Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής
και Υπολογιστών

# Κατανεμημένη Αναλυτική Επεξεργασία Ροών Δικτυακών Δεδομένων σε Πραγματικό Χρόνο

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

**ΓΕΩΡΓΙΟΣ ΤΟΥΛΟΥΠΑΣ**

**Επιβλέπων :** Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Αθήνα, Οκτώβριος 2015

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής
και Υπολογιστών

# Κατανεμημένη Αναλυτική Επεξεργασία Ροών Δικτυακών Δεδομένων σε Πραγματικό Χρόνο

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

**ΓΕΩΡΓΙΟΣ ΤΟΥΛΟΥΠΑΣ**

**Επιβλέπων :** Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 19η Οκτωβρίου 2015.

| | | |
|---|---|---|
| ........................................... | ........................................... | ........................................... |
| Νεκτάριος Κοζύρης | Νικόλαος Σ. Παπασπύρου | Γεώργιος Γκούμας |
| Καθηγητής Ε.Μ.Π. | Αν. Καθηγητής Ε.Μ.Π. | Λέκτορας Ε.Μ.Π. |

Αθήνα, Οκτώβριος 2015

...........................................

**Γεώργιος Τουλούπας**

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

## Περίληψη

TODO

## Λέξεις κλειδιά

Distributed systems, Storm

# Abstract

TODO

## Key words

Distributed systems, Storm

# Ευχαριστίες

TODO

Γεώργιος Τουλούπας,

Αθήνα, 19η Οκτωβρίου 2015

# Contents

# List of Tables

# List of Figures

Chapter 1

# Introduction

## 1.1 Motivation

test[1]

increasing volume of IP traffic

IXP traffic can provide insight about the whole internet

processing big data made possible with distributed frameworks

## 1.2 Objectives

The goal of thesis is to design and implement a system that allows the execution of SQL queries that join a real-time data stream and an external dataset.

scalability, fault tolerence, extensibility, low latency

our use case is network data, but can be modified for different datasets

the external datasets can be of any size

explain IXP network data usecase

## 1.3 Thesis Outline

In Chapter 2 we provide the necessary theoretical background so that the reader can familiarize themselves with the frameworks and technologies used in in the thesis. More specifically, we present the characteristics, architecture and key concepts of Kafka, Storm, HDFS, HBase and Phoenix.

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.

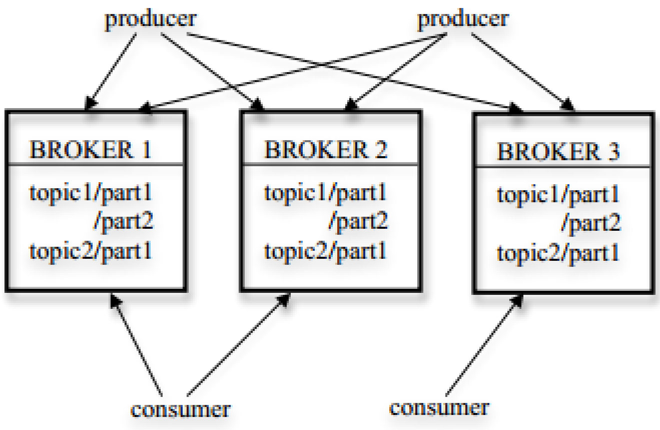


**Figure 2.1:** Kafka architecture

For each topic, the Kafka cluster maintains a partitioned log with the structure depicted in 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.

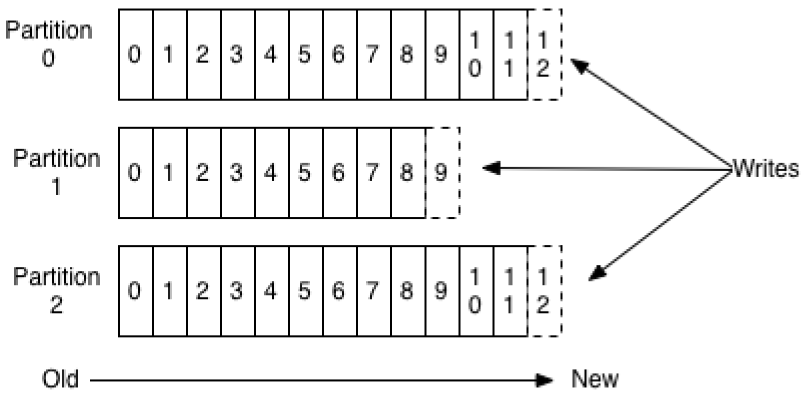


**Figure 2.2:** Kafka topic structure

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.

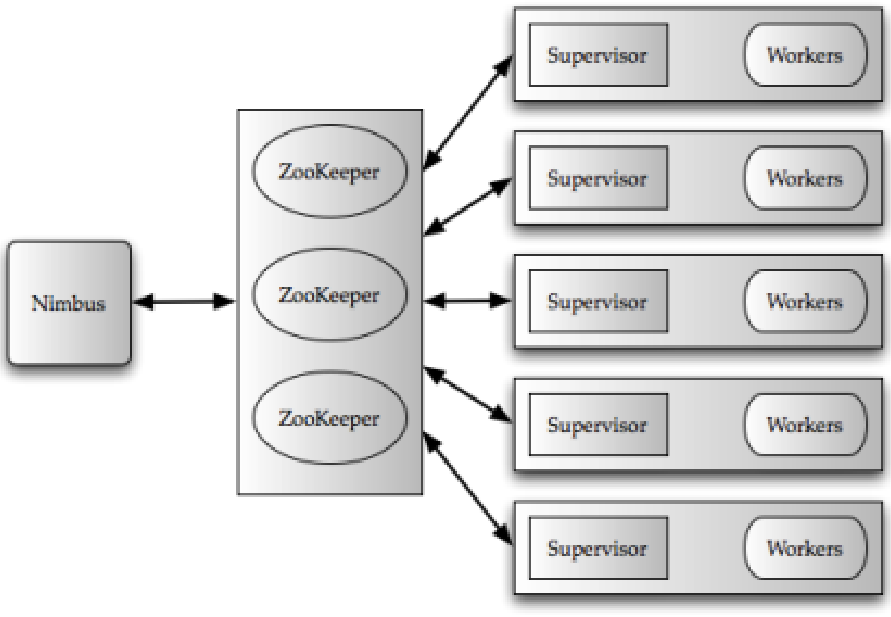


**Figure 2.3:** Storm architecture

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.

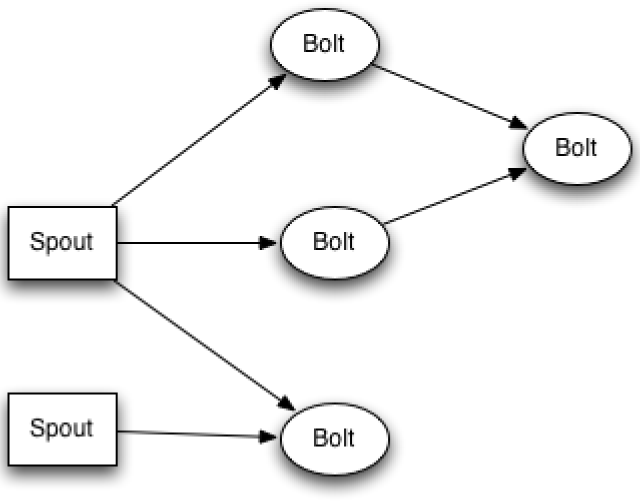


**Figure 2.4:** Example storm topology

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.

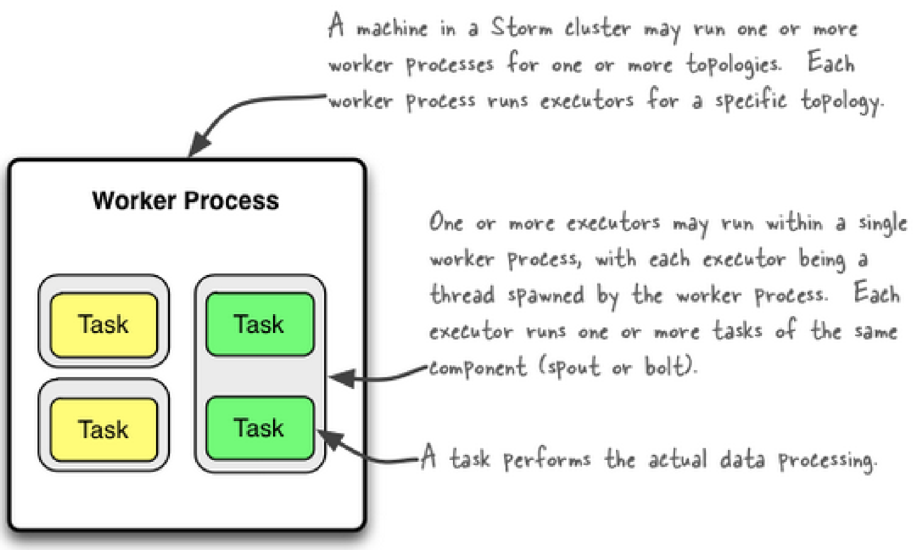

A machine in a Storm cluster may run one or more worker processes for one or more topologies. Each worker process runs executors for a specific topology.
Worker Process
Task
Task
Task
Task
One or more executors may run within a single worker process, with each executor being a thread spawned by the worker process. Each executor runs one or more tasks of the same component (spout or bolt).
A task performs the actual data processing.

**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.

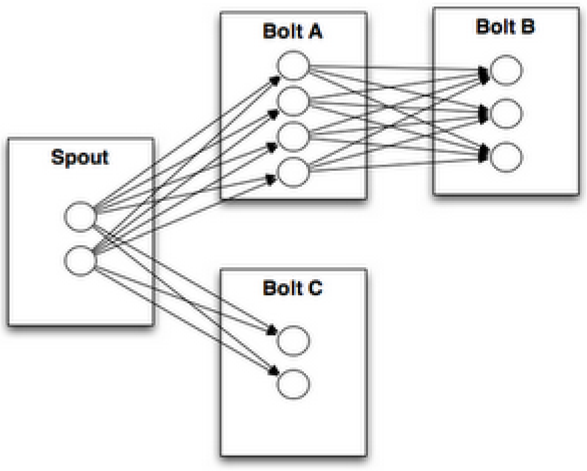


**Figure 2.6:** Task-level execution of a topology

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.

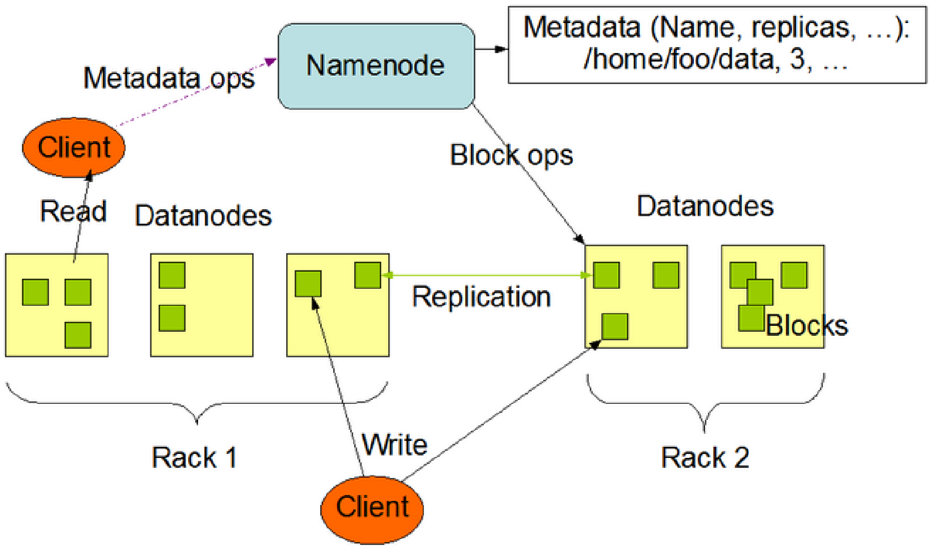


**Figure 2.7:** HDFS architecture

## 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.

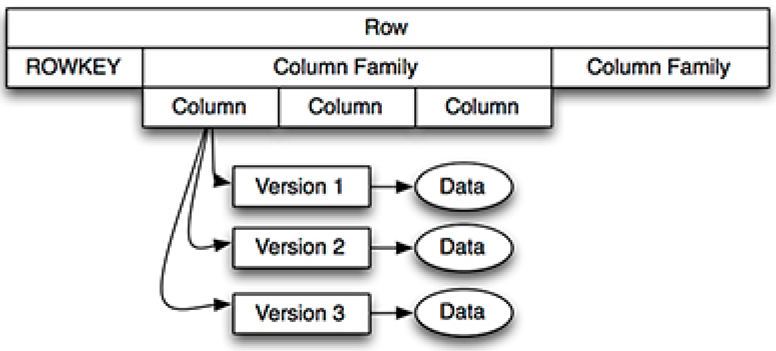


**Figure 2.8:** HBase data model

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.

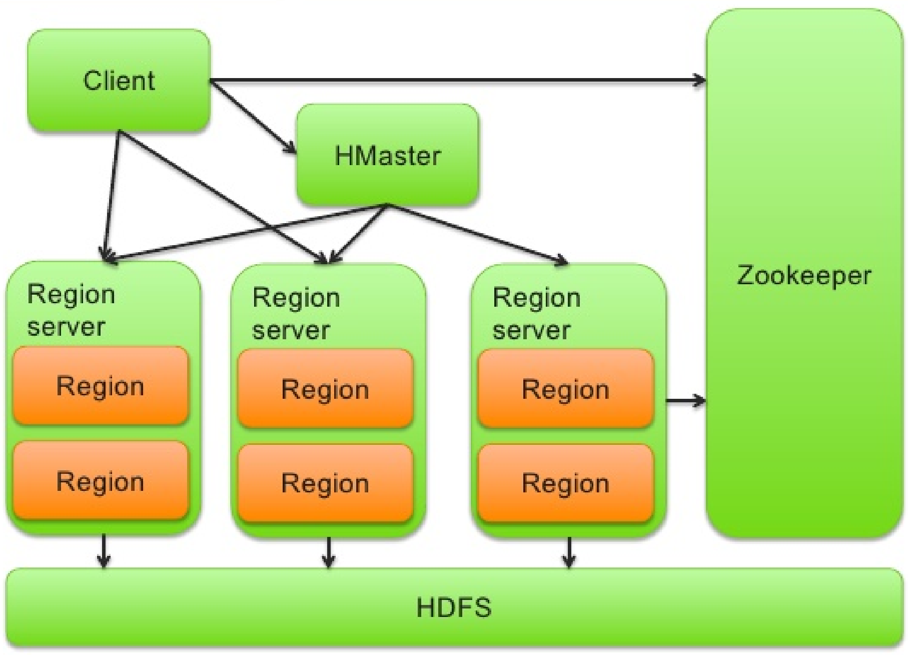


**Figure 2.9:** HBase architecture

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.

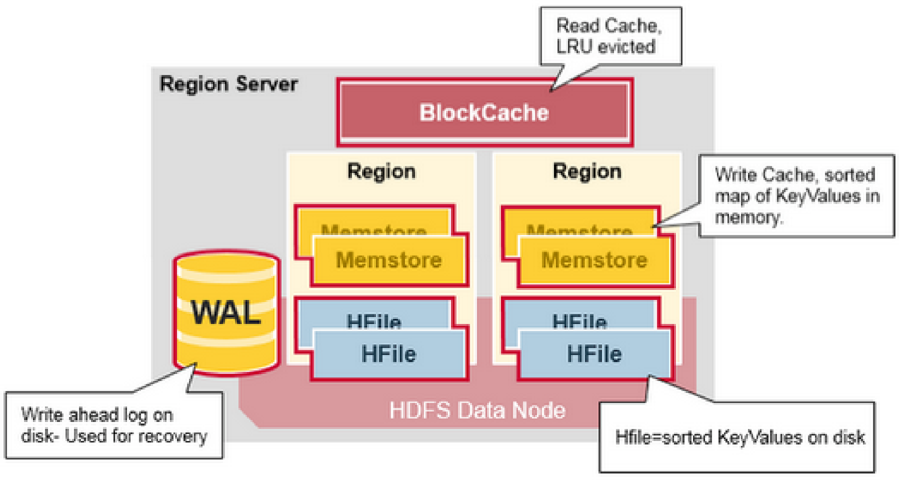

Read Cache, LRU evicted
Region Server
BlockCache
Region
Region
Write Cache, sorted map of KeyValues in memory.
Memstore
Memstore
WAL
HFile
HFile
HDFS Data Node
Write ahead log on disk- Used for recovery
Hfile=sorted KeyValues on disk

**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.

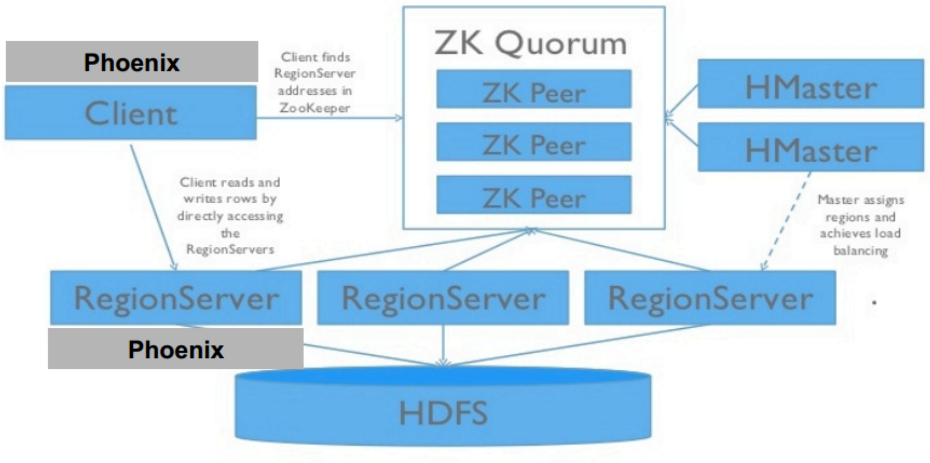

Phoenix
Client
Client finds RegionServer addresses in ZooKeeper
ZK Quorum
ZK Peer
ZK Peer
ZK Peer
HMaster
HMaster
Client reads and writes rows by directly accessing the RegionServers
Master assigns regions and achieves load balancing
RegionServer
RegionServer
RegionServer
Phoenix
HDFS

**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. An example topN query for a pair of columns is presented below.

```
SELECT column1, column2, COUNT(*) AS pairCount
FROM table
WHERE column3 > 0
GROUP BY column1, column2
ORDER BY pairCount DESC
LIMIT 10;
```

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 System Overview

TODO: Rework overview

As mentioned before, the goal of thesis is the design and implementation of a system that allows the execution of SQL queries that join a real-time data stream and an external dataset. The prerequisites for the system are extensibility, scalablitily, fault tolerance and most importantly enabling the execution of low latency SQL queries.

--------------------- join performance an arbitrary time window of the data stream. increasing the query latency multiple

Performing an SQL join combines records from two tables. The join effectively creates a third table which combines the information from both of them. This is at some expense in terms of the time it takes to compute the join.

While it is also possible to simply maintain a denormalized table if speed is important, duplicate information may take extra space, and add the expense and complexity of maintaining data integrity if data which is duplicated later changes.

Denormalization is the process of attempting to optimize the read performance of a database by adding redundant data or by grouping data. In some cases, denormalization is a means of addressing performance or scalability in relational database software. Denormalization introduces a trade-off, speeding up reads while slowing down writes

move the join cost to the Storm processing pre-join ----------------------

From a high level, the system implemented for our IXP network data usecase consists of 4 major parts that can be seen in Figure 3.1. In the first part, the network data is generated by the switches of an IXP and collected by a host running a Kafka producer. There, the useful fields are extracted from the captured packets and published to the Kafka topic. The second component of the system is the Kafka topic that stores the data stream at the Kafka cluster. In the next part, the data stream is processed by a Storm topology. The topology contains the IP to AS Bolt, that performs the join of the data stream and the AS dataset in-memory, since the size of the dataset is small enough. It also contains the IP to DNS Bolt, that performs the join of the data stream and the Reverse DNS dataset using Get operations on the HBase table where the dataset is stored, since it does not fit in the bolt's memory. Finally, in the last part the denormalized network data is stored at a Phoenix table in HBase, allowing Phoenix clients to perform low latency SQL queries to it.

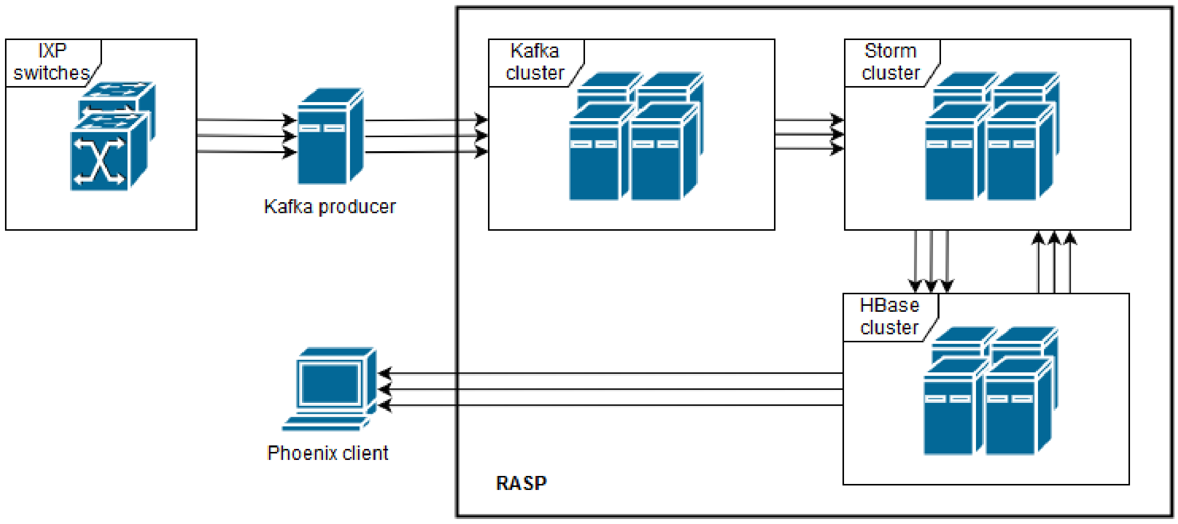


**Figure 3.1:** Storm architecture overview

The system's **scalability** is achieved by using distributed frameworks and technologies for it's implementation. Kafka topics consist of partitions that are distributed over a cluster of Kafka brokers. Storm topologies run over a cluster of Supervisors and multiple instances of any component of the topology (spout or bolt) can run at the same time. The output Phoenix table is stored in HBase and subsequently in the HDFS, which both distributed technologies run on clusters of DataNodes and RegionServers respectively. Moreover, Phoenix can parallelize queries to take full advantage of the HBase cluster.

**Fault tolerance** is very important for our system since it will be constantly running over extended periods of time, processing real-time data. First of all, Kafka topic partitions can be replicated across multiple Kafka brokers, allowing data input by the Kafka producer and consumption by the Storm topology in the case of a broker failure. Storm topologies are also fault tolerant and in case of a Supervisor failure Nimbus reassigns the tasks as necessary. Storm also keeps track of failed tuples and is able to replay them since Kafka retains a topic's data for a configurable period of time. This allows us to restart Storm topologies without skipping any data. Finally, the output Phoenix table that is stored in HBase is replicated by the underlying HDFS, allowing its data to be available in the case of a DataNode or RegionServer failure.

Using the Storm framework provides **extensibility** to our system. Extending the functionality of the Storm topology is as simple as adding an extra bolt to the topology. For example, the processing for the join of the data stream with a new external dataset can be added by implementing the new bolt and placing it before the output Phoenix Bolt. We discern two cases with respect to the size of the external dataset. If the dataset's size is small enough, we can load it in the bolt's memory and perform the join in-memory. Otherwise, when the dataset does not fit in memory, we store it in an HBase table and perform the join using Get operations.

In the following Sections of this Chapter we offer a detailed description for all of the system's components.

## 3.2 Data Generation and Input

### 3.2.1 IXP Switch

The data stream that is processed by our system is generated by an *sFlow agent* running on a switch that processes traffic in an IXP. sFlow is an industry standard technology for monitoring high speed switched networks and is supported by multiple network device manufacturers. The sFlow agent performs random sampling of packets processed by the switch. By default, the agent samples the first 128 bytes of 1 in every 2048 packets.

The flow samples are sent as *sFlow datagrams* (UDP packets) to the *sFlow collector*, described in Subsection 3.2.2. The sFlow collector can accept sFlow datagrams from multiple sFlow agents, allowing us to process a data stream that combines flow samples generated by multiple switches that are used in the same IXP.

### 3.2.2 Kafka Producer

The sFlow datagrams are sent by the sFlow agents of the IXP switches to an sFlow collector running at a specified host. This sFlow collector collects the flow samples from all the switches and makes them available for further processing. In our implementation we use `sflowtool`, a tool functions as an sFlow collector and translates the flow samples to a simple-to-parse ASCII format.

The same host runs a Kafka producer script that preprocesses the flow samples and publishes the useful information to a Kafka topic. This script reads the output of our sFlow collector `sflowtool` and extracts the following useful fields for each sampled packet:

- `sourceIP`: source IP address in dot-decimal notation
- `destinationIP`: destination IP address in dot-decimal notation
- `protocol`: IP protocol number (6 for TCP, 17 for UDP)
- `sourcePort`: source port number
- `destinationPort`: destination port number
- `ipSize`: total length of the IP packet
- `dateTime`: Unix timestamp of the packet's capture time in microseconds. This field is generated by the script while preprocessing each packet.

After the extraction, we compose a message containing the fields in CSV format. The script is running a Kafka producer that publishes these messages to the Kafka topic topic `netdata` that is stored at the Kafka cluster.

Algorithm 1 outlines the script implementation.

**Algorithm 1** Kafka Producer

```
 1: for line in sFlowToolOutput do
 2:     fields = line.split(",")
 3:     sourceIP = fields[9]
 4:     destinationIP = fields[10]
 5:     protocol = fields[11]
 6:     sourcePort = fields[14]
 7:     destinationPort = fields[15]
 8:     ipSize = fields[17]
 9:     dateTime = int(time.time()*1000000)
10:     message = "{},{},{},{},{},{},{}".format(sourceIP, destinationIP, protocol, sourcePort,
    destinationPort, ipSize, dateTime)
11:     kafkaProducer.send_messages("netdata", message)
12: end for
```

Messages can be sent to a Kafka topic either synchronously or asynchronously. Synchronous send publishes the messages immediately, whereas asynchronous send accumulates them in memory batches multiple messages in a single request. As we will see in Subsection 5.3.1 batching can greatly increase the performance of the producer, therefore we choose to use asynchronous send.

## 3.3 Kafka Topic

The preprocessed messages containing the useful fields in CSV format are stored at the `netdata` Kafka topic in the Kafka cluster. For scalability and load balancing we set the number of the topic's partitions equal to the number of the brokers of the Kafka cluster. In this way, the write and read requests of the producer and the consumers respectively are distributed over the cluster.

To provide fault tolerance, we also set a replication factor of 2 for the topic. This means that every partition is replicated and stored in 2 brokers, the leader that handles all the requests for the partition, and the follower that is replicating the leader. In case of failure on the leader, the follower can take over and handle the requests for the partition.

As we mentioned in Section 2.1, all published messages remain stored at the brokers for a configurable period of time, whether or not they have been consumed. This allows the Storm topology to replay previously read messages in case of failure. The default data retention window for the topic is 7 days.

## 3.4 Storm Topology

The Storm topology is the heart of our system. This is where the processing of the data stream is performed. The topology consists of 1 spout and 4 bolts in a pipeline setup: Kafka Spout, Split Fields Bolt, IP to AS Bolt, IP to DNS Bolt and Phoenix Bolt. In short, the topology reads messages from a Kafka topic, extracts the useful fields from the messages, performs the join of the data stream and the external datasets and finally stores the augmented data in a Phoenix table. The topology has acking enabled, which guarantees that every message from the topic will be processed and will be replayed in case it fails.

The overview of the Storm topology for our network data usecase can be seen in Figure 3.2. As we will see in Section **??**, the functionality of the topology can be extended by adding more bolts that perform the join of the data stream and another external dataset right before the Phoenix Bolt.

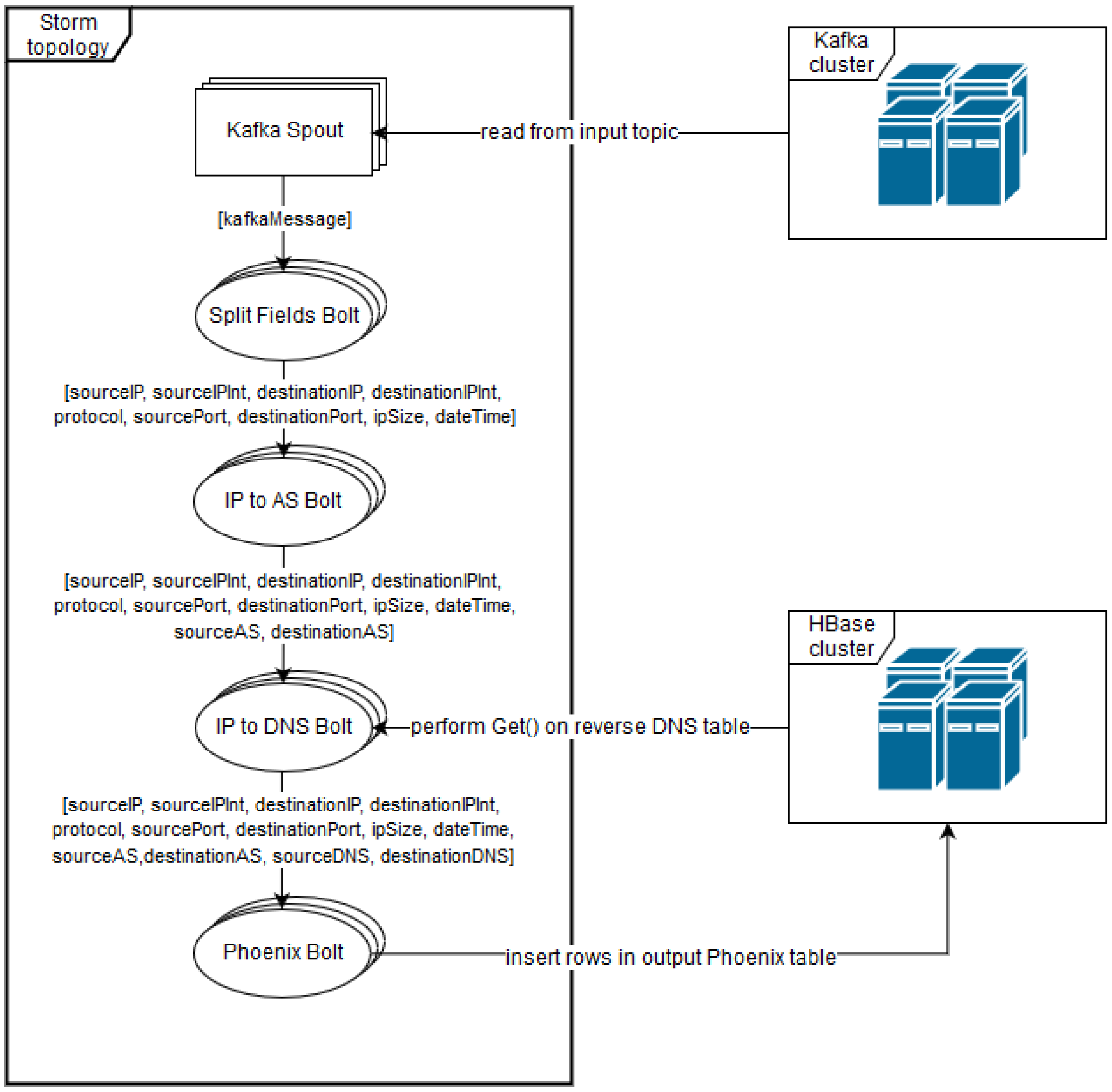

Storm topology
Kafka Spout
read from input topic
Kafka cluster
[kafkaMessage]
Split Fields Bolt
[sourceIP, sourceIPInt, destinationIP, destinationIPInt, protocol, sourcePort, destinationPort, ipSize, dateTime]
IP to AS Bolt
[sourceIP, sourceIPInt, destinationIP, destinationIPInt, protocol, sourcePort, destinationPort, ipSize, dateTime, sourceAS, destinationAS]
HBase cluster
IP to DNS Bolt
perform Get() on reverse DNS table
[sourceIP, sourceIPInt, destinationIP, destinationIPInt, protocol, sourcePort, destinationPort, ipSize, dateTime, sourceAS,destinationAS, sourceDNS, destinationDNS]
Phoenix Bolt
insert rows in output Phoenix table

**Figure 3.2:** Storm topology overview

### 3.4.1 Kafka Spout

The source of data stream in our topology is the Kafka Spout. The spout is a Kafka consumer that reads messages from the `netdata` Kafka topic and emits them to the Split Fields Bolt. The maximum parallelism of the Kafka spout is the number of the topic's partitions, because any instances of the spout further than that would not read any data.

The Kafka Spout stores the offset of the consumer for each partition of the topic in Zookeeper. In this way, if a failure happens the topology can be restarted and resume reading messages from the last one that was executed successfully by the topology.

### 3.4.2 Split Fields Bolt

The tuple emitted by the Kafka Spout has a single field: a message from the topic containing the useful fields of the packet in CSV format. The Split Fields Bolt extracts these fields from the message. In addition to that the bolt computes the integer representations of the source and destiantion IP addresses, which are usually more useful than the IP addresses in dot-decimal notation.

After processing the Kafka message, the Split Fileds Bolt emits a tuple containing the following fields: sourceIP, sourceIPInt, destinationIP, destinationIPInt, protocol, sourcePort, destinationPort, ipSize, dateTime.

### 3.4.3 IP to AS Bolt

The natural join of the data stream and the Autonomous System dataset is performed by the IP to AS Bolt. The Autonomous System dataset maps IP addresses to AS number and name. The data contained in the dataset are stored in CSV format and have 3 fields: the first IP address contained in the AS, the last IP address contained in the AS and the AS number and name. The IP adresses are stored in their integer representation format. The dataset file must be stored in a location accessible by all the Storm supervisors, such as the HDFS. Further information on the dataset is available in Subsection 5.1.2.

The defining characteristic of the Autonomous System dataset is that its size (13 MB) is small enough to fit in the memory, which is the optimal way to perform the join of the stream and the dataset. During the initialization of the topology the `prepare` method of the IP to AS Bolt is called and loads the dataset in a TreeMap structure. A TreeMap is a map implementation based on red-black trees, a variation of binary search trees. For each record of the dataset we insert 2 records in the TreeMap, containing the start and stop IP address for each AS along with the AS number and name.

The helper method `ipToAS` takes an IP address in integer representation format as input and returns the name and number of the AS it belongs. More specifically, by using the TreeMap's ceilingKey and get methods we find the first AS boundary IP address larger or equal to the input IP address. If this address is equal to the input IP address or coresponds to the last address of an AS IP address range, then the AS it belongs is the one we are looking for and its number and name is returned by the method. Otherwise the IP address provided does not belong to any AS according to the dataset and the `ipToAS` method returns the String `"null"`.

For every tuple received by the bolt, the `execute` method is called. Using the sourceIPInt and destinationIPInt fields as input to the method `ipToAS` we determine the sourceAS and destinationAS fields that denote the source and destination AS number and name respectively. The new fields are appended to the received fields and all of them are emitted to the next bolt of the topology.

Algorithm 2 outlines the IP to AS Bolt implementation.

**Algorithm 2** IP to AS Bolt

```
function PREPARE
    asMap = new TreeMap<Long, String[]>()
    for line in ipToASFile do
        fields = line.split(",")
        asMap.put(fields[0], [fields[2], "start"])
        asMap.put(fields[1], [fields[2], "stop"])
    end for
end function
function IPTOAS(ipInt)
    as = "null"
    key = asMap.ceilingKey(ipInt)
    if key != null then
        value = asMap.get(key)
        if (key == ipInt) || (value[1].equals("stop")) then
            as = value[0]
        end if
    end ifreturn as
end function
function EXECUTE(tuple)
    sourceIPInt = tuple.getField("sourceIPInt")
    destinationIPInt = tuple.getField("destinationIPInt")
    outputValues = tuple.getValues()
    outputValues.add(ipToAS(sourceIPInt))
    outputValues.add(ipToAS(destinationIPInt))
    collector.emit(outputValues)
end function
```

### 3.4.4 IP to DNS Bolt

The natural join of the data stream and the Reverse DNS dataset is performed by the IP to DNS Bolt. The Reverse DNS dataset maps IP addresses to domain names. The data contained in the dataset have 2 fields: the IP address in dot-decimal notation and the corresponding domain name. Further information on the dataset is available in Subsection 5.1.3.

The defining characteristic of the Reverse DNS dataset is that its size (55 GB uncompressed) is larger than the memory size, therefore loading it in every bolt's memory is not an option. To make the dataset available to the bolts, we store it in the `rnds` HBase table, where the IP addresses are used as the row key and the domain names are stored in the column `d:dns`. This allows the bolts to perform Get operations on the table for an IP address row key to get the corresponding domain name.

HBase can perform low latency Get operations by using *Bloom filters*. A Bloom filter, is a data structure which is designed to predict whether a given element is a member of a set of data. A positive result from a Bloom filter is not always accurate, but a negative result is guaranteed to be accurate. In HBase, Bloom filters a lightweight in-memory structure to reduce the number of disk reads for a given Get operation to only the HFiles likely to contain the desired row.

The helper method `ipToDNS` takes an IP address in dot-decimal notation as input and returns corresponding domain name. More specifically, a Get operation is performed on the `rdns` HBase table for the input IP address row key. If the Get is successful, the corresponding domain name is the value of the column `d:dns` of the returned row, and is afterwards returned by the method. Otherwise the IP address provided does not have a corresponding domain name according to the dataset and the `ipToDNS` method returns the String `"null"`.

For every tuple received by the bolt, the `execute` method is called. Using the sourceIP and destinationIP fields as input to the method `ipToDNS` we determine the sourceDNS and destinationDNS fields that denote the source and destination domain names respectively. The new fields are appended to the received fields and all of them are emitted to the next bolt of the topology.

Algorithm 3 outlines the IP to DNS Bolt implementation.

**Algorithm 3** IP to DNS Bolt

```
1:  function ipToDNS(ip)
2:      table = new HTable("rdns")
3:      g = new Get(ip)
4:      res = table.get(g)
5:      dns = res.getValue("d", "dns")
6:      if dns == null then
7:          dns = "null"
8:      end ifreturn dns
9:  end function
10: function EXECUTE(tuple)
11:     sourceIP = tuple.getField("sourceIP")
12:     destinationIP = tuple.getField("destinationIP")
13:     outputValues = tuple.getValues()
14:     outputValues.add(ipToDNS(sourceIP))
15:     outputValues.add(ipToDNS(destinationIP))
16:     collector.emit(outputValues)
17: end function
```

### 3.4.5 Phoenix Bolt

The last component of the topology is the Phoenix Bolt, which inserts the augmented data stream into the `netdata` Phoenix table. The table is described in detail in Section 3.5. This bolt uses the Phoenix JDBC driver and performs an `UPSERT VALUES` query that includes all the fields received by the bolt. `UPSERT` queries are the only way to insert data in a table in Phoenix. This query inserts the row if not present, otherwise it updates the row's values in the table. In our case where the primary key of the table is the packet timestamp which is monotonically increasing this query behaves like `INSERT VALUES`.

```
UPSERT INTO netdata VALUES (dateTime, sourceIP, sourceIPInt, destinationIP,
    destinationIPInt, protocol, sourcePort, destinationPort, ipSize, sourceAS,
    destinationAS, sourceDNS, destinationDNS);
```

## 3.5 Phoenix Table

After being processed by the Storm topology, the augmented data stream is stored at the `netdata` Phoenix table in HBase. The design of this table is important because it affects the way queries are executed. In our use case, the queries performed will be topN AS or topN DNS queries over a time window for the data.

The queries performed on the table have a time window constraint. To benefit from HBase Scan operations that perform sequential reads, we want to use the packet's capture timestamp as the row key in the underlying HBase table. In this way, the table is sorted by capture timestamp and the data for any time window are stored sequentially. To achieve this we use the packet's capture timestamp as the primary key of the Phoenix table. The capture timestamp in microseconds can be used as the primary key since it is unique for each packet.

The use case queries concern either AS or DNS information. In HBase only the column families needed for the query are cached. Having separate column families containing AS, DNS and other information reduces query latency by reducing the data that have to be cached during each query. Therefore we separate the table's columns in 3 column families: one for the AS fields, another for DNS fields and a default column family that contains the rest of the packet's fields.

In HBase every cell value is always (when stored, transfered or cached) accompanied by its row key, column name and timestamp. Since the table will store millions of cells the column names will be repeates several millions of times in our data. This means that if the column names are large then the table size will be significantly increased. This is why we try to minimize the column names by keeping the column family and column qualifier names as small as possible.

Another way to reduce the table size is by utilizing Phoenix data types. Using the appropriate data type for each column reduces the size of each row, which improves query performance. For example, instead of storing the capture timestamp as string, we use the `BIGINT` type. The current UNIX timestamp in microseconds has 16 digits. As a string this needs 16 bytes to be stored, whereas a `BIGINT` needs only 8 bytes.

Having all the aforementioned design choices taken into consideration we create the `netdata` Phoenix table with the columns listed below. The dots in the column names separate the column families from the column qualifiers created in the underlining HBase table.

- `t`: Unix timestamp of the packet's capture time in microseconds, used as the primary key
- `d.ipS`: source IP address in dot-decimal notation
- `d.ipSI`: integer representation of the source IP address
- `d.ipD`: destination IP address in dot-decimal notation
- `d.ipDI`: integer representation of the destination IP address
- `d.proto`: IP protocol number of the packet
- `d.portS`: source port number
- `d.portD`: destination port number
- `d.size`: total length of the IP packet
- `as.asS`: AS number and name of the source IP address
- `as.asD`: AS number and name of the destination IP address
- `dns.dnsS`: domain name of the source IP address
- `dns.dnsD`: domain name of the destination IP address

The final table creation query, including the optimizations that will be described in Chapter 4 is presented below.

```
CREATE TABLE netdata (
    t BIGINT PRIMARY KEY,
    d.ipS VARCHAR,
    d.ipSI BIGINT,
    d.ipD VARCHAR,
    d.ipDI BIGINT,
    d.proto SMALLINT,
    d.portS INTEGER,
    d.portD INTEGER,
    d.size INTEGER,
    as.asS VARCHAR,
    as.asD VARCHAR,
    dns.dnsS VARCHAR,
    dns.dnsD VARCHAR
)
SALT_BUCKETS = 4,
DEFAULT_COLUMN_FAMILY = 'd',
DATA_BLOCK_ENCODING = 'NONE',
COMPRESSION = 'SNAPPY';
```

Chapter 4

# HBase and Phoenix Optimizations

## 4.1 HDFS Short-Circuit Local Reads

In HDFS, all reads normally go through the DataNode. When a RegionServer asks the DataNode to read a file, the DataNode reads that file off of the disk and sends the data to the RegionServer over a TCP socket. The downside of this approach for local reads is the overhead of the TCP protocol in the kernel, as well as the overhead of DataTransferProtocol used for the communication with the DataNode.

When the RegionServer is co-located with the data and short-circuit local reads are enabled, local reads bypass the DataNode. This allows the RegionServer to read the data directly from the local disk. Short-circuit local reads provide a substantial performance boost in data transfer from the disk to the BlockCache when the data is local.

We evaluate the effect of enabling HDFS short-circuit local reads on Subsection 5.6.1.

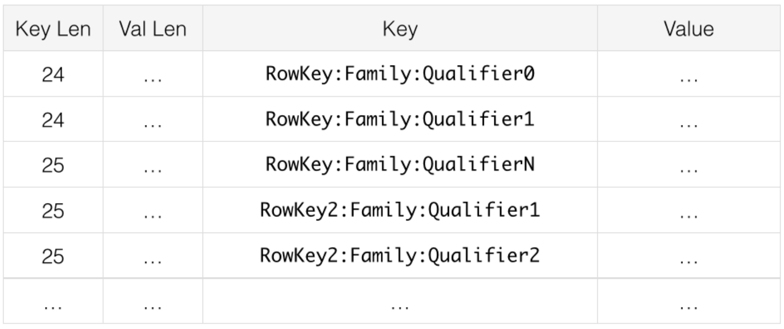
## 4.2 Compression and Data Block Encoding

Physical data size on disk can be decreased by using compression and data block encoding. Compression reduces the size of large, opaque byte arrays in cells, and can significantly reduce the storage space needed to store uncompressed data. Data block encoding attempts to limit duplication of information in keys, taking advantage of some of the fundamental designs and patterns of HBase, such as sorted row keys and the schema of a given table. Compression and data block encoding can be used together on the same column family.

Aside from on-disk data size, compression and data block encoding can reduce the data size in the BlockCache. Data is cached by default on their encoded format. Moreover, compressed BlockCache can be enabled, allowing compressed data to be cached in their compressed and encoded on-disk format.

Between all of our compression options, Snappy is the most fitting to our use case, since minimizing query latency is our priority. It does not aim for maximum compression, but instead aims for very high speeds and reasonable compression. Compared to gzip, Snappy is an order of magnitude faster for most inputs, but he compression ratio is 20% to 100% lower.

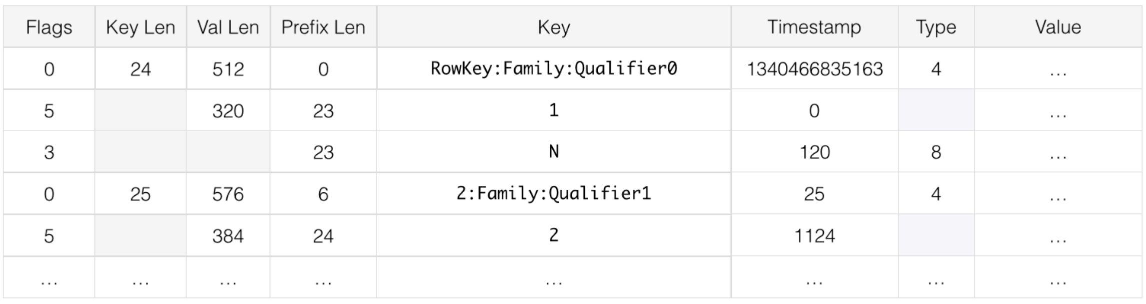
Regarding data block encoding, Fast Diff enabled by default in HBase. The format in which non-encoded data are stored in the HFile often results in multiple similar keys for each row, as seen in Figure 4.1.



| Key Len | Val Len | Key | Value |
|---|---|---|---|
| 24 | … | `RowKey:Family:Qualifier0` | … |
| 24 | … | `RowKey:Family:Qualifier1` | … |
| 25 | … | `RowKey:Family:QualifierN` | … |
| 25 | … | `RowKey2:Family:Qualifier1` | … |
| 25 | … | `RowKey2:Family:Qualifier2` | … |
| … | … | … | … |

**Figure 4.1:** Column family stored with no encoding

Fast Diff works similar to Diff encoding, but uses a faster implementation. In Diff encoding, the most important feature is an extra column which holds the length of the prefix shared between the current key and the previous key. In addition, the timestamp is stored as the difference from the previous row's timestamp, rather than being stored in full. Figure 4.2 shows the same data with Figure 4.1 stored with Diff encoding.



| Flags | Key Len | Val Len | Prefix Len | Key | Timestamp | Type | Value |
|---|---|---|---|---|---|---|---|
| 0 | 24 | 512 | 0 | `RowKey:Family:Qualifier0` | 1340466835163 | 4 | … |
| 5 |  | 320 | 23 | `1` | 0 |  | … |
| 3 |  |  | 23 | `N` | 120 | 8 | … |
| 0 | 25 | 576 | 6 | `2:Family:Qualifier1` | 25 | 4 | … |
| 5 |  | 384 | 24 | `2` | 1124 |  | … |
| … | … | … | … | … | … | … | … |

**Figure 4.2:** Column family stored with Diff encoding

Both compression (with compressed BlockCache enabled) and data block encoding reduce the in-cache data size. This means that more rows can be cached at the same time, while data transfer time from the disk to the BlockCache for the same data is reduced. However every time the cached data is used in a query they must be decompressed or decoded or both. These performance hits increase query latency, while is our priority is to minimize it.

To achieve the best in-cache query latency we decide to use Snappy compression for our final Phoenix table, in conjunction with enabled compressed BlockCache and no data block encoding. The experiment on which we base this decision is presented on Subsection 5.6.2.

## 4.3 Disable BlockCache on the Reverse DNS Table

The Reverse DNS dataset is stored in the `rdns` HBase table. This table has Snappy compression and no data block encoding to reduce on-disk size and avoid the decoding performance hit on read. The on-disk size of the compressed table is 12GB.

We investigate the read access pattern on the table by the IP to DNS Bolt. Since the IP addresses on the packets are random, the reads are performed on the table `rdns` are random too. Every read caches the HFile it hits, which does not provide any benefit since `rdns` does not fit into the BlockCaches of the RegionServers. Moreover, constantly caching different HFiles of `rdns` throws out of the cache

HFiles of the output table `netdata`. Subsequent queries will have to cache these HFiles again, which increases the query latency.

To alleviate this problem we disable BlockCache on `rdns`, thus allowing the `netdata` table to fully take advantage of the cache.

## 4.4 Salting

Rows in HBase are sorted lexicographically by row key. The row key for the HBase table where our Phoenix table is stored must be the timestamp associated with the packet, in order to optimize scans for queries over a specified time range. Since the timestamp is always increasing for live data, the row key is also monotonically increasing.

However, monotonically increasing row keys are a common source of hotspotting. When records with sequential keys are being written to HBase all writes hit one region which is served by one RegionServer. This uneven write load distribution limits the write throughput to the capacity of a single RegionServer instead of making use of multiple nodes in the HBase cluster. Moreover, hotspotting overwhelms the RegionServer responsible for hosting that region, causing performance degradation and potentially leading to region unavailability.

Salting the row key provides a way to mitigate the problem. Salting refers to adding a randomly-assigned prefix to the row key to cause it to sort differently than it otherwise would. The number of possible prefixes correspond to the number of regions you want to spread the data across. For example we can salt the row key by using this:

```
newKey = (++index % BUCKETS_NUMBER) + originalKey
```

In this listing, the `newKey` is produced by prefixing the `originalKey` a salt denoting the salt bucket. The salted records are be split into multiple buckets served by different RegionServers. The row keys of bucketed records are no longer in the original sequence, however records within in each bucket preserve their original sequence, as seen in Figure 4.3.

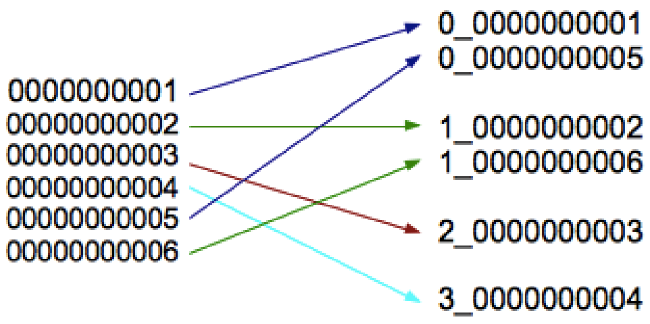


**Figure 4.3:** HBase row key prefix salting

Since data is placed in multiple buckets during writes, we have to read from all of those buckets when doing scans based on original start and stop keys and merge-sort the data. These scans can be run in parallel on the different RegionServers serving the salt buckets, which may lead to an increase in read performance.

Phoenix provides a way to transparently salt the row key with a salting byte for a particular table. To distribute the load evenly among all the nodes of the HBase cluster we set the number of salt buckets equal to the number of the RegionServers. The effect of salting on writes and reads is evaluated on Subsections 5.4.5 and 5.6.4 respectively.

# Chapter 5

# Evaluation

## 5.1 Datasets

### 5.1.1 IXP Traffic Dataset

The data used for the evaluation of the system is network traffic collected by GR-IX. GR-IX is the Greek Internet Exchange Point (IXP), through which Greek Internet Service Providers (ISPs) exchange traffic between their networks without using their upstream transit providers. GR-IX was founded in 2009 as a successor of AIX (Athens Internet Exchange), which was in operation since 2000. The management and operation of the exchange is done by the Greek Research and Technology Network (GRNET).

GR-IX is handling aggregate traffic that is at peaking multiple Gigabytes per second. Using the packet sampling tool sFlow, IP packet were captured with a random sampling of 1 out of 2000 over a period of six months (July 2013 to February 2014). During this period of time 1.9 billion packets were captured, which translates to an average of 110 packets sampled per second. The captured packets were preprocessed to extract the following useful fields:

- `sourceIP`: source IP address in dot-decimal notation
- `destinationIP`: destination IP address in dot-decimal notation
- `protocol`: IP protocol number (6 for TCP, 17 for UDP)
- `sourcePort`: source port number
- `destinationPort`: destination port number
- `ipSize`: total length of the IP packet
- `dateTime`: Unix timestamp of the packet's capture time in microseconds

### 5.1.2 Autonomous System Dataset

One of the external datasets used by the topology is the GeoLite ASN IPv4 database. This dataset maps IPv4 addresses to Autonomous System Numbers (ASN), including the names of each Autonomous System. This dataset created by MaxMind and is updated every month. The dataset comes in a CSV file, having a size of 13 MB. This file is stored at HDFS in order to be available to the Storm Supervisors. The data contained in this dataset have the following fields:

- `ipIntStart`: integer representation of the first IP address contained in the AS
- `ipIntEnd`: integer representation of the last IP address contained in the AS
- `as`: AS number and name

### 5.1.3 DNS Dataset

The other external dataset used by the topology is the Rapid7 Reverse DNS dataset. This dataset maps IPv4 addresses to domain names. Rapid7 Labs creates this data by performing a DNS PTR lookup for all IPv4 addresses. It is updated every 2 weeks and is made available at The Internet-Wide Scan Data Repository(scans.io). The data format is a gzip-compressed CSV file, having a size of 5.7 GB compressed and 55 GB uncompressed, while containing 1.2 billion records. The fields of the dataset are:

- `ip`: IP address in dot-decimal notation
- `dns`: domain name

The Reverse DNS dataset is stored in the `rdns` HBase table. The field `ip` is used as the row key and `dns` is stored at a column.

## 5.2 Cluster Description

To execute the following experiments, we use virtual machines (VMs) on the Openstack cluster hosted by the Computing Systems Laboratory (CSLab) of the School of Electrical and Computer Engineering, NTUA.

For the performance tuning experiments we create 10 virtual machines:

- **Zookeeper:** This is a Zookeeper server running in standalone mode, running the QuorumPeerMain application. Zookeeper is providing coordination between the nodes of the Kafka, Storm and HBase clusters.
- **Master:** This node is running the master applications for all the clusters. Master runs the Nimbus daemon for Storm, a NameNode and SecondaryNameNode for HDFS and an HMaster for HBase.
- **Storm cluster:** The Storm cluster consists of 4 virtual machines running the Supervisor daemon.
- **Kafka and HBase cluster:** The Kafka and HBase clusters are co-hosted on 4 virtual machines. Each machine runs a Kafka server, an HDFS DataNode and HRegionServer for HBase. The RegionServer is allowed to have 5 GB of maximum heap size. Kafka CPU usage is very low on all of our benchmarks (around 3%), therefore co-hosting it with HBase is justified.

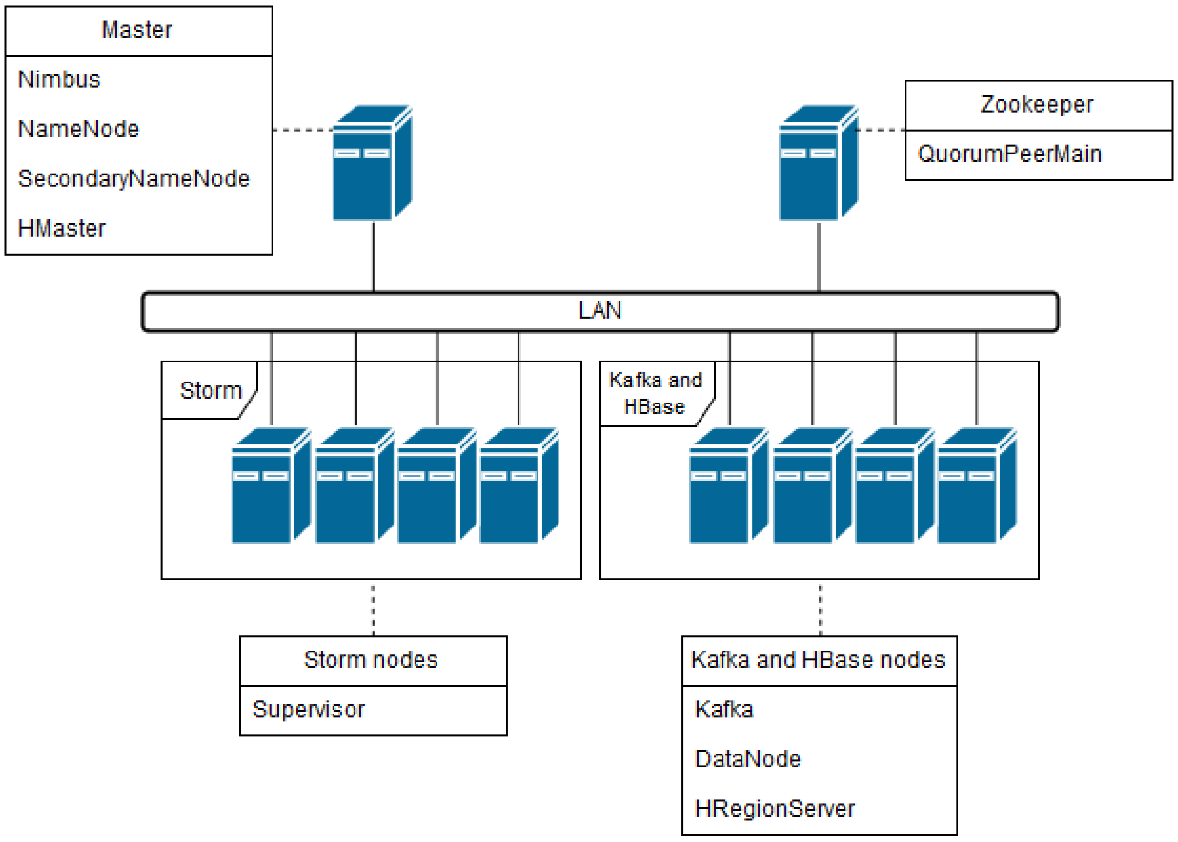


**Figure 5.1:** Cluster deployment diagram

The deployment diagram for the clusters is presented in Figure 5.1. Each of the virtual machines has the specifications listed in Table 5.1. The versions of the software used in our experiments are listed in Table 5.2.

| Component | Description |
|---|---|
| CPU | 4 cores @ 2.4 GHz |
| RAM | 8 GB |
| Disk | 80 GB |

**Table 5.1:** Virtual machine hardware specifications

| Software | Version |
|---|---|
| Zookeeper | 3.4.6 |
| Kafka | 2.10-0.8.2.1 |
| Storm | 0.9.4 |
| Hadoop | 2.6.0 |
| HBase | 1.1.0.1 |
| Phoenix | 4.4.0-HBase-1.1 |

**Table 5.2:** Software versions

To conduct the scalability experiments, we increase the number of nodes in the Storm and Kafka/HBase clusters up to 16 for each. The rest of the deployment details remain the same.

## 5.3 Kafka Performance and Scalability

To measure Kafka performance and scalability we use the performance tools ProducerPerformance and TestEndToEndLatency shipped with the Kafka installation. For the following experiments, we set the size of the messages generated by the tools to 62 bytes, to match the average size of the messages produced by real IXP traffic. The topic we use has 4 partitions and a replication factor of 2 for fault tolerance. Replication during the experiments is asynchronous, meaning that the broker acknowledges the write as soon as it has written it to its local log without waiting for the other replicas to also acknowledge it.

### 5.3.1 Producer Batch Size

The producer can be configured to accumulate data in memory and to send out larger batches in a single request for each partition. Batching leads to larger network packets and larger sequential disk operations on the brokers, which allows Kafka to turn a stream of random message writes into linear writes. This increases performance on both the producer and the broker.

We experiment with different batch sizes and measure the message input throughput for our topic. The effects of batching on throughput can be observed in Table 5.3 and Figure 5.2.

| Batch size | Throughput (messages/sec) |
|---|---|
| 100 | 12658 |
| 200 | 25516 |
| 400 | 49397 |
| 800 | 104597 |
| 1600 | 188730 |
| 3200 | 293877 |
| 6400 | 381859 |

**Table 5.3:** Producer batch size effect on topic throughput

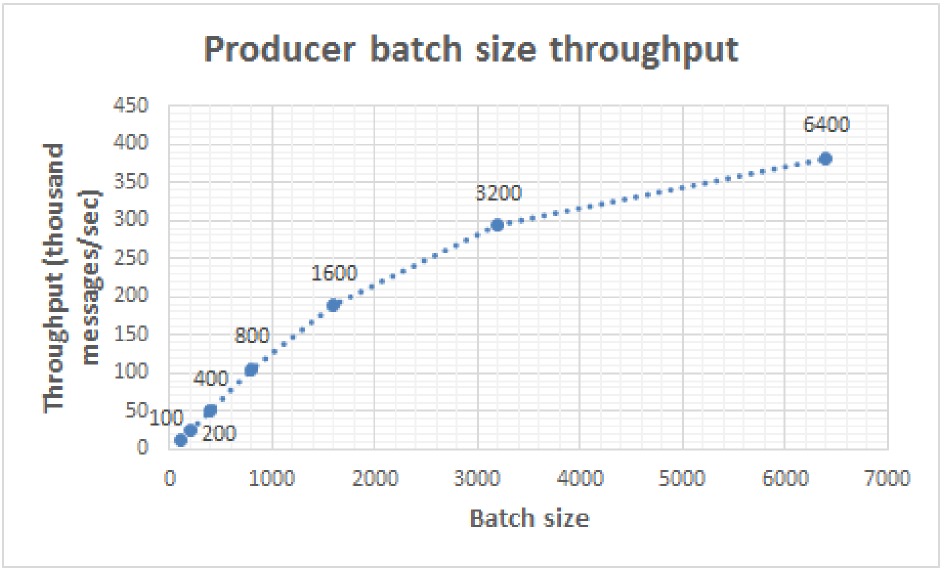


**Figure 5.2:** Producer batch size effect on topic throughput

Even though a bigger batch size can increase throughput by orders of magnitude, it also increases the time a message is waits in the producer to be sent in the next batch. Since even a low batch size 100 can achieve greater throughput (12658 messages/sec) than the storm topology in the maximum configuration of our scalability experiment (3988 messages/sec as we will see in Section 5.5), we opt to choose a small batch size in order to reduce message latency. To allow the producer to handle bursts of more packets, we use batch size **200** for our producer. The rest of the experiments are performed with batch size 200.

### 5.3.2 End-to-End Latency

End-to-end latency is the time it takes for a message sent by a producer to be delivered to a consumer. For this experiment, the performance tool TestEndToEndLatency creates a producer and a consumer and repeatedly times how long it takes for a producer to send a message to the Kafka cluster and then be received by the consumer.

The average Kafka end-to-end latency is measured at **2.871 msec**. Other latency percentiles are presented at Table 5.4.

| Percentile | Latency (msec) |
|---|---|
| 50th | 2 |
| 99th | 4 |
| 99.9th | 10 |

**Table 5.4:** End-to-end latency percentiles

### 5.3.3 Multiple Producers

In this experiment we multiple producers that create messages for a single topic and measure the aggregate message input throughput for the topic. The producers are running on different machines.

Figure 5.3 shows that the aggregate message input throughput increases linearly with the number of the simultaneous producers. This allows us to expand our system to collect and store in Kafka data from multiple different producer sources, in our case multiple IXP switches.

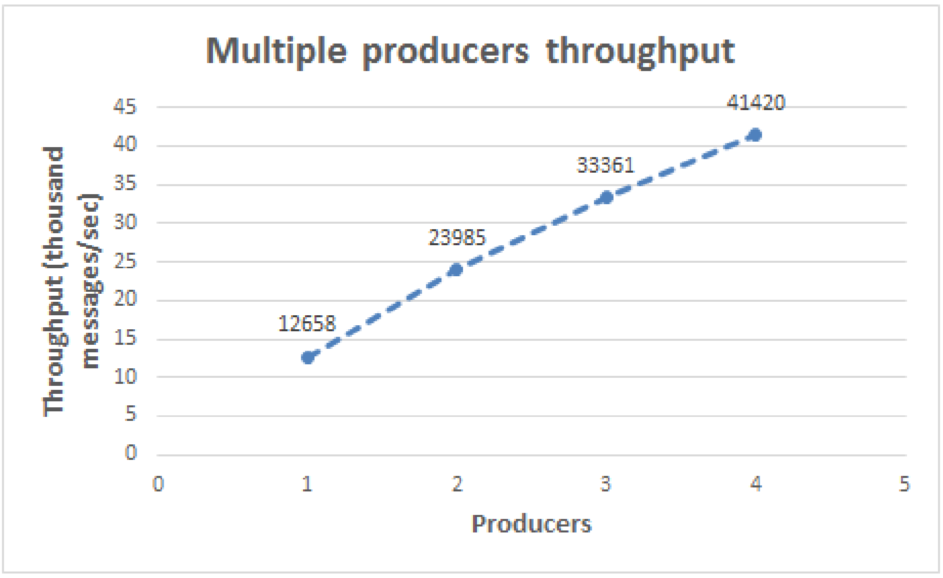


**Figure 5.3:** Topic throughput scalability with the number of producers

### 5.3.4 Kafka Scalability

Partitions allow the topic to scale in size by being distributed over the brokers of the cluster and act as the unit of parallelism, providing load balancing over the write and read requests of the producers and the consumers respectively.

To evaluate the scalability of the Kafka topic with the Kafka cluster size, we measure the message input throughput for clusters with different numbers of brokers. The number of the topic's partitons is adjusted according to the number of the brokers.

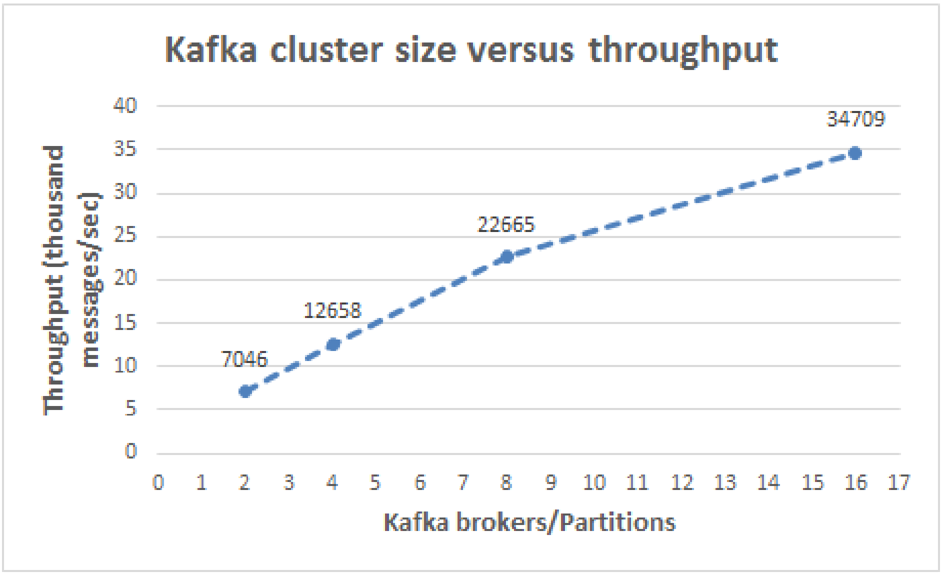


**Figure 5.4:** Topic throughput scalability with Kafka cluster size

## 5.4 Storm Performance Tuning

### 5.4.1 Parallelism Tuning

To achieve maximum topology throughput, we experiment with the parallelism of its components (spout and bolts). Parallelism tuning in Storm is performed with the help of the capacity metric.

The capacity metric tells us what percentage of the time in the last 10 minutes the bolt spent executing tuples. If this value is close to 1, then the bolt is 'at capacity' and is a bottleneck in our topology. The solution to at-capacity bolts is to increase the parallelism of that bolt. The listing used to compute the capacity metric is:

```
capacity = (executedTuplesNumber * averageExecuteLatency) / measurementTime
```

During our parallelism tuning experiments, when we see that a bolt's capacity is close to 1 we increase its parallelism in the next experiment. We continue until we achieve maximum topology throughput. The parallelism and capacity for each bolt during the parallelism tuning experiments are presented on Table 5.5. The name of each experiment denotes the parallelism of each component of the topology: Kafka Spout - Split Fields Bolt - IP to AS Bolt - IP to DNS Bolt - Phoenix Bolt.

Note that capacity is computed based on topology statistics, therefore its value may sometimes be larger than 1. The parallelism of the Kafka Spout is always 4 to match the number of the topic's partitions.

| Experiment | Split Fields Bolt | | IP to AS Bolt | | IP to DNS Bolt | | Phoenix Bolt | |
|---|---|---|---|---|---|---|---|---|
| | Parallelism | Capacity | Parallelism | Capacity | Parallelism | Capacity | Parallelism | Capacity |
| 4-4-4-4-4 | 4 | 0.013 | 4 | 0.011 | 4 | 0.600 | 4 | **0.987** |
| 4-4-4-12-12 | 4 | 0.021 | 4 | 0.042 | 12 | 0.491 | 12 | **1.068** |
| 4-4-4-12-20 | 4 | 0.040 | 4 | 0.043 | 12 | **1.043** | 20 | **0.911** |
| 4-4-4-16-28 | 4 | 0.043 | 4 | 0.062 | 16 | 0.879 | 28 | **1.049** |
| 4-4-4-16-36 | 4 | 0.067 | 4 | 0.077 | 16 | 0.816 | 36 | **1.086** |

**Table 5.5:** Bolt capacity during parallelism tuning experiments

As we can see in Figure 5.5 we can achieve maximum topology throughput with the parallelism combination **4-4-4-16-28**. We use these parallelism settings in the rest of the benchmarks.

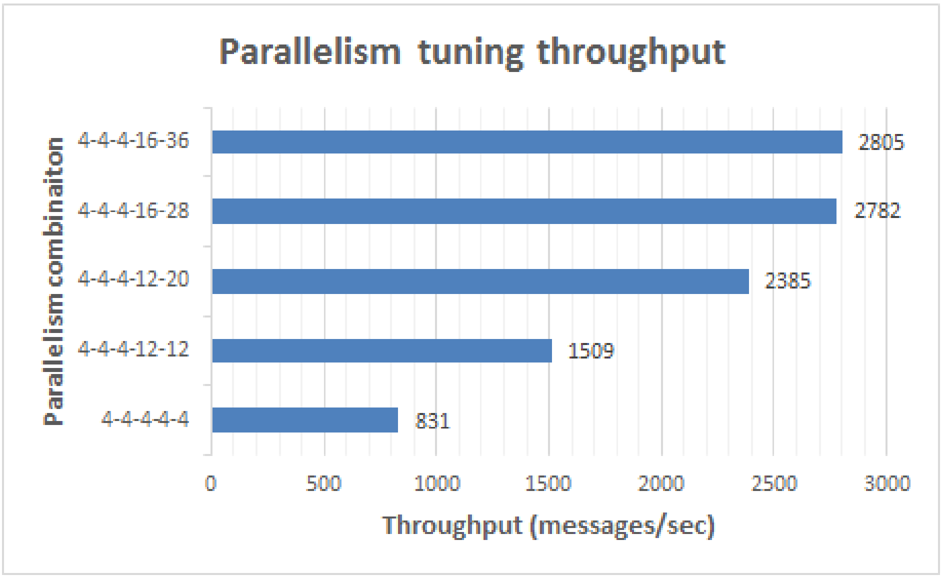


**Figure 5.5:** Topology throughput during parallelism tuning experiments

We also record the average CPU utilization for the Storm and HBase clusters during the tuning experiments and present them in Figure 5.6. We notice that the CPUs of the Storm and HBase clusters are not saturated at maximum topology throughput, which indicates that the topology workload is I/O intensive. This was to be expected since the IP to DNS Bolt and the Phoenix Blot perform reads and writes respectively to HBase tables.

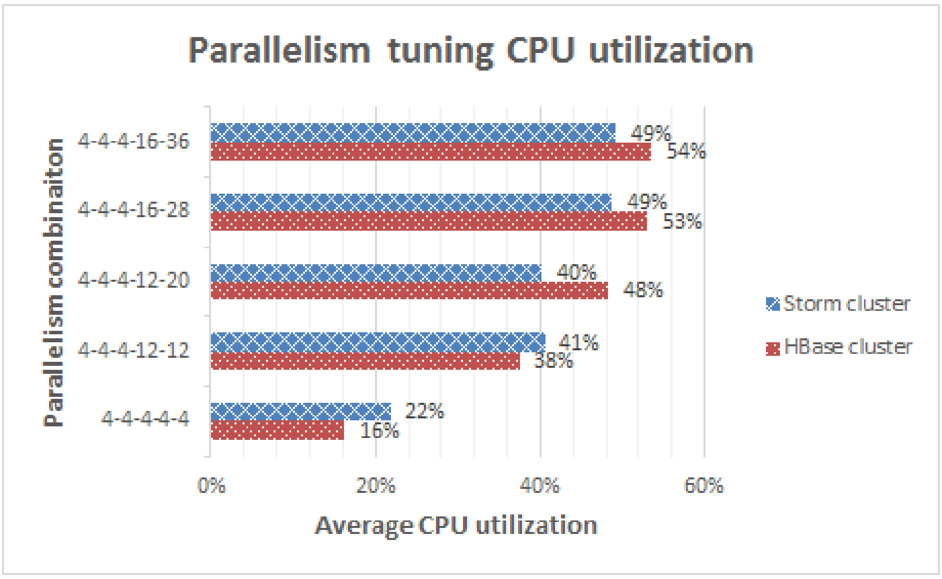


**Figure 5.6:** Average CPU utilization for the Storm and HBase clusters during parallelism tuning experiments

### 5.4.2 Maximum Pending Tuples

Storm topologies have a maximum spout pending tuples parameter. This value puts a limit on the number of tuples that can be in flight (have not yet been acked or failed) in a Storm topology at any point of time. The need for this parameter comes from the fact that Storm uses queues to dispatch tuples from one task to another task. If the consumer side of the queue is unable to keep up with the tuple rate, then the queue starts to build up. Eventually tuples timeout at the spout and get replayed to the topology, thus adding more pressure on the queues. To avoid this failure case, Storm allows the user to put a limit on the number of tuples that are in flight in the topology. Setting a small maximum pending tuples number can starve the topology from tuples, while a sufficiently large value can overload the topology with a huge number of tuples to the extent of causing failures and replays.

We experiment with the maximum pending tuples value while feeding the topology with messages at the maximum rate determined in Subsection 5.4.1 (around 2800 messages/sec). The results presented on Table 5.3 indicate that we can achieve maximum throughput by setting the value of the maximum pending tuples parameter over 100. To allow the topology to handle bursts of more messages, we use the value **200** for the maximum pending spout tuples parameter. In the case of a message burst, the in-flight tuples will spend more time in the internal queues of the topology, but their number will still be limited by the parameter to avoid failures by overloading.

| Maximum pending tuples | Throughput (messages/sec) |
|---|---|
| 10 | 1347 |
| 50 | 2075 |
| 100 | 2782 |
| 200 | 2723 |
| 500 | 2752 |

**Table 5.6:** Effect of maximum pending tuples on topology throughput

### 5.4.3 Bolt Execute Latencies

A useful metric that allows us to identify the bottlenecks in our topology is the execute latency of each bolt. Execute latency is the average time a tuple spends in the execute method of a bolt.

We record the execute latencies for each bolt of the topology at maximum throughput and present them in Table 5.7. We also compare their relative sizes in Figure 5.7. It is clear that the tuples spend practically all of their execute time in the IP to DNS Bolt and the Phoenix Bolt. This was to be expected since these bolts perform reads and writes to HBase tables, while the other bolts execute simple commands in memory.

| Bolt | Execute latency (msec) |
|---|---|
| Split Fields Bolt | 0.047 |
| IP to AS Bolt | 0.052 |
| IP to DNS Bolt | 4.779 |
| Phoenix Bolt | 7.784 |

**Table 5.7:** Average execute latency for each bolt of the topology

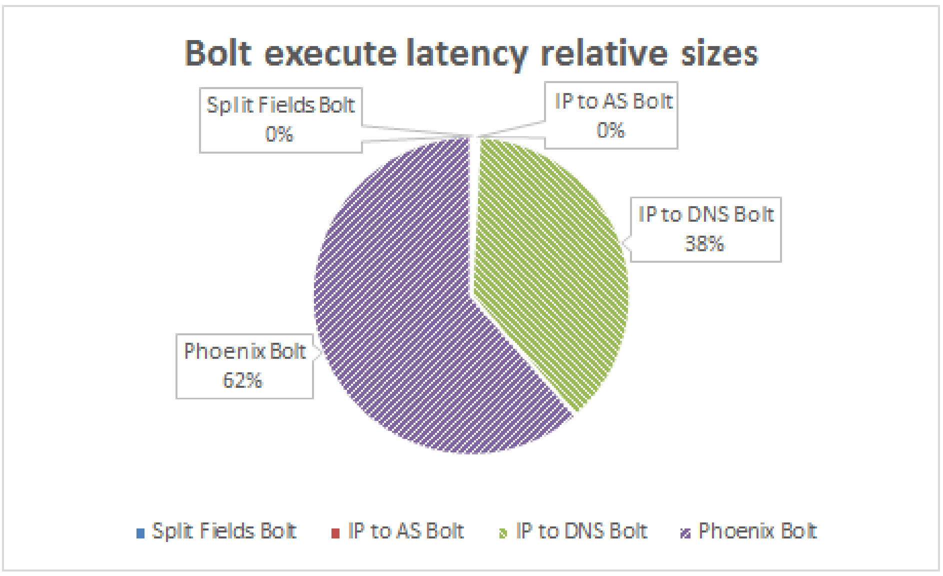


**Figure 5.7:** Relative sizes of execute latencies for the bolts of the topology

### 5.4.4 Total System Latency

An important performance indicator for our system is total system latency, the time it takes for a message to be sent by the Kafka producer to the topic, consumed by the Kafka Spout, processed by the bolts of the topology and eventually be stored in the Phoenix table and made available for queries.

To compute total system latency we feed the topology with real-time messages and query the table for the row with the latest timestamp. By comparing this timestamp to the current time we can measure the total system latency. Complete system latency is measured at **1.161 sec** on average at max topology throughput.

### 5.4.5 Salting Write Performance

HBase sequential write suffers from RegionServer hotspotting since the row key is monotonically increasing. Salting the row key provides a way to balance load among the RegionServers, as we described in Section 4.4.

In this experiment we use a salted and a non-salted table and we compare the throughput of the topology, the write request and CPU utilization on the RegionServers, as well as the execute latency of the Phoenix Bolt. The salted table has 4 salt buckets that are split among the 4 RegionServers of the HBase cluster.

Figures 5.8 and 5.9 demonstrate that salting serves its purpose of eliminating write hostspotting. Whereas all the write requests are directed to a single RegionServer for the non-salted table, the load is evenly distributed for the salted table. Note that higher aggregate CPU utilization while using the salted table is linked to better use of the cluster's resources, leading to higher topology throughput.

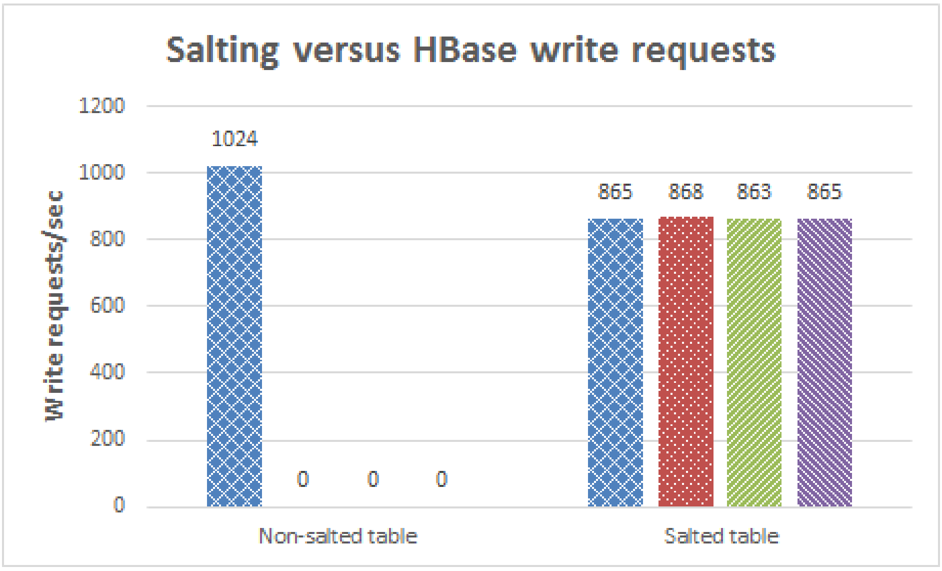


**Figure 5.8:** Salting effect on HBase write request distribution

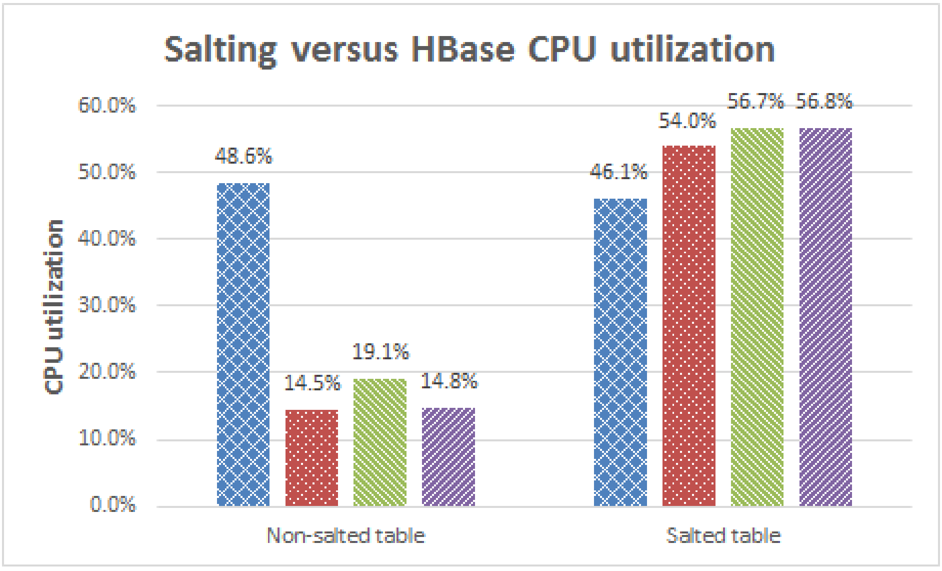


**Figure 5.9:** Salting effect on HBase cluster CPU utilization

Salting also decreases the Phoenix Bolt's execute latency by **74%**, as we can see in Figure 5.10. The execute latency of the Bolt when writing to the non-salted table was increased due to the strain put on the RegionServer that handled all the write requests.

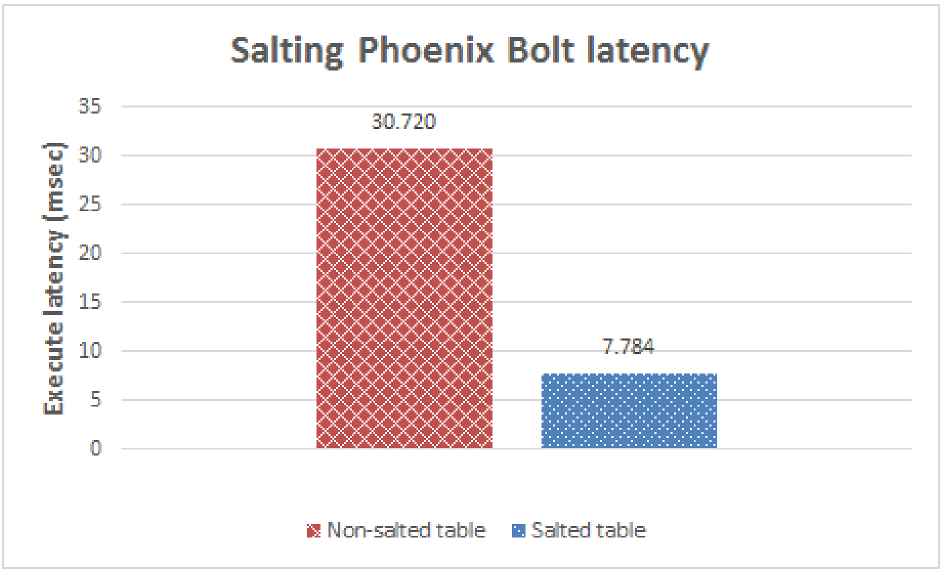


**Figure 5.10:** Salting effect on Phoenix bolt latency

Finally, Figure 5.11 demonstrates that salting massively increases the topology throughput by **140%**.

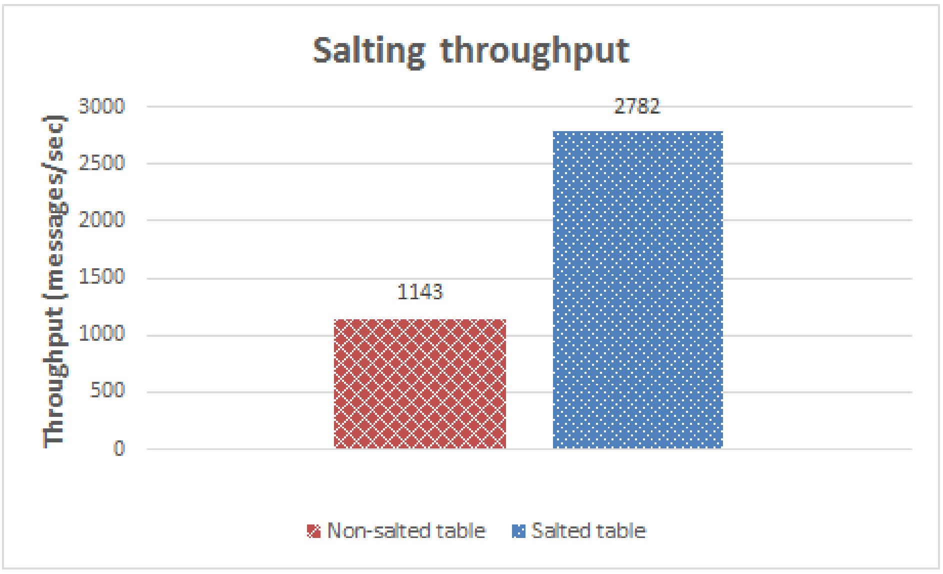


**Figure 5.11:** Salting effect on topology throughput

## 5.5 Storm Scalability

To evaluate the scalability of the topology with Storm and HBase cluster size, we measure the topology's throughput for different cluster sizes. We increase Storm and HBase cluster sizes simultaneously, meaning that on each test there are as many Supervisors as RegionServers. We also adjust accordingly the number of partition for the topic, the component parallelism in the topology and the number of salt buckets for the table. Finally, after any change to the size HBase cluster we distribute the `rdns` table evenly among the RegionServers and compact it for data locality.

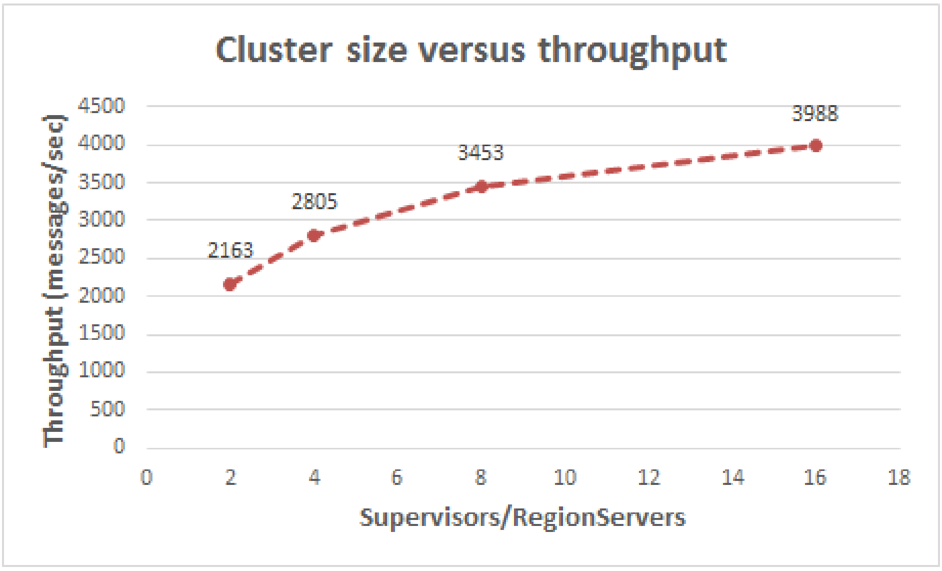


**Figure 5.12:** Topology throughput scalability with Storm and HBase cluster size

The average CPU utilization for the Storm and HBase clusters during the scaling experiments is presented in the following diagram.

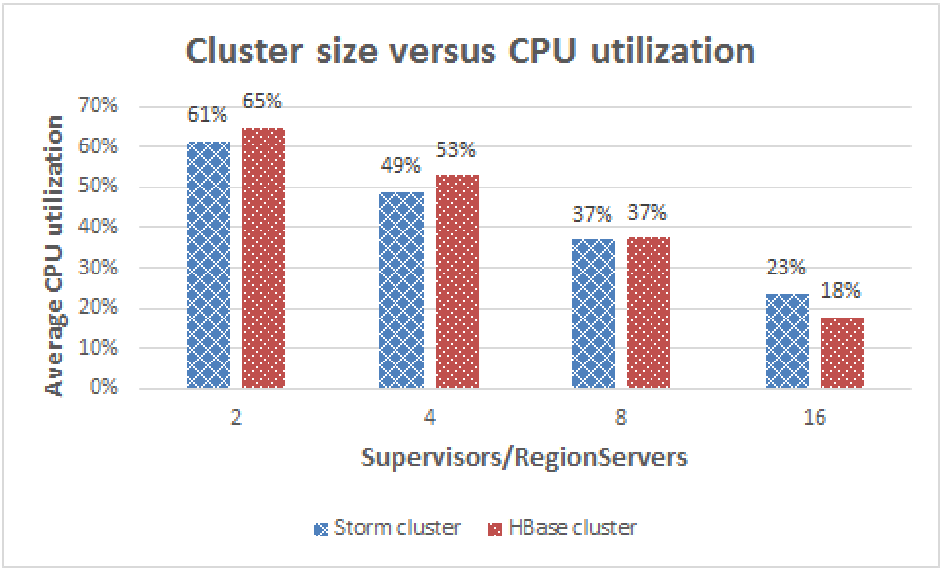


**Figure 5.13:** Average CPU utilization during scalability experiments

We notice that CPUs are underused for larger cluster sizes, which indicates that disk I/O throughput does not increase proportionally with cluster size in our environment. This explains the diminishing increases in throughput as we increase the cluster size.

## 5.6 HBase and Phoenix Performance Tuning

The comparison basis of the following benchmarks is our final Phoenix table, after all optimizations. The table uses Snappy compression and no data block encoding, is split in 3 column families and is salted in 4 buckets. All the tables are compacted and their regions are distributed evenly among the RegionServers. Queries are performed over 10 million rows that are already cached in the BlockCache, unless stated otherwise.

We perform the following two types of queries:

```sql
SELECT COUNT(*) FROM TABLE netdata;
```

The *count* query iterates over the rows of the default column family. Is useful to measure read performance.

```sql
SELECT as.asS, as.asD, COUNT(*) AS pairCount
FROM netdata
GROUP BY as.asS, as.asD
ORDER BY pairCount DESC
LIMIT 10;
```

The *topN AS* query returns the top 10 AS pairs that exchanged packets in this table. We also perform the *topN DNS* alternative on some benchmarks, however it is more computationally intensive, since the `GROUP BY` clause creates many more distinct pairs for domain names than for autonomous systems. This leads to significantly bigger sets that have to be sorted and subsequently larger query latency.

### 5.6.1 HDFS Short-Circuit Local Reads

As we described on Section 4.1, when HDFS short-circuit local reads are enabled the RegionServer reads local data directly from the disk instead of going through the DataNode. This speeds up data transfer from the disk to the BlockCache when the data is local.

In this experiment we perform a count query over 1 million rows, at first with HDFS short-circuit local reads enabled and afterwards disabled. We measure the total query latency, which includes data transfer time to the BlockCache as well as query processing time.

When HDFS short-circuit local reads are enabled total query time is reduced by **62%**, as we can see in Figure 5.14.

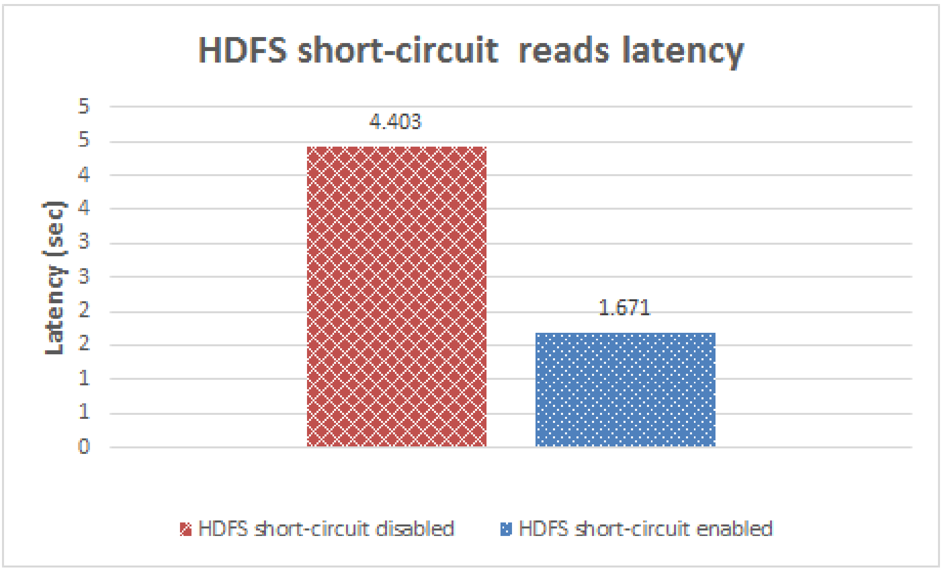


**Figure 5.14:** Enabling HDFS short-circuit for local reads effect on count query latency

### 5.6.2 Compression and Data Block Encoding

Compression and data block encoding can be used to reduce on-disk data size as well as in-cache data size, as we described in Section 4.2. However this comes with a performace hit for decompression, decoding or both when reading the cached data.

In our experiment we compare on-disk size and in-cache query latency forthe following tables:

- The first table has Fast Diff encoding enabled for all of its column families.
- The second table has Snappy compression enabled for all of its column families. Compressed BlockCache is enabled.
- The last table has both Fast Diff encoding enabled and Snappy compression enabled for all of its column families. Compressed BlockCache is enabled.

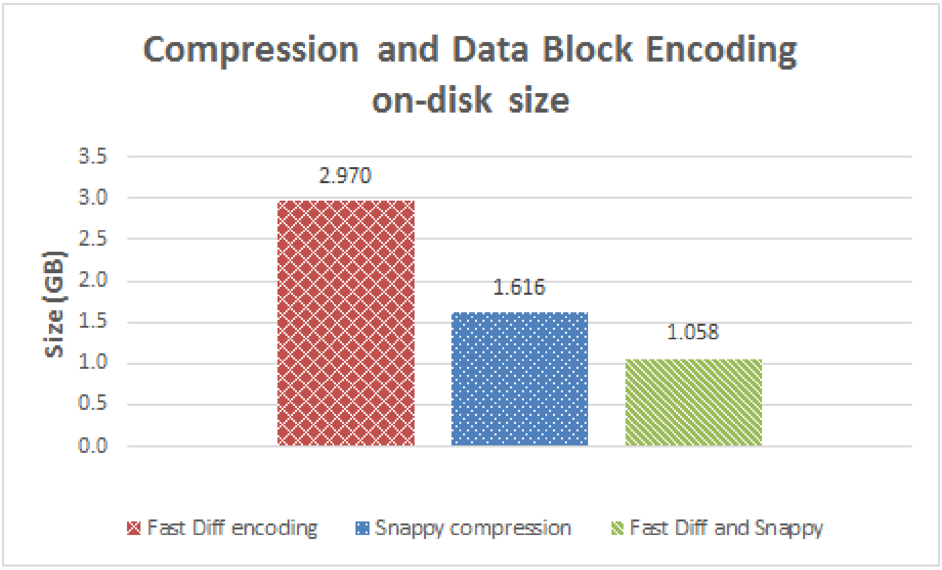


**Figure 5.15:** Compression and data block encoding effect on the on-disk size of a 10 million row table

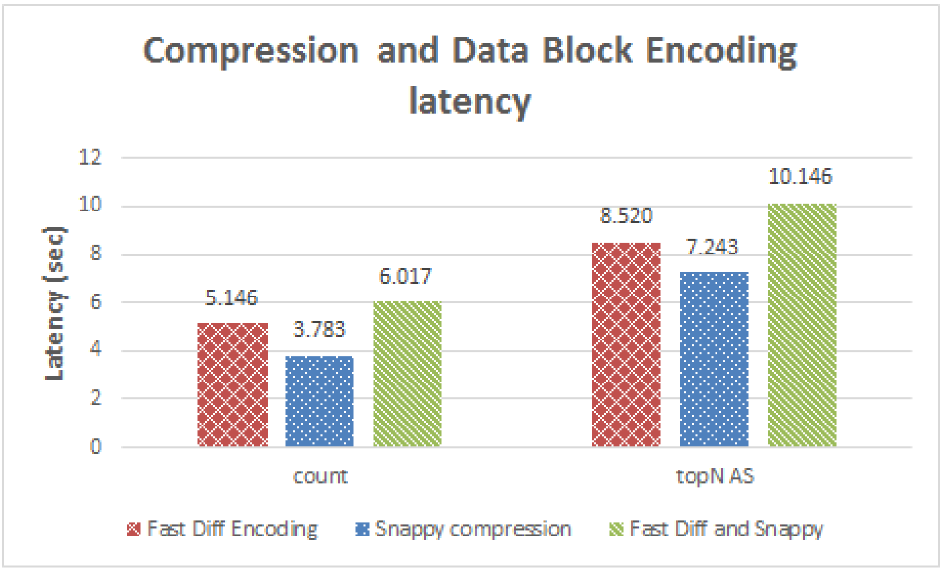


**Figure 5.16:** Compression and data block encoding effect on query latency

As we can see in Figure 5.15, using both compression and data block encoding reduces the data size further than the other options. Reduced data size allows more rows can me cached at the same time and reduces data transfer time from the disk to the BlockCache.

However the best in-cache query latency is achieved by compression alone. The data size difference between the second and the third tables is not big enough to outweigh the query latency advantage of the compressed table. This is the reason why we chose compression and no data block encoding for our final table.

### 5.6.3 Number of Column Families

We compare query performance between our final table, that includes three column families (default, as, dns), and the table containing the same data in one column family. Data is cached by column family, which means that count queries only cache the default family and topN AS queries cache only the as family.

Total size of the final table is divided between the three column families with the percentages shown in Figure 5.17.

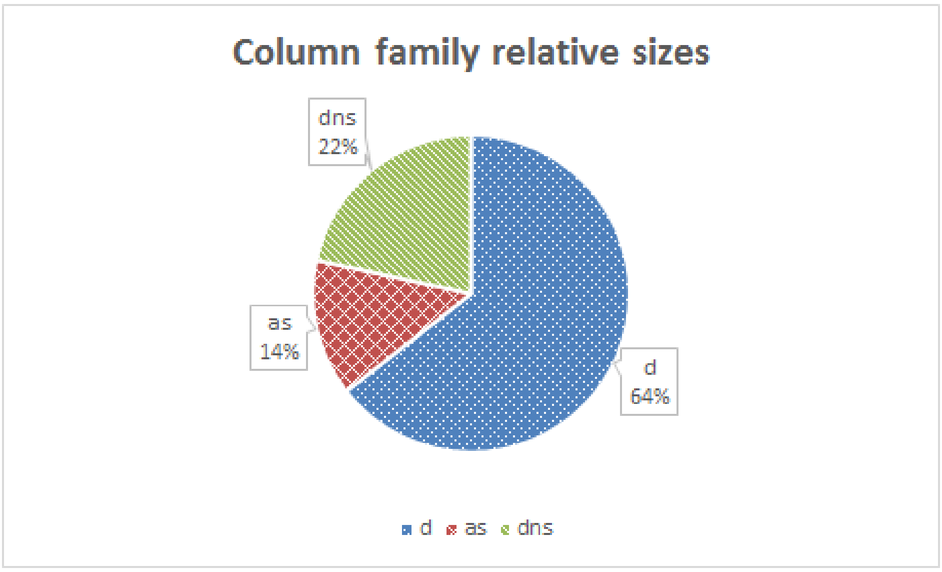


**Figure 5.17:** Relative sizes of the three column families of the table

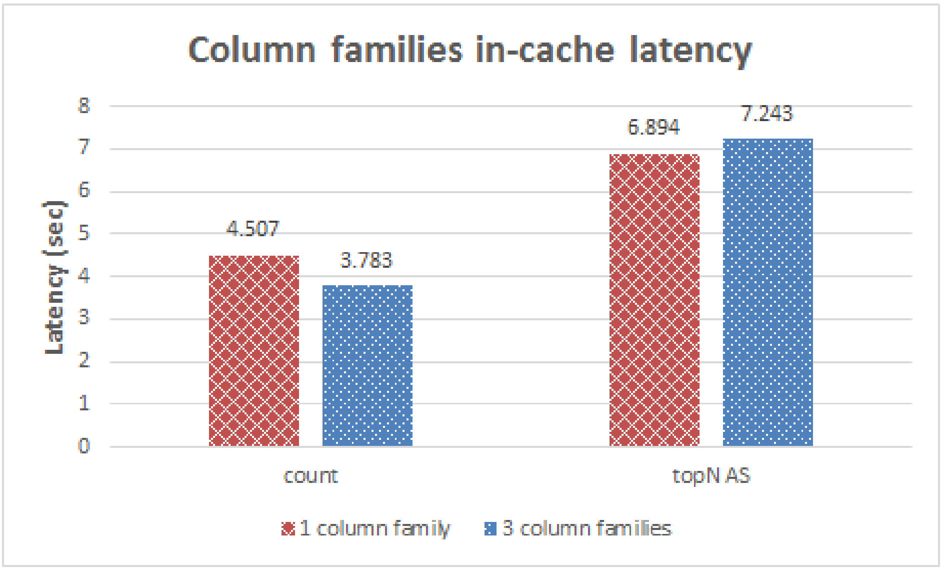


**Figure 5.18:** In-cache query latency for tables with different column family setups

Even though the tables seem to have similar performance we must remember that the latencies on the diagram are measured on already cached data. The real difference is presented on the next diagram where complete query time is measured for 1 million row queries.

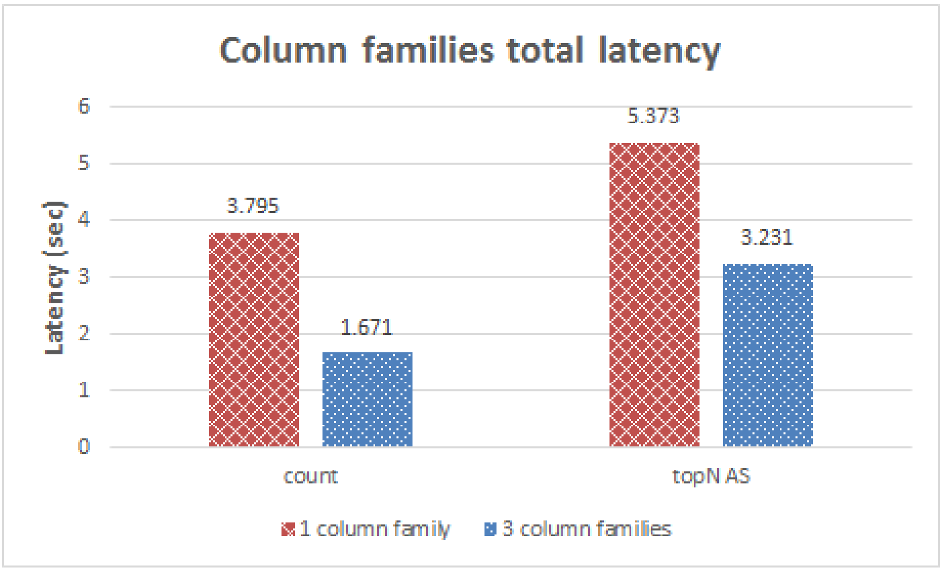


**Figure 5.19:** Total query latency for tables with different column family setups

### 5.6.4 Salting Read Performance

As we mentioned in Section 4.4, aside from write throughput salting can also improve read throughput. Phoenix scans the salted data, sorted within each bucket, in parallel and merge-sorts them in the Phoenix client.

In this experiment we perform a count and a topN AS query on a non-salted and a salted table and compare the query latencies. Salting speeds up count and topN AS queries by **68%**, as we can see in Figure 5.20.

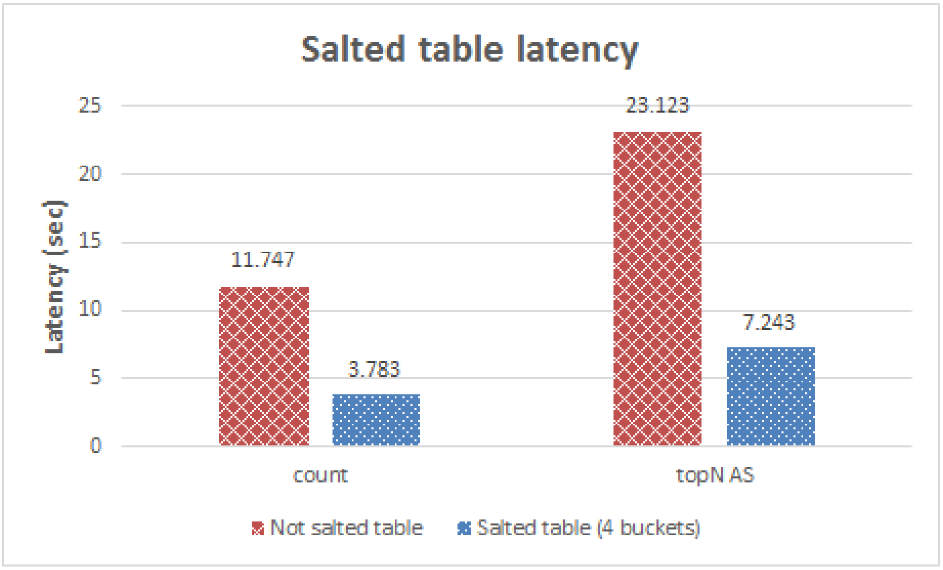


**Figure 5.20:** Salting effect on query latency

## 5.7 HBase and Phoenix Scalability

### 5.7.1 Table Rows

To evaluate the scalability of our table with the size of the data, we measure the query latency for tables with different numbers of rows.

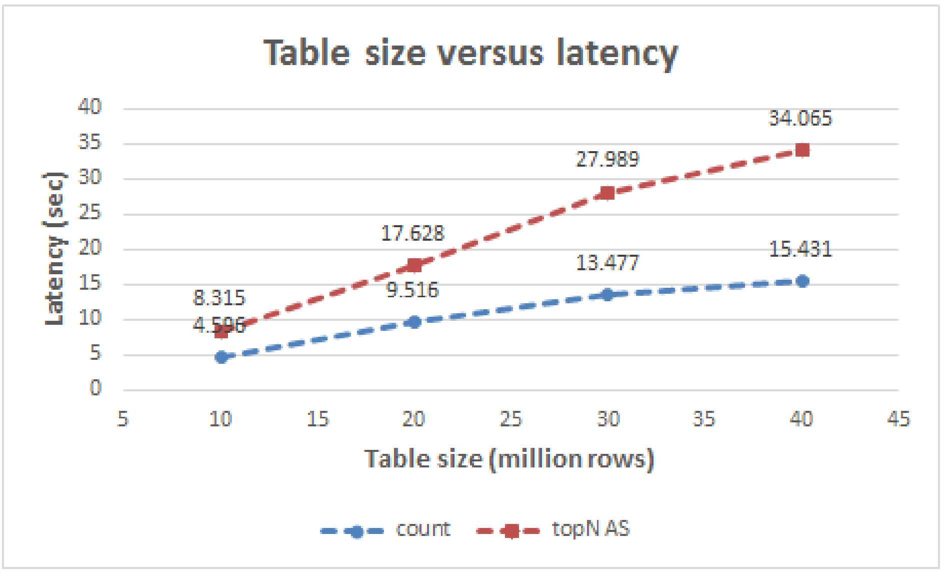


**Figure 5.21:** Count and topN AS query latency scalability with table size

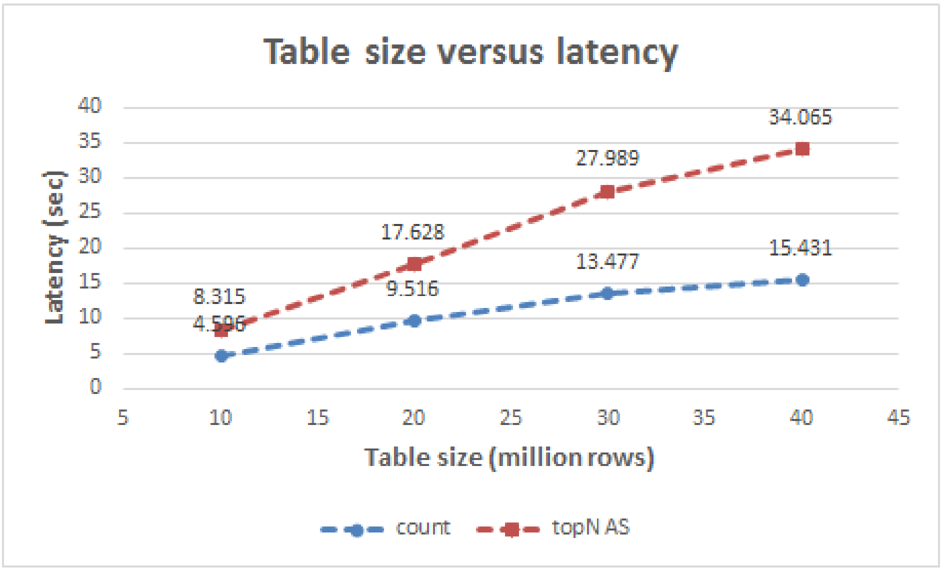


**Figure 5.22:** TopN DNS query latency scalability with table size

### 5.7.2 HBase Cluster Size

To evaluate the scalability of our table with the HBase cluster size, we measure the query latency for clusters with different numbers of RegionServers. The number of the table's salt buckets is adjusted according to the number of the RegionServers.

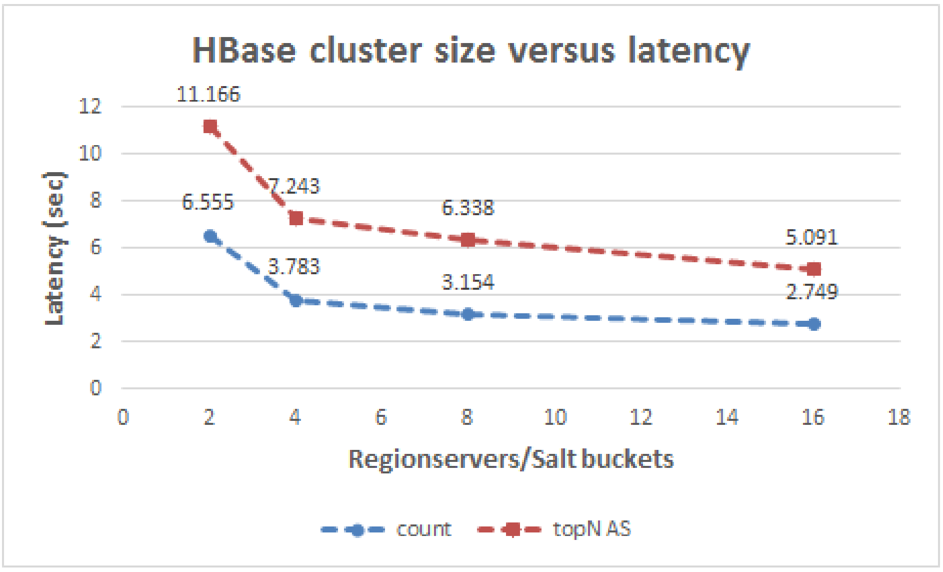


**Figure 5.23:** Query latency scalability with HBase cluster size

### 5.7.3 Multiple Simultaneous Queries

In this experiment we perform simultaneously the same query from multiple Phoenix clients and measure the average query latency. The clients are running on different machines.

Figure 5.24 shows that multiple queries performed at the same time from different have an additive impact on the average query latency. Since reducing query latency is our priority, multiple simultaneous queries should be avoided.

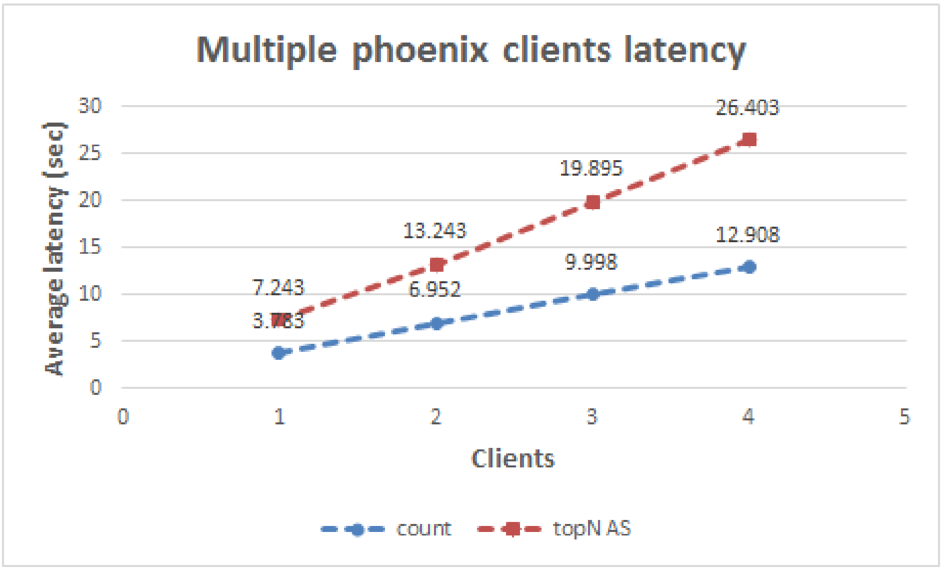


**Figure 5.24:** Query latency scalability with the number of Phoenix clients

Chapter 6

# Conclusion

## 6.1 Concluding Remarks

TODO

## 6.2 Future Work

TODO
Compare Trident, Spark, Shamza implementations
Phoenix alternatives (SQL on HBase)

# Bibliography

[1] Geolite database. http://dev.maxmind.com/geoip/legacy/geolite.