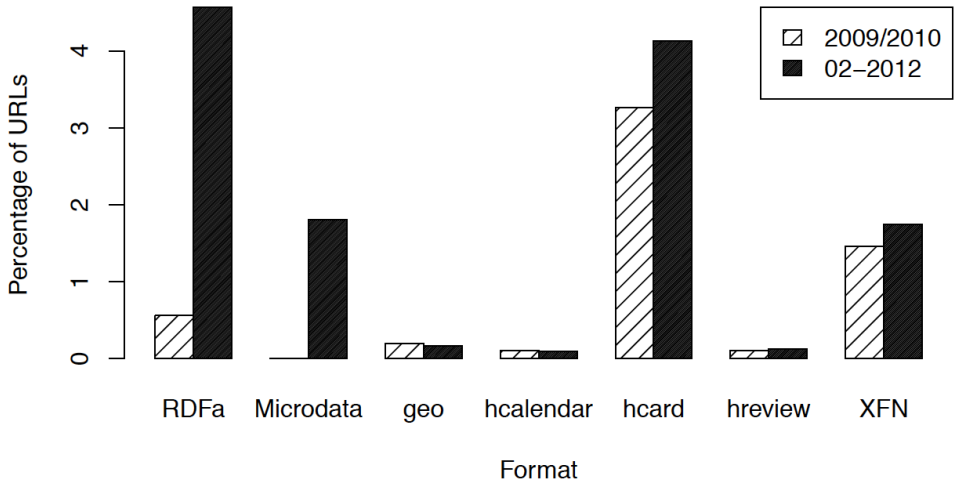

Figure 8: Common Crawl corpora - format distribution

3.4 Types of Data

While each Microformat can only be used to annotate the specific types of data it was designed for, RDFa and Microdata are able to use arbitrary vocabularies. Therefore, the format comparison alone does not yield insight into the types of data being published. RDFa and Microdata both support the definition of a data type for the annotated entities. Thus simple counting the occurrences of these types can give an indicator of their popularity. The top 20 values for type definitions of the RDFa data within the 2012 corpus are given in Table 6. Type definitions are given as shortened URLs, using common prefixes[6]. Note that gd: stands for Google's Data-Vocabulary, one of the predecessor of Schema.org.

We have then manually grouped the 100 most frequently occurring types by entity count into groups. These groups are given in Table 8. The most frequent types were from the area of website structure annotation, where for example navigational aids are marked. The second most popular area is information about people, businesses and organizations in general, followed by media such as audio files, pictures and videos. Product offers and corresponding reviews represent the fourth most frequent group, and geographical information such as addresses and coordinates was least frequent. Groups below 1 % frequency are not given.

Table 9 shows the same analysis for the Microdata data within the 2012 corpus. Apart from variations in the specific percentages, the same groups were found to be most frequently used. An interesting observation was that only two of the 100 most frequently occurring types were not from of the Schema.org namespace, confirming the overwhelming prevalence of types from this namespace, which is not surprising since the Microdata format itself was made popular by this initiative. This is also shown in the listing of the 20 most frequent type definitions given in Table 7, where all URLs originate from either the Schema.org domain or the Data-Vocabulary domain.

[6] See: http://prefix.cc/popular.file.ini

Table 6: Top 20 Types for RDFa

Type	Entities
gd:Breadcrumb	13,541,661
foaf:Image	4,705,292
gd:Organization	3,430,437
foaf:Document	2,732,134
skos:Concept	2,307,455
gd:Review-aggregate	2,166,435
sioc:UserAccount	1,150,720
gd:Rating	1,055,997
gd:Person	880,670
sioctypes:Comment	666,844
gd:Product	619,493
gd:Address	615,930
gd:Review	540,537
mo:Track	444,998
gd:Geo	380,323
mo:Release	238,262
commerce:Business	197,305
sioctypes:BlogPost	177,031
mo:SignalGroup	174,289
mo:ReleaseEvent	139,118

Table 7: Top 20 Types for Microdata

Type	Entities
gd:Breadcrumb	18,528,472
schema:VideoObject	10,760,983
schema:Offer	6,608,047
schema:PostalAddress	5,714,201
schema:MusicRecording	2,054,647
schema:AggregateRating	2,035,318
schema:Product	1,811,496
schema:Person	1,746,049
gd:Offer	1,542,498
schema:Article	1,243,972
schema:WebPage	1,189,900
gd:Rating	1,135,718
schema:Review	1,016,285
schema:Organization	1,011,754
schema:Rating	872,688
gd:Organization	861,558
gd:Product	647,419
gd:Person	564,921
gd:Review-aggregate	539,642
gd:Address	538,163

Table 8: Entities by area for RDFa

Area	% Entities
Website Structure	29 %
People, Organizations	12 %
Media	11 %
Products, Reviews	10 %
Geodata	2 %

Table 9: Entities by area for Microdata

Area	% Entities
Website Structure	23 %
Products, Reviews	19 %
Media	15 %
Geodata	8%
People, Organizations	7%

3.5 Extraction Process

The Common Crawl datasets are stored in the AWS Simple Storage Service[7] (S3) in 100MB compressed arc files. To avoid additional time costs for downloading, we performed the extraction in the Amazon Elastic

[7] See: http://aws.amazon.com/s3/

Compute Cloud[8] (EC2). The main criteria here are the costs to achieve a certain task. Extracting structured data had to be performed in a distributed way in order to finish this task in a reasonable time. We developed two extraction tools to identify structured data in the HTML pages and transform them into RDF-quads:

- Based on a Hadoop MapReduce job: The Common Crawl foundation provides a custom `arc-InputSplitFormat` class, which allows us to generate input splits with individual homepages as records for the map tasks. In a map task the structured data from the homepages are extracted and transformed to RDF-quads. The subsequent reduce task simple writes the extracted quads to disk.
- Based on the Amazon Simple Queue Service (SQS): SQS provides a message queue implementation, which we used to coordinate 100 extraction nodes. Since the Common Crawl corpora are already partitioned into separate files of around 100MB each, we added the identifiers of each of these files as messages to the queue. The extraction nodes share this queue and take file identifiers from it. The corresponding file was then downloaded from S3 to the node. The compressed archive was split into individual web pages. We extracted the structured data from the pages. The resulting RDF-quads were written back to S3 together with extraction statistics and later collected. This process is illustrated in Figure 9.

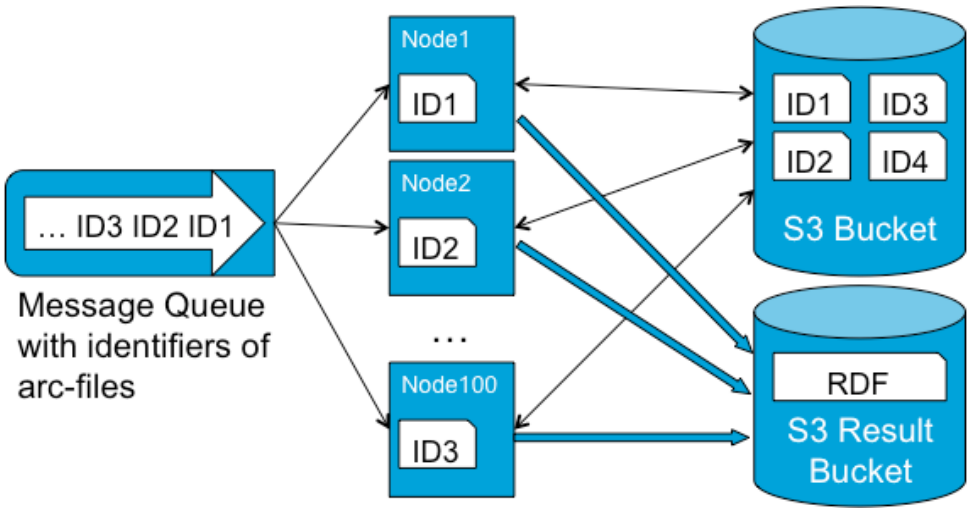

Figure 9: Web Data Commons extraction process on Amazon SQS

Both tools are publicly available[9]. For the actual extraction of data from both corpora we used the SQS-based approach, because Amazon charges additional fees for using Hadoop infrastructure on EC2.

On each page, we ran our RDF extractor based on the Anything To Triples (Any23) library[10]. Any23 parses web pages for structured data by building a DOM tree and then evaluates XPath expressions to extract the structured data. While profiling, we found this tree generation to account for much of the parsing cost, and we have thus searched for a way to reduce the number of times this tree is built. Our solution was to run regular expressions against each archived web page prior to extraction, which detected the presence of structured data within the HTML page, and only to run the Any23 extractor when the regular expression found potential matches. The extraction process on a singular Node is illustrated in Figure 10.

[8] See: http://aws.amazon.com/ec2/
[9] See: https://subversion.assembla.com/svn/commondata/
[10] See: http://incubator.apache.org/any23/

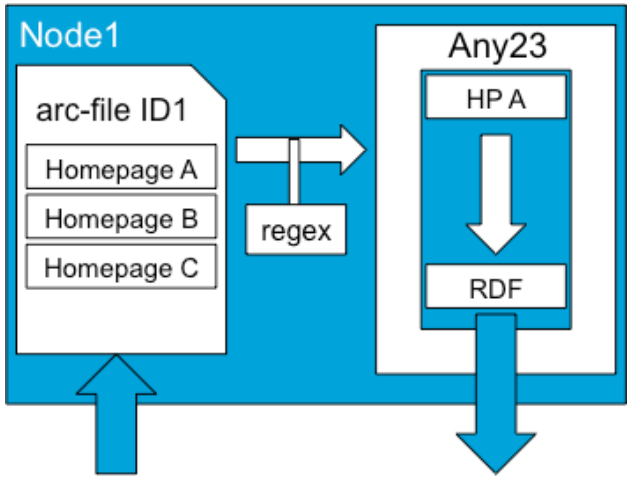

Figure 10: Extraction of structured data on a singular node

The costs for parsing the 28.9 Terabytes of compressed input data of the 2009/2010 Common Crawl corpus, extracting the RDF data and storing the extracted data on S3 totaled 576 EUR (excluding VAT) in Amazon EC2 fees. We used 100 spot instances of type c1.xlarge for the extraction, which altogether required 3,537 machine hours. For the 20.9 Terabytes of the February 2012 corpus, 3,007 machine hours at a total cost of 523 EUR were required.

3.6 Data transformation to CSV-Tables

Additionally to the RDF-Quads format of the extracted data from the Web Data Commons project, we also wanted to provide CSV-tables of the structured data with a fixed schema (i.e., Microformats). For data with a fixed schema we can leverage that their structure is homogeneous, i.e., extracted RDF quads from Microformat data use only a constrained set of properties. The values for these properties (for the complete Microformat dataset) can be extracted with a SPARQL query. Note that this is not possible for RDFa data and Microdata, since they allow for the use of arbitrary vocabularies, which would imply the need to design an individual SPARQL query for every used vocabulary.

We used Hadoop MapReduce Jobs to execute the queries and transform the extracted values into CSV tables. A distributed framework was necessary to execute the queries over billions of triples in a reasonable amount of time.

For the necessary infrastructure we made use of the KIT OpenCirrus[11] Hadoop Cluster. OpenCirrus is a collaboration of several organizations to provide an open cloud-computing research test bed designed to support research into the design, provisioning and management of services at a global scale. For our analysis we used 54 work nodes, each with a 2.27 GHz 4-Core CPU and 100GB RAM.

For the MapReduce jobs we leveraged that an extracted Microformat dataset never originates from several homepages. The extracted RDF-quads contain in the fourth position of every quad (i.e., context node) the URL of the homepage they were extracted from. This implies that every quad in a set of RDF-quads that matches a graph pattern in one of our SPARQL queries has the same context node. Therefore we developed an `N-Quad-RecordReader` that generates input splits for the map tasks, which contain records of the form `<context, triples>.` The records are key/value pairs with the set of all triples as value, which have the same context node. The key is simply the URL of the homepage, from which the triples originate. Note that the `N-Quad-RecordReader` is able to identify the records in a scalable manner without reading the complete dataset for every record, due to the fact that the RDF-quads are already sorted by their context node. Figure 11 Illustrates how the `N-Quad-RecordReader` works.

11 See: https://opencirrus.org/

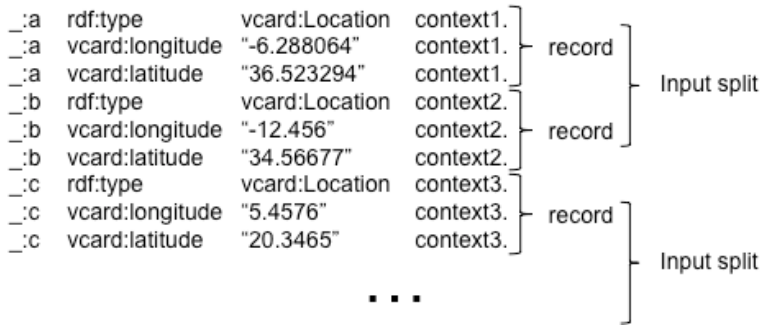

Figure 11: Creating records with the same context node

The records in the input splits now contain reasonable sized chunks of RDF to execute our SPARQL queries in the map tasks. The extracted values for the individual entities are passed to the reduce tasks together with the original homepage URL. The reducers perform a clean-up (e.g., escaping existing commas in extracted literal) and format the values into lines of CSV-tables. Finally the generated tables are committed to the HDFS. Figure 12 illustrates the complete MapReduce process. We were able to generate the CSV-tables on average within 1 hour for every Microformat with our approach.

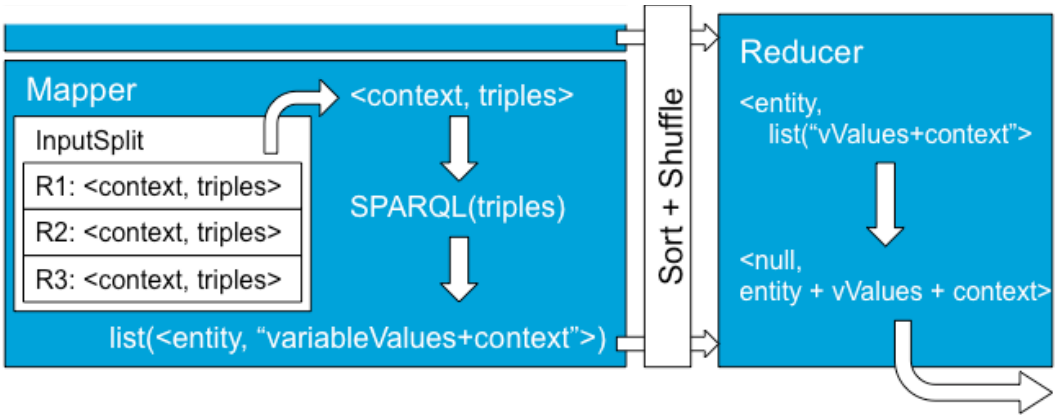

Figure 12: MapReduce job to generate CSV-tables from RDF-quads

3.7 Summary

We extracted structured data from two of the largest publicly available web corpora to this date and analyzed the extracted data. This analysis of the two Common Crawl corpora has shown that the percentage of web pages that contain structured data has increased from 6 % in 2010 to 12 % in 2012. The analysis showed an increasing uptake of RDFa and Microdata, while the Microformat deployment stood more or less constant. Furthermore, we provide detailed statistics about the popularity of types in the different formats.

We have shown how such an extraction over a very large dataset is possible within a reasonable amount of time, using affordable resources on the Amazon EC2 Cloud.

The extracted data itself can be downloaded as RDF-quads or CSV-tables. The latter implied a transformation of the extracted data, which we performed using Hadoop.

4 SciLens – A Local-Area Research Cluster for Petabyte Scale Scientific Data Management Techniques

In this section, we report on our experiences with the design and utilisation of ***SciLens***, a local-area cluster machine that has been built to facilitate research on data management issues for data-intensive scientific applications. *The main contribution of the SciLens cluster is that it provides a platform on which we can carry out experiments to help develop, examine, compare and improve new DBMS techniques for the scientific applications. SciLens enables various possibilities to run experiments on large clusters using real-world large datasets, obtained directly from different scientific disciplines.*

This section is further organised as follows. First, we give background information about the research on DBMS support for data-intensive sciences. We will briefly describe the data management problems in the data-intensive sciences, the mismatch between what the scientists need and what the current DBMSs provide, our plans to advance the state-of-the-art of the scientific data management and analysis, and the need of a SciLens cluster to help realising the plans. Then, we discuss the design issues of SciLens. Thirdly, we describe the final architecture of SciLens. Fourthly, we discuss two main lessons we have learnt during the building of SciLens. Fifthly, we describe the tools we have developed to deploy the relational DBMS MonetDB[12] on SciLens. Finally, we conclude with an outlook to future work.

4.1 Background

4.1.1 The Fourth Paradigm in evolution

To truly understand the complexities of global warming, gene diversity and the wonders of the universe, the sciences more and more rely on database technology to digest and manipulate large amounts of raw facts. It requires support for the daily aggregation of Terabytes of science data, a synergy with large-scale existing science repositories (e.g., based on FITS[13], NetCDF[14]), efficient support for the preferred computational paradigm (e.g., array processing environment like MATLAB), and linked-in libraries (e.g., the statistical package R). The ability to explore huge digital artifacts assembled in data warehouses, databases and files at unprecedented speed becomes the pivot to keep abreast in science. Significant advances in data warehouse technology are needed to aid scientists in this quest for quantitative insight in and model driven validation of phenomena coming from, e.g., astronomical observations, life-science experiments, neuroinformatics and environmental sensor-based monitoring systems (*Cyberinfrastructure Vesion*, 2007). The creators of such technology are the catalyst for future Nobel Prize winners.

Jim Gray (1944-2007) chartered this road ahead as *the Fourth Paradigm* (Hey et al., 2009), i.e., *scientific discoveries are increasingly enabled by scalable database management technology*. It is the next step in evolution which started with the first paradigm based on thousands of years of *experimentation*, the second paradigm of the last 500 years based on *theory* formulation, and the third paradigm of the last 50 years based on *simulation* through computational science. The fourth paradigm closes the loop by providing the setting to comparatively assess results obtained through observation and produced by simulations against those predicted by models.

This data explosion is unfolding due to better experimental instruments and large-scale sensor networks. They form the prime basis to advance science in the years ahead. Scientific databases are far larger, and growing faster, than their commercial counterparts (Becla and Lim, 2008). The number of table attributes can run into thousands of tables degenerate to key-value stores, such as for protein databases, and the number of rows in a table can run into the billions, such as for astronomical catalogs.

This makes it hard for scientists to grind its content for in-sights using their daily tools, e.g., spreadsheets and statistical packages. As pointed out in the recent study on patterns of information use in life sciences (*Patterns of information*, 2009), scientists "share concerns about receiving, generating, handling and managing huge volumes of information and data, in particular genome sequence data, as well as accessing,

[12] See: http://www.monetdb.org/

[13] See: http://heasarc.nasa.gov/docs/heasarc/fits.html

[14] See: http://www.unidata.ucar.edu/software/netcdf/

organising and analysing vast quantities of varied data, and then making sense of them... Researchers fear that there will be too much data to handle, process or even look at".

4.1.2 Where do DBMSs fall short

Current database management systems, developed mostly for administrative tasks, lack components required to reach the goals set for the Fourth Paradigm wave. Major innovations in this domain partly overhauling the DBMS architecture, is widely recognized as a grand challenge in the research community (Stonebraker et al., 2009; Becla and Lim, 2009; Gray et al., 2005; Gray and Szalay, 2004).The heart of a scientific data warehouse is its database system, running on a modern distributed platform, and used for both direct interaction with data gathered from experimental devices and management of the derived knowledge using workflow software. However, most (commercial) DBMS offerings fall short for scientific data management in one or more of the following areas:

- **Multi-paradigm data models.** Database management systems are mostly focused on the relational data model and its algebraic computation, but in science the *array*-programming paradigm is prevalent. This lack of computational expressiveness reduces database systems to managing *blobs* of non-interpreted data, which are handed over to a linked-in library for further processing. However, the mixture of both paradigms, which allows switching between array- and tabular- views at low cost, provides opportunities currently only found hard coded in specialized user-programs. For example, image segmentation can be partly based on the relational selection mechanism, while subsequent smoothing may rely on the array view of the same data for speed.
- **Event stream processing.** Sensor-based devices and networks are evolving at great speed. All nodes are connected into a worldwide-distributed application to share events. They call for a direct feed into the science warehouse, followed by complex-event-processing techniques to react instantly to unexpected events. The even more ambitious web-of-things would create a potential flooding of streaming data that can only be handled by application logic close to the source. Hints on what to expect can be found in, e.g., the radio astronomy setting of LOFAR[15] with multi-gigabits per second event streams.
- **Sequence matching.** Sequence information from biology is encoded in large strings. Nowadays, a data warehouse of gene sequences is achievable, even for archiving sequence data from a large number of individuals simultaneously. But pattern directed search over such large repositories is hardly supported. Techniques developed for this field have not yet found their way into generic offerings of database suppliers.
- **Provenance.** Scientific database are mostly append only. Data comes from observations, but more importantly, a lot of metadata are added based on the analysis of the observations. To ensure correct future interpretation, the precise sequence of operations, from observation, cleaning to knowledge derivations, should be maintained. Such provenance issues are hardly supported in database systems. Instead, workflow systems currently fill this gap.
- **Science library integration.** The operations provided by a relational DBMS are geared mainly towards administrative applications. The use of a relational DBMS for scientific applications stresses its functionality to link-in existing science libraries as User Defined Functions (UDFs). However, the impedance mismatch between data structures and operational behaviour often lead to the non-acceptable approach to simply extract data from a database, run a program against it, and update the database with the results.
- **Multi-scale query processing.** The size of scientific databases renders traditional query processing flawed. Expecting real-time response for ad-hoc queries over multi-terabytes science databases will simply fail unless extreme hardware investments are made. A multi-scale, statistically sound organisation of the database, which allows gradual expansion of the area of interest to explore for query answering, is a necessary route to explore. A weighted sampling as proposed in (Sidirourgos et al., 2011) is a promising step into this direction.
- **Summarization.** The scheme deployed in a database management system to simply produce a result set fails due to its size and difficulties of the ad-hoc user when he attempts to precisely formulate the

[15] See: http://www.lofar.org/

queries. Scientists do not exactly know up front what they are looking for. The ultimate query is a result of an interactive process, driven by partial answers. The process of harvesting the knowledge through ad hoc querying will be greatly improved by on-the-fly summarization of multi-gigabytes. However, it is an area largely unexplored in the context of database research.

- **Scale-up and scale-out.** Traditional distributed and parallel database techniques require a predictable workload. This is rarely the case in science. Attempts to scale out into Cloud computing using, e.g., Map-Reduce techniques, gives a short-term answer for massive data-parallel problems. It fails, however, to address the inherent complexity of distributed processing, where each sub-query contains cross correlation to other elements in the database (which need not reside on the same node).
- **Green computation.** The predominant approach in business intelligence applications is to throw hardware at the database challenges to reach satisfactory response time. Hundreds of disks distributed over tens of nodes to realize just a few GB/sec bandwidths are common. Despite continuously dropping hardware prices, such solutions remain expensive and energy-consumptive. Moreover, scientists are too budget constrained to permit them to follow this brute force approach. Much can be gained if we can harness the database management challenges into energy saving solutions. The ideal would be a 10 TB data warehouse in cool and silent "shoeboxes" on the desk with enough processing capacity for science applications at an affordable price level of say $500.

4.1.3 Who will rescue us from a data overload?

Scientific data management in astronomy during the 80s and 90s has shown that reliance on the third-generation programming expertise of domain specialists is bound to fail. It does not provide for a scalable solution, and distracts the scientists from going after the information contained in their experimental data. Wearied of following hype topics of limited impact, technology inventors, such as Jim Gray and Alex Szalay, have been called upon to cross the chasm between disciplines. They proofed the productivity gains in science from using a relational DBMS and its query language SQL.

The third paradigm focused on large-scale simulation. It triggered a software infrastructure to scale-out computational jobs over computer clusters and later over a dynamically changing network, nicknamed the Cloud. Middleware and grid computing support packages were developed. However, the focus was on the computational aspects, leaving the data management to a relational system and/or the underlying file stores. The recent revival of data-parallel programming, i.e., Map-Reduce, addresses only part of the issues at stake.

Although middleware tools form a large area on their own, scientific workflow packages rely strongly on DBMS features, document information retrieval and semantic web technology, e.g., high volume RDF processing. This often leads to under utilization of DBMSs in science projects. The DMBSs degrade to an administration tool for a large passive repository of data files. To illustrate, multi-tier data infrastructure of the LHC (Large Hadron Collider) consists for >95% of HDF5[16] (Hierarchical Data Format 5) files with database management primarily devoted to workflow management. The scale and flexibility required for data intensive research extend beyond this brick-and-mortar solution for most real-world information systems. For example, in astronomy, planned instruments will produce >100 petabyte by the end of this decade producing catalogs with 3000 million galaxy surveys[17]. Likewise the amount of data produced by environmental sensory systems is expected to explode. In seismology, the amount of event data collected in 1998 was a trickle 2 MB, while the planned combined sensor network will produce > 10TB/year as of 2011. Data ingress speed is actually doubling every year.

Current database management systems lack components required to reach the goals set for the Fourth Paradigm. *The heart of a scientific data warehouse is its database system, running on a modern distributed platform, and used for both direct interaction with data gathered from experimental devices and management of the derived knowledge using workflow software.* However, most (commercial) DBMS offerings fall short for scientific data management in one or more of the aforementioned areas. To tackle these problems, we are addressing three pivotal shortcomings of database system architectures for science, using MonetDB as a frame of reference:

[16] See: http://www.hdfgroup.org/HDF5/

[17] See for instance the LSST(Large Synoptic Survey Telescope) project: http://www.lsst.org/lsst/

- **Data Vaults**: to research the architectural consequences for blending database technology over existing science file repositories and evolving scientific instruments, which would make data available through a DBMS beyond the hitherto limitation of meta-data in relational stores.
- **SciQL**: Having thus opened the treasuries of experimental data for further analysis, we address the limitations of the query language SQL. We are developing a natural extension to SQL, called *SciQL*, which provides a symbiosis between the relational- and array- paradigms. We are developing solutions to cope with both (sparse) arrays and data cubes running into the hundreds of TB, but also to blend this with relational query processing and optimization schemes. A true multi-paradigm symbiosis will emerge.
- **Data Cyclotron**: scale-up and scale-out is pursued through an elastic distributed processing system, called the *Data Cyclotron*, which addressed in particular the requirements of knowledge discovery algorithms using an elastic (virtual) ring of processing nodes to boost query processing.

4.1.4 Why do we need yet another cluster?

The fourth paradigm time window calls for leading edge computer science and engineering research activity to avoid that next generation scientific instruments produce data we are not able to digest. ***The actions called for should go beyond theory and (micro-) benchmarks to assess more-and-more complex data structures and algorithms.*** Instead, Jim Gray's laws for data engineering should lead the way (Gray et al., 2005). They stimulate the need for reliance on realistic data, a focus on scale-out of the solutions, and moving analysis towards the data itself. All this should be done within ***a methodology that combines early articulation of the 20 queries to describe the challenge and an approach based on evolving full-functional exemplars.***

The proof of the pudding is in the eating, which also amounts to high-risk science and engineering. In this light we continue the tradition of the CWI database research group to develop solutions up to the point they can be shared with and picked up through an open-source solution. All software will become part of the MonetDB suite. This calls for a hardware infrastructure *to validate our algorithms and share experiences through access to real-life application demonstrators*. The SciLens cluster has been built to fulfil this request.

HPC v.s. DIC. Many large clusters already exist, e.g., the SARA Lisa cluster[18] and the Amazon EC2 cloud[19], on which experiments of distributed and parallel computing can be conducted. However, the existing clusters are not suitable for the database experiments we want to carry out for several reasons.

First, the existing clusters generally focus on supporting High-Performance Computing (HPC), for which extremely fast floating point computations is the primary issue; while in data-intensive database research, dealing with a large amount of data is one of the primary issues, so we need fast data access and high I/O bandwidth. For instance, in the SARA Lisa cluster, there is only one storage node. So for this cluster, a dataset of 4TB is already too large, also transferring the whole dataset from the storage node to the computation nodes for each experiment is high time consuming.

Second, the existing (mostly HPC-oriented) clusters are mainly used for productions. They are not intended for users who want to explore new possibilities, but are intended to facilitate mature applications, e.g., calculations for chemists and weather forecast. This means that the existing clusters mainly focus on providing a stable environment, while for our data-intensive database research, we want to be able to experiment with different hardware configurations (e.g., different network configurations) and different versions of the software including the unstable versions. These needs cannot be satisfied by the existing Cloud or cluster systems.

Finally, in database research, we often need to examine in details the behaviour of both hardware and software, so at a certain point, we need to access the kernel information of the testing systems to find out, e.g., what is actually going on and how the hardware is being use. Accessing the low-level system information is usually not allowed for ordinary users on the aforementioned clusters.

[18] See: https://www.sara.nl/systems/lisa/

[19] See: http://aws.amazon.com/ec2/

4.2 Design Considerations

Database systems are very large software systems, which bring together a plethora of algorithmic knowledge, optimization techniques, and plan architecture considerations. Scientific database research call for a real-life sized system to evaluate technology and extract user experiences. The design of the SciLens infrastructure should maximise external exposure and obtain deep results on scalability in heterogeneous clusters. We use the Amdahl Blade principle as the basic criterion for the design of SciLens.

Amdahl Blade Computing. Computer systems aimed at driving data intensive research differentiate from computational science, e.g., super-computers, by a much more prominent balance of the system components in the storage stack. Queries over large-scale database are more susceptible to IO bandwidth and latency than the number of CPU cycles thrown at it when data is available in the core. Even with careful algorithmic design, most database systems tend to stall on access to memory or disk.

Amdahl formulated several rules for a balanced system[20]:

- The speedup of a distributed system should be proportional to its longest sequential code paths.
- The bandwidth should reach one bit of IO/sec per instruction/sec (BW).
- The amount of main memory should reach one byte per instruction/sec (MEM).
- Each IO operation is equivalent to 50.000s of instructions.

In database queries, many blocking operations, e.g., sorting and relational joins, provide a strong upper bound on the attainable speedup. Indexing and proximity based algorithms in database systems are inherently random access operations, which affect the effective bandwidth usage. Modern multi-core computers move further away from Amdahl's laws. Large super computers have low numbers on bandwidth and memory. For the IBM Blue Gene the bandwidth BW= 0.013 bit of IO/sec per instruction/sec and the memory factor MEM=0.471 byte per instruction/sec. A state-of-the-art experimental system at Johns Hopkins University reaches BW=0.66 and MEM=1.0. Pioneering work of Szalay (Szalay et al., 2002) underpins the need for different system configurations to deal with the requirements posed. It identifies two recent technological innovations as critical: Solid State Disks (SSD) combine high I/O rates with low power and energy efficient CPUs, e.g., Intel's Atom family of processors designed for the mobile market. They showed that it is possible to use these to build balanced, so called *Amdahl blades* offering very high performance per Watt while maintaining a low total-cost-of-ownership.

4.3 Architecture of the SciLens Cluster

4.3.1 The dream

Figure 13 shows the original architecture of SciLens. The infrastructure is based on the Amdahl blade philosophy and uses common-of-the-shelf components. The initial system configuration is a 4-tier cluster: a computational top tier with 1 node called *Diamond*, a high-end tier with 16 nodes called Rocks, a Cloud-oriented tier with 64 nodes called *Bricks*, and an energy-conservative tier with 256 nodes called *Pebbles*. Each tier consists of a homogeneous platform. All tiers come with the same global hardware resources of 1 TB RAM and 128 TB disk space. The tiers are assembled in both an InfiniBand network and a Gb (Gigabit) Ethernet to enable topology reconfiguration and slicing over all tiers. *The infrastructure is specifically tuned towards data intensive work, i.e., it is focused on massive IO instead of raw compute power.*

The key rationale for a multi-layer structure is to create a setting where comparative real-life experiments are feasible. The applications available and foreseen call for multi-terabyte storage settings. In turn, this calls for sizable tiers to explore the limits of distribution. For example, a 4 TB science database like Skyserver[21] can be handled by a single *Rock.* The effects of migration down the hierarchy may well uncover the rules to stop at the *Bricks* layer. Or trigger research into new database optimization and algorithms. The nodes in each tier are affordable systems, without exotic hardware components.

[20] See: http://en.wikipedia.org/wiki/Amdahl%27s_law

[21] See: http://www.sdss.org/

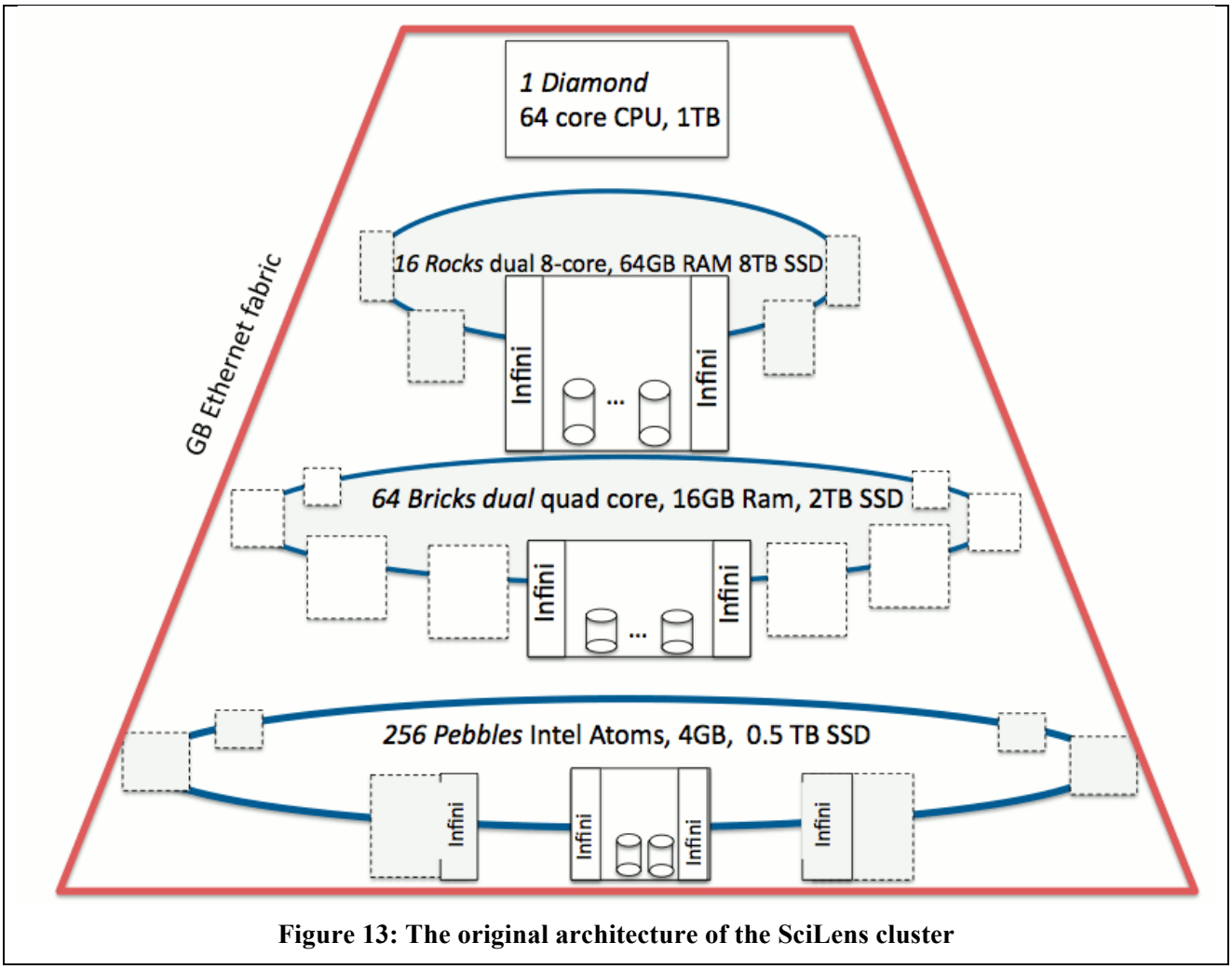

Figure 13: The original architecture of the SciLens cluster

Using homogeneous tiers with each 1TB of RAM and 128TB of disk is needed to support scalability experiments. In the cause of the lifetime of this system, we expect experiments to call for > 50TB of data. The current infrastructure already faces serious data administration challenges due to an 'overloaded' 20TB disk farm. Furthermore, it should be homogeneous, because it is a known fact in database areas, that even a minor change in hardware configuration or the operating system version can have a disturbing effect on the performance characteristics. For example, going from Linux Fedora 10 to 11 caused MonetDB performance to drop significantly (>20%), and it is only partly recovered in Fedora 12. This is without changing the DBMS software stack itself.

Each tier is initially physically organised as a ring with InfiniBand. The ratio is to provide a system infrastructure were a large distributed ring buffer of 1TB of memory becomes available for data intensive applications. For read dominant applications, this gives an easy to program backbone to share data of common interest and still can realise a high throughput of the application code.

The multi-tier structure provides valuable insight in algorithms, parallelism when the code moves up or down the tier. However, with a slight change in topology, slices over all tiers can be cut out to create a multilevel caching structure from different components. For example, a *Rock* node could be supported by a series of *Pebbles*. With proper query planning, this creates a setting where large intermediate results are handled without transport cost in the *Rock,* while the massive scan operations on the bulk data can be subcontracted to a large number of the slower *Pebbles*.

The *Pebble* layer is configured as an outward look into energy conservative computing. It has no spinning parts and the CPU power is much less than one ordinary expects from a desktop. The large number of nodes, hooked up into two different network infrastructures (i.e., InfiniBand and Gb Ethernet) enables in depth studies on the algorithmic consequences. In scientific database applications with less intermediate data productions, e.g., life sciences, it may well be that the Gb Ethernet is sufficient. Contrarily, for data mining applications the low latency in the InfiniBand ring is preferred.

The projected top node *Diamond* is scheduled as the bridge towards affordable supercomputing on the desktop. It should ideally be a single node machine with 1TB of DRAM where the CPU characteristics are either the proper number of cores or a mixture with high-end graphics processor cards (GPUs). The hardware trend would not call for this investment before 2013.

Although each node in the system is an Amdahl blade, the combined effect on the total system performance is an open research question. *This envisioned architecture provides a controlled laboratory setting to deepen our understanding of the algorithmic trade-offs of local, remote, or in flux (streaming) processing on large amounts of data.* The multi-tier layers provide the setting for quantitative evaluation against, e.g., level of parallelism, system complexity and homogenous/heterogeneous virtual machine platforms. Each tier is large enough to support at least the astronomy SkyServer[22] and seismology ORFEUS[23] applications on a continual basis. By switching between the tiers, their relative performance and cost structure can be determined. Carving out slices over all tiers provide a hierarchical data cache. The InfiniBand fabrique permits variations in topology and exploration of different algorithms for distributed shared memory. The combined storage can be used as a staging area for specific experiments, e.g., TPC-H[24] and SS-DB[25] benchmarks, or other emerging user applications, e.g., Pan-STARRS[26], LOFAR[27] and the initial SKA[28] test cases.

The SciLens infrastructure is expected to become a landmark in data intensive research through its comparative application portfolio at realistic scales. The SciLens infrastructure is configured to facilitate real life application deployment. Studying them at full scale provides valuable insights in the requirements on database technology. The direct access to change the system configuration (e.g., network topology) and control over the complete DBMS software stack (2M lines of code) enables validation experiments hitherto not achievable.

[22] See: http://www.sdss.org/

[23] See: http://www.orfeus-eu.org/

[24] See: http://www.tpc.org/tpch/

[25] Available at: http://www-conf.slac.stanford.edu/xldb10/docs/ssdb_benchmark.pdf

[26] See: http://pan-starrs.ifa.hawaii.edu/public/

[27] See: http://www.lofar.org/

[28] See: http://www.skatelescope.org/

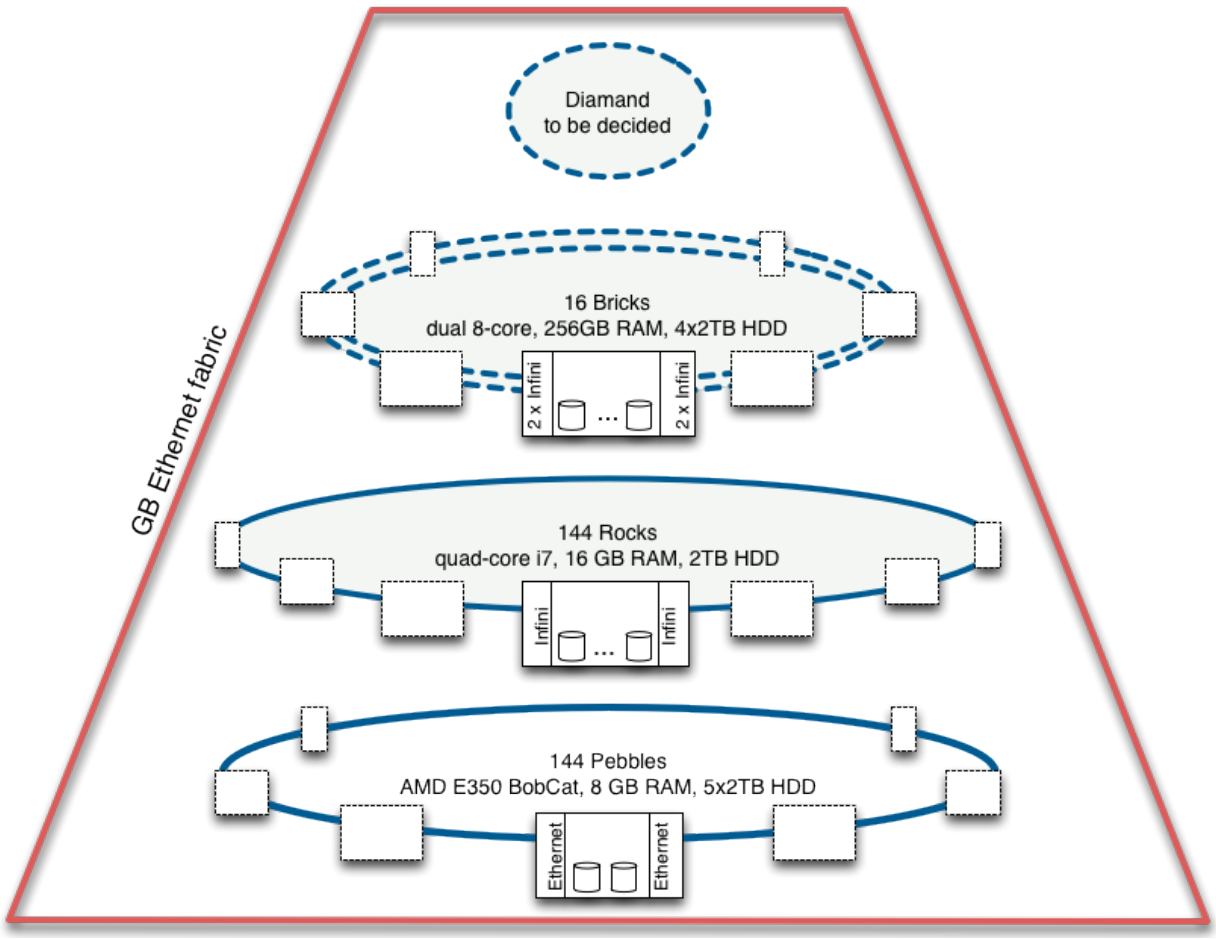

Figure 14: The actual architecture of the SciLens cluster

4.3.2 From dream to reality

Since 2011, we have been conducting the first round of building and installing machines for the two lower most layers of the SciLens cluster. After having done some tests (described in Section 4.4.1), we have decided to make some changes to the selected hardware in the original SciLens architecture to achieve the best trade-off between maximising the amount of hardware we can buy and the performance and price of the hardware. The actual architecture of SciLens is depicted in Figure 14: The actual architecture of the SciLens clusterThe configuration of the two lower most layers is as follows.

The Pebbels. The bottom layer is called the *Pebbles*, which consists of 144 low-end systems with a low-performance processor but a lot of hard discs (HDDs). The *Pebbles* are each equipped with an AMD E350 BobCat processor and 8GB memory. Each Pebble has five 2TB HDDs. One disc is used for the system and some scratch space. The remaining four discs are configured as a 4-way RAID 0 disc[29] to achieve a bandwidth of almost 400GB/sec. The Pebbles are intended to store large datasets that are rarely used. For reasons of prices, the Pebbles are connected with a 1Gb (gigabit) Ethernet, instead of an InfiniBand network.

The Rocks. The layer above the Pebbles is called the *Rocks*[30], which consists of 144 moderate systems that are comparable with contemporary desktop machines. Each Rock has an Intel® Core™ i7-2600K Quad-Core processor with Hyper-Threading and 16GB memory. All Rocks are connected through both a 1Gb Ethernet

[29] See: http://en.wikipedia.org/wiki/RAID

[30] In the final architecture of SciLens, the names of the two middle layers are switched due to some human factors. So, in the end, the layer above Pebbles is called Rocks and the layer above the Rocks is called Bricks.

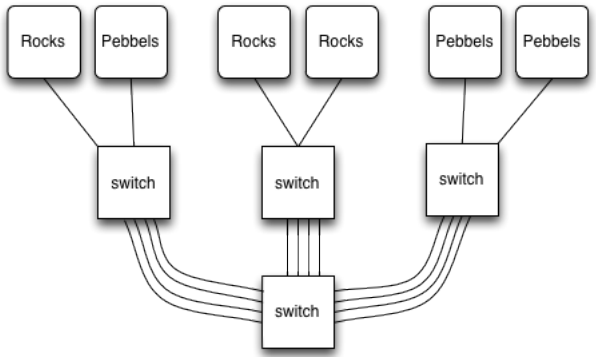

Figure 15: Schema of the SciLens Ethernet network (not all nodes and switches are shown). Nodes are connected to switches, and switches are inter-connected with 4x1Gb Ethernet.

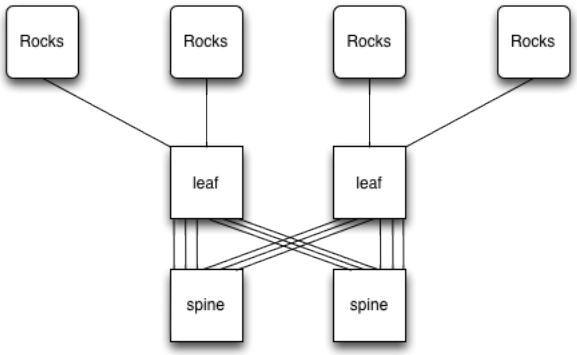

Figure 16: The setup of the InfiniBand network in SciLens. Each node is connected to one leaf switch, which in turn, is connected to all four spine switches with three cables.

network and a 40Gb Quad Data Rate (QDR) InfiniBand network. At the moment, each Rock has only one 2TB hard disc, but there is room for two SSDs if needed.

The Ethernet network. The Ethernet network of SciLens has a standard setup, which schema is depicted in Figure 15. Currently, it has 10 switches each with 48 ports. Each host is connected to one switch using one port. The network uses a star topology, i.e., there is one switch that connects all other 9 switches using 4 cables of 1Gb connection. One of the 9 switches is only used for administrative purpose, which leaves 8 switches to be used for connecting the hosts. The switches are fully managed switches, which means that we have the full control of the configuration of the network. This allows us to try out different network settings, such as Virtual LANs (VLANs)[31] with varying Quality of Service (QoS) and all kinds of RFCs (i.e., different internet protocols). The connections between the switches can be the bottleneck when large amount of data need to be transfers through the switches. However, for reasons of price, we decided to not connect the switches using 10Gb links yet. This can be considered during the second round.

The Ethernet is wired in such a way that from the 8 switches that have hosts connected, 2 of them are filled with Pebbles and 2 of them are filled with Rocks. The other 4 switches are filled half with Rocks and half with Pebbles. This way, if one wants to do experiments with larger clusters, one can use 36 machines connected to a single switch, which avoid possible bottlenecks between the switches. If one wants to do experiments on different machines, i.e., some pebbles and some rocks, one can also avoid having to transfer a large amount of data between switches.

The InfiniBand network. The Rocks are also connected with an InfiniBand network, because next to the standard Ethernet network, we also want to have a network that provides high bandwidth between nodes and switches. This way, we can have a large cluster in which we can move around large datasets. The InfiniBand network has 12 Mellanox switches[32], 8 of them are *leaf switches* and the remaining 4 are *spine switches*[33]. The switches are so-called "half-blocking", which means that they only have half of the potential bandwidth between the switches, because to get the maximum amount bandwidth, we need four more switches. The schema of the InfiniBand network is depicted in Figure 16. Each Rock is connected to one leaf switch using a QDR HBA (Host Bus Adapter) port of 40Gb/sec, which in turn, is connected to all four spine switches with three cables. The leaf switches are not interconnected to each other, neither are the spine switches. In the InfiniBand network, the swiches are connected through such a high-speed network that they will not create potential bottlenecks for data transferring. Thus, in the InfiniBand network, transferring data from any machine to any other machine in the network goes extremely fast.

[31] See: http://en.wikipedia.org/wiki/Virtual_LAN

[32] See: http://www.mellanox.com/

[33] The names “leaf” and “spine” are standard terms in InfiniBand networks, which indicate the role of the switches. Leaf switches are switches to which the hosts are connected, while spine switches are only used to connect the leaf switches.

The Ganglia monitoring system. To monitor the health and activities of the SciLens cluster, we use Ganglia[34]. Ganglia is an open-source scalable distributed monitoring system for high-performance computing systems such as clusters and Grids. It is based on a hierarchical design targeted at federations of clusters. Ganglia supports clusters up to 2000 nodes in size. Figure 17 shows a "physical view" of the SciLens cluster, which lists all machines in the cluster with a brief description of the hardware resources on each machine. Figure 18 shows a "full view" of the SciLens cluster, which displays various statistics about the health of the machines and the activities. Figure 19 shows the "node view" of a single node *pebble011.scilens* in the cluster, with extended information about the hardware and software resources on this node, and some brief statistics of the past and current usage of the node. Figure 20 shows the "host view" of the same node, which displays more elaborated statistic information.

[34] See: http://ganglia.sourceforge.net/

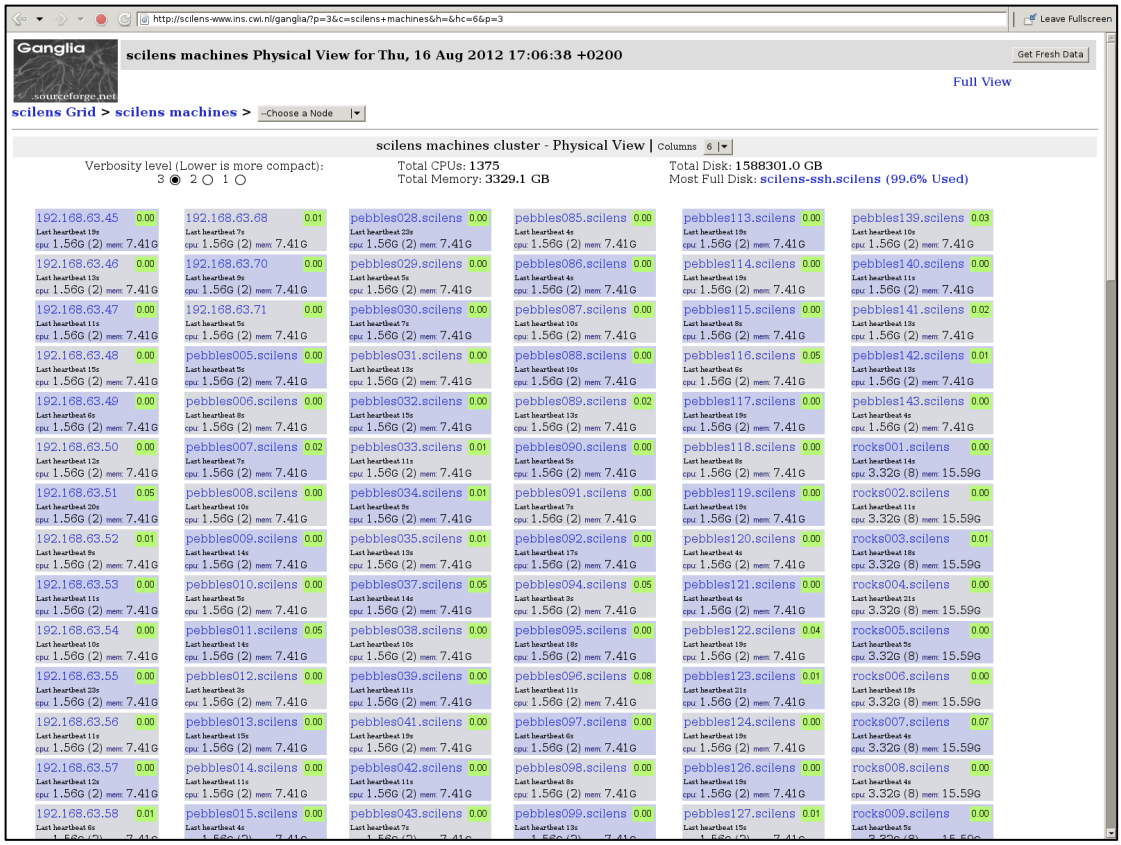

Figure 17: The physical view of the SciLens cluster: a list of all machines in the cluster with a brief description of the hardware resources on each machine.

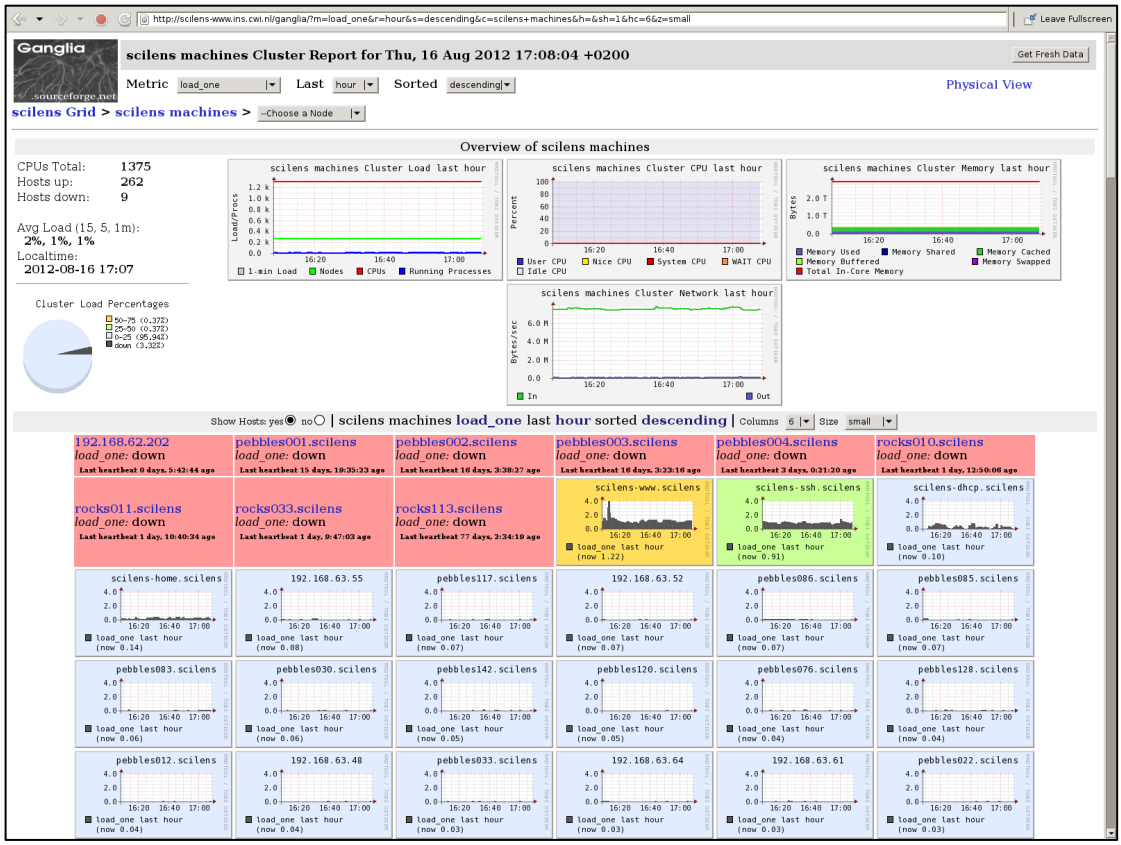

Figure 18: The full view of the SciLens cluster: various statistics about the health of the machines and the past and current activities.

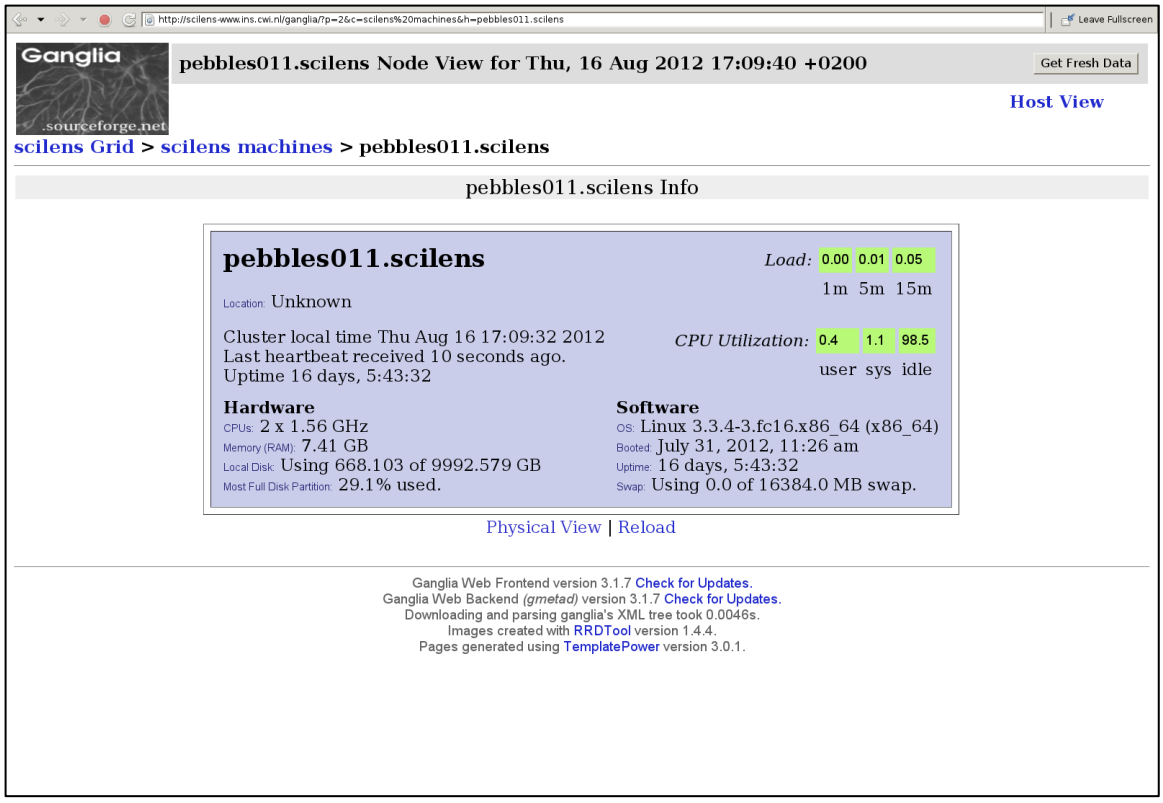

Figure 19: The node view of a single node *pebble011.scilens* in the cluster: extended information about the hardware and software resources on this node, and some brief statistics of the past and current usage of the node.

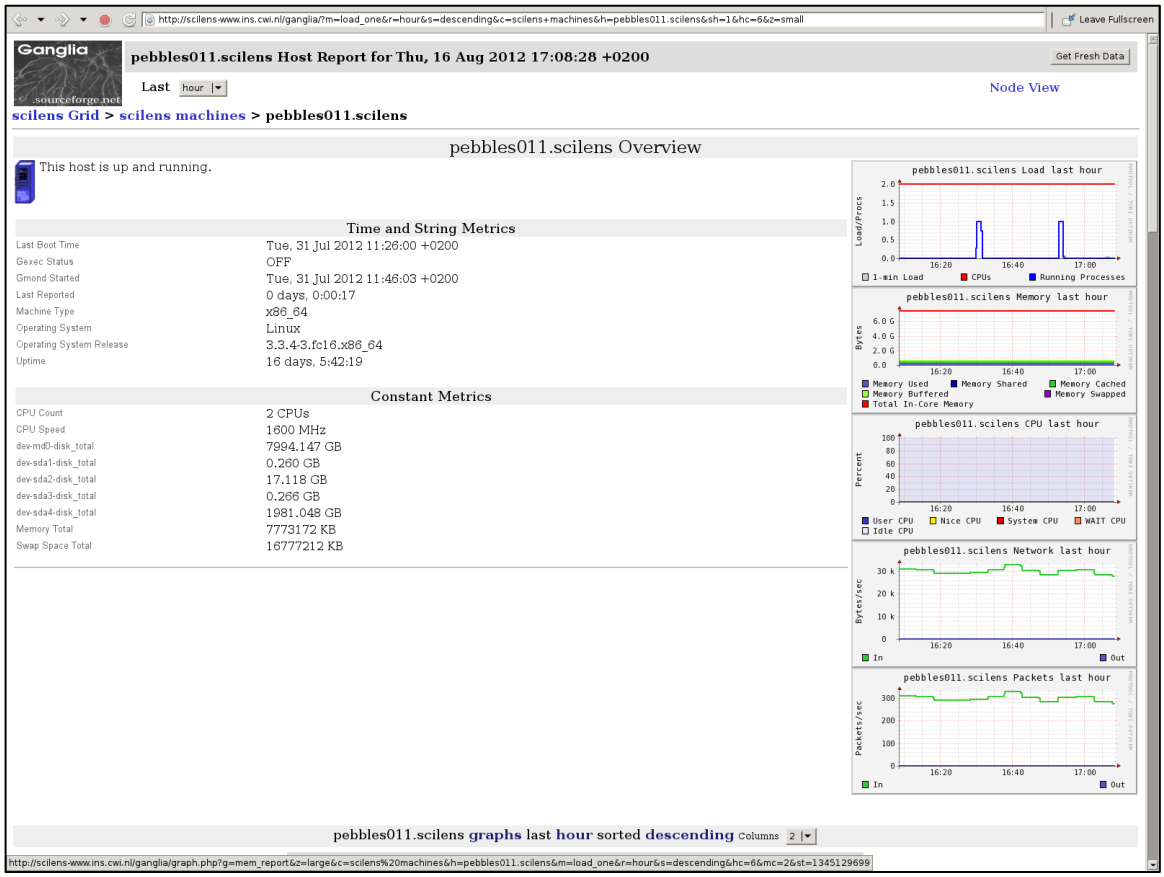

Figure 20: The host view of a single node *pebble011.scilens* in the cluster: displays elaborated statistic information.

4.4 Lessons Learnt

From the building of the first round of SciLens, we have learnt two main lessons. The one is how to select the hardware to achieve the best trade-off between price and performance. The other one is how to manage a cluster that is intended for experimental use. Below we elaborate both lessons.

AMD E350 BobCat processors are used in the Pebbles, which is highly comparable with the original Intel Atom processors. The advantage of the AMD BobCat processors is that they can work with 8GB memory.

4.4.1 How to select the best suitable hardware

Before placing the final order for the SciLens machines, we have carried out extensive tests to assess if the initial architecture of SciLens is indeed optimal, in terms of *the price and performance of the selected hardware* and *their power usage*. So, in the spring of 2011 we assembled 12 test machines with highly different configurations, e.g., different processors, different HDDs, both HDDs and SSDs, different RAID cards, to evaluate the original design of the two bottom tiers. We even tried different power supplies and casings, to see how we can put a maximal amount of hardware in a minimal amount of casings, so that we can minimise the necessary server room space for the cluster machines.

The test. The test machines were assembled from Commercially available Off-The-Shelf (COTS) elements and included Intel® Atom™ D425@1.8Ghz, Intel® Atom™ 330@1.6 GHz, Intel® Core™ i3-540@3.07Ghz, Intel® Pentium® Processor E5700@3.00GHz, Intel® Core™ i3-M350@2.27Ghz, Intel® Core™ i7-860@2.8 GHz, Intel® Core™ i7-970@ 2.8Ghz, Intel® Core™ i7-2600, AMD Athlon™ II X2 250, and AMD BobCat. It covered a complete range of affordable processors with 1 to 6 cores. For hard drives we experimented with RAID-0 configurations from 1-4 160 GB, 500 GB and 2TB SATA disks and 128 GB, 250 GB SSDs. All systems were mounted on state-of-the art motherboards.

Given the large number of parameters, a heuristic search was initiated to find the 'best' solution for the bottom layers. The search components were based on measuring pure disk IO bandwidth, network capabilities, compilation of the MonetDB system and two database warehouse benchmarks, i.e., TPC-H[35] and the air-traffic benchmark[36]. The latter are considered representative for the work we intend to perform on the machine. In addition, we measured electricity consumption of the complete system in idle mode, boot and high-stress situations.

The conclusions. The selection process uncovered the weakness of the presumed energy conservative Intel Atoms. Using these systems for database processing turned out to both poor in performance and far more energy draining. For example, the complete workload on an Intel® Atom™ took around 5.2 hours compared to on the Intel® Core™ i7-970 processor in 1.2 hours. Although the peak energy drain was 40 against 132 Watt, the total energy consumed by the Intel® Core™ i7-970 was better as the machine could be turned off after an hour. This phenomena has been observed in most cloud settings as well; migration of work and turning off machines is by far the best energy saver. Furthermore, the prices of the processors do not scale linearly with their performance. So, for instance, the Intel® Core™ i7 processors are much faster than the i3 processors, but the i7 processors only cost about €50 more. Therefore, we choose the i7 processors for the Rocks.

Both the Intel® Atom™ processors and AMD BobCat processors are slow but energy saving processors. The Intel® Atom™ processors are more popular, but the AMD BobCat processors are highly comparable with them in techniques, price and presentation. We finally choose AMD BobCat over Intel® Atom™, because AMD BobCat is able to work with 8GB memory, while Intel® Atom™ only works with up to 4GB memory.

The disk specific benchmark[37] confirmed the well-known dependency on read/write mix and the block sizes. However, in assembling the systems the motherboard chip sets had sufficient impact on the results measured.

[35] See: http://www.tpc.org/tpch/

[36] A description is available at:

http://www.mysqlperformanceblog.com/2009/10/02/analyzing-air-traffic-performance-with-infobright-and-monetdb/

[37] Just some standard I/O tests that simply read/write GBs of consecutively stored data from/to the disk.

In a 4-way RAID 0 setting, the SATA HDDs topped at 380 MB/sec and the peak for the SSD was 2 GB/sec using an expensive JBOD card[38]. However, although a 4-way RAID 0 with SSDs is five times faster than the SATA disks in pure I/O operations, the benefit of faster SSDs was not fully reflected in the database workload performance. In the TPC-H benchmark results, using SSDs only leads to about a factor 2 of performance improvement compared with a similar machine configured with SATA HDDs.

4.4.2 How to manage an experimental cluster

As pointed out before (Section 4.1.4), the intended usage of the SciLens cluster is rather different than the existing HPC-oriented (Cloud) clusters. Because SciLens will be used by a relatively small number of users to try out highly experimental software, we choose a minimalistic management strategy to keep the cluster maintainable and dynamic (i.e., can be easily updated to different versions of software), while allowing any specific user requests to be discussable.

A minimal set of software. To avoid duplicate maintenance work, and also because other Linux distributions are not better and other (commercial) OS is not an option, the same OS as used on the CWI desktop machines, i.e., Fedora, is used on the SciLens machines. However, experiments with alternatives OS are possible by request. Moreover, we only install a minimal amount of software packages on the SciLens machiens (~800 compared with ~3800 on a normal desktop) that are required to run experiments and only provide command line interface. This is because SciLens should only be used for experiments, it should not be used for daily work, e.g., software development or writing documents. There is (almost) no connections to the outside world. Transferring data to the SciLens machines needs to be done through the CWI desktops. The idea behind this is that usually data only need to be copied to the SciLens machine once, and used for all experiments one wants to do. Finally, no backups of the SciLens machines are made, since backing up PBs of data is not feasible. Thus, owners of important datasets have to take care of the backups themselves.

No scheduler. Currently, we do not use any scheduler to control the usage of the machines by the users. This is because schedulers are generally difficult to use, so they should be avoided whenever possible; also if there is enough resources, there is no need to limit the usage. Without any monitoring, the users are expected to check the availability of a node before using it; also if a large number of nodes are needed, the user should inform the user group to see if there are any objections. Thus, if it turns out to be necessary later on, we might consider using a scheduler for, e.g., long running tests requiring many machines.

No guarantees. Because we want to do all kinds of new experiments on the SciLens cluster, so, unlike the normal desktop systems that are expected to be stable and always working, experimental software will be installed and used on the SciLens machines. Because the behaviours of those experimental software are highly unpredictable, it is not possible to prevent them from causing problems (e.g., system crashes). It is also not possible to predict, after a software or machine crash, if it is possible and how long it takes to bring a machine back online. In this sense, the cluster is always in a beta stage. Although urgent experiments can get extra machines and help, users of the SciLens cluster should never make any assumption of the availability of the machines and software.

[38] JBOD stands for *J*ust a *b*unch *o*f *d*isks, which is a non-RAID drive architecture, containing an array of drives, each of which is accessed directly as an independent drive. See: http://en.wikipedia.org/wiki/Non-RAID_drive_architectures

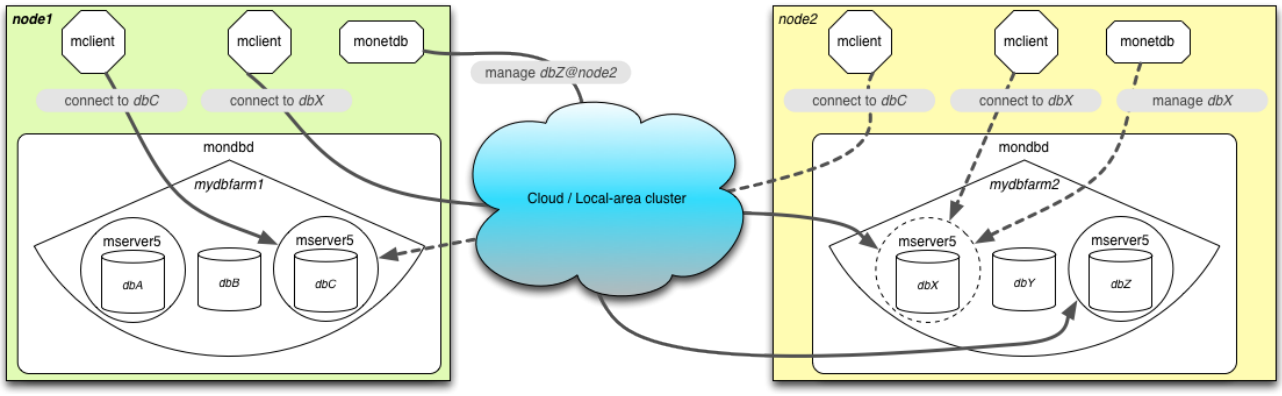

Figure 21: Distributed database management of MonetDB for Cloud environment and local-area clusters.

4.5 Deploying MonetDB on SciLens

In this subsection, we describe the tools we have developed that together support managing (i.e., create, start, stop and checking status) and querying databases that reside on different nodes in a Cloud or LAN environment, such as the SciLens cluster. The tools include the following programs:

- `monetdbd`: the MonetDB database server daemon. It is responsible for i) serving all databases in one local database farm to local or remote clients; and ii) redirecting a local client to a remote `monetdbd` if necessary.
- `monetdb`: control a single MonetDB database server instance, which can be either local or remote.
- `mserver5`: the MonetDB server version 5 of the data processing engine, which serves a single database to a (local or remote) client.
- `mclient`: the MonetDB command-line client tool[39], which can be used to connect to a database server and perform queries[40].

Figure 21 demonstrates how the tools interact with each other. Two nodes are connected through a Cloud or local-area cluster. A `monetdbd` is running on node1 and serving three databases, *dbA*, *dbB* and *dbC*, in its local database farm *mydbfarm1*. Two of the three databases, *dbA* and *dbB*, have already been started, indicated by a surrounding `mserver5` process. So, queries on these two databases will be answered by the corresponding `mserver5`. The situation on node2 is similar. The main difference is that only one database, *dbZ*, is already up and running. The database *dbX* is started on demand after a client connection; its surrounding `mserver5` process is indicated using the dashed line.

On node1, the left-most `mclient` sends a request to the local `monetdbd` to connect to database *dbC*. Since *dbC* is on the local host and is up and running, a connection between the requesting `mclient` and the `mserver5` in charge of *dbC* is established immediately. Then, the right-most `mclient` sends a request to the local `monetdbd` to connect to the database *dbX*. Since *dbX* is not locally available, the `monetdbd` on node1 searches in the local network for the remote `monetdbd` that serves *dbX*. This action is called *neighbour discovery*. The `monetdbd` instance on node2 will be found, which starts a `mserver5` to serve the database *dbX*. Finally, the `monetdbd` instance on node1 transparently redirects the requesting the right-most `mclient` to the `mserver5` serving *dbX* on node2. On node2, the `mclient` requests are handled in a similar way. The request from the left-most `mclient` to connect to the database *dbC* on node1 is

[39] MonetDB also supports other ways to connect to a database server, such as ODBC, JDBC, PHP, Python Perl and Ruby bindings.

[40] The primary query language is SQL. MonetDB also support other query languages such as JAQL, SciQL and SPARQL, but their support is in a less mature stage.

automatically redirect to the mserver5 on node1 serving *dbC*. The request from the right-most mclient to connect to the local database *dbX* is handled by a connection to the mserver5 that has just been started to serve *dbX*. So, the monetdbd daemon program provides a single access point for the database users (e.g., human users or (web) applications) to databases distributed over multiple nodes, but takes the burden of managing distributed databases (e.g., tracing their exact locations) from the users.

Figure 21 also shows that with the tool monetdb, one (e.g., the database administrator) can manage any single MonetDB database server mserver5 from any node in the network. Note the difference between the requests sent by the two monetdb instances on node1 and node2, respectively. Because the monetdb on node1 wants to manage a remote database *dbZ@node2*, its request explicitly identifies the remote node on which the database is running. Another difference is between an mclient and a monetdb. An mclient is not aware of the exact location of the database to which it wants to connect to, so an mclient always sends its connecting request to its local monetdbd which will take care of establishing a connection with the desired database. However, a monetdb is aware of the location of the database it wants to manage, so the connection request goes directly to the monetdbd on the target host. This choice is motivated by the fact that the administrator of a local-area cluster usually has the full knowledge of the distribution of all databases in the system.

In the reminder of this section, we elaborate how each of these tools work. Particularly, how a monetdbd discovers remote databases (Section 4.5.1.1), and how it handles multiple clients (Section 4.5.1.2). The man pages with complete usage information of these tools are included in Appendix I - Appendix IV.

4.5.1 The MonetDB Database Server daemon: monetdbd

The MonetDB Database Server daemon is called monetdbd. It can be started as follows:

```
monetdbd command [command_args] dbfarm
```

Where command is one of: create, start, stop, get, set, version or help. The database farm dbfarm to operate on must always be given to monetdbd explicitly. The complete usage information of monetdbd and its arguments and options is given in Appendix I.

The program is mainly meant to be used as a daemon, but it also allows setting up and changing the configuration of database farms. The use of monetdbd is either as user-oriented way to configure, start and stop a database farm, or to be started from a startup script, such as from */etc/init.d/* on Linux systems or smf(5) on Solaris systems, as part of a system startup.

A monetdbd instance manages one local cluster based database farm, referred to as the *dbfarm*, which is a directory in the system. Nowadays, the *dbfarm* location always has to be given as argument to monetdbd.

Within its local cluster monetdbd takes care of starting up databases when necessary, and stopping them either upon request via monetdb (see Section0 below) or when being shut down. Client database connections are made against monetdbd initially which redirects or proxies the client to the appropriate database process, started on the fly when necessary.

When started, monetdbd runs by default in the background, sending log messages to *merovingian.log*[41], until being sent a stop, terminate or interrupt signal, possibly using the stop command of monetdbd.

monetdbd uses a *neighbour discovery* scheme to detect other monetdbd processes running in the local network. Databases from those remote instances are made available to a local client that is trying to connect with a remote database. Remote databases never override local databases, and their availability is controlled by the remote monetdbd process. See also the sharing capabilities of monetdb (Section0) and *Remote Databases* (Section4.5.1.1) below.

[41] monetdbd is formerly known as merovingian. It was renamed to monetdbd since the name merovingian proved to be confusing to most regular end-users. Internally, monetdbd uses the name merovingian at many places for historical reasons.

4.5.1.1 Remote Databases

The neighbour discovery capabilities of `monetdbd` allow a user to contact a remote database transparently, as if it were a local database. By default, a `monetdbd` announces all its local databases in the network, so that a neighbouring `monetdbd` can pick them up to make them available for its local users. This feature can be disabled globally, or on database level. For the latter, the `monetdb` (see Section0 below) utility can be used to change the share property of a database.

While neighbour discovery in itself is sufficient to locate a database in a cluster, it is limited in expressiveness. For instance, database names are assumed to be unique throughout the entire system. This means local databases overshadow remote ones, and duplicate remote entries cannot be distinguished. To compensate for this, `monetdbd` allows adding a *tag* to each database that is being shared. This tag is sent in addition to the database name, and only understood by other `monetdbd` programs.

Tags are arbitrary ASCII-strings matching the pattern *[A−Za−z0−9./]+*. There are no assumed semantics in the tag, which allows for multiple approaches when using the tag. The tag is always used in combination with the database name. For this, the '/' character is used as the separator, which hence suggests the user to use that character as the separator for multilevel tags. `monetdbd` allows common path globbing using '*' on tags, which allows for many use cases. Consider for instance three databases with their tags: *dbX/master/tableQ*, *dbY/slave/tableQ* and *dbZ/slave/tableQ*. A default match has implicit '/*' added to the search, making more generic search strings match more specific ones. Hence, a connection with database *dbX* is the same as *dbX/**, which hence matches *dbX/master/tableQ*. Similarly, a database connection for **/master* matches the same database as before. Note that the implicit '/*' is not added if that would cause no matches, such as for **/master/tableQ*, which would return all masters for *tableQ*, which in our example is only *dbX*. In contrast, a database connect for **/slave/tableQ* matches with either *dbY* or *dbZ*. `monetdbd` returns these two options to the client in a round-robin fashion, such that subsequent connects for the same pattern result in a load-balanced connect to either of both databases.

With tags in use, one can possibly make distinction between databases with setup similar to the above example. The previous example could hence also be setup as follows: *tableQ/master*, *tableQ/slave* and *tableQ/slave*. Connecting to *tableQ/slave* would now return either of the two matching databases even though they are not unique (apart from the host they are located on, which is not shown in the example). While being confusing for humans, for `monetdbd` it is the same situation as in the previous example. However, because globbing allows making things easier to understand, tags for both slaves could be changed to *slaveX* or *slave/X* and use the necessary pattern to match them. It is up to the user to decide how to use the tags.

4.5.1.2 Multiplex-Funnels

`monetdbd` implements multiplex-funnel capabilities. As the name suggests two techniques are combined, the multiplexer and the funnel.

The *funnel* capability limits the access to the database to one client at a time. That is, if multiple clients connect to the funnel, their queries will be serialised such that they are executed one after the other. An effect of this approach is that clients no longer have an exclusive channel to the database, which means that individual queries from one client may have been interleaved with queries from others. This most notably makes SQL transaction blocks unreliable with a funnel. The funnel, hence, is meant to scale down a large amount of clients that perform short-running (read-only) queries, as typically seen in web-based query loads.

When a funnel is defined to use multiple databases, the funnel adds a *multiplexer* to its query channel. A multiplex-funnel sends each query to all of the defined databases. This behaviour can be quite confusing at first, but proves to be useful in typical sharding configurations, where in particular simple selection queries have to be performed on each of the shards. The multiplexer combines the answers from all defined databases in one single answer that it sends back to the client. However, this combining is without any smart logic, that is, the multiplexer does not evaluate the query it is running, but just combines all answers it receives from the databases. This results in, e.g., as many return tuples for a `SELECT COUNT(*)` query, as there are databases defined.

Due to the two above-mentioned characteristics, a multiplex-funnel has some limitations. As mentioned before, transactions over multiple queries are likely not to result in the desired behaviour. This is due to each

query to the funnel is required to be self-contained. Further, since for each query, the results from multiple servers have to be combined into one, that query must only return a single response, i.e., multi-statement queries are most likely causing the funnel to respond with an error, or return garbled results. Last, the size of each query is limited to currently about 80K. While this size should be sufficient for most queries, it is likely not enough for, e.g., `COPY INTO` statements. Apart from the data transfer implications, such statements should not be used with the funnel, as the results will be undefined due to the limited query buffer. Applications using the funnel should aim for short and single-statement queries that require no transactions.

4.5.2 Control a MonetDB Database Server instance: `monetdb`

The controlling program of a MonetDB Database Server instance is call `monetdb`. It can be started as follows:

```
monetdb [monetdb_options] command [command_options][command_args]
```

Where `command` is one of: `create`, `destroy`, `lock`, `release`, `status`, `start`, `stop`, `kill`, `set`, `get`, `inherit`, `discover`, `help` or `version`. The commands facilitate adding, removing, maintaining, starting and stopping a database inside the MonetDB Database Server. `monetdb` allows an administrator of the MonetDB Database Server to perform various operations on the databases in the cluster. It relies on a `monetdbd` running on the target host in the background for all operations. The complete usage information of `monetdb` and its arguments and options is given in Appendix II.

4.5.3 The MonetDB server version 5: `mserver5`

The current MonetDB server that performs all processing on request of clients for a certain database is called `mserver5`[42]. Note that while `mserver5`is the process that does the actual work, as we have explained before, it is recommended to start, monitor and connect to an `mserver5` process through `monetdbd`.

This man-page given in Appendix III describes the options that `mserver5` understands. However, it is only intended for people who really need to work with `mserver5` itself. In regular cases, the programs `monetdbd` and `monetdb` control the many options, and allow adjusting them to appropriate values where sensible. For normal usage, it is preferred to apply any configuration through these programs.

When the build-time configuration did not disable this, an `mserver5` process presents the user with a console prompt. On this prompt, MAL commands can be executed. The architecture is setup to handle multiple streams of requests. The first thread started represents the server, which is the console prompt, reading from standard input and writing to standard output.

The server thread started remains in existence until all other threads die. The server can be stopped by pressing *Ctrl-D* or entering the command `\q` in its console, or by sending it a termination signal (`SIGINT`, `SIGTERM`, `SIGQUIT`).

4.5.4 The MonetDB command-line tool: `mclient`

`mclient` is the command-line interface to a MonetDB server. It can be started as follows:

```
mclient [ options ] [ file or database [ file ... ] ]
```

If the `--statement=query (-s query)` option is given, the query is executed. If any files are listed after the options, queries are read from the files and executed. The special filename '-' refers to the standard input. Note that if there is both a `--statement` option and *filename* arguments, the query given with `--statement` is executed first. If no `--statement` option is given and no files are specified on the command line, `mclient` reads queries from standard input.

When reading from standard input, if standard input is a terminal or if the `--interactive (-i)` option is given, `mclient` interprets lines starting with a '\' (backslash) character specially. See the section BACKSLASH COMMANDS in Appendix IV.

Before `mclient` starts parsing command line options, it reads a *.monetdb* file. If the environment variable `DOTMONETDBFILE` is set, it reads the file pointed to by that variable instead. When unset, `mclient`

[42] The number 5 indicates the version number of the DBMS kernel.

searches for a *.monetdb* file in the current working directory, and if that does not exist, in the current user's home directory. This file can contain defaults for the flags `user`, `password`, `language`, `save_history`, `format` and `width`. For example, an entry in a *.monetdb* file that sets the default language for `mclient` to MAL looks like this: `language=mal`. To disable reading the *.monetdb* file, set the variable `DOTMONETDBFILE` to the empty string in the environment.

The complete usage information of `mclient` and its arguments and options is given in Appendix IV.

4.6 Conclusion

In the section, we have reported our experience on the designing, building and managing of the SciLens cluster, a large-scale local-area cluster intended for database research for data-intensive scientific application. We have also described four tools that have been developed to run distributed MonetDB databases on a Cloud or local-area network.

Currently, we are working on the second round to install machines for the two upper layers. The Bricks layer will typically contain machines that are comparable with contemporary server systems. We are considering the option 16 machines, with each of them equipped with two 8-core Intel® Xeon® processors, 256GB memory and four 2TB HDDs. We plan to connect the Bricks with two InfiniBand networks and two 1Gb Ethernet networks. The configuration for the top layer is still rather open. Basically, it will have 1, 2 or 4 high-end machines with, e.g., 8 core, 2 TB RAM. Another option we want to consider is to get 10Gb Ethernet between the switches, or even between some leaf machines. Thirdly, depending on the price and availability, we want to consider adding SSDs to the Rocks and/or in the high-end machines. Finally, we want to consider adding GPUs into the top layer high-end machines, since CPU-GPU cross device query processing is gaining increasing interest from the database research community (Pirk, 2012).

5 CumulusRDF: Linked Data Management on Nested Key-Value Stores

Many datasets on the Linked Data web cover descriptions of millions of entities. DBpedia (Auer et al. 2007), for example, describes 3.5m things with 670m RDF triples, and LinkedGeoData describes around 380m locations with over 2bn RDF triples. Standard file systems are ill equipped for dealing with these large amounts of files (each description of a thing would amount to one file). Thus, data publishers typically use full-fledged RDF triple stores for exposing their datasets online. Although the systems are in principle capable of processing complex (and hence expensive) queries, the offered query processing services are curtailed to guarantee continuous service[43].

At the same time, there is a trend towards specialised data management systems tailored to specific use cases (Stonebaker et al. 2005). Distributed key-value stores, such as Google Bigtable (Chang et al. 2006), Apache Cassandra (Lakshman et al. 2009) or Amazon Dynamo (DeCandia et al. 2007), sacrifice functionality for simplicity and scalability. These stores operate on a key-value data model and provide basic key lookup functionality only. Some key-value stores (such as Apache Cassandra and Google's Bigtable) also provide additional layers of key-value pairs, i.e. keys can be nested. We call these stores nested key-value stores. As we will see, nesting is crucial to efficiently store RDF data. Joins, the operation underlying complex queries, either have to be performed outside the database or are made redundant by de-normalising the storage schema. Given the different scope, key-value stores can be optimised for the requirements involving web scenarios, for example, high availability and scalability in distributed setups, possibly replicated over geographically dispersed data centres.

Common to many key-value stores is that they are designed to work in a distributed setup and provide replication for fail-safety. The assumption is that system failures occur often in large data centres and replication ensures that components can fail and be replaced without impacting operation.

The goal of our work (Ladwig et al. 2011) is to investigate the applicability of key-value stores for managing large quantities of Linked Data. We present our main contributions:

- We devise two RDF storage schemes on Google BigTable-like (nested) key-value stores supporting Linked Data lookups and atomic triple pattern lookups (Section 5.1).
- We compare the performance of the two storage schemes implemented on Apache Cassandra, using a subset of the DBpedia dataset with a synthetic and a real-world workload obtained from DBpedia's access log (Section 5.2).

We discuss related work in Section 5.3 and conclude with a summary and an outlook to future work in Section 5.4.

5.1 Storage Layouts

In the following we describe indexing schemes for RDF triples on top of nested key/value stores. The goals for the index scheme are:

- To cover all six possible RDF triple pattern with indices to allow for answering single triple patterns directly from the index. In other words, we aim to provide a complete index on RDF triples (Harth et al. 2005).
- To employ prefix lookups so that one index covers multiple patterns; such a scheme reduces the number of indices and thus index maintenance time and amount of space required.

Ultimately, the index should efficiently support basic triple pattern lookups and (as a union of those) Linked Data lookups which we evaluate in Section 5.2.

[43] Restrictions on query expressivity are common in other large-scale data services, for example in the Google Datastore.
See: http://code.google.com/appengine/docs/ python/datastore/overview.html/#Queries_and_Indexes

5.1.1 Nested Key-Value Storage Model

Since Google's paper on Bigtable (Chang et al. 2006), a number of systems have been developed that mimic Bigtable's mode of operation. Bigtable pioneered the use of the key-value model in distributed storage systems and inspired systems such as Amazon's Dynamo (DeCandia et al. 2007) and Apache's Cassandra (Lakshman et al. 2009). We illustrate the model in Figure 22.

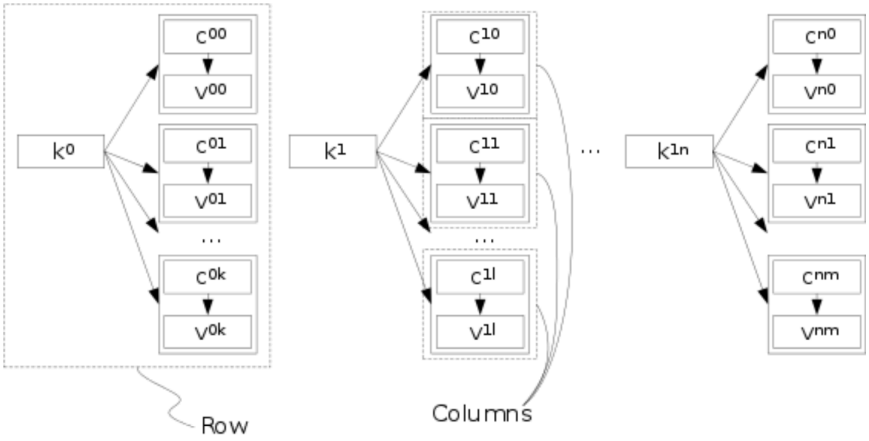

Figure 22: Key-value storage model comprising rows and columns. A lookup on the row key (k) returns columns, on which a lookup on column keys (c) returns values (v).

We use the notation `{ key:value }` to denote a key-value pair. We denote concatenated element with, e.g., `sp`, constant entries with `'constant'` and an empty entry with `-`. The storage model thus looks like the following:

```
{ row key : { column key : value } }
```

Systems may use different strategies to distribute data and organise lookups. Cassandra uses a distributed hash table structure for network access: storage nodes receive a hash value; row keys are hashed and the row is stored on a node, which is closest in numerical space to the row key's hash value. Only hash lookups thus can be performed.

Although there is the possibility for configuring an order preserving partitioner for row keys, we dismiss that option, as skewed data distribution easily can lead to hot-spots among the storage nodes.

Another restriction is that entire rows are stored on a single storage node - data with the same row key always ends up on the same machine. Columns, on the other hand, are stored in order of their column keys, which means that the index allows for range scans and therefore prefix lookups.

Please note that keys have to be unique; both row keys and column keys can only exist in the index once, which has implications on how we can store RDF triples.

Cassandra has two further features which our index schemes use: supercolumns and secondary indices. Supercolumns add an additional layer of key-value pairs; a storage model with supercolumns looks like the following:

```
{ row key : { supercolumn key : { column key : value }}}
```

Supercolumns can be either stored based on the hash value of the supercolumn key or n sorted order. In addition, supercolumns can be further nested.

Secondary indices, another feature of Cassandra we use for one of the storage layouts, allow to map column values to row keys:

```
{ value : row key }
```

Applications could use the regular key-value layout to also index values to row keys, however, secondary indices are "shortcuts" for such indices. In addition, secondary indices are built in the background without requiring additional maintenance code.

We identify two feasible storage layouts: one based on supercolumns (“Hierarchical Layout”), and one based on a standard key-value storage model (“Flat Layout”), but requiring a secondary index given restrictions in Cassandra.

5.1.2 Hierarchical Layout

Our first layout scheme builds on supercolumns.

The first index is constructed by inserting (s, p, o) triples directly into a supercolumn three-way index, with each RDF term occupying key, supercolumn and column positions respectively, and an empty value. We refer to that index as SPO, which looks like the following:

```
{ s : { p : { o : - } }
```

For each unique `s` as row key, there are multiple supercolumns, one for each unique `p`. For each unique `p` as supercolumn key, there are multiple columns, one for each `o` as column key. The column value is left empty.

Given that layout, we can perform (hash-base) lookups on `s`, (sorted) lookups on `p` and (sorted) lookups on `o`. We construct the POS and OSP indices analogously. We use three indices to satisfy all six RDF triple patterns as listed in Table 10. The patterns (spo) and (???) can be satisfied by any of the three indices.

Table 10: Triple patterns and respective usable indices to satisfy triple pattern

Triple Pattern	Index
(spo)	SPO, POS, OSP
(sp?)	SPO
(?po)	POS
(s?o)	OSP
(?p?)	POS
(s??)	SPO
(??o)	OSP
(???)	SPO, POS, OSP

5.1.3 Flat Layout

We base our second storage layout on the standard key-value data model. As columns are stored in a sorted fashion, we can perform range scans and therefore prefix lookups on column keys. We thus store (s, p, o) triples as

```
{ s : { po : - } }
```

where `s` occupies the row-key position, `p` the column-key position and `o` the value position.

We use a concatenated `po` as column key as column keys have to be unique. Consider using

```
{s :  { p : { o }}}
```

as layout, which `p` as column key and `o` as value. In RDF, the same predicate can be attached to a subject multiple times, which violates the column key uniqueness requirement. We would like to delegate all low level index management to Cassandra, hence we do not consider maintaining lists of `o`’s manually.

The SPO index satisfies the triple patterns *(s*??),(sp?) and *(spo)* with direct lookups. To perform a *(sp?)* lookup on the SPO index, we look for the column keys matching `p` (via prefix lookup on the column key `po`) on the row with row key `s`.

We also need to cover the other triple patterns. Thus, two more indices are constructed by inserting (p, o, s) and (o, s, p) re-orderings of triples, leading to POS and OSP indices. In other words, to store the SPO, POS

and OSP indices in a key-value store, we create a column family for each index and insert each triple three times: once into each column family in different order of the RDF terms.

There is a complication with the POS index though, because RDF data is skewed: many triples may share the same predicate (Harth et al. 2007). A typical case are triples with `rdf:type` as predicate, which may represent a significant fraction of the entire dataset. Thus, having P as the row key will result in a very uneven distribution with a few very large rows. As Cassandra is not able to split rows among several nodes, large rows lead to problems (at least a very skewed distribution, out of memory exceptions in the worst case). The row size may exceed node capacity; the uneven distribution makes load balancing problematic. We note that skewed distribution may also occur on other heavily used predicates (and possibly objects, for example in case of heavily used class URIs), depending on the dataset. For an in-depth discussion of uneven distribution in RDF datasets see Duan et al. 2011

To alleviate the issue, we take advantage of Cassandra's secondary indexes.

First, we use PO (and not P) as a row key for the POS index which results in smaller rows and also better distribution, as less triples share predicate and object than just the predicate. The index thus looks like the following:

```
{ po : { s : - } }
```

In each row there is also a special column 'p' which has P as value:

```
{ po : { 'p' : p } }
```

Second, we use a secondary index which maps column values to row keys, resulting in an index which allows for retrieving all PO row keys for a given P. Thus, to perform a *(?p?)* lookup, the secondary index is used to retrieve all row keys that contain that property. For *(?po)* lookups, the corresponding row is simply retrieved.

In effect, we achieve better distribution at the cost of a secondary index and an additional redirection for *(?p?)* lookups.

5.2 Evaluation

We now describe the experimental setup and the results of the experiments. We perform two sets of experiments to evaluate our approach:

- first, we measure triple pattern lookups to compare the hierarchical (labelled Hier) and the flat storage (labelled Flat) layout;
- second, we select the best storage layout to examine the influence of output format on overall performance.

5.2.1 Setting

We conducted all benchmarks on four nodes in a virtualised infrastructure (provided via OpenNebula[44], a cloud infrastructure management system similar to EC2). Each virtualised node has 2 CPUs, 4 GB of main memory and 40 GB of disk space connected to a networked storage server (SAN) via GPFS[45]. The nodes run Ubuntu Linux. We used version 1.6 of Sun's Java Virtual Machine and version 0.8.1 of Cassandra. The Cassandra heap was set to 2GB, leaving the rest for operating system buffers. We deactivated the Cassandra row cache and set the key cache to 100k. The Tomcat[46]server was run on one of the cluster nodes to implement the HTTP interface. Apache JMeter[47] was used to simulate multiple concurrent clients.

[44] See: http://opennebula.org
[45] See: http://www-03.ibm.com/systems/software/gpfs/
[46] See: http://tomcat.apache.org/
[47] See: http://jakarta.apache.org/jmeter/

5.2.2 Dataset and Queries

For the evaluation we used the DBpedia 3.6 dataset[48] excluding page links; characteristics of that DBpedia subset are listed in Table 11. We obtained DBpedia query logs from 2009-06-30 to 2009-10-25 consisting of 87,203,310 log entries.

Table 11: DBpedia dataset characteristics

Name	Value
Distinct triples	120,436,315
Distinct subjects	18,324,688
Distinct predicates	42,004
Distinct objects	40,531,020

We constructed two query sets:

- for testing single triple pattern lookups, we sampled 1 million S, SP, SPO, SO, O patterns from the dataset;
- for Linked Data lookups we randomly selected 2 million resource lookup log entries from the DBpedia logs (which, due to duplicate lookups included in the logs, amount to 1,241,812 unique lookups).

The triple pattern lookups were executed on CumulusRDF using its HTTP interface. The output of such a lookup is the set of triples matching the pattern, serialised as RDF/XML.

The output of a Linked Data lookup on URI u is the union of two triple pattern lookups: 1) all triples matching pattern *(u??)* and 2) a maximum of 10k triples matching *(??u)*. The number of results for object patterns is limited in order to deal with the skewed distribution of objects, which may lead to very large result sets. A similar limitation is used by the official DBpedia server (where a maximum of 2k triples in total are returned).

5.2.3 Results: Storage Layout

Table 12 shows the size and distribution of the indices on the four machines. The row size of a Cassandra column family is the amount of data stored for a single row-key. For example, for the OSP index, this is the number of triples that share a particular object.

Table 12: Index size per node in GB. POS Flat is not feasible due to skewed distribution of predicates. The size for POS Sec includes the secondary index.

Index	Node 1	Node 2	Node 3	Node 4	Std.Dev.	Max. Row
SPO Hier	4.41	4.40	4.41	4.41	0.01	0.0002
SPO Flat	4.36	4.36	4.36	4.36	0.00	0.00044
OSP Hier	5.86	6.00	5.75	6.96	0.56	1.16
OSP Flat	5.66	5.77	5.54	6.61	0.49	0.96
POS Hier	4.43	3.68	4.69	1.08	1.65	2.40
POS Sec	7.35	7.43	7.38	8.05	0.33	0.56
POS Flat	-	-	-	-	-	-

[48] See: http://wiki.dbpedia.org/Downloads36

The load distribution shows several typical characteristics of RDF data and directly relates to the dataset statistics from Table 11. The small maximum row size for SPO shows that there are very few triples that share a subject. The large maximum row size of the OSP index indicates that there are a few objects that appear in a large amount of triples, i.e., the distribution is much more skewed (this is usually due the small number of classes that appear as objects in the many `rdf:type` triples).

Here, we can also see the difference between the hierarchical and secondary POS indices. The hierarchical POS index only uses the predicate as key, leading to a very skewed distribution (indicated by the maximum row size) and a very uneven load distribution among the cluster nodes (indicated by the high standard deviation). The POS index using a secondary index fares much better as it uses the predicate and object as row key. This validates the choice of using secondary index for POS over the hierarchical layout.

5.2.4 Results: Queries

The performance evaluation consists of two parts: first, we use triple pattern lookups to compare two CumulusRDF storage layouts and, second, we examine the influence of output formats using Linked Data lookups.

Triple Pattern Lookups Figure 23 shows the average number of requests/s for the two CumulusRDF storage layouts (flat and hierarchical) for 2, 4, 8, 16 and 32 concurrent clients. Overall, the flat layout outperforms the hierarchical layout. For 8 concurrent clients, the flat layout delivers 1.6 times as many requests per second as the hierarchical layout. For both layouts the number of requests per second does not increase for more than 8 concurrent clients, indicating a performance limit. This may be due to the bottleneck of using a single HTTP server (as we will see in the next section).

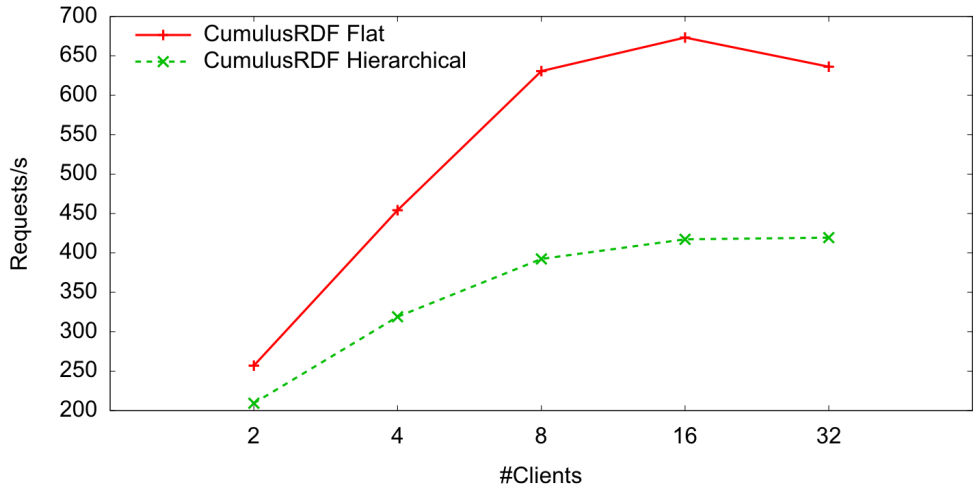

Figure 23: Requests per second for triple pattern lookups with varying number of clients.

Figure 24 shows the average response times from the same experiment, broken down by pattern type (S, O, SP, SO and SPO). This shows the differences between the two CumulusRDF storage layouts. While the hierarchical layout performs better for S, O and SP patterns, it performs worse for SO and SPO patterns. The worse performance for SO and SPO is probably due to inefficiencies of Cassandra super columns. For example, Cassandra is only able to de-serialise super columns as a whole, which means for SPO lookups all triples matching a particular subject and predicate are loaded. This is not the case for the flat layout (which does not use super columns): here, Cassandra is able to load only a single column, leading to much better response times.

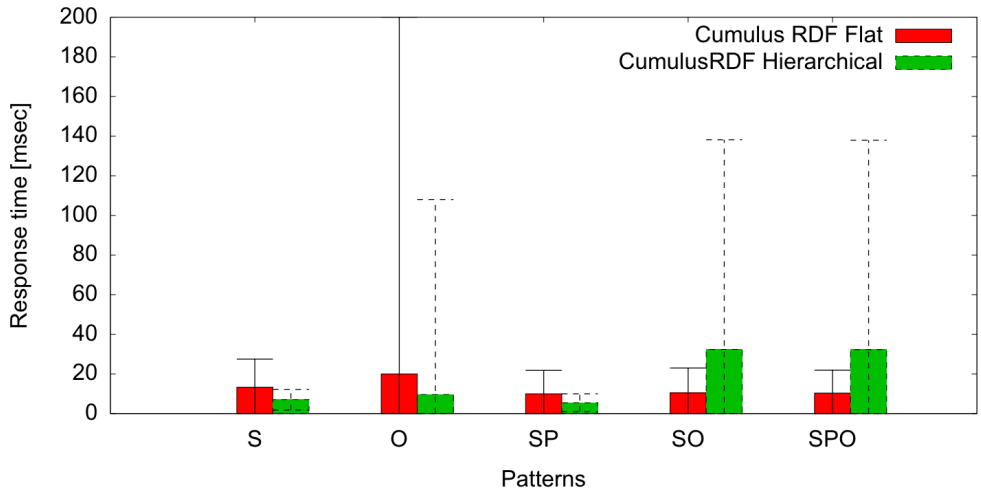

Figure 24: Average response times per triple pattern (for 8 concurrent clients). Please note we omitted patterns involving the POS index (P and PO). Error bars indicate the standard deviation.

Linked Data Lookups Figure 25 shows the average number of requests per second of the flat layout of CumulusRDF with two different output formats: RDF/XML and NTriples. As CumulusRDF stores RDF data in N3 notation the N-Triples output is cheaper to generate (for example, it does not require the escaping that RDF/XML does). This is reflected in the higher number of lookups per second that were performed using N-Triples output. The difference between the two output formats is more pronounced for a higher number of concurrent clients: for 4 clients N-Triples is 12% faster than RDF/XML, whereas for 8 clients the difference is 26%. This indicates that the performance is mainly limited by the Tomcat server, whose web app performs the output formatting.

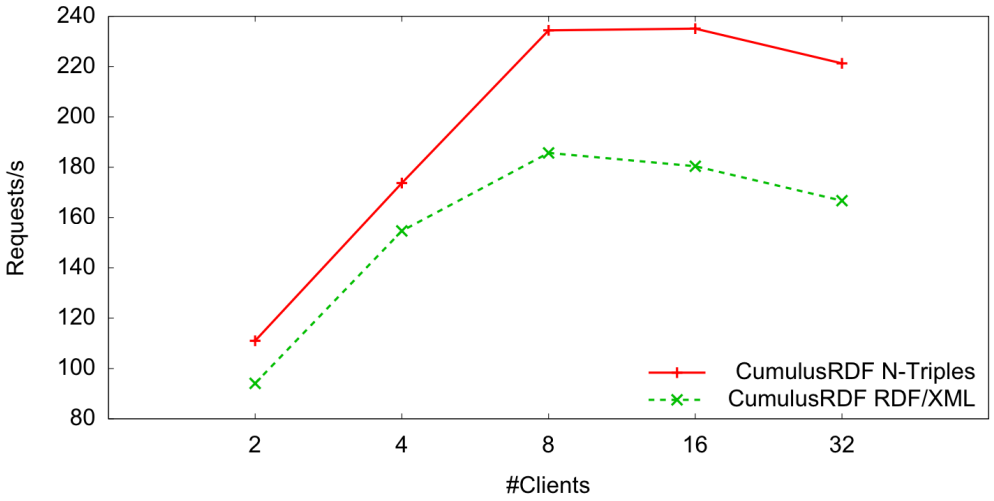

Figure 25: Requests per second Linked Data lookups (based on flat storage layout) with varying number of clients (including measure of effect of serialisation).

5.2.5 Conclusion

Overall, the evaluation shows that the flat layout outperforms the hierarchical layout. We also see that Apache Cassandra is a suitable storage solution for RDF, as indicated by the preliminary results. As the different layouts of CumulusRDF perform differently for different types of lookups, the choice of storage layout should be made considering the expected workload. From the experiments we can also see that the output format also has a large impact on performance. It may be beneficial to store the RDF data in an already escaped form if mainly RDF/XML output is desired.

5.3 Related Work

A number of RDF indexing schemes have been devised. YARS (Harth et al. 2005) described the idea of "complete" indices on RDF with context (quads), with an index for each possible triple pattern. Similar indices are used by RDF-3X (Neumann et al. 2008) and Hexastore (Weiss et al. 2008). All of these systems are built to work on single machines. In our work we use complete indices for RDF triples, implemented in three indices over a distributed key-value store.

Abadi et al. 2007 introduced an indexing scheme on C-Store for RDF called "vertical partitioning", however, with only an index on the predicate while trying to optimise access on subject or object via sorting of rows. The approach is sub-optimal on datasets with many predicates as subject or object lookups without specified predicate require a lookup on each of the predicate indices. For a follow-up discussion on C-Store we refer the interested reader to (Sidirougos et al. 2008).

The skewed distribution for predicates on RDF datasets and the associated issues in a distributed storage setting have been noted in (Harth et al. 2007).

Stratustore is an RDF store implemented over Amazon's SimpleDB (Stein et al. 2010). In contrast to Stratusstore, which only makes use of one index on subject (and similarly the RDF adaptor for Ruby[49]), we index all triple patterns. In addition, Cassandra has to be set up on a cluster while SimpleDB is a service offering, accessible via defined interfaces.

5.4 Conclusion and Future Work

We have presented and evaluated two index regimen for RDF on nested key-value stores to support Linked Data lookups and basic triple pattern lookups. The flat indexing scheme has given best results; in general, our implementation of the flat indexing scheme on Apache Cassandra can be seen as a viable alternative to full-fledged RDF stores in scenarios where large amounts of small lookups are required. We are working on packaging the current version of CumulusRDF for publication as open source at http://code.google.com/p/cumulusrdf/. We would like to add functionality for automatically generating and maintaining dataset statistics to aid dataset discovery or distributed query processors. Future experiments include measures on workloads involving inserts and updates.

[49] See: http://rdf.rubyforge.org/cassandra/

6 News Aggregator – A High-Volume Data Aggregation Pipeline

In this section, we describe a news aggregator pipeline, which was partially developed within the PlanetData project and showcases the handling of relatively large amounts of data.

The aggregator represents the first stage in processing and analysing textual streams – their acquisition, pre-processing into a form suitable for further analyses, and an efficient mechanism for their distribution.

6.1 Overview

The news aggregator is a piece of software developed at JSI which provides a real-time aggregated stream of textual news items provided by RSS-enabled news providers across the world. The pipeline performs the following main steps:

1) Periodically crawls a list of RSS feeds and a subset of Google News and obtains links to news articles
2) Downloads the articles, taking care not to overload any of the hosting servers
3) Parses each article to obtain
    a. Potential new RSS sources, to be used in step (1)
    b. Cleartext version of the article body
4) Process articles with Enrycher (see Section 6.3.2)
5) Expose two streams of news articles (cleartext and Enrycher-processed) to end users.

6.2 System Architecture

Refer to Figure 26 for a schematic overview of the architecture. The first part of the aggregator is based around a PostgreSQL database running on a Linux server. The database contains a list of RSS feeds which are periodically downloaded by the RSS monitoring component. RSS feeds contain a list of news article URLs and some associated metadata, such as tags, publication date, etc. Articles that are not already present in the database are added to a list of article URLs, and marked for download. Tags and publication date are also stored alongside, if found in the RSS.

A separate component periodically retrieves the list of new articles and fetches them from the web. The complete HTML is stored in the database, and simultaneously sent to a set of cleaning processes over a *0mq* message queue.

The cleaning process converts the HTML into UTF-8 encoding, determines which part of the HTML contains the useful text, and discards the remainder and all of the tags. Finally, a classifier is used to determine the primary language.

The cleaned version of the text is stored back in the database, and sent over a message queue to consumers.

Documents in English language are sent to the Enrycher web service, where named entities are extracted and resolved, and the entire document is categorized into a DMOZ topic hierarchy.

Both the cleartext and the enriched versions of documents are fed to a filesystem cache, which stores a sequence of compressed xml files, each containing a series of documents in the order they have arrived through the processing pipeline. The caching service exposes an HTTP interface to the web through an Apache transparent proxy, serving those compressed xml files on user request.

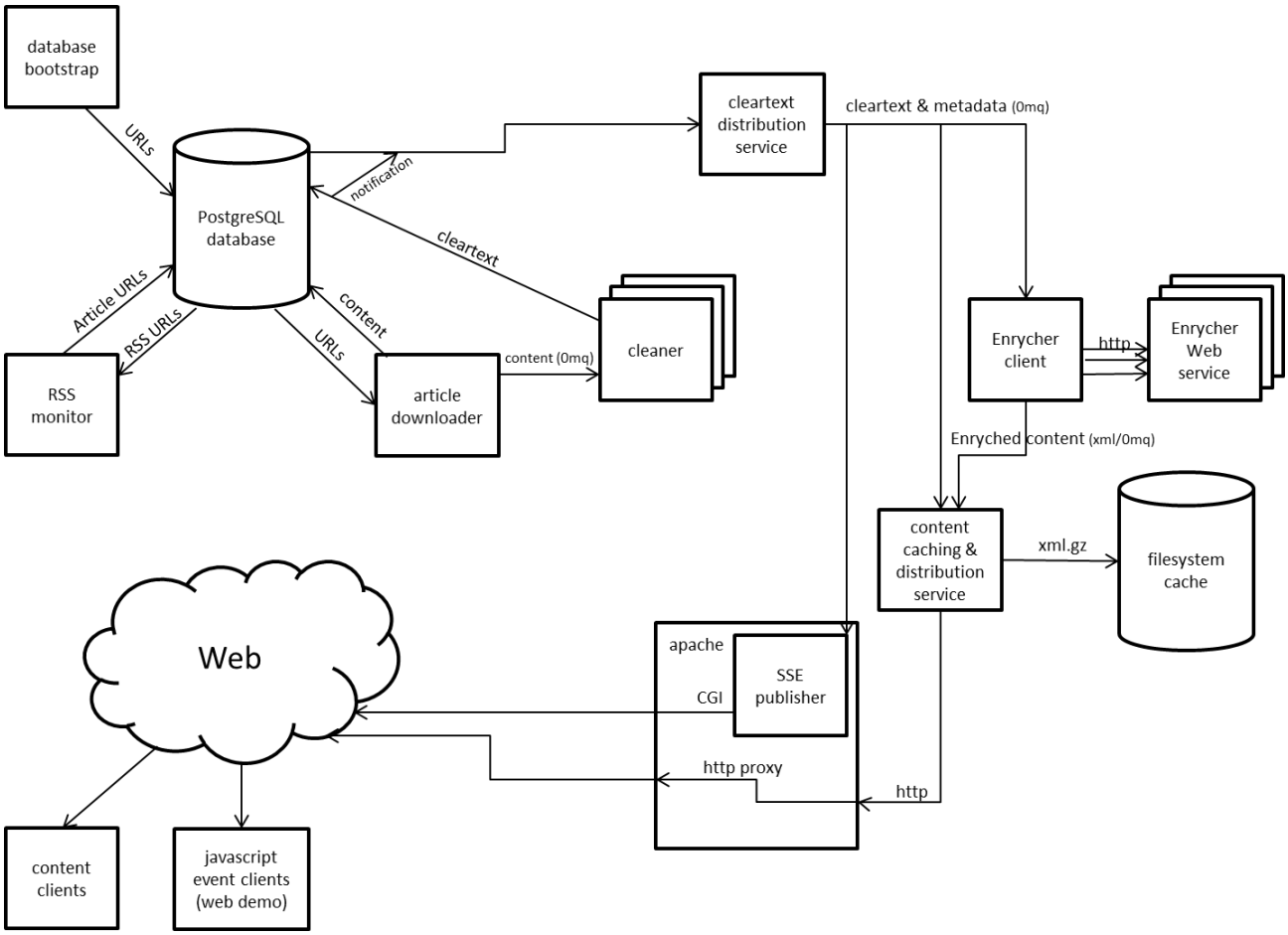

Figure 26. News aggregator system architecture.

The Apache server also hosts a CGI process capable of generating HTML5 server-side events, which contains the article metadata and cleartext as payload. These events can be consumed using Javascripts EventSource object in a web browser.

6.3 Data Preprocessing

Data preprocessing is an important part of the pipeline, both in terms of the added value provides and in terms of challenges posed by the data volume. The articles themselves are certainly useful, but almost any automated task dealing with them first needs to transform the raw HTML into a form more suitable for further processing. We therefore perform the preprocessing ourselves; this is much like the practice followed by professional data aggregation services like Spinn3r or Gnip.

In terms of data volume, preprocessing is the most interesting stage and the one at which the most tradeoff can be made. The present data download rate of about one article per second is nothing extreme, especially if we consider scaling to multiple processing nodes; however, it is nontrivial in that adding complex preprocessing steps (e.g. full syntactic parsing of text) or drastically increasing data load (e.g. including a 10% sample of the Twitter feed) would turn preprocessing into a bottleneck and require us to scale the architecture.

6.3.1 Extracting article body from web pages

Extracting meaningful content from the HTML is the most obviously needed preprocessing step. As this is a pervasive problem, a lot has been published on the topic; see e.g. Pasternack (2009), Arias (2009), and Kohlschütter (2010). We initially implemented the algorithm by Pasternack because of its simplicity and reported state-of-the-art performance. The algorithm scores each token (a word or a tag) in the document

based on how probable it is to comprise the final result (the scores are trained); then it extracts the maximum token subsequence.

Datasets

We tested the initial algorithm on three manually developed datasets. Each of the three consists of 50 articles, each from a different web site.

- english – English articles only.
- alphabet – Non-English articles using an alphabet, i.e. one glyph per sound. This includes e.g. Arabic.
- syllabary – Non-English articles using a syllabary, i.e. one glyph per syllable. This boils down to Asian languages. They lack word boundaries and have generally shorter articles in terms of glyphs. Also, the design of Asian pages tends to be slightly different.

Some of the input pages (about 5%), realistically, also do not include meaningful content. This is different from other data sets but very relevant to our scenario. Examples are paywall pages and pages with a picture bearing a single-sentence caption.

The fact that each of the 150 articles comes from a different site is crucial – most of the papers on this topic evaluate on a dataset from a small number of sites, which leads to overfitting and poor performance in the general case. This was also the case with Pasternack's algorithm. In particular, it tends to chip off the beginning and end of the article body if there is a link in vicinity; and it tends to include discussion threads appearing under the article body as the "benefit" of comments' text outweighs the markup between the body and the comments.

As the performance was unsatisfactory, we developed three new algorithms.

Algorithms

- WWW – an improved version of Pasternack, it extracts *two* most promising contiguous chunks of text from the article to account for the fact that the first paragraph is often placed separately from the main article body.
- WWW++ – a combination of WWW and heuristic pre- and post-processing to account for the most obvious errors of WWW. For instance, preprocessing tries to remove comments.
- DOM – a completely heuristics-based approach which requires the DOM tree to be computed. With the fast libxml package, this is not a limiting factor. The core of the heuristic is to take the first large enough DOM element that contains enough promising <p> elements. Failing that, take the first <td> or element which contains enough promising text. The heuristics for the definition of "promising" rely on metrics found in other papers as well; most importantly, the amount of markup within a node. Importantly, none of the heuristics are site-specific.

In all three algorithms, all pages are first normalized to the utf8 character set using the BeautifulSoup[50] package (which in turn uses a combination of http headers, meta tags and the chardet tool).

Evaluation

We evaluated two of the three pairs of algorithms by comparing per-article performance. We did compare WWW and DOM; based on informal inspection of outputs, DOM would be certain to perform better.

Dataset \ Algo	WWW vs WWW++ number of articles where one of the algorithms performs better			WWW++ vs DOM number of articles where one of the algorithms performs better		
	WWW	**tie**	**WWW++**	**WWW++**	**tie**	**DOM**
English	2	43	4	7	34	8
alphabet	4	37	8	6	36	7
syllabary	0	44	6	2	12	32

[50] See: http://www.crummy.com/software/BeautifulSoup/

Table 13. Performance comparison of webpage chrome removal algorithms

The differences between the algorithms are statistically significant with a 5% confidence interval only on the syllabary dataset; it is however clear from the data that overall, WWW++ performs better than WWW and DOM performs better still. DOM is therefore our algorithm of choice.

For DOM, we additionally performed an analysis of errors on all three datasets. As the performance did not vary much across datasets, we present the aggregated results. For each article, we manually graded the algorithm output as one of the following:

- **Perfect [66.3%]** – The output deviates from the golden standard by less than one sentence or not at all: a missing section title or a superfluous link are the biggest errors allowed. This also includes cases where the input contains no meaningful content and the algorithm correctly returns an empty string.
- **Good [22.1%]** – The output contains a subset or a superset of the golden standard. In vast majority of the cases, this means a single missing paragraph (usually the first one which is often styled and positioned on the page separately) or a single extraneous one (short author bio or an invitation to comment on the article). A typical serious but much rarer error is the inclusion of visitors' comments in the output.
- **Garbage [5.8%]** – The output contains mostly or exclusively text that is not in the golden standard. These are almost always articles with a very short body and a long copyright disclaimer that gets picked up instead.
- **Missed [5.8%]** – Although the article contains meaningful content, the output is an empty string, i.e. the algorithm fails to find any content.

If we combine "Perfect" and "Good" (where the outcome is most often only a sentence away from the perfect match) into a "Positive" score, both precision and recall for DOM are 94%. This (article-based) metric is arguably comparable with the word- or character-based metrics employed in some other papers on state of the art methods (Kohlschütter 2010); those also report precision and accuracy of at most 95%.

6.3.2 Extracting semantic information from unstructured text

For most of semantic processing, we rely on Enrycher (Štajner 2009) running as a service. In order to increase error resiliency, improve the utilization of the service and avoid undue delays in the preprocessing pipeline, we access the service in a multithreaded fashion. For performance evaluation and other information, please refer to the paper by Štajner.

Enrycher annotates each article with named entities appearing in the text (resolved to Wikipedia when possible), discerns its sentiment and categorizes the document into the general-purpose DMOZ category hierarchy.

We also annotate articles with a language; detection is provided by a combination of Google's open-source Compact Language Detector library for mainstream languages and a separate Bayesian classifier. The latter is trained on character trigram frequency distributions in a large public corpus of over a hundred languages. We use CLD first; for the rare cases where the article's language is not supported by CLD, we fall back to the Bayesian classifier. The error introduced by automatic detection is below 1% (McCandless, 2011).

6.4 Data Properties

In no particular order, we list some statistics of the data provided by the news aggregator.

6.4.1 Sources

The crawler actively monitors about 75000 feeds from 1900 sites. The list of sources is constantly being changed – stale sources get removed automatically, new sources get added from crawled articles. In addition, we occasionally manually prune the list of sources using simple heuristics as not all of them are active, relevant or of sufficient quality. The feed crawler has inspected about 350000 RSS feeds in its lifetime. The list was bootstrapped from publically available RSS compilations.

Besides the RSS feeds, we use Google News (news.google.com) as another source of articles. We periodically crawl the US English edition and a few other language editions, randomly chosen at each crawl. As news articles are later parsed for links to RSS feeds, this helps diversify our list of feeds while keeping the quality high.

We also support additional custom news sources, provided by payable services and/or project partners in various projects.

The sources are not limited to any particular geography or language.

6.4.2 Language distribution

We cover 37 languages at an average daily volume of 100 articles or more. English is the most frequent with an estimated 54% of articles being in that language. German, Spanish and French are represented by 3 to 10 percent of the articles. Other languages reaching at least 1% of the corpus are Chinese, Slovenian, Portugese, Korean, Italian and Arabic.

6.4.3 Data volume

The crawler currently downloads between 50000 and 100000 articles per day which amounts to roughly one article per second. The current archive contains about 40 million articles and begins in May 2008.

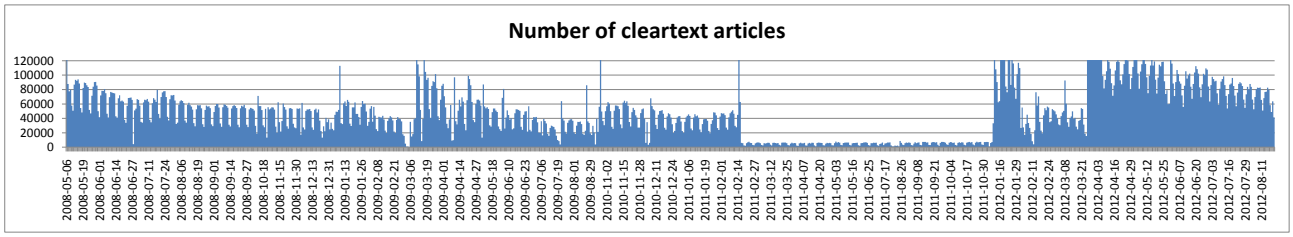

Figure 27. The daily number of downloaded articles. A weekly pattern is nicely observable. Through most of 2011, only Google News was used as an article source, hence the significantly lower volume in that period.

The median and average article body length is 1750 and 2400 bytes, respectively.

6.4.4 Responsiveness

We poll the RSS feeds at varying time intervals from 5 minutes to 12 hours depending on the feed's past activity. Google News is crawled every two hours. All crawling is currently performed from a single machine; precautions are taken not to overload any news source with overly frequent requests.

Based on articles where the RSS feed provides the original time of publication, we estimate 70% of articles are fully processed by our pipeline within 3 hours of being published, and 90% are processed within 12 hours.

6.5 Data Dissemination

Two streams of news are made available:

1) The stream of all downloaded news articles
2) The stream of English articles, processed with Enrycher

Figure 28. Demo preview of the news stream at http://newsfeed.ijs.si/.

6.5.1 Storing and Serving the Data

Each stream of articles is serialized into XML and segmented by time into .gz files several megabytes in size. The data is made available through a simple HTTP service that can be polled for new files periodically. Although this scheme introduces minor additional delays, it has the major benefit of keeping the server-side cache simple and easy to maintain.

Independent of this server-side, filesystem-based cache, a complete copy of the data is still kept in the traditional structured database. This is the only copy guaranteed to be consistent and contain all the data; from it, the XML files can be regenerated at any time. This is particularly useful in case of XML format changes and/or improvements to the preprocessing pipeline.

For a client to access the data, a request can be made to a URL of the form

`http://newsfeed.ijs.si/stream/`*`STREAM_NAME`*`?after=`*`TIMESTAMP`*

This returns the oldest available .gz file created after *`TIMESTAMP`* or HTTP error 404 if no recent enough file exists.

The `after` parameter is optional; if omitted, the oldest .gz file overall is returned. (We will attempt to maintain a month's worth of articles available through this API. The remainder of the archive is available to project partners on request.) *`TIMESTAMP`* must be given in the ISO 9601 format `yyyy-mm-ddThh:mm:ssZ` (`Z` and `T` are literals).

We also provide a python script which polls the server at one-minute intervals and copies new .gz files to the local disk as they are made available on the server. See http://newsfeed.ijs.si/.

Note: Due to the streaming nature of the pipeline, the articles in .gz files are only approximately chronologically sorted; they are sorted in the order in which they were processed rather than published.

6.5.2 Stream Serialization Format and Contents

Each .gz file contains a single XML tree. The root element, `<article-set>`, contains zero or more articles in the following format (elements marked with "?" are only present when we successfully extract the corresponding piece of information):

```
<article id="internal article ID; consistent across streams">
   <source>
      <uri> URL from which the article was discovered; typically the RSS feed
</uri>
      <title> Title for the source </title>
      <type> MAINSTREAM_NEWS </type>
```

```
      <location?>
         <longitude> publisher longitude in degrees </longitude>
         <latitude> publisher latitude in degrees </latitude>
         <city?> publisher city </city>
         <country?> publisher country </country>
      </location>
      <tags?> RSS-provided tags; the tag vocabulary is not controlled
         <tag> some_tag </tag>
         <tag> another_tag </tag>
      </tags>
   </source>
   <uri> URL from which the article was downloaded </uri>
   <publish-date?> The publication time and date.</publish-date>
   <retrieve-date> The retrieval time and date.</retrieve-date>
   <lang> 3-letter ISO 639-2 language code </lang>
   <location?>
      <longitude> story content longitude in degrees </longitude>
      <latitude> story content latitude in degrees </latitude>
      <city?> story city </city>
      <country?> story country </country>
   </location>
   <tags?> RSS-provided tags; the tag vocabulary is not controlled
      <tag> some_tag </tag>
      <tag> another_tag </tag>
   </tags>
   <img?> The URL of a related image, usually a thumbnail. </img>
   <title> Title. Can be empty if we fail to identify it. </title>
   <body> See below. </body>
</article>
```

The `<article-body>` element contains

- for the cleartext stream: cleartext body of the article, segmented into paragraphs with
- for the Enrycher stream: an XML subtree as returned by Enrycher. See documentation at http://enrycher.ijs.si/ for the exact format. Content-wise, the subtree contains document categorization into DMOZ, named entity detection and named entity resolution (i.e. entities are linked to DBpedia and YAGO).

7 Conclusion

In this deliverable we give guidance on the deployment and use of large-scale data management tools and infrastructure.

We reported on our collected experience to derive best practices and enable the PlanetData partners to

- Optimize analysis tasks for performance and cost reduction.
- Develop Strategies to design and manage local area clusters.
- Maintain scalable data storage and provisioning systems.
- Acquire large data sets and pre-process them for further use.

In particular we detailed best practices for data analysis with Apache Hadoop and MapReduce. We focused on typical challenges when developing MapReduce analysis tasks (e.g., joins) and provided methods to resolve resulting problems. Furthermore we gave a guide on the design of Hadoop clusters to reduce the execution time of analysis tasks.

With regard to data analysis we additionally described the Web Data Commons project as showcase on how to perform an analysis over large datasets. Concretely we extracted structured data from two large Web corpora using Amazon cloud infrastructure and post-processed the data with a Hadoop MapReduce job.

We addressed infrastructure design with SciLens, a local area cluster. From the design of SciLens we derived best practices on how to select hardware to achieve the best trade-off between price and performance and how to manage a cluster that is intended for experimental use.

With regard to data storage and provisioning we described CumulusRDF, where we investigate the feasibility of using a distributed nested key/value store as an underlying storage component to achieve scalable storage and retrieval system. Our results indicate that the chosen approach is competitive to state-of-the-art distributed RDF stores.

Furthermore we described how the acquisition of large amounts of data and it's processing into a suitable form for further analysis can be handled with the News Aggregator Pipeline, a large-scale data acquisition tool.

This deliverable serves as guidance for further research of all PlanetData members on large-scale data.

References

Abadi, D. J., Marcus, A., Madden, S. R., Hollenbach, K. (2007), *Scalable semantic web data management using vertical partitioning,* In Proceedings of the 33rd International Conference on Very Large Data Bases, 411–422. VLDB Endowment.

Arias, J., Deschacht, K., & Moens, M. (2009). *Language independent content extraction from web pages*. Proceedings of the 9th Dutch-Belgian information retrieval workshop.

Auer, S., Bizer, C., Kobilarov, G., Lehmann, J., Cyganiak, R., Ives, Z. G. (2007), *Dbpedia: A nucleus for a web of open data*, In Proceedings of the 6th International Semantic Web Conference, 722–735.

Becla, J. and Lim, K.T. (2008). *Report from the first Workshop on Extremely Large Databases*. In Data Science Journal, Vol 7, pages 1—13.

Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., Gruber, R. E. (2006), *Bigtable: a distributed storage system for structured data,* In Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation, 15–15.

Cyberinfrastructure Vision for 21st Century DisCoVery. (2007) NSF07-28, March 2007. National Science Foundation (USA). http://www.nsf.gov/pubs/2007/nsf0728/

Dean, J., Ghemawat, S., 2008. *MapReduce: simplified data processing on large clusters*, In Communications of the ACM 51, 1.

DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W. (2007), *Dynamo: Amazon's highly available key-value store,* In Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles, 205–220. ACM.

Duan, S., Kementsietsidis, A., Srinivas, K., Udrea, O. (2011), *Apples and oranges: a comparison of rdf benchmarks and real rdf datasets,* In Proceedings of the 2011 International Conference on Management of Data, 145–156. ACM.

Ganz, J. F., Chute, C., Manfrediz, A., Minton, S., Reinsel, D., Schlichting, W., Toncheva, A. (2008). *The Diverse and Exploding Digital Universe,* IDC, http://www.emc.com/collateral/analyst-reports/diverse-exploding-digital-universe.pdf

Gray, J., Liu, D. T., Nieto-Santisteban, M. A., Szalay, A. S., DeWitt, D. J., and Heber, G. (2005). *Scientific Data Management in the Coming Decade*. In the Computing Research Repository (CoRR), volume abs/cs/0502008.

Gray, J., Szalay, A. S. (2004). *Where the Rubber Meets the Sky. Bridging the Gap between Databases and Science*. In IEEE Data Eng. Bull. 27(4): 3—11.

Harth, A., Decker, S. (2005), *Optimized Index Structures for Querying RDF from the Web*, In Proceedings of the 3rd Latin American Web Congress, 71–80. IEEE Computer Society.

Harth, A., Umbrich, J., Hogan, A., Decker, S. (2007) *Yars2: A federated repository for querying graph structured data from the web,* In 6th International Semantic Web Conference, 211–224.

Hey, T., Tansley, S., Tolle, K. (Eds.). (2009). *The Fourth Paradigm: Data-intensive Scientific Discoveries*. Redmond, Washington: Microsoft research. Retrieved from http://research.microsoft.com/en-us/collaboration/fourthparadigm/

Huang, J., Daniel J. Abadi, K. R. (2011), *Scalable SPARQL Querying of Large RDF Graphs*, in Proceedings of the VDLB Endowment 4(11): 1123-1134

Klyne, G., Carroll, J. J. (2004) *Resource Description Framework (RDF): Concepts and Abstract Syntax,* W3C Recommendation, http://www.w3.org/TR/rdf-concepts/

Kohlschütter, C., Fankhauser, P., & Nejdl, W. (2010). *Boilerplate detection using shallow text features*. Proceedings of WSDM 2010.

Ladwig, G., Harth A. (2011), *CumulusRDF: Linked Data Management on Nested Key-Value Stores* Proceedings of the 7th International Workshop on Scalable Semantic Web Knowledge Base Systems ISWC.

Lakshman A., Malik, P. (2009), *Cassandra: a structured storage system on a p2p network,* In Proceedings of the 21st Annual Symposium on Parallelism in Algorithms and Architectures, 47–47. ACM.

Lam, C. (2011), *Hadoop in Action*, Manning Publications Co., Stamford.

Mika, P. (2011). *Microformats and RDFa deployment across the Web* http://tripletalk.wordpress.com/2011/01/25/rdfa-deployment-across-the-web/

Mühleisen, H., Bizer, C. (2012), *Web Data Commons - Extracting Structured Data from Two Large Web Corpora*. Invited paper at the 5th Linked Data on the Web.

Neumann, T., Weikum, G. (2008). *RDF-3X: a RISC-style Engine for RDF*. Proceedings of the VLDB Endowment, 1(1):647–659.

Pasternack, J., & Roth, D. (2009). *Extracting article text from the web with maximum subsequence segmentation.* Proceedings of the 18th WWW conference

Patterns of information use and exchange: case studies of researchers in the life sciences. (2009). A report by the Research Information Network and the British Library. http://rinarchive.jisc-collections.ac.uk/our-work/using-and-accessing-information-resources/patterns-information-use-and-exchange-case-studie

Pirk, H. (2012). *Efficient Cross-Device Query Processing*. In Proceedings of the International Conference on Very Large Data Bases. Istanbul, Turkey. VLDB Endowment.

Sidirourgos, L., Goncalves, R., Kersten, M., Nes, N., Manegold, S. (2008*), Column-store support for rdf data management: not all swans are white,* Proceedings of the VLDB Endowment, 1(2):1553–1563.

Sidirourgos, L., Kersten, M.L., and Boncz, P. (2011). *SciBORQ: Scientific data management with Bounds On Runtime and Quality*. In CIDR.

Štajner, T., Rusu, D., Dali, L., & Fortuna, B. (2009). *Enrycher: service oriented text enrichment*. Proc. of SiKDD.

Stein R., Zacharias, V. (2010), *Rdf on cloud number nine*. In Workshop on NeFoRS: New Forms of Reasoning for the Semantic Web: Scalable & Dynamic.

Stonebraker, M., Becla, J, DeWitt, D. J., Lim, K. T., Maier, D., Ratzesberger, O., and Zdonik, S. B. (2009). *Requirements for Science Data Bases and SciDB*. In CIDR.

Stonebraker, M., Cetintemel, U. (2005), *"one size fits all": An idea whose time has come and gone,* In Proceedings of the 21st International Conference on Data Engineering, 2–11. IEEE Computer Society.

Szalay, A. S., Gray, J., Thakar, A., Kunszt, P. Z., Malik, T., Raddick, J., Stoughton, C., van den Berg, J. (2002). *The SDSS SkyServer: public access to the sloan digital sky server data*. In SIGMOD Conference.

Weiss, C., Karras, P., Bernstein, A. (2008) *Hexastore: sextuple indexing for semantic web data management. Proceedings of the VLDB Endowment*, 1(1):1008–1019.

White, Tom (2010), *Hadoop: The Definive Guide (2nd Edition)*, O'Reilly Media, Sebastopol.

Appendix I The `monetdbd` man-page

NAME

`monetdbd` – the MonetDB Database Server daemon

SYNOPSIS

```
monetdbd command [command_args] dbfarm
```

DESCRIPTION

`monetdbd` is the MonetDB Database Server daemon. The program is mainly meant to be used as daemon, but it also allows to setup and change the configuration of a dbfarm. The use of `monetdbd` is either as user-oriented way to configure, start and stop a database farm, or to be started from a startup script, such as from /etc/init.d/ on Linux systems or `smf`(5) on Solaris systems, as part of a system startup.

`monetdbd` is the system formerly known as `merovingian`. It was renamed to `monetdbd` since the name `merovingian` proved to be confusing to most regular end-users. Internally, `monetdbd` uses the name `merovingian` at many places for historical reasons.

A `monetdbd` instance manages one local cluster based, which is a directory in the system, referred to as the dbfarm. Nowadays, the dbfarm location always has to be given as argument to `monetdbd`.

Within its local cluster `monetdbd` takes care of starting up databases when necessary, and stopping them either upon request via `monetdb`(1) or when being shut down. Client database connections are made against `monetdbd` initially which redirects or proxies the client to the appropriate database process, started on the fly when necessary.

When started, `monetdbd` runs by default in the background, sending log messages to merovingian.log, until being sent a stop, terminate or interrupt signal, possibly using the stop command of `monetdbd`.

`monetdbd` uses a neighbour discovery scheme to detect other `monetdbd` processes running in the local network. Databases from those remote instances are made available to a locally connecting client. Remote databases never override local databases, and their availability is controlled by the remote `monetdbd` process. See also the sharing capabilities of `monetdb`(1) and the REMOTE DATABASES section below.

COMMANDS

The commands for `monetdbd` are `create`, `start`, `stop`, `get`, `set`, `version`, and `help`. The commands facilitate initialising a dbfarm, starting and stopping the MonetDB Database Server, and retrieving or setting options.

`create dbfarm`

Initialises a new database farm, such that a MonetDB Database Server can be started on that location. All necessary directories are attempted to be created, and an initial properties file is created in the directory itself. `dbfarm` must be a location addressable in the local filesystem hierarchy.

`start [-n] <dbfarm>`

Starts `monetdbd`, the MonetDB Database Server, on the given `dbfarm`. When the `-n` flag is given, `monetdbd` will not fork into the background, but instead remain attached to the calling environment, until given a stop signal.

`stop <dbfarm>`

Sends a stop signal to the `monetdbd` process responsible for the given `dbfarm`.

`get <all | property[,property[,..]]> <dbfarm>`

Prints the requested properties, or all known properties, for the given dbfarm. For each property, its value is printed. Some properties are virtual, and given for information purposes only, they cannot be modified using the set command.

`set property=value <dbfarm>`

Sets property to value for the given database. For a list of properties, run `monetdbd get all`. Some properties require a restart of the MonetDB Database Server in order to take effect. The set command, will however always write the property, and tell the running `monetdbd` to reload the properties file (if running). For an explanation of the properties, see the CONFIGURATION section below.

CONFIGURATION

`monetdbd` reads its properties from the *.merovingian_properties* file inside the dbfarm. This file is created by the `create` command. This file is not meant to be edited manually, instead it should be updated using the set command. The following properties can be set:

`logfile`

This property points to the file where all log messages are written to. It is relative to the dbfarm directory, but can be absolute to point to e.g. another medium. Changing this property takes effect immediately at runtime.

`pidfile`

`monetdbd` stores the process ID of the background server in the file pointed to by this property. The same rules apply as for the `logfile` property.

`sockdir`

For faster access, `monetdbd` uses UNIX domain sockets for its control mechanism and regular database connections. The sockets are placed as files in the filesystem hierarchy. The `sockdir` property controls in which directory they are placed. In general this setting should not be changed.

`port`

This property specifies which TCP port `monetdbd` should listen to for connection requests. Defaults to 50000.

`control`

For remote management of `monetdbd`, the control property specifies whether or not to enable remote management. Note that for remote management, a passphrase is required, see below. It defaults to false for security reasons. Changing this property takes effect immediately at runtime.

`passphrase`

To control `monetdbd` from a remote machine, a passphrase is necessary, to be given to `monetdb`(1). The passphrase can be either given as hashed value prefixed by the hash type in curly braces (e.g. *{SHA512}xxx...*) or as plain text value which will be hashed automatically. Note that the only hash accepted is the one specified at configure time, which is SHA512. Changing this property takes effect immediately at runtime.

`discovery`

Specifies whether neighbour discovery is to be enabled using UDP broadcasts or not. The broadcasts are done on the same portnumber as the port setting.

`discoveryttl`

`monetdbd` publishes locally available databases to others periodically. The interval used here, defined in seconds, depends on the time-to-live of the databases before they need to get refreshed. The default is 600 seconds (10 minutes), which should keep traffic in your

network fairly low. Additions and removals are processed immediately regardless of this timeout. If you are in a network environment where physical network links disappear often, you may want to decrease this value to more quickly remove no longer reachable databases.

`exittimeout`

mservers that were started by the MonetDB Database Server are shut down when `monetdbd` is shut down. Setting the exittimeout property to a positive non-zero value will shut down each running mserver with the given time-out in seconds. If the time-out expires, the mserver process is killed using the `SIGKILL` signal. A time-out value of 0 means no mservers will be shut down, and hence they will continue to run after `monetdbd` has shut down. Note that this particular configuration is extremely inconvenient. The default time-out is 60 seconds. If your databases are rather large and find your databases consistently being killed by monetdbd upon shutdown, you may want to increase this time-out. Changing this property `takes` effect immediately at runtime.

`forward`

`monetdbd` has two ways in which it can "attach" a connecting client to the target database. The first method, redirect, uses a redirect sent to the client with the responsible mserver process. The second method, proxy, proxies the client to the mserver over `monetdbd`. While redirect is more efficient, it requires the connecting client to be able to connect to the mserver. In many settings this may be undesirable or even impossible, since a wide range of open ports and routing are necessary for this. In such case the proxy technique of `monetdbd` is a good solution, which also allows a `monetdbd` instance on the border of a network to serve requests to nodes in the local (unreachable) network. Note that for local databases, the proxy method uses a UNIX domain socket feature to pass file-descriptors to the local mserver. This effectively is as efficient as the redirect approach, but still hides away the mservers properly behind `monetdbd`. Hence, in practice it is only relevant for connections to remote databases to use redirects instead of proxies. Changing this property takes effect immediately at runtime.

SIGNALS

`monetdbd` acts upon a number of signals as is common for a daemon.

`SIGINT, SIGTERM, SIGQUIT`

Any of these signals make `monetdbd` enter the shutdown sequence. This sequence involves cleanly shutting down listener sockets, shutting down all started databases and finally terminating itself.

`SIGHUP`

When this signal is received by `monetdbd` it will reopen the logfile as pointed to by the logfile setting. Before it reopens the logfile, it will re-read the properties file from the dbfarm, which might result in opening a different file to continue logging.

RETURN VALUE

`monetdbd` returns exit code 0 if it was able to successfully perform the requested action, e.g. start, stop, etc. When an error occurs during the action, that prevents `monetdbd` from successfully performing the action, the exit code 1 is returned.

SEE ALSO

`monetdb`(1) `mserver5`(1)

Appendix II The `monetdb` man-page

NAME

`monetdb` − control a MonetDB Database Server instance

SYNOPSIS

```
monetdb [monetdb_options] command [command_options] [command_args]
```

DESCRIPTION

`monetdb` allows an administrator of the MonetDB Database Server to perform various operations on the databases in the server. It relies on `monetdbd`(1) running in the background for all operations.

OPTIONS

`monetdb_options` affect all commands and control the general behaviour of `monetdb`.

`-q`

Supresses all standard progress messages, only writing output to `stderr` if an error occurred.

`-h` *`hostname`*

Connect to `hostname` instead of attempting a connection over the local UNIX socket. This allows `monetdb` to connect to a remote `monetdbd`(1). The use of this option requires `-P` (see below).

`-p` *`port`*

Connects to the given port number instead of the default (50000). Requires `-h` to be given as option too.

`-P` *`passphrase`*

Specifies the passphrase necessary to login to a remote `monetdbd`(1). This option requires `-h` to be given as well. A bad passphrase causes `monetdb` to fail to login, and hence fail to perform any remote action.

`-v`

Show version, equal to `monetdb` version.

COMMANDS

The commands for the `monetdb` utility are `create`, `destroy`, `lock`, `release`, `status`, `start`, `stop`, `kill`, `set`, `get`, `inherit`, `discover`, `help` and `version`. The commands facilitate adding, removing, maintaining, starting and stopping a database inside the MonetDB Database Server.

For all commands, database arguments can be glob-like expressions. This allows to do wildcard matches. For details on the syntax, see EXPRESSIONS.

`create [-m pattern] database [database ...]`

Initialises a new database in the MonetDB Database Server. A database created with this command makes it available under its database name, but not yet for use by clients, as the database is put into maintenance mode. This allows the database administrator to perform initialisation steps before releasing it to users. See also `monetdb lock`. The name of the database must match the expression *[A−Za−z0−9−_]+*.

`-m pattern`

With the `-m` flag, instead of creating a database, a multiplex-funnel is created. See section MULTIPLEX-FUNNEL in `monetdbd`(1). The pattern argument is not fully the same as a

pattern for connecting or discovery. Each parallel target for the multiplex-funnel is given as *username+password@pattern* sequence, separated by commas. Here the pattern is an ordinary pattern as would be used for connecting to a database, and can hence also be just the name of a database.

`destroy [-f] database [database ...]`

Removes the given database, including all its data and logfiles. Once destroy has completed, all data is lost. Be careful when using this command.

`-f`

By default, a confirmation question is asked, however the `-f` option, when provided, suppresses this question and removal is executed right away. Note that you cannot destroy a running database, bring it down first using the `stop` command.

`lock database [database ...]`

Puts the given database in maintenance mode. A database under maintenance can only be connected to by an administrator account (by default the "monetdb" account). A database which is under maintenance is not started automatically by `monetdbd`(1), the MonetDB Database Server, when clients request for it. Use the `release` command to bring the database back for normal usage. To start a database which is under maintenance for administrator access, the `start` command can be used.

`release database [database ...]`

Brings back a database from maintenance mode. A released database is available again for normal use by any client, and is started on demand. Use the `lock` command to take a database under maintenance.

`status [-lc] [-s states] [database ...]`

Shows the state of the given database, or, when none given, all known databases. Three modes control the level of detail in the displayed output. By default a condensed one-line output per database format is used. This output resembles pretty much the output of various `xxxstat` programs, and is ideal for quickly gaining an overview of the system state.

`-c`

The `-c` flag shows the most used properties of a database. This includes the state of the database (running, crashed, stopped), whether it is under maintenance or not, the crash averages and uptime statistics. The crash average is the number of times the database has crashed over the last 1, 15 or 30 starts. The lower the average, the healthier the database is.

`-l`

Triggered by the `-l` flag, a long listing is used. This listing spans many rows with on each row one property and its value separated by a colon (':'). The long listing includes all information that is available.

`-s`

The `-s` flag controls which databases are being shown, matching their state. The required argument to this flag can be a combination of any of the following characters. Note that the order in which they are put also controls the order in which the databases are printed. `r`, `s`, `c` and `l` are used to print a started (running), stopped, crashed and locked database respectively. The default order which is used when the `-s` flag is absent, is `rscl`.

`start [-a] database [database ...]`

`stop [-a] database [database ...]`

`kill [-a] database [database ...]`

Starts, stops or kills the given database, or, when -a is supplied, all known databases. The kill command immediately sends a SIGKILL and should only be used as last resort for a database that doesn't respond any more. Killing a database may result in (partial) data loss. It is more common to use the stop command to stop a database. It will first attempt to stop the database, waiting for mero_exittimeout seconds and if that fails, kill the database. When using the start command, monetdb(1) will output diagnostic messages if the requested action failed. When encountering an error, one should always consult the logfile of monetdbd(1) for more details. For the kill command a diagnostic message indicating the database has crashed is always emitted, due to the nature of that command. Note that in combination with -a the return code of monetdb(1) indicates failure if one of the databases had a failure, even though the operation on other databases was successful.

get <all | property[,property[,..]]> [database ...]

Prints the requested properties, or all known properties, for the given database. For each property its source and value are printed. Source indicates where the current value comes from, e.g. the configuration file, or a local override.

set property=value database [database ...]

Sets property to value for the given database. For a list of properties, run monetdb get all. Most properties require the database to be stopped when set.

shared=<yes|no|tag>

Defines if and how the database is being announced to other monetdbds or not. If not set to yes or no the database is simply announced or not. Using a string, called tag the database is shared using that tag, allowing for more sophisticated usage. For information about the tag format and use, see section REMOTE DATABASES in the monetdbd(1) manpage. Note that this property can be set for a running database, and that a change takes immediate effect in the network.

nthreads=<number>

Defines how many worker threads the server should use to perform main processing. Normally, this number equals the number of available CPU cores in the system. Reducing this number forces the server to use less parallelism when executing queries, or none at all if set to 1.

optpipe=<string>

Each server operates with a given optimiser pipeline. While the default usually is the best setting, for some experimental uses the pipeline can be changed. See the mserver5(1) manpage for available pipelines. Changing this setting is discouraged at all times.

readonly=<yes|no>

Defines if the database has to be started in read-only mode. Updates are rejected in this mode, and the server employs some read-only optimisations that can lead to improved performance.

nclients=<number>

Sets the maximum amount of clients that can connect to this database at the same time. Setting this to a high value is discouraged. A multiplex-funnel may be more performant, see MULTIPLEX-FUNNEL below.

inherit property database [database ...]

Like set, but unsets the database-local value, and reverts to inherit from the default again.

discover [expression]

Returns a list of remote monetdbds and database URIs that were discovered by monetdbd(1). All databases listed can be connected to via the local MonetDB Database

Server as if it were local databases using their database name. The connection is redirected or proxied based on configuration settings. If expression is given, only those discovered databases are returned for which their URI matches the expression. The expression syntax is described in the section EXPRESSIONS. Next to database URIs the hostnames and ports for `monetdbd`s that allow to be controlled remotely can be found in the discover list masked with an asterisk. These entries can easily be filtered out using an expression (e.g. "*mapi:monetdb:**") if desired. The control entries come in handy when one wants to get an overview of available `monetdbd`s in e.g. a local cluster. Note that for `monetdbd` to announce its control port, the `mero_controlport` setting for that `monetdbd` must be enabled in the configuration file.

`-h`

`help [command]`

Shows general help, or short help for a given command.

`-v`

`version`

Shows the version of the `monetdb` utility.

EXPRESSIONS

For various options, typically database names, expressions can be used. These expressions are limited shell-globbing like, where the * in any position is expanded to an arbitrary string. The * can occur multiple times in the expression, allowing for more advanced matches. Note that the empty string also matches the *, hence "*de*mo*" can return "*demo*" as match. To match the literal '*' character, one has to escape it using a backslash, e.g. "*".

RETURN VALUE

The `monetdb` utility returns exit code 0 if it successfully performed the requested command. An error caused by user input or database state is indicated by exit code 1. If an internal error in the utility occurs, exit code 2 is returned.

SEE ALSO

`monetdbd`(1) `mserver5`(1)

Appendix III The `mserver5` man-page

NAME

`mserver5` − the MonetDB server version 5

DESCRIPTION

`mserver5` is the current MonetDB server that performs all processing on request of clients for a certain database.

Note that while `mserver5` is the process that does the actual work, it is usually more common to start, monitor and connect to the `mserver5` process through `monetdbd`(1).

This man-page describes the options that `mserver5` understands. It is intended for people who really need to work with mserver5 itself. In regular cases, the programs `monetdbd`(1) and `monetdb`(1) control the many options, and allow to adjust them to appropriate values where sensible. For normal usage, it is preferred to apply any configuration through these programs.

OPERATION

When the build-time configuration did not disable this, an `mserver5` process presents the user with a console prompt. On this prompt, MAL commands can be executed. The architecture is setup to handle multiple streams of requests. The first thread started represents the server, which is the console prompt, reading from standard input and writing to standard output.

The server thread started remains in existence until all other threads die. The server is stopped by *Ctrl-D* on its console, typing `\q` or by sending it a termination signal (`SIGINT`, `SIGTERM`, `SIGQUIT`).

MSERVER5 OPTIONS

`mserver5` can be started with options and scripts as arguments. The MAL scripts will be executed directly after startup on the console, which eases e.g. testing of MAL scripts directly, without starting a client.

`--dbfarm=<path>`

Path where `mserver5` should find a database. Shorthand for option `gdk_dbfarm`. Default value: `@localstatedir@/monetdb5/dbfarm`.

`--dbname=<name>`

Database name to start, within the dbfarm. Shorthand for option `gdk_dbname`. Default value: `demo`.

`--dbinit=<stmt>`

MAL statement to execute as part of the startup of the server.

`--config=<file>`

Config file to read options from. This file can contain all options as can be set with the `--set` flag. See CONFIG FILE FORMAT.

`--daemon=<yes|no>`

Disable the console prompt, do not read commands from standard input. Default: `no`

`--set <option>=<value>`

Set individual configuration option. For possible options, see PARAMETERS sections.

`--help`

Print list of options.

`--version`

Print version and compile configuration.

GDK PARAMETERS

GDK (Goblin Database Kernel) is the current columnar storage kernel engine of the MonetDB 5 database. It is the component that manages and performs operations on BATs (Binary Association Tables), single columns. The parameters here affect the behaviour of GDK which may nagatively impact performance if set wrongly. The kernel tries to choose the values for optimal performance. Changing these parameters is discouraged.

`gdk_mem_bigsize`

Memory chunks of size >= `gdk_mem_bigsize` (in bytes) will be mmaped anonymously. Default: `1<<20 == 1024576 == 1 MiB`

`gdk_vmtrim`

Enable or disable the `vmtrim` thread which tries to unload memory that is not in use. Default: `yes`

`gdk_alloc_map`

This parameter is mainly for debugging with `valgrind`. Also, you need to tell `valgrind` to use 8-byte alignment, hence: "`valgrind --alignment=8 mserver5 --set gdk_alloc_map=no ...`" This feature should better be disabled for normal use. Default: `no`

`gdk_debug`

You can enable debug output for specific kernel operations. By default debug is switched off for obvious reasons. The value of `gdk_debug` is an integer, which value can be (a combination of):

```
      1 = thread-specific debug output
      2 = CHECKMASK    = property enforcing on new BATs
      4 = MEMMASK      = memory allocation
      8 = PROPMASK     = property checking on all values:
                         tells about wrongly set properties
     16 = IOMASK       = major IO activity
     32 = BATMASK      = BAT handling
     64 = PARSEMASK    = parser debugging
    128 = PARMASK      = Thread management
    256 = TRGMASK      = Event management
    512 = TMMASK       = Transaction management
   1024 = TEMMASK      = Locks and Triggers
   2048 = DLMASK       = Dynamic loading
   4096 = PERFMASK     = BBP Performance (?)
   8192 = DELTAMASK    = Delta debugging (?)
  16384 = LOADMASK     = Module loading
  32768 = YACCMASK     = Yacc specific error messages
2097152 = ALGOMASK     = show join/select algorithm chosen
4194304 = ESTIMASK     = show result size estimations
                         (for join, select)
```

```
   8388608 = XPROPMASK     = extended property checking:
                             tells also about not set properties
  16777216 = JOINPROPMASK  = disable property checking with
                             join & outerjoin (e.g., for
                             performance measurements)
  33554432 = DEADBEEFMASK  = disable "cleaning" of freed memory
                             in GDKfree() (e.g., for performance
                             measurements)
  67108864 = ALLOCMASK     = exhaustive GDK malloc & free tracing
                             for debugging (GDK developers, only)
 134217728 = OPTMASK       = trace the actions, decisions and
                             effects of MAL optimizers
 268435456 = EXTENDMASK    = trace/debug HEAPextend;
                             used only for development & debugging
536870912 = FORCEMITOMASK = forcefully activate mitosis even on
                             small tables, i.e., split small tables
                             in as many (tiny) pieces as there are
                             cores (threads) available;
                             this allows us to test mitosis
                             functionality without requiring large
                             data sets (--- at the expense of a
                             potentially significant interpretation
                             overhead for unnecessary large plans);
                             used only for development & testing;
                             set automatically by Mtest.py
```

Note that `mserver5` recognizes a series of command line options that sets one or more of these debug flags as well:

```
--threads       (1 | PARMASK)
--memory        (MEMMASK)
--properties    (CHECKMASK | PROPMASK | BATMASK)
--io            (IOMASK | PERFMASK)
--transactions  (TMMASK | DELTAMASK | TEMMASK)
--modules       (DLMASK | LOADMASK)
--algorithms    (ALGOMASK | ESTIMASK)
--xproperties   (XPROPMASK )
--performance   (JOINPROPMASK | DEADBEEFMASK)
--optimizers    (OPTMASK)
--forcemito     (FORCEMITOMASK)
```

Default: 0

MSERVER5 PARAMETERS

mserver5 instructs the GDK kernel through the MAL (MonetDB Assembler Language) language. MonetDB 5 contains an extensive optimiser framework to transform MAL plans into more optimal or functional (e.g. distributed) plans. These parameters control behaviour on the MAL level.

mal_listing

You can enable the server listing the parsed MAL program for any script parsed on the command line. The value of mal_listing is an integer that has the following possible values:

```
 0 = Disable
 1 = List the original input
 2 = List the MAL instruction
 4 = List the MAL type information
 8 = List the MAL UDF type
16 = List the MAL properties
32 = List the hidden details
64 = List the bat tuple count
```

Default: 0

monet_vault_key

The authorisation tables inside mserver5 can be encrypted with a key, such that reading the BATs does not directly disclose any credentials. The monet_vault_key setting points to a file that stores a secret key to unlock the password vault. It can contain anything. The file is read up to the first null-byte (‘ ’), hence it can be padded to any length with trailing null-bytes to obfuscate the key length. Generating a key can be done for example by using a tool such as pwgen and adding a few of the passwords generated. Make sure not to choose a too small key. Note that on absence of a vault key file, some default key is used to encrypt the authorisation tables. Changing this setting (effectively changing the key) for an existing database makes that database unusable, as no one is any longer able to login. If you use monetdbd(1), a per-database vault key is set.

max_clients

Controls how many client slots are allocated for clients to connect. This setting limits the maximum number of connected clients at the same time. Note that MonetDB is not designed to handle massive amounts of connected clients. The funnel capability from monetdbd(1) might be a more suitable solution for such workloads.

Default 64.

SQL PARAMETERS

The SQL component of MonetDB 5 runs on top of the MAL environment. It has its own SQL-level specific settings.

sql_debug

Enable debugging using a mask. This option should normally be disabled (0). Default: 0

sql_optimizer

The default SQL optimizer pipeline can be set per server. See the optpipe setting in monetdb(1) when using monetdbd. During SQL initialization, the optimizer pipeline is checked against the dependency information maintained in the optimizer library to ensure there are no conflicts and at least the pre-requisite optimizers are used. The setting of sql_optimizer can be either the list of optimizers to run, or one or more variables

containing the optimizer pipeline to run. The latter is provided for readability purposes only. Default: `default_pipe`

The following are possible pipes to use:

`minimal_pipe`

The minimal pipeline necessary by the server to operate correctly.

```
minimal_pipe=inline,remap,deadcode,multiplex,garbageCollector
```

`default_pipe`

The default pipeline contains as of Feb2010 mitosis-mergetable-reorder, aimed at large tables and improved access locality.

```
default_pipe=inline,remap,evaluate,costModel,coercions,emptySet,aliases,mitosis,merge
table,deadcode,commonTerms,joinPath,reorder,deadcode,reduce,dataflow,history,multiple
x,garbageCollector
```

`no_mitosis_pipe`

The `no_mitosis` pipeline is identical to the default pipeline, except that optimizer mitosis is omitted. It is used mainly to make some tests work deterministically, and to check/debug whether "unexpected" problems are related to mitosis (and/or mergetable).

```
no_mitosis_pipe=inline,remap,evaluate,costModel,coercions,emptySet,aliases,mergetable
,deadcode,commonTerms,joinPath,reorder,deadcode,reduce,dataflow,history,multiplex,gar
bageCollector
```

`sequential_pipe`

The sequential pipeline is identical to the default pipeline, except that optimizers mitosis & dataflow are omitted. It is use mainly to make some tests work deterministically, i.e., avoid ambigious output, by avoiding parallelism.

```
sequential_pipe=inline,remap,evaluate,costModel,coercions,emptySet,aliases,mergetable
,deadcode,commonTerms,joinPath,reorder,deadcode,reduce,history,multiplex,garbageColle
ctor
```

`nov2009_pipe`

The default pipeline used in the November 2009 release.

```
nov2009_pipe=inline,remap,evaluate,costModel,coercions,emptySet,aliases,mergetable,de
adcode,constants,commonTerms,joinPath,deadcode,reduce,dataflow,history,multiplex,garb
ageCollector
```

Debugging the optimizer pipeline: the best way is to use `mdb` (MAL debugger) and inspect the information gathered during the optimization phase. Several optimizers produce more intermediate information, which may shed light on the details. The `opt_debug` bitvector controls their output. It can be set to a pipeline or a comma-separated list of optimizers you would like to trace. It is a server wide property and cannot be set dynamically, as it is intended for internal use.

CONFIG FILE FORMAT

The conf-file readable by mserver5 consists of parameters of the form "*name=value*".

The file is line-based, each newline-terminated line represents either a comment or a parameter.

Only the first equals sign in a parameter is significant. Whitespace before or after the first equals sign is not stripped. Trailing whitespace in a parameter value is retained verbatim.

Any line beginning with a hash (#) is ignored, as are lines containing only whitespace.

The values following the equals sign in parameters are all a string where quotes are not needed, and if written be part of the string.

SEE ALSO

`monetdbd`(1), `monetdb`(1), `mclient`(1)

Appendix IV The `mclient` man-page

NAME

`mclient` – the MonetDB command-line tool

SYNOPSIS

```
mclient [ options ] [ file or database [ file ... ] ]
mclient --help
```

DESCRIPTION

MonetDB is a database management system that is developed from a main-memory perspective with use of a fully decomposed storage model, automatic index management, extensibility of data types and search accelerators, SQL- and XML- frontends.

`mclient` is the command-line interface to the MonetDB server.

If the `--statement=query (-s query)` option is given, the query is executed. If any files are listed after the options, queries are read from the files and executed. The special filename - refers to standard input. Note that if there is both a `--statement` option and filename arguments, the query given with `--statement` is executed first. If no `--statement` option is given and no files are specified on the command line, `mclient` reads queries from standard input.

When reading from standard input, if standard input is a terminal or if the `--interactive (-i)` option is given, `mclient` interprets lines starting with '\' (backslash) specially. See the section BACKSLASH COMMANDS below.

Before `mclient` starts parsing command line options, it reads a .monetdb file. If the environment variable `DOTMONETDBFILE` is set, it reads the file pointed to by that variable instead. When unset, `mclient` searches for a *.monetdb* file in the current working directory, and if that doesn't exist, in the current user's home directory. This file can contain defaults for the flags `user`, `password`, `language`, `save_history`, `format` and `width`. For example, an entry in a *.monetdb* file that sets the default language for `mclient` to `mal` looks like this: `language=mal`. To disable reading the *.monetdb* file, set the variable `DOTMONETDBFILE` to the empty string in the environment.

OPTIONS

General Options

`--help (-?)`

Print usage information and exit.

`--encoding=`*`encoding`* `(-E` *`encoding`*`)`

Specify the character encoding of the input. The option applies to both the standard input of `mclient` and to the argument of the `--statement (-s)` option but not to the contents of files specified on the command line (except for - which refers to standard input) or files specified using the `\<` command (those must be encoded using UTF-8). The default encoding is taken from the locale.

`--language=`*`language`* `(-l` *`language`*`)`

Specify the query language. The following languages are recognized: `mal` and `sql`. A unique prefix suffices. When the `--language` option is omitted, the default of `sql` is assumed.

`--database=`*`database`* `(-d` *`database`*`)`

Specify the name of the database to connect to. The `-d` can be omitted if an equally named file does not exist in the current directory. As such, the first non-option argument will be interpreted as database to connect to if the argument does not exist as file.

`--host=`*`hostname`* `(-h` *`hostname`*`)`

Specify the name of the host on which the server runs (default: `localhost`). When the argument starts with a forward slash (`/`), host is assumed to be the directory where the UNIX sockets are stored for platforms where these are supported.

`--port=`*`portnr`* `(-p` *`portnr`*`)`

Specify the portnumber of the server (default: 50000).

`--interactive[=`*`timermode`*`] (-i[`*`timermode`*`])`

When reading from standard input, interpret lines starting with `\` (backslash) specially. See the section BACKSLASH COMMANDS below. This is the default if standard input is a terminal. The optional *`timermode`* argument controls the format of the time reported for queries. Note that no space is allowed between `-i` and *`timermode`*. The default mode is `human` which adjusts the time precision to the measured value. The modes `ms`, `s` and `m` force millisecond, second and minute + second precision respectively.

`--user=`*`user`* `(-u` *`user`*`)`

Specify the user to connect as. If this flag is absent, the client will ask for a user name, unless a default was found in *.monetdb* file. Note that user must follow immediately after the option.

`--format=`*`format`* `(-f` *`format`*`)`

Specify the output format. The possible values are `sql`, `csv`, `tab`, `raw`, and `xml`. `csv` is comma-separated values, `tab` is tab-separated values, `raw` is no special formatting (data is dumped the way the server sends it to the client), `sql` is a pretty format which is meant for human consumption, and `xml` is a valid (in the XML sense) document. In addition to plain `csv`, two other forms are possible. `csv=c` uses `c` as column separator; `csv+c` uses `c` as column separator and produces a single header line in addition to the data.

`--echo (-e)`

Echo the query. Note that using this option slows down processing.

`--history (-H)`

Load and save the command line history (default off).

`--log=`*`logfile`* `(-L` *`logfile`*`)`

Save client/server interaction in the specified file.

`--statement=`*`stmt`* `(-s` *`stmt`*`)`

Execute the specified query. The query is run before any queries from files specified on the command line are run.

`--timezone (-z)`

Do not tell the client's timezone to the server.

`--Xdebug (-X)`

Trace network interaction between `mclient` and the server.

`--pager=`*`cmd`* `(-|` *`cmd`*`)`

Send query output through the specified `cmd`. One `cmd` is started for each query. Note that the `|` will have to be quoted or else the shell will interpret it.

SQL Options

`--null=`*`nullstr`* `(-n` *`nullstr`*`)`

Set the string to be used as NULL representation when using the `sql`, `csv`, or `tab` output formats. If not used, NULL values are represented by the string "null" in the `sql` output format, and as the empty string in the `csv` and `tab` output formats. Note that an argument is required, so in order to use the empty string, use `-n ""` (with the space) or `--null=`.

`--autocommit (-a)`

Switch `autocommit` mode off. By default, `autocommit` mode is on.

`--rows=`*`nr`* `(-r` *`nr`*`)`

If specified, query results will be paged by an internal pager at the specified number of lines.

`--width=`*`nr`* `(-w` *`nr`*`)`

Specify the width of the screen. The default is the (initial) width of the terminal.

`--dump (-D)`

Create an SQL dump.

`--inserts (-N)`

Use `INSERT INTO` statements instead of `COPY INTO + CSV` values when dumping the data of a table. This option can be used when trying to load data from MonetDB into another database, or when e.g. JDBC applications are used to reload the dump.

BACKSLASH COMMANDS

General Commands

`\?`

Show a help message explaining the backslash commands.

`\q`

Exit `mclient`.

`\<` *`file`*

Read input from the named *`file`*.

`\>` *`file`*

Write output to the named *`file`*. If no *`file`* is specified, write to standard output.

`\|` `command`

Pipe output to the given command. Each query is piped to a new invocation of the command. If no command is given, revert to writing output to standard output.

`\h`

Show the `readline`(3) history.

`\L` *`file`*

Log client/server interaction in the given *`file`*. If no *`file`* is specified, stop logging information.

`\X`

Trace what `mclient` is doing. This is mostly for debugging purposes.

`\e`

Echo the query in SQL formatting mode.

`\f` *`format`*

Use the specified *format* mode to format the output. Possible modes the same as for the `--format (-f)` option.

`\w width`

Set the maximum page width for rendering in the sql formatting mode. If width is −1, the page width is unlimited, when width is 0, use the terminal width. If width is greater than 0, use the given width.

`\r rows`

Use an internal pager using rows per page. If rows is −1, stop using the internal pager.

SQL Commands

`\D`

Dump the complete database. This is equivalent to using the program `msqldump`(1).

`\D` *table*

Dump the given *table*.

`\d`

Alias for `\dvt`.

`\d[Stvsfn]+`

List database objects of the given type. Multiple type specifiers can be used at the same time. The specifiers `S`, `t`, `v`, `s`, `f` and `n` stand for System, table, view, sequence, function and schema respectively. Note that `S` simply switches on viewing system catalog objects, which is orthogonal on the other specifiers.

`\d[Stvsfn]+` *object*

Describe the given *object* in the database using SQL statements that reconstruct the object. The same specifiers as above can be used, following the same rules. When no specifiers are given, `vt` is assumed. The object can be given with or without a schema, separated by a dot. The object name can contain the wildcard characters `*` and `_` that represent zero or more, and exactly one character respectively. An object name is converted to lowercase, unless the object name is quoted by double quotes (`"`). Examples of this, are e.g. ***.mytable*, *tabletype*** or "*myschema.FOO*". Note that wildcard characters do not work in quoted objects. Quoting follows SQL quoting rules. Arbitrary parts can be quoted, and two quotes following each other in a quoted string represent the quote itself.

`\A`

Enable auto commit mode.

`\a`

Disable auto commit mode.

EXAMPLES

Efficiently import data from a CSV (comma-separated values) file into a table. The file must be readable by the server. *$file* is the absolute path name of the file, $table is the name of the table, *$db* is the name of the database.

```
mclient -d $db -s "COPY INTO $table FROM '$file' USING DELIMITERS ',','\\n','\"'"
```

Efficiently import data from a CSV file into a table when the file is to be read by mclient (e.g. the server has no access to the file). *$file* is the (absolute or relative) path name of the file, *$table* is the name of the table, *$db* is the name of the database.

```
mclient -d $db -s "COPY INTO $table FROM STDIN USING DELIMITERS ',','\\n','\"'" - < $file
```

Note that in this latter case, if a count of records is supplied, it should be at least as large as the number of records actually present in the CSV file. This, because otherwise the remainder of the file will be interpreted as SQL queries.

See http://www.monetdb.org/Documentation/Manuals/SQLreference/CopyInto for more information about the `COPY INTO` query.

SEE ALSO

`msqldump`(1)