NATIONAL TECHNICAL UNIVERSITY OF ATHENS
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING
ΤΟΜΈΑΣ ΤΕΧΝΟΛΟΓΊΑΣ ΠΛΗΡΟΦΟΡΙΚΉΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Datix-realtime

DIPLOMA PROJECT

ΓΙΩΡΓΟΣ ΤΟΥΛΟΎΠΑΣ

Supervisor : Νικόλαος Σ. Παπασπύρου
Αν. Καθηγητής Ε.Μ.Π.

Athens, October 2015

NATIONAL TECHNICAL UNIVERSITY OF ATHENS
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING
ΤΟΜΈΑΣ ΤΕΧΝΟΛΟΓΊΑΣ ΠΛΗΡΟΦΟΡΙΚΉΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Datix-realtime

DIPLOMA PROJECT

ΓΙΩΡΓΟΣ ΤΟΥΛΟΎΠΑΣ

Supervisor : Νικόλαος Σ. Παπασπύρου
Αν. Καθηγητής Ε.Μ.Π.

Approved by the examining committee on the October 17, 2015.

Νικόλαος Σ. Παπασπύρου Αν. Καθηγητής Ε.Μ.Π.	Πέτρος Χ. Παπαδόπουλος Επίκ. Καθηγητής Ε.Μ.Π.	Γεώργιος Χ. Νικολάου Καθηγητής Ε.Κ.Π.Α.

Athens, October 2015

Γιώργος Τουλούπας
Electrical and Computer Engineer

Περίληψη

Σκοπός της παρούσας εργασίας είναι

Λέξεις κλειδιά

Distributed systems, Storm

Abstract

The purpose of this diploma dissertation is

Key words

Distributed systems, Storm

Ευχαριστίες

Ευχαριστώ θερμά

Γιώργος Τουλούπας,

Αθήνα, 17η Οκτωβρίου 2015

Η εργασία αυτή είναι επίσης διαθέσιμη ως Τεχνική Αναφορά CSD-SW-TR-42-14, Εθνικό Μετσόβιο Πολυτεχνείο, Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών, Εργαστήριο Τεχνολογίας Λογισμικού, Οκτώβριος 2015.

URL: `http://www.softlab.ntua.gr/techrep/`
FTP: `ftp://ftp.softlab.ntua.gr/pub/techrep/`

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

test[Chur32]

1.2 Objectives

1.3 Thesis Outline

Chapter 2

Theoretical Background

2.1 Kafka

Kafka is a distributed, partitioned, replicated commit log service, that provides the functionality of a messaging system. It is used for collecting and delivering high volumes of data with low latency. Apache Kafka was originally developed by LinkedIn, and was subsequently open sourced in 2011. In 2012 Kafka became an Apache Top-Level Project.

The basic concepts of Kafka are the following:

- A *topic* defines a stream of messages of a particular type.
- A *producer* is a process that publishes messages to a topic.
- The published messages are stored at a cluster comprised of servers called *brokers*. All coordination between the brokers is done through a Zookeeper cluster.
- A *consumer* is a process that subscribes to one or more topics and processes the feed of published messages.

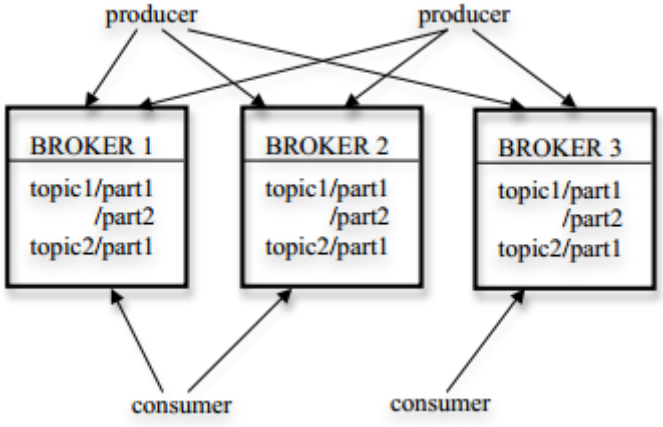

Figure 2.1: Kafka architecture

For each topic, the Kafka cluster maintains a partitioned log with the structure depicted on Figure 2.2. A *partition* is essentially a commit log to which an ordered, immutable sequence of messages that is continually appended. Every message is assigned an offset: a sequential id number that uniquely identifies the message within the partition. Kafka only provides a total order over messages within a partition, not between different partitions in a topic.

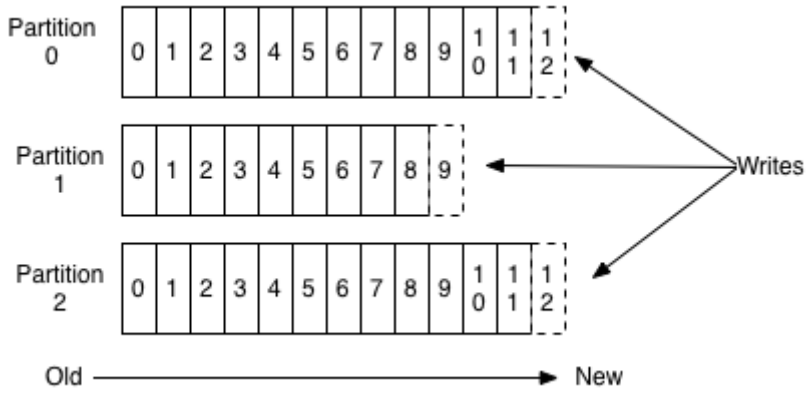

Figure 2.2: Kafka topic structure

All published messages remain stored at the brokers for a configurable period of time, whether or not they have been consumed. Kafka's performance is effectively constant with respect to data size, allowing a big volume of data to be retained. The only metadata retained for each consumer is the offset of the consumer in the log. By controlling this offset the consumer can read messages in any order. For example a consumer can advance its offset linearly as it reads messages or even reset to an older offset to reprocess them.

The partitions in the log serve several purposes. Firstly, they allow the log to scale in size, by being distributed over the brokers of the cluster. Moreover, the partitions are replicated across a configurable number of brokers to provide fault tolerance. For each partition one broker acts as the ”leader”, handling all the requests for the partition, and zero or more brokers act as ”followers”, replicating the leader. Finally, partitions act as the unit of parallelism and provide load balancing over the write and read requests of the producers and the consumers respectively.

2.2 Storm

2.2.1 Introduction

Storm is a real-time fault-tolerant and distributed stream data processing system. It was originally created by BackType and was subsequently open sourced after being acquired by Twitter in 2011. Storm is an Apache Top-Level Project since 2014. The basic Storm data processing architecture consists of streams of tuples flowing through topologies. A topology is a directed graph where the vertices represent computation and the edges represent the data flow between the computation components. Vertices are divided into spouts and bolts, that define information sources and manipulations respectively.

Storm demonstrates the following key properties:

- **Scalable:** Storm topologies are inherently parallel and run across a cluster of machines. Different parts of a topology can be scaled individually by tweaking their parallelism. Moreover, nodes can be added or removed from the Storm cluster without disrupting the existing topologies.
- **Resilient:** Storm is designed to be fault-tolerant. If there are faults or failures during the execution of a topology, Storm will reassign tasks as necessary.
- **Efficient:** Storm must have good performance characteristics, since it is used in real-time applications. To achieve this Storm uses a number of techniques, including keeping all its storage and computational data structures in memory.
- **Reliable:** Storm guarantees every tuple will be fully processed by tracking the lineage of every tuple as it advances through the topology.

- **Easy to monitor:** Storm provides easy-to-use administration tools that help end-users immediately notice if there are failure or performance issues associated with Storm.

2.2.2 Storm Architecture

A Storm cluster consists of one master node and one or more worker nodes. The *master node* runs the *Nimbus* daemon that is responsible for distributing code around the cluster, assigning tasks to machines, and monitoring for failures.

Every *worker node* runs a *Supervisor* daemon that listens for work assigned to its machine and starts and stops *worker processes* as necessary based on what Nimbus has assigned to it. Each worker process executes a subset of a topology.

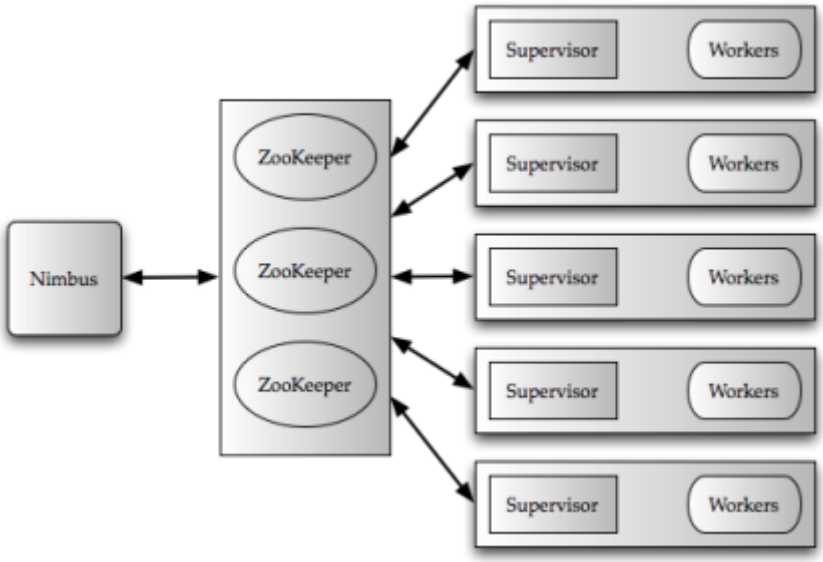

Figure 2.3: Storm architecture

All coordination between Nimbus and the Supervisors is done through a Zookeeper cluster. Additionally, the Nimbus daemon and Supervisor daemons are fail-fast and stateless, because all state is kept in Zookeeper or on local disk. This design leads to Storm clusters being incredibly stable, allowing the cluster to recover even if Nimbus or the Supervisors are killed and restarted afterwards.

2.2.3 Topologies

The core abstraction in Storm is the *stream*, an unbounded sequence of tuples. A *tuple* is a named list of values, and a field in a tuple can be an object of any type. The basic primitives Storm provides for doing stream transformations are spouts and bolts.

A *spout* is a source of streams in a computation. Usually a spout reads from a queueing broker such as Kafka, but a spout can also generate its own stream or read from a streaming API.

A *bolt* consumes any number of input streams, does some processing, and possibly emits new streams. Most of the logic of a computation goes into bolts, such as functions, filters, streaming joins, streaming aggregations, databases queries, etc.

Networks of spouts and bolts are packaged into a topology which is the top-level abstraction is submitted to Storm clusters for execution. A *topology* is a graph of stream transformations where each vertex is a spout or bolt. Edges in the graph indicate which bolts are subscribing to which streams. Topologies run indefinitely when deployed.

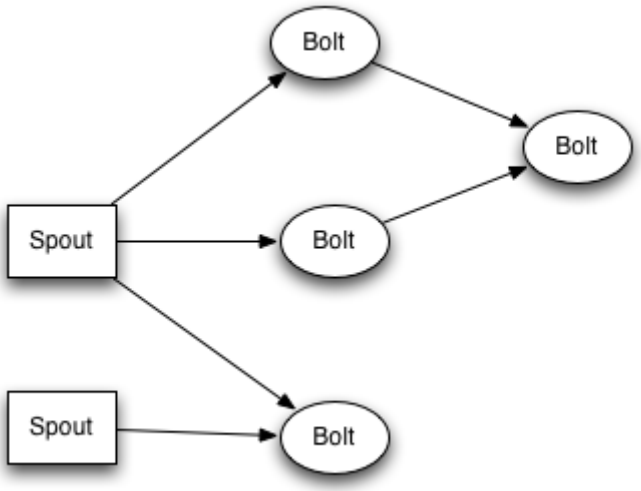

Figure 2.4: Example storm topology

Each component (spout or bolt) in a Storm topology executes in parallel. The degree of parallelism for each component can be configured and Storm will spawn that number of threads across the cluster to do the execution.

2.2.4 Parallelism in Storm

There are three main entities that are used to actually run a topology in a Storm cluster: worker processes, executors and tasks. The relationships between them are illustrated in Figure 2.5.

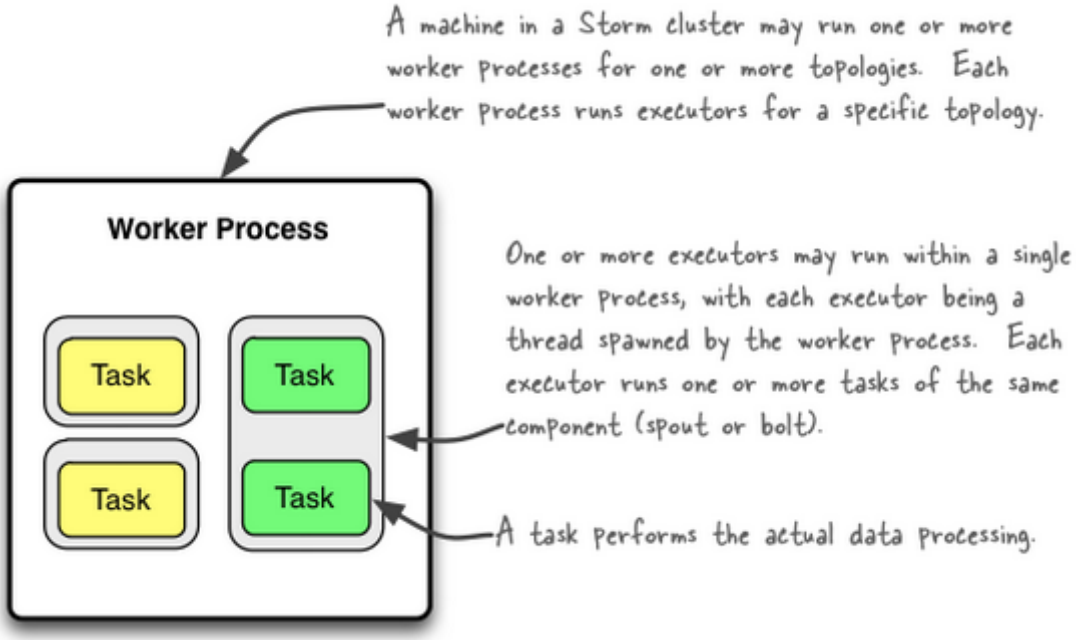

Figure 2.5: The relationships between worker processes, executors and tasks

A *worker process* runs a JVM and executes a subset of a topology. Each worker process belongs to a specific topology and may run one or more executors.

An *executor* is a thread spawned by a worker and may run one or more tasks for the same topology component. All of the tasks belonging to the same executor are run serially, since every executor

always corresponds to one thread.

A *task* performs the actual data processing for a topology component. Each spout or bolt is executed as many tasks across the cluster. The number of tasks for a component is static, in contrast to the number of executors for a component that can be changed after the topology has been started. By default, Storm will run one task per executor.

2.2.5 Stream Groupings

A *stream grouping* defines how a stream between two components (spout to bolt or bolt to bolt) is partitioned among the tasks of each component. For example, the way tuples are emitted between the sets of tasks corresponding to Bolt A and Bolt B in Figure 2.6 is defined by a stream grouping.

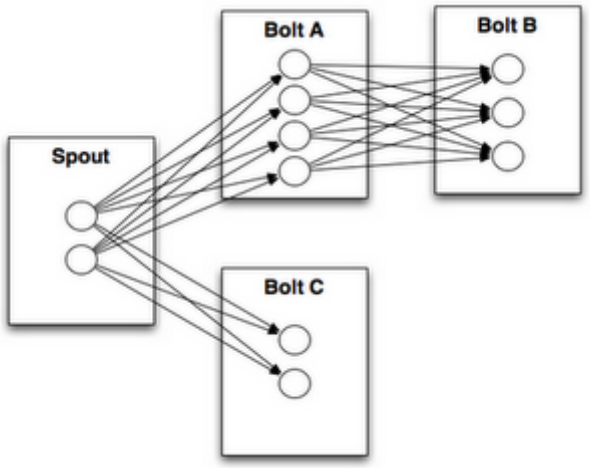

Figure 2.6: Task-level execution of a topology

Storm supports the following stream groupings:

- **Shuffle grouping:** Tuples are randomly and evenly distributed across the bolt's tasks.
- **Fields grouping:** The stream is partitioned by the fields specified in the grouping. This guarantees that tuples with the same values on the specified fields are emitted to the same task.
- **All grouping:** The stream is replicated across all the bolt's tasks.
- **Global grouping:** The entire stream goes to a single one of the bolt's tasks.
- **Local grouping:** If there are one or more of the bolt's tasks in the same worker process, tuples will be shuffled to just those in-process tasks.

2.3 Hadoop Distibuted File System

The Hadoop Distributed File System (HDFS) is a distributed file system designed to run on commodity hardware, inspired by the Google File System. It can reliably store very large files across machines in a large cluster and provides high throughput access to large data sets. HDFS is the storage part of the Hadoop framework, an Apache Top-Level Project since 2006.

Each file on HDFS is stored as a sequence of blocks of the same size, except the last block. Blocks belonging to a file are replicated for fault tolerance. The block size and replication factor are configurable per file.

HDFS has a master/slave architecture. An HDFS cluster consists of a single NameNode and one DataNode per node in the cluster. *NameNode* is a master server that manages the file system

namespace and regulates access to files by clients. Each *DataNode* manages storage attached to the node that it run on. HDFS exposes a file system namespace and allows user data to be stored in files. Every file is split into blocks that are stored in a set of DataNodes. The NameNode determines the mapping of blocks to DataNodes and executes file system namespace operations like opening, closing, and renaming files and directories. The DataNodes are responsible for serving read and write requests from the file system's clients and also perform block creation, deletion, and replication upon instruction from the NameNode.

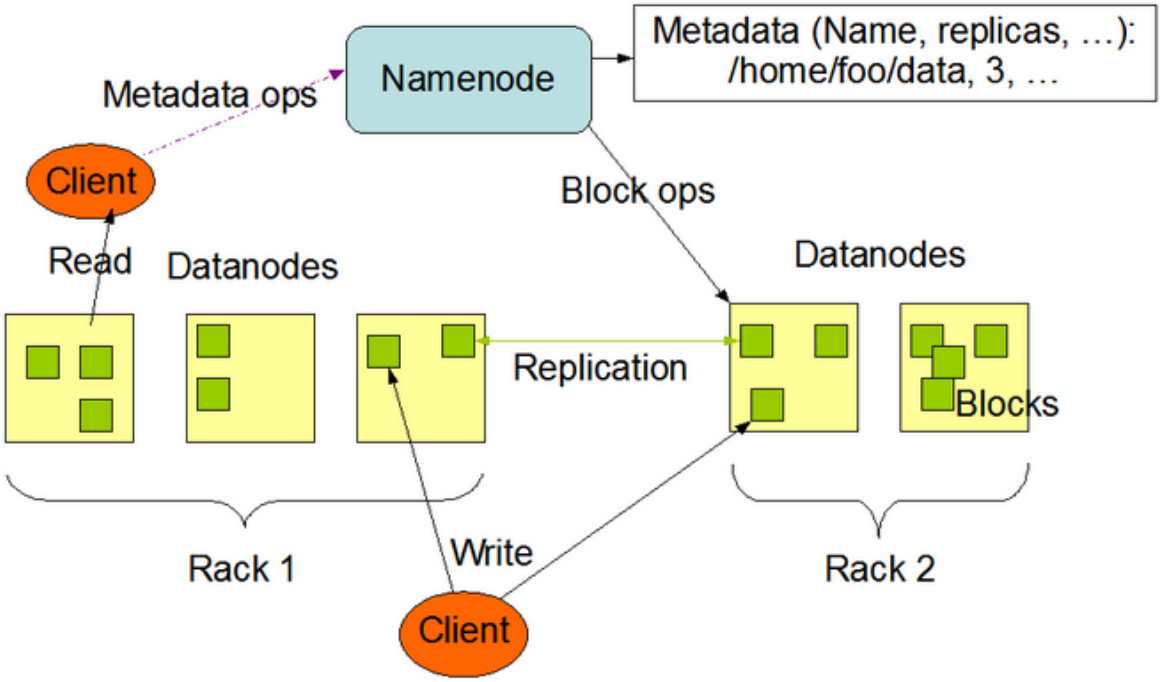

Figure 2.7: HDFS architecture

2.4 HBase

2.4.1 Introduction

HBase is a distributed non-relational database modeled after Google's BigTable, that runs on top of the HDFS. It provides a fault-tolerant way of storing large quantities of sparse data, while allowing random, real-time access to them. HBase is an Apache Top-Level Project since 2010.

HBase offers the following key features:

- **Linear and modular scalability:** HBase clusters expand by adding RegionServers that are hosted on commodity class servers, increasing storage and as well as processing capacity.
- **Strictly consistent reads and writes:** HBase guarantees that all writes happen in an order, and all reads are seeing the most recent committed data.
- **Automatic sharding of tables:** HBase tables are distributed on the cluster via regions, and regions are automatically split and re-distributed as data grows.
- **Automatic failover support between RegionServers:** If a RegionServers fails, the regions it was hosting are reassigned between the available RegionServers.

- **Integration with Hadoop MapReduce:** HBase supports massively parallelized processing via MapReduce for using HBase as both source and sink.
- **Block Cache and Bloom Filters:** HBase supports a Block Cache and Bloom Filters for real-time queries.

2.4.2 HBase Data Model

The data model of HBase is very different from that of relational databases. As described in the Bigtable paper, it is a sparse, distributed, persistent multidimensional sorted map. The map is indexed by a row key, column key, and a timestamp.

$$(rowkey, column, timestamp) \rightarrow value$$

The basic elements of the HBase data model and the relations between them are presented below:

- **Table:** HBase organizes data into tables.
- **Row:** Within a table, data is stored according to its row. A row consists of a row key and one or more columns with values associated with them. Rows are identified uniquely and sorted alphabetically by their row key.
- **Column:** A column consists of a column family and a column qualifier, which are delimited by a : (colon) character.
- **Column Family:** Data within a row is grouped by column family. Column families physically colocate a set of columns and their values. Each column family has a set of storage properties. For these reasons, column families must be declared up front at schema definition. Every row in a table has the same column families, though a given row might not store data in all of its families.
- **Column Qualifier:** Data within a column family is addressed via its column qualifier. Though column families are fixed at table creation, column qualifiers are mutable and may differ greatly between rows.
- **Cell:** A combination of row, column family, and column qualifier uniquely identifies a cell. The data stored in a cell is that cell's value.
- **Timestamp:** Values within a cell are versioned. A timestamp is written alongside each value, and is the identifier for a given version of a value.

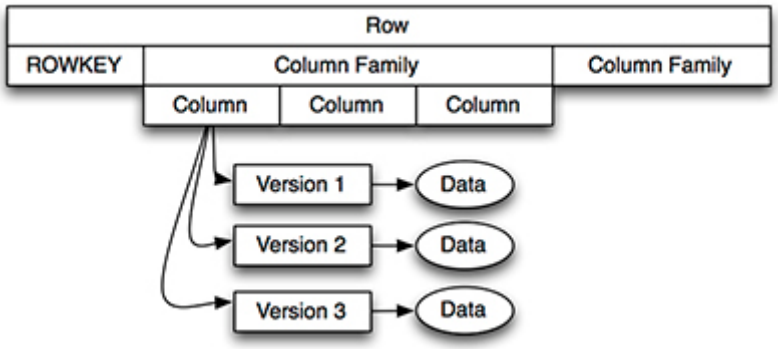

Figure 2.8: HBase data model

There are four primary data model operations in HBase:

- **Get:** Returns the values for a specified row.
- **Put:** Adds a row to a table, if the key is new. If the key already exists, the row is updated.
- **Scan:** Returns the values for a range of rows. Filters can also be used to narrow down the results.
- **Delete:** Marks a row for deletion by adding a Tombstone marker. These rows are cleaned up during the next major compactions of the table.

2.4.3 HBase Architecture

In HBase, tables are are divided horizontally by row key range into *Regions*. Regions are vertically divided by column families into *Stores*, which are saved as files in HDFS (*HFiles*). Figure 2.9 illustrates the architecture of HBase.

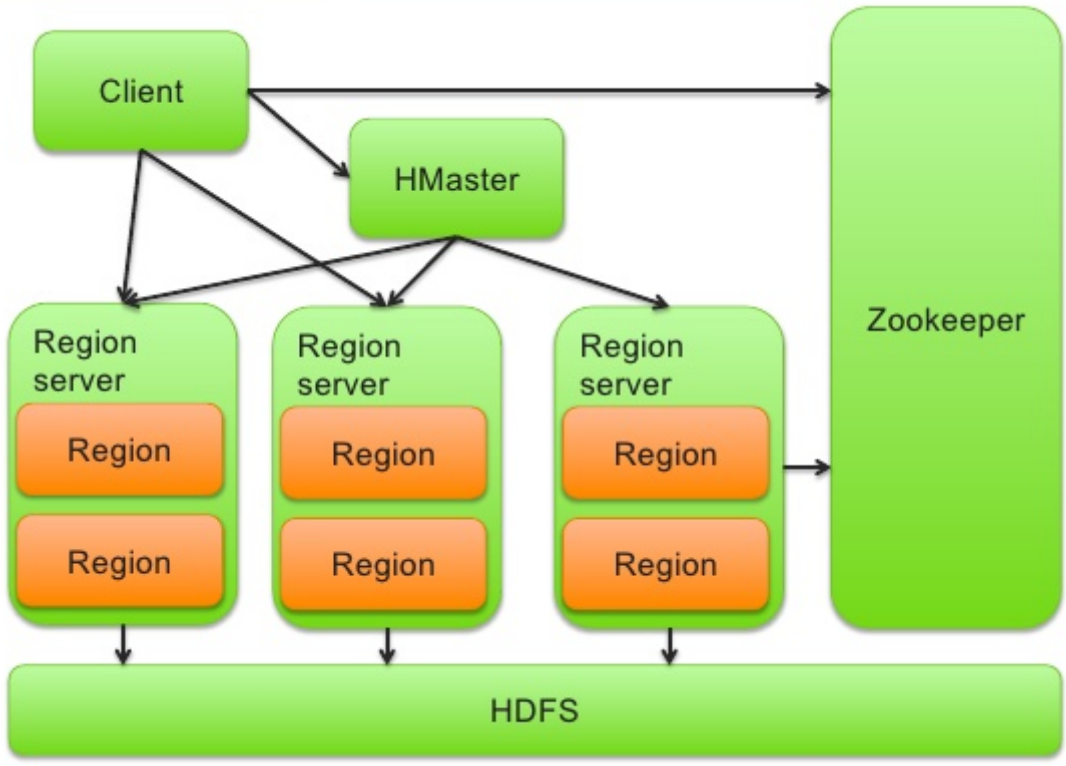

Figure 2.9: HBase architecture

An HBase cluster is composed of two types of servers in a master/slave type architecture. The *HMaster* is responsible for monitoring all RegionServer instances in the cluster, and is the interface for all metadata changes. *RegionServers* are responsible for serving and managing Regions. They are collocated with the HDFS DataNodes, which enables data locality for the data served by the RegionServers. HBase uses ZooKeeper as a distributed coordination service to maintain server state in the cluster. Zookeeper maintains which servers are alive and available, and provides server failure notification.

Every RegionServer has the following components:

- **WAL:** Write Ahead Log is used to store new data that hasn't yet been persisted to permanent storage. The WAL is used for recovery in the case of failure.
- **BlockCache:** Keeps data blocks resident in memory after they are read. Least Recently Used data is evicted when full.

- **MemStore:** Stores in-memory new data which has not yet been written to disk. There is one MemStore per column family per Region. Once the MemStore fills, its contents are written to disk as additional HFiles.
- **Hfile:** Stores the rows as sorted key-values on disk.

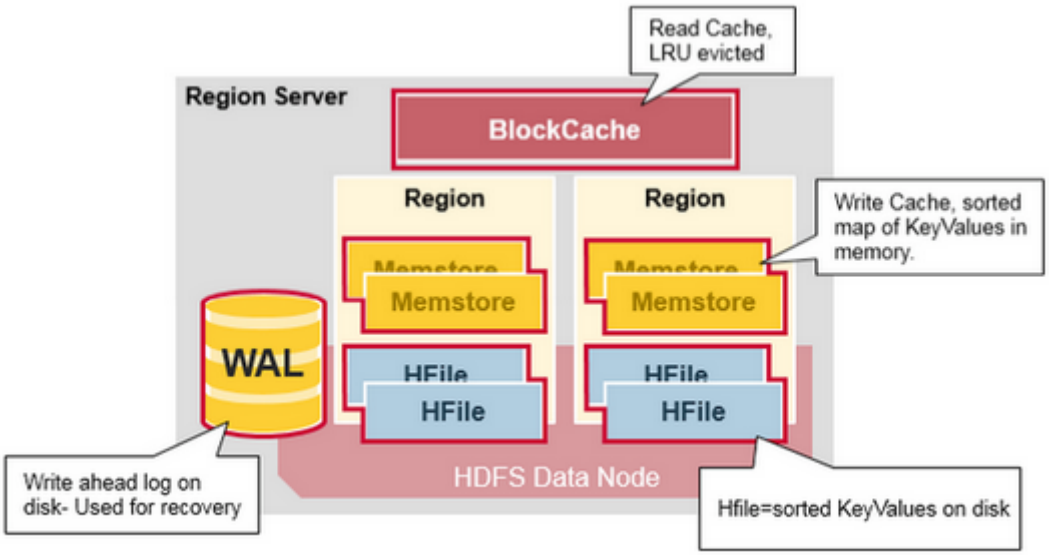

Figure 2.10: RegionServer components

2.5 Phoenix

2.5.1 Introduction

Apache Phoenix is relational database layer for HBase, targeting low latency queries over HBase data. Phoenix provides a JDBC driver that hides the intricacies of Hbase enabling users to create, delete, and alter SQL tables, views, indexes, and sequences, upsert and delete rows singly and in bulk and query data through SQL. Phoenix began as an internal project by the company Salesforce and was subsequently open-sourced and became a top-level Apache project on 2014.

2.5.2 Phoenix Data Model

The relational elements of the Phoenix data model are mapped to their respective counterparts in the HBase data model:

- A Phoenix table is mapped to an HBase table.
- The Phoenix table's columns that are included in the primary key constraint are mapped together to the HBase row key.
- The rest of the columns are mapped to HBase columns, consisting of a column family and a column qualifier.

Columns in a Phoenix table are assigned an SQL datatype. Phoenix serializes data from their datatype to byte arrays when upserting, because HBase stores everything as a byte array. In this way Phoenix allows typed access to HBase data.

2.5.3 Phoenix Architecture

On the client-side Phoenix is a JDBC driver that hides an HBase client from the user. The Phoenix driver compiles queries and other statements into native HBase client calls, enabling the building of low latency applications.

On the server-size a Phoenix jar is installed in every RegionServer, allowing Phoenix to take advantage of coprocessors and custom filters that HBase provides in order to increase performance. Coprocessors perform operations on the server-side thus minimizing client/server data transfer and custom filters prune data as close to the source as possible.

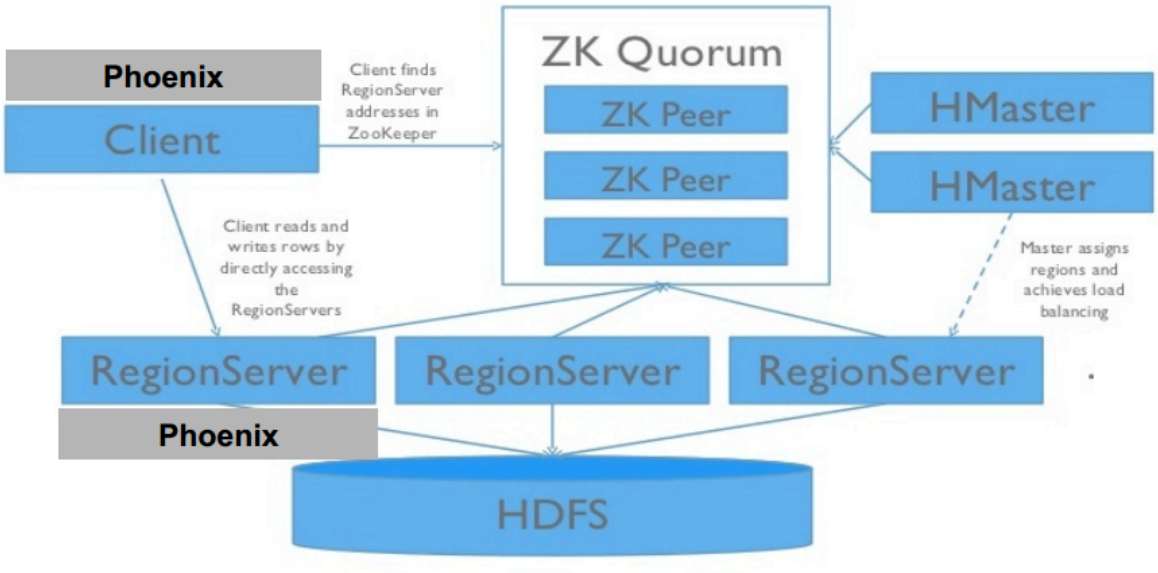

Figure 2.11: Phoenix and HBase architecture

2.5.4 TopN Queries

TopN queries return the top N rows, where top is determined by the ORDER BY clause and N is defined by the LIMIT clause of the SQL query. The execution of this query needs to make a pass through all the rows restricted by the WHERE clause and sort the results of the GROUP BY, which is very computationally expensive for large tables. In order to decrease execution time, Phoenix handles these queries in a different way, using an approximate algorithm described below.

Firstly, the Phoenix client issues parallel scans restricted by the WHERE clause of the query. The parallel scans are *chunked* by region boundaries and guideposts. *Guideposts* are a set of keys per Region per column family collected by Phoenix at an equal byte distance from each other, that act as hints to improve the parallelization of queries on their Region. The rows that satisfy the WHERE clause are grouped for each chunk in parallel on the server-side by the TopN coprocessor, according to the GROUP BY clause. The TopN coprocessor of each RegionServer keeps only the top N rows for each chunk. Afterwards, the Phoenix client receives the top N rows for each chunk, does a final merge sort and returns the top N rows.

Chapter 3

System Description

3.1 High Level Architecture

3.2 System Components

3.3 Optimizations

Chapter 4

Evaluation

4.1 Dataset Description

4.2 Cluster Description

4.3 Benchmarks

Chapter 5

Conclusion

5.1 Concluding Remarks

5.2 Future Work

Bibliography

[Chur32] A. Church, "A Set of Postulates for the Foundations of Logic", *Annals of Mathematics*, vol. 33, no. 1, pp. 346–366, 1932.