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 Σ. ΤΟΥΛΟΥΠΑΣ

Επιβλέπων : Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Αθήνα, Νοέμβριος 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 Σ. ΤΟΥΛΟΥΠΑΣ

Επιβλέπων : Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 2η Νοεμβρίου 2015.

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

Αθήνα, Νοέμβριος 2015

Γεώργιος Σ. Τουλούπας

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

Περίληψη

Στην παρούσα διπλωματική, σχεδιάζουμε και υλοποιούμε ένα κατανεμημένο σύστημα το οποίο επιτρέπει την εκτέλεση SQL ερωτημάτων που πραγματοποιούν συνένωση μιας ροής δεδομένων πραγματικού χρόνου και ενός εξωτερικού συνόλου δεδομένων. Η περίπτωση χρήσης για την οποία υλοποιούμε αυτό το σύστημα είναι η εκτέλεση topN SQL ερωτημάτων που πραγματοποιούν συνένωση μιας ροής δικτυακών δεδομένων πραγματικού χρόνου, που παράγεται από δειγματοληψία κίνησης ενός IXP, και εξωτερικών συνόλων δεδομένων που περιλαμβάνουν Autonomous System και DNS πληροφορίες.

Για να επιτύχουμε χαμηλό χρόνο απόκρισης στα ερωτήματα, η συνένωση πραγματοποιείται σε πραγματικό χρόνο χρησιμοποιώντας το Storm processing framework και η αποκανονικοποιημένη ροή δεδομένων αποθηκεύεται σε ένα Phoenix table, επιτρέποντας έτσι σε όλα τα επόμενα ερωτήματα να εκτελούνται χωρίς να χρειάζεται ξανά ο υπολογισμός της συνένωσης κατά το χρόνο εκτέλεσης. Το σύστημα χρησιμοποιεί τις κατανεμημένες τεχνολογίες Kafka, Storm και HBase, οι οποίες εξασφαλίζουν την κλιμακωσιμότητά του και την ανοχή του σε σφάλματα. Επιπλέον, το Storm προσφέρει επεκτασιμότητα στο σύστημα επιτρέποντάς μας να προσθέσουμε με εύκολο τρόπο νέα εξωτερικά σύνολα δεδομένων κάθε μεγέθους, τα οποία συνενώνονται με τη ροή δικτυακών δεδομένων.

Επιπρόσθετα, εφαρμόζουμε ένα συνδυασμό βελτιστοποιήσεων στο HBase cluster και στο Phoenix table, οι οποίες μειώνουν ακόμα περισσότερο το χρόνο απόκρισης των ερωτημάτων. Τέλος, αξιολογούμε την επίδοση διαφόρων παραμέτρων του συστήματος και πειραματιζόμαστε με την κλιμακωσιμότητα του συστήματος.

Λέξεις κλειδιά

Επεξεργασία σε Πραγματικό Χρόνο, Ανάλυση Δικτυακών Δεδομένων, Κατανεμημένα Συστήματα, Kafka, Storm, Hadoop, HBase, Phoenix

Abstract

In this thesis, we design and implement a distributed system that allows the execution of low latency SQL queries that join a real-time data stream and an external dataset. The use case for which we implement this system is the execution of topN SQL queries that join a real-time network data stream, generated by sampling IXP traffic, and external datasets containing Autonomous System and DNS information.

To achieve low query latency, the join is performed in real time using the Storm processing framework and the denormalized data stream is stored at a Phoenix table, allowing all subsequent queries to be performed without the need to compute the join on query time. The system utilizes distributed technologies such as Kafka, Storm and HBase, which ensure its scalability and fault tolerance. Moreover, Storm provides extensibility to the system, allowing us to easily add more external datasets of any size that are joined with the network data stream.

We also apply a combination of optimizations to the HBase cluster and the Phoenix table that further reduce query latency. Finally, we evaluate the performance of the system for various parameters while tuning and applying optimizations, and experiment with the system's scalability.

Key words

Real-time Processing, Network Analytics, Distributed Systems, Kafka, Storm, Hadoop, HBase, Phoenix

Ευχαριστίες

Με την παρούσα διπλωματική εργασία ολοκληρώνεται ένα σημαντικό κεφαλαίο της ακαδημαϊκής μου πορείας. Στο σημείο αυτό θα ήθελα να ευχαριστήσω τα πρόσωπα που με βοήθησαν σε αυτή τη διαδρομή.

Αρχικά θα ήθελα να ευχαριστήσω θερμά τον καθηγητή μου Νεκτάριο Κοζύρη, για τη δυνατότητα που μου έδωσε με αυτό το θέμα να ασχοληθώ σε βάθος με τον τομέα των κατανεμημένων συστημάτων.

Ιδιαίτερα θα ήθελα να ευχαριστήσω τον μεταδιδακτορικό ερευνητή Γιάννη Κωνσταντίνου για την βοήθεια και την καθοδήγηση του κατά την εκπόνηση της παρούσας εργασίας.

Επίσης θα ήθελα να ευχαριστήσω τους φίλους μου, συμφοιτητές και μη, που ομόρφυναν τα χρόνια της φοιτητικής μου ζωής.

Τέλος, θα ήθελα να ευχαριστήσω την οικογένειά μου για την υπομονή και την κατανόησή τους, καθώς και για την συνεχή στήριξη που μου παρείχαν όλα αυτά τα χρόνια.

Γεώργιος Σ. Τουλούπας,

Αθήνα, 2η Νοεμβρίου 2015

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Over the past decades, the Internet is continuously growing, driven by ever greater amounts of online information and knowledge, commerce, entertainment and social networking. Recent studies forecast that global Internet traffic will grow with a compound annual growth rate of 26% over the next years, reaching 136.1 Exabytes per month in 2019, up from 42.4 Exabytes per month in 2014 [10]. The key elements that will shape Internet traffic in the coming years include the increase in the number of the Internet users, the proliferation of networked devices such as tablets and smartphones, faster broadband speeds, advanced video services and increased IP traffic deriving from cellular data connections [11].

Large portions of the Internet traffic are routed through Internet Exchange Points (IXPs). An IXP consists of one or more network switches, to which Internet Service Providers (ISPs) connect and exchange Internet traffic between their networks. The IXP allows these networks to interconnect directly, rather than through their upstream transit providers, thereby reducing costs, latency and bandwidth.

Recent studies have shown that large IXPs have visibility to a large fraction of the Internet and fit the role of being global Internet vantage points [32]. Therefore, one can extract information about the global state of the Internet by analyzing the traffic of a large IXP over a sufficient period of time. The typical approach to perform network traffic analysis on a large IXP is by sampling the traffic over a period of time and saving the capture in a file. Then the capture is processed in a centralized manner by a script, where the network traffic analysis is performed. This approach has two main drawbacks. From the one hand it does not scale for a larger amount of network data. From the other hand processing offline network traffic captures limits the “freshness” of the data.

Over the past years, a variety of distributed technologies and frameworks have been developed to process and store big data [2, 39, 3]. Distributed technologies such as Kafka, Storm, HDFS, HBase and Phoenix can be used to implement scalable systems that process and analyze data streams in real time. By using such technologies for processing and analyzing the IXP network traffic, the issues mentioned in the previous paragraph can be alleviated.

1.2 Objectives

The objective of this thesis is the design and implementation of a distributed system that allows the execution of SQL queries that join a real-time data stream and an external dataset. The top priority of the system is minimizing the execution latency of these queries, which contains two sub-objectives. From the one hand, the latency between the issue of the query and the moment we receive the query response must be minimized. From the other hand, the delay between the data generation and the moment they are available for querying must also be as small as possible, since we are dealing with a real-time data stream.

The rest of the prerequisites for the design and the implementation are the scalability, fault tolerance and extensibility of the system. All of the system's components should use distributed technologies to ensure scalability with the data stream throughput. Moreover, the technologies used should provide fault tolerance, since the system will be constantly running over extended periods of time, processing real-time data. Finally the system should be extensible, by allowing additional external datasets of any size to be joined with the data steam without the need for drastic changes in the implementation.

The use case for which we implement this system is the execution of SQL queries that join a real-time network data stream, generated by sampling IXP traffic, and external datasets containing Autonomous System and DNS information. The network data stream contains useful fields extracted from the headers of the sampled packet, whereas the external datasets can map IP addresses to Autonomous Systems and domain names. The critical difference between these two external datasets is their size, which as we will see affects the way the join can be performed. The specific queries that we intend to perform in this use case are topN AS and topN DNS queries, which return the top 10 Autonomous System and domain name pairs respectively for the IXP traffic over a specified time window.

The novelty of this thesis consists in combining state-of-the-art distributed technologies and techniques to minimize the execution latency of SQL queries that join a real-time data stream and an external dataset. We present an implementation that performs the join once during processing and allows subsequent queries to execute without the need to perform it again. Moreover, we provide a way of extending the system by adding external datasets of any size that are joined with the data stream. Finally, we apply a combination of optimizations that increase the system's performance.

1.3 Related Work

Over the recent years, various systems have been proposed to perform network analytics. In the following list we present some of them that are related to this thesis:

- **Datix** [37] is a distributed analytics system for network traffic data that relies on smart partitioning storage schemes to support fast join algorithms and efficient execution of filtering queries. However, Datix is built upon batch processing technologies such as MapReduce and Hive, which limits its scope to offline data processing.

- **Bro** [24] is a network monitoring framework that can be used for collecting and analyzing real-time network traffic. A Bro cluster can be deployed to achieve scalability. Unfortunately Bro does not integrate a storage solution for the processed data.

- **DBStream** [28] is a real-time network traffic monitoring system which allows fast and flexible analysis across multiple data sources. It is based on the Data Stream Warehousing paradigm, which provides the means to handle both real-time and historical data. The crucial drawback of DBStream is that it is lacking scalability.

- **CellIQ** [35] is a real-time cellular network analytics system that supports complex analysis tasks. This system is not fit for our IXP traffic use case, because it is optimized for cellular network analytics, by leveraging the spatial and temporal locality cellular network data.

- **FCCE** [38] is a distributed, low latency key-value data management system. It is optimized to extract, store, retrieve, and correlate features from diverse data sources, including real-time data streams. While it can be used for our use case, FCCE does offer SQL support and only provides put and get operations similar to those of HBase.

1.4 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.

In Chapter 3 we describe the system's design and implementation. We provide a high-level overview of the system and its characteristics, followed by detailed information for its components, including the data generation and input part, the Kafka topic, the Storm topology and the Phoenix table.

In Chapter 4 we present the optimizations that we apply on the HBase cluster and the Phoenix table to increase the system's performance. More specifically, we describe the effects of HDFS short-circuit local reads, compression and data block encoding, disabling BlockCache on the Reverse DNS table and salting.

In Chapter 5 we evaluate the performance of the system. Firstly we describe the datasets used, as well as the evaluation cluster. Next, we perform experiments to evaluate the performance and the scalability of the Kafka, Storm, HBase and Phoenix components of the system.

In Chapter 6 we provide some concluding remarks as well as propositions for future work on the system.

Chapter 2

Theoretical Background

2.1 Kafka

Apache Kafka [36, 5] 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 [9].
- A *consumer* is a process that subscribes to one or more topics and processes the feed of published messages.

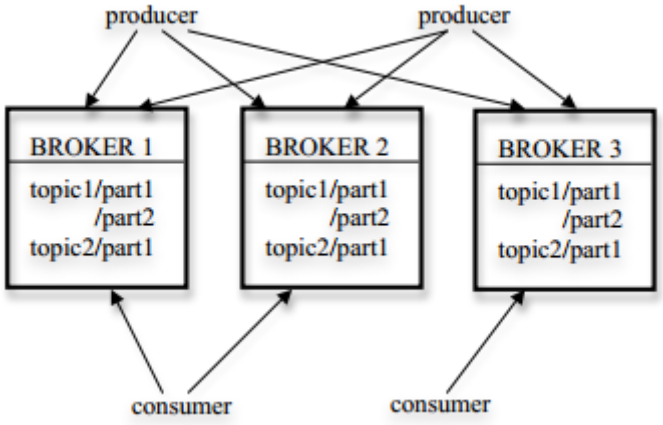

Figure 2.1: Kafka architecture

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.

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

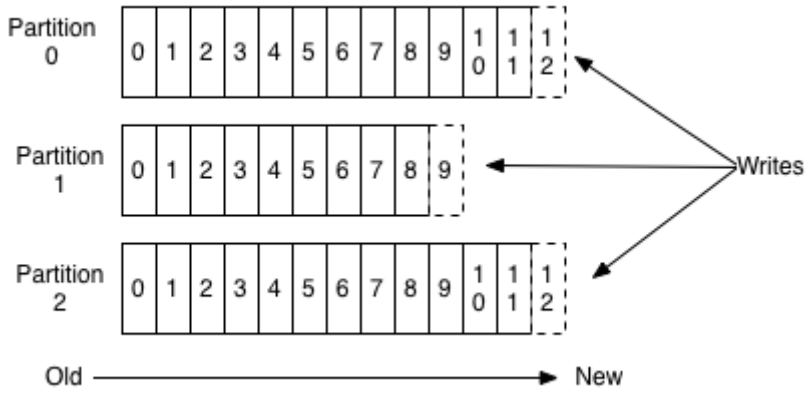

Figure 2.2: Kafka topic structure

offset of the consumer in the topic. 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 topic serve several purposes. Firstly, they allow the topic 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 for the write and read requests of the producers and the consumers respectively.

2.2 Storm

2.2.1 Introduction

Apache Storm [39, 8] is a real-time fault-tolerant 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 the 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 failures or performance issues associated with it.

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 the execution 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.

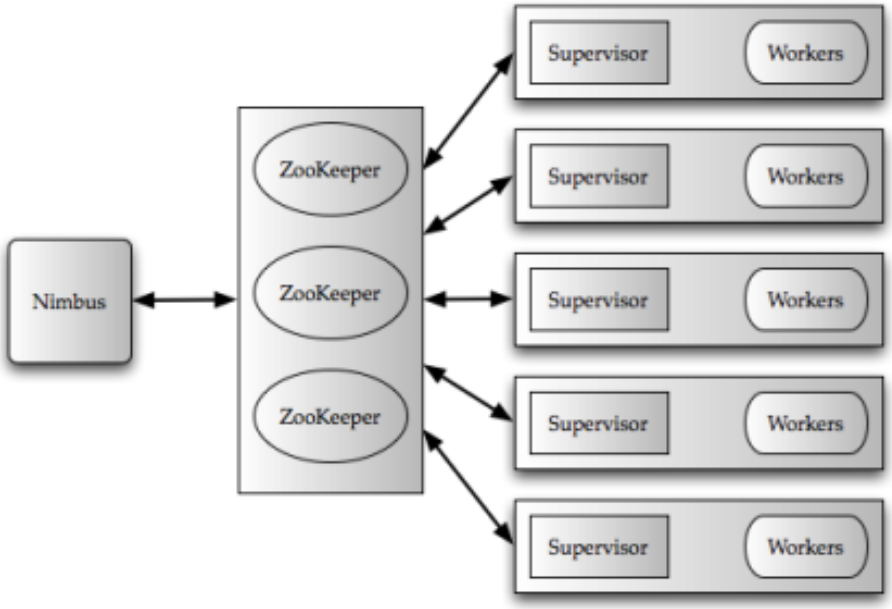

Figure 2.3: Storm architecture

All coordination between Nimbus and the Supervisors is done through a Zookeeper cluster. 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 queuing 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.

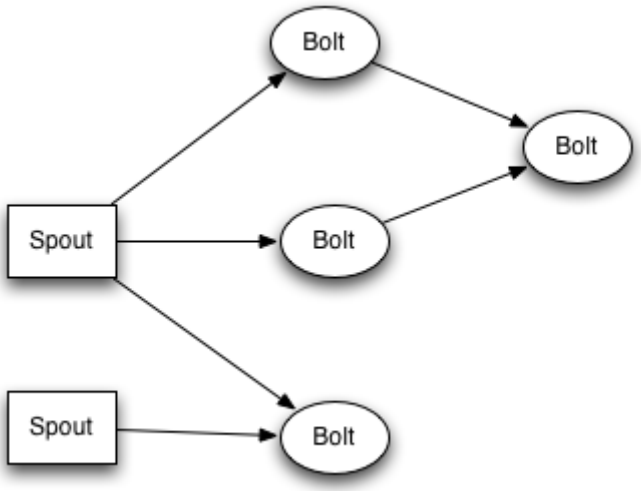

Figure 2.4: Example storm topology

Each component in a Storm topology (spout or bolt) 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 [26]. The relationships between them are illustrated in Figure 2.5.

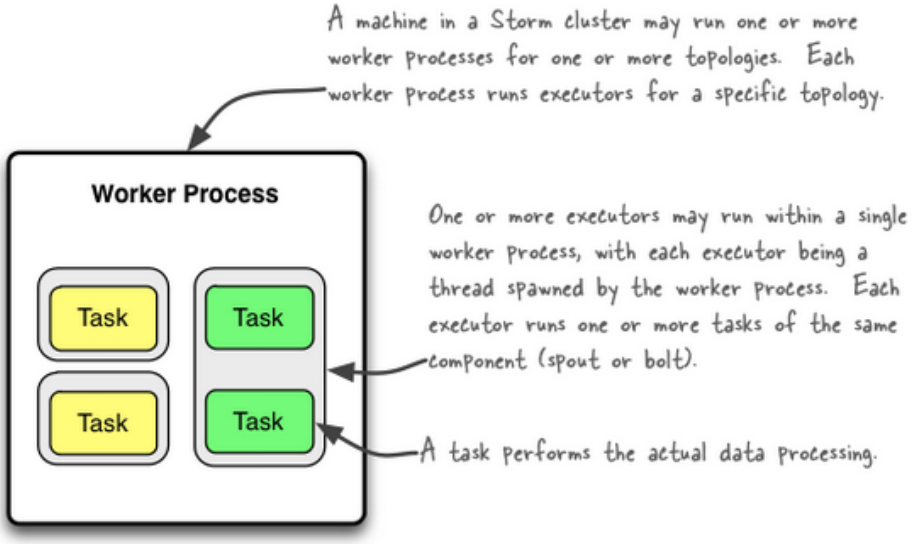

Figure 2.5: The relationships between worker processes, executors and tasks

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 which 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.

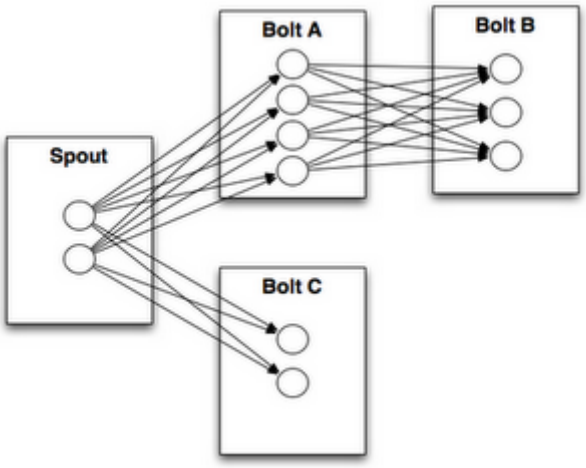

Figure 2.6: Task-level execution of a topology

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 Distributed File System

The Hadoop Distributed File System (HDFS) [2, 30] is a distributed file system designed to run on commodity hardware, inspired by the Google File System [33]. It can reliably store very large files across machines in a large cluster and provides high throughput access to large data sets. The 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 for the last block. Blocks belonging to a file are replicated for fault tolerance. The block size and replication factor are configurable per file.

The HDFS has a master/slave architecture. An HDFS cluster consists of a single NameNode and one DataNode per node in the cluster. The *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. The 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.

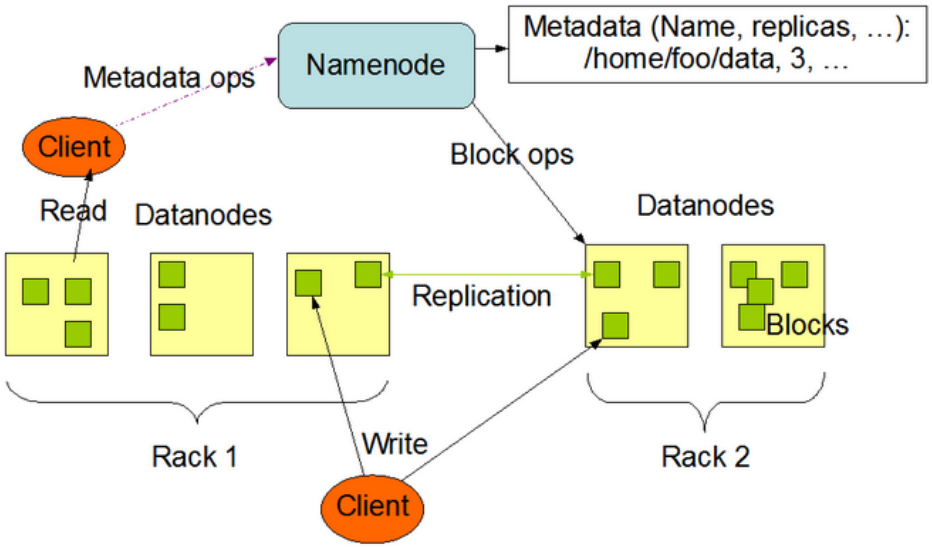

Figure 2.7: HDFS architecture

2.4 HBase

2.4.1 Introduction

Apache HBase [3] is a distributed non-relational database modeled after Google's BigTable [31], 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 redistributed 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.

- **BlockCache and Bloom filters:** HBase supports a BlockCache 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 [31], 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 co-locate 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.

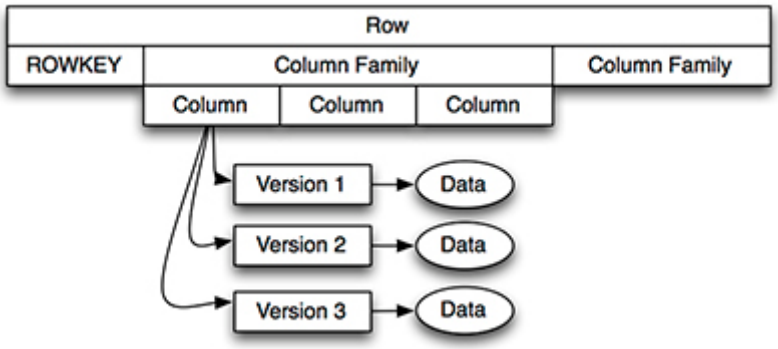

Figure 2.8: HBase data model

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 compaction of the table.

2.4.3 HBase Architecture

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

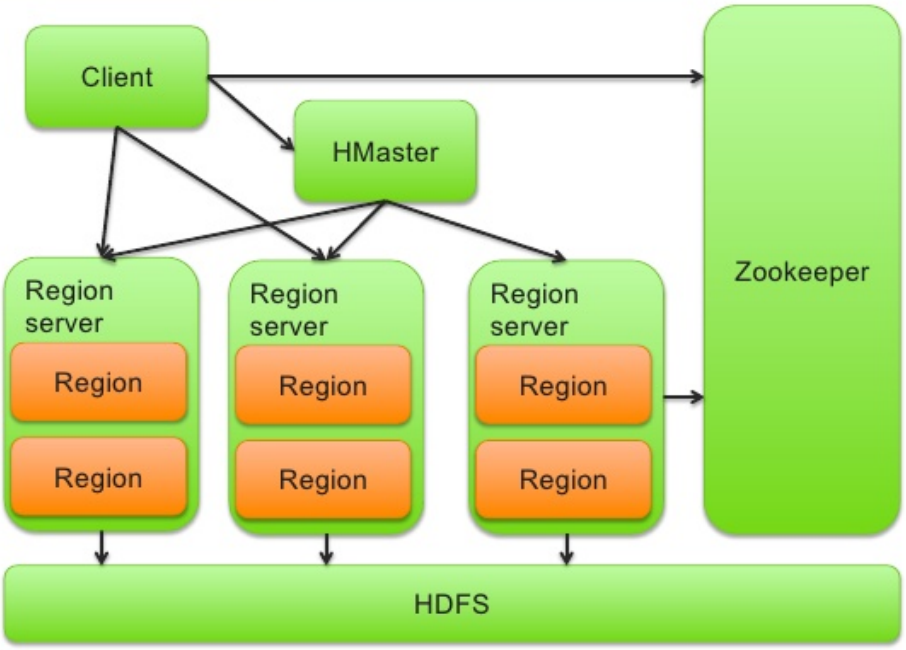

Figure 2.9: HBase architecture

An HBase cluster is composed of two types of servers in a master/slave type architecture [1]. 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:** The Write Ahead Log stores new data that has not yet been persisted to permanent storage. It 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.

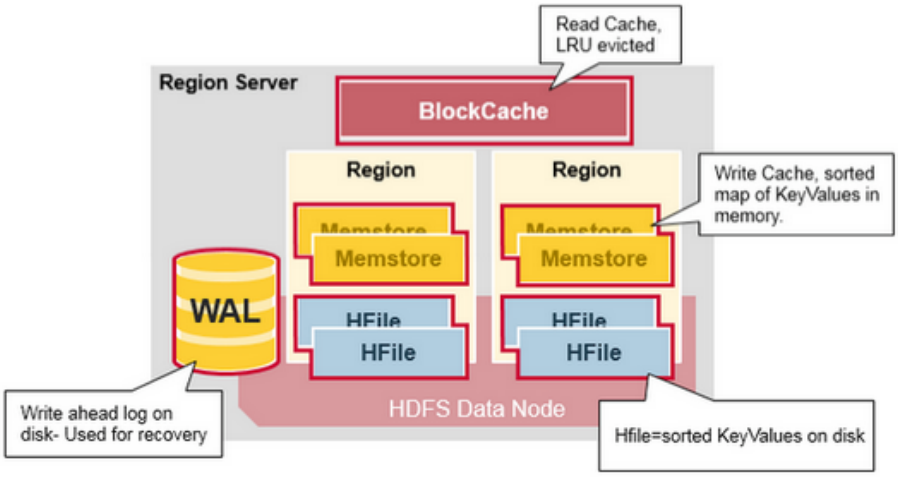

Figure 2.10: RegionServer components

2.5 Phoenix

2.5.1 Introduction

Apache Phoenix [6] 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-side, 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. Copro-

cessors perform operations on the server-side, thus minimizing client/server data transfer and custom filters prune data as close to the source as possible.

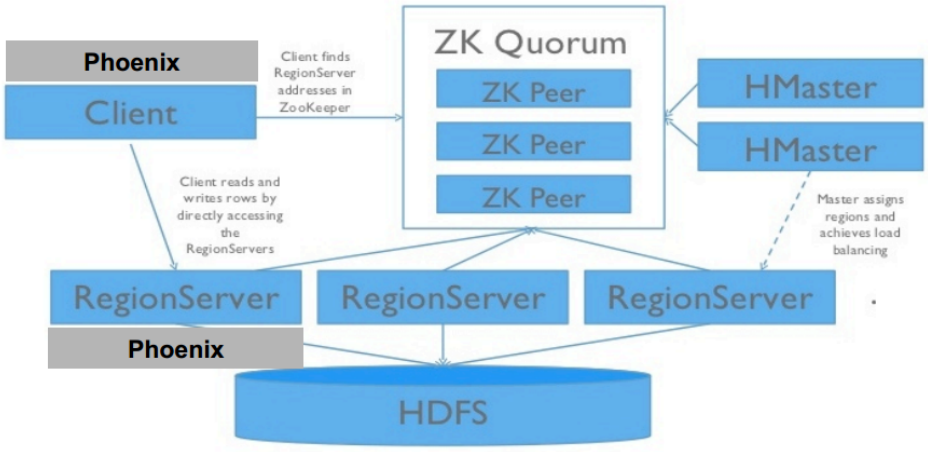

Figure 2.11: Phoenix and HBase architecture

2.5.4 TopN Queries

TopN queries return the top N rows, where top rows are 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 tableName
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 that satisfy 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 the approximate algorithm described below.

Firstly, the Phoenix client issues parallel scans filtering rows according to 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 partial top N rows for each chunk, does a final merge sort and returns the top N rows requested by the query.

Chapter 3

System Description

3.1 System Overview

As mentioned before, the objective of this thesis is the design and implementation of a distributed 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 scalability, fault tolerance, extensibility, but most importantly enabling the execution of low latency SQL queries. This means that the latency between the issue of the query and the moment we receive the query response must be minimized. The delay between the data generation and the moment they are available for querying must also be as small as possible, since we are dealing with a real-time data stream.

Performing an SQL join combines records from two tables. The join effectively creates a third table which combines the information from both of them. Performing a join can be expensive in terms of the time it takes to compute it, especially if one or both of the tables are large in size. Since minimizing the join query latency is our priority, we can store the stream of data combined with the external data information in a single denormalized table. *Denormalization* is the process of attempting to optimize the read performance of a database by adding redundant data, an example of which can be seen in Table 3.1. A Storm topology can compute the join of the data stream and the external dataset in real time and store the denormalized data stream at a Phoenix table in HBase.

LastName	DepartmentID
Jones	2
Wagner	1
Gray	1
Draper	3
Nolan	2

(a) Employee table

DepartmentID	DepartmentName
1	Sales
2	Engineering
3	Marketing

(b) Department table

LastName	DepartmentID	DepartmentName
Jones	2	Engineering
Wagner	1	Sales
Gray	1	Sales
Draper	3	Marketing
Nolan	2	Engineering

(c) Denormalized table

Table 3.1: Denormalization example

This design decision allows all subsequent queries that combine the data stream and the external dataset to be performed directly on the denormalized Phoenix table, without the need to perform the computationally expensive join on query time. Denormalization introduces a trade-off, speeding up

reads from queries while slowing down writes to the table, since the join is performed by the Storm topology. However, if the system processing rate can handle the real-time data generation rate, slower writes are not a problem.

From a high level, the system implemented for our IXP network data use case 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 headers of the captured packets and published to the Kafka topic. The second component of the system is the Kafka topic that temporarily 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.

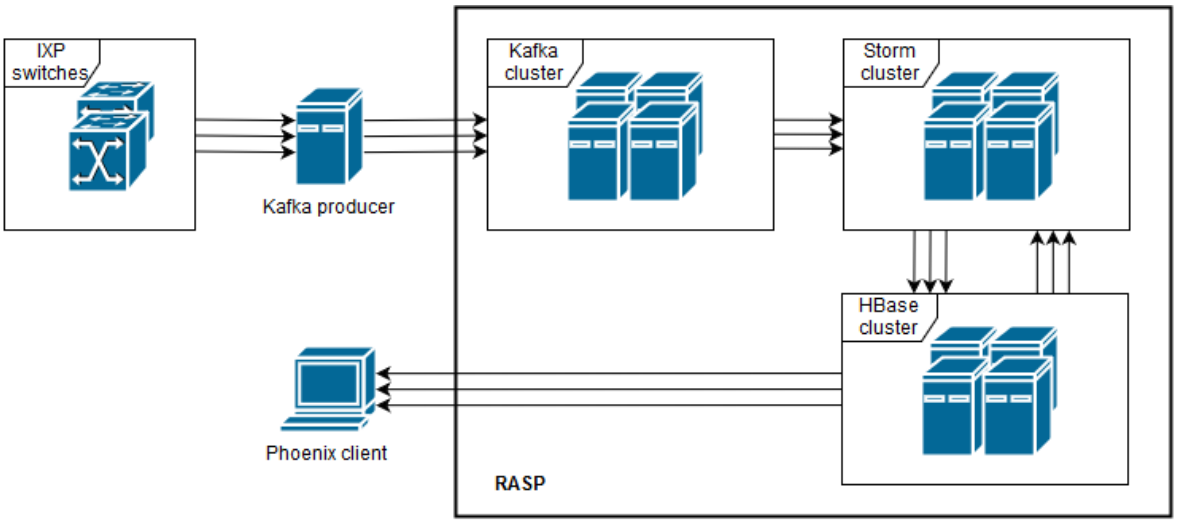

Figure 3.1: Storm architecture overview

The system's **scalability** is achieved by using distributed frameworks and technologies for its 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 are both distributed technologies that 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 for 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 even 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 for a new dataset 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 [21] 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 to the 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 of the switches and makes them available for further processing. In our implementation we use `sflowtool` [22], 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 fields 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 `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 [5]. 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. To ensure 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 of the topic 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 one spout and four 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 denormalized data stream 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 of failure.

The overview of the Storm topology for our network data use case can be seen in Figure 3.2. 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.

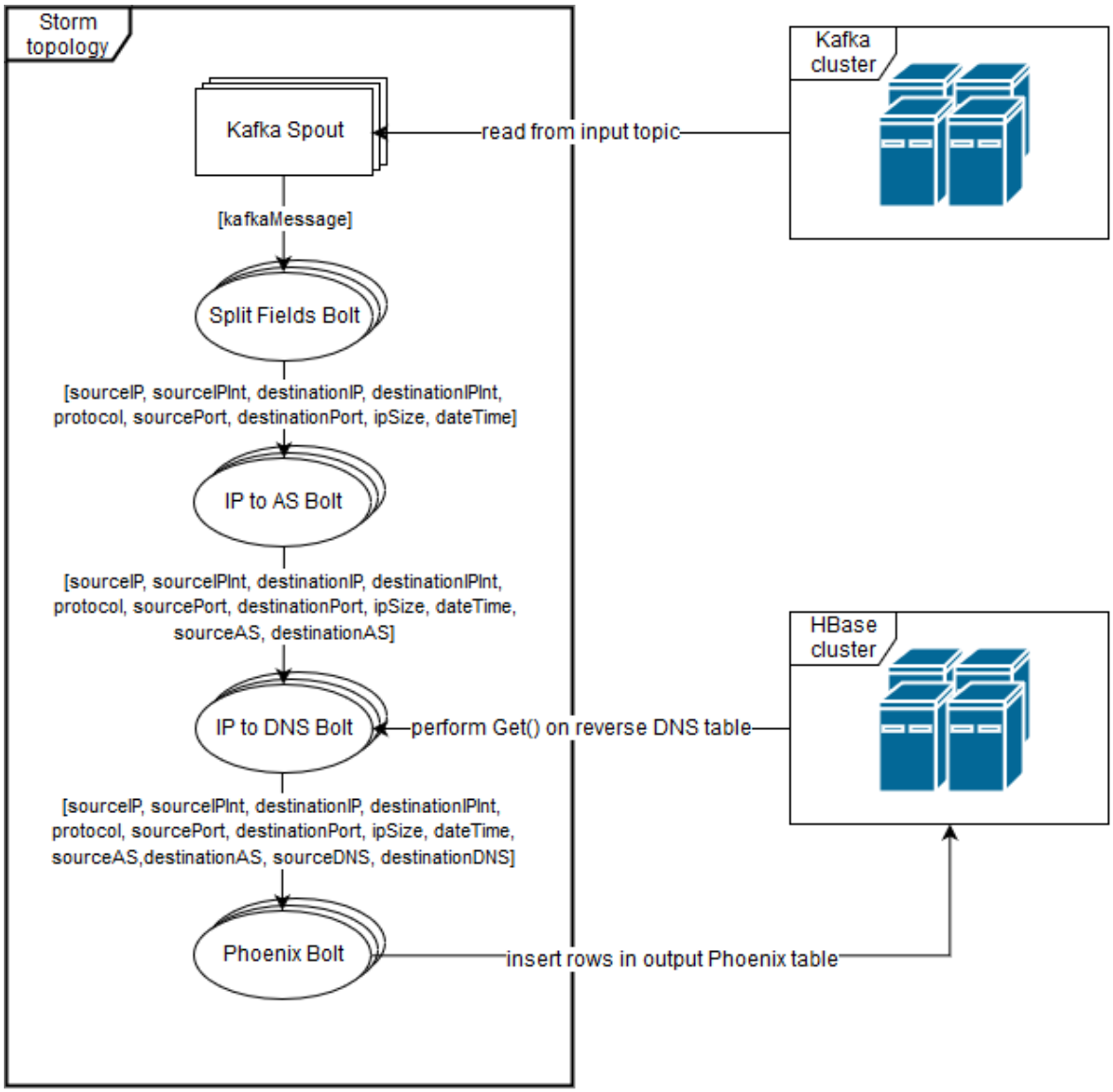

Figure 3.2: Storm topology overview

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 destination IP addresses, which are usually more useful than the IP addresses in dot-decimal notation.

After processing the Kafka message, the Split Fields 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 join of the data stream and the Autonomous System dataset is performed by the IP to AS Bolt. The Autonomous System dataset maps IP address ranges 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 addresses are stored in their integer representation format. The dataset file must be stored in a location accessible by all of the 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, that allow searching in $O(\log n)$ time [12]. For each record of the dataset we insert two records in the TreeMap, containing the start and the stop IP address for each AS along with the AS number and name, as seen in Algorithm 2.

Algorithm 2 IP to AS Bolt

```
1:  function prepare
2:      asMap = new TreeMap<Long, String[]>()
3:      for line in ipToASFile do
4:          fields = line.split(",")
5:          asMap.put(fields[0], [fields[2], "start"])
6:          asMap.put(fields[1], [fields[2], "stop"])
7:      end for
8:  end function
9:  function ipToAS(ipInt)
10:     as = "null"
11:     key = asMap.ceilingKey(ipInt)
12:     if key != null then
13:         value = asMap.get(key)
14:         if (key == ipInt) || (value[1].equals("stop")) then
15:             as = value[0]
16:         end if
17:     end ifreturn as
18: end function
19: function execute(tuple)
20:     sourceIPInt = tuple.getField("sourceIPInt")
21:     destinationIPInt = tuple.getField("destinationIPInt")
22:     outputValues = tuple.getValues()
23:     outputValues.add(ipToAS(sourceIPInt))
24:     outputValues.add(ipToAS(destinationIPInt))
25:     collector.emit(outputValues)
26: end function
```

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 corresponds 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.

3.4.4 IP to DNS Bolt

The 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 `rdns` 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 bolt to perform `Get` operations on the table for an IP address row key to receive the corresponding domain name.

HBase can perform low latency `Get` operations by using *Bloom filters* [29]. 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 that reduces 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 denormalized 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 computed by the Storm topology, the denormalized 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 HBase table is sorted by capture timestamp and the data for any time window are stored sequentially. To achieve this in Phoenix, 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 [4]. 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, transferred or cached) accompanied by its row key, column name and timestamp. Since the table will store millions of cells the column names will be repeated several millions of times in our data [4]. 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 Phoenix SQL statement used to create the final `netdata` table, including the optimizations that will be described in Chapter 4, is the following.

```
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 the HDFS, all reads normally go through the DataNode. When a RegionServer asks the DataNode to read a file, the DataNode reads that file from 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 [17, 18]. 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 in 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 [4]. *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. In addition to that, 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 [23] 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 the compression ratio is 20% to 100% lower.

Regarding data block encoding, Fast Diff [4] is 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.

Fast Diff works similar to Diff encoding, but uses a faster implementation. The most important feature of Diff encoding 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.

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

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 in Subsection 5.6.2.

4.3 Disabling BlockCache on the Reverse DNS Table

The Reverse DNS dataset is stored in the `rdns` HBase table. This table is stored with 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 inspect 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 Phoenix 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 underlying 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 window. 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 [16]. 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. In addition to that, 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 [20, 16]. 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 the following:

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

In this listing, the `newKey` is produced by prefixing the `originalKey` with 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.

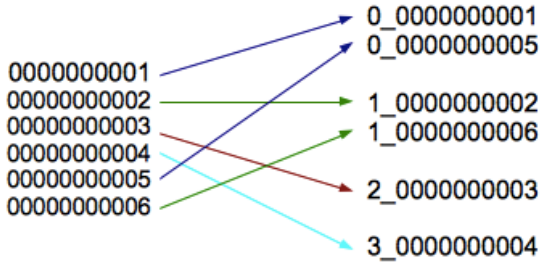

Figure 4.3: HBase row key prefix salting

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 in 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 [15]. GR-IX is the Greek IXP, through which 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 exchange is managed and operated by the Greek Research and Technology Network (GRNET).

GR-IX is handling aggregate traffic peaking at multiple Gigabytes per second. Using the packet sampling tool sFlow, IP packets were captured with a random sampling rate 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

5.1.2 Autonomous System Dataset

One of the external datasets used by the topology is the GeoLite ASN IPv4 database [14]. This dataset maps IPv4 address ranges to Autonomous System Numbers (ASN) and is updated by MaxMind 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 it 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 [19]. 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) operating 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 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 a 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 during all of the benchmarks (around 3%), therefore co-hosting it with HBase does not interfere with performance.

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.

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.

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

Table 5.1: Virtual machine hardware specifications

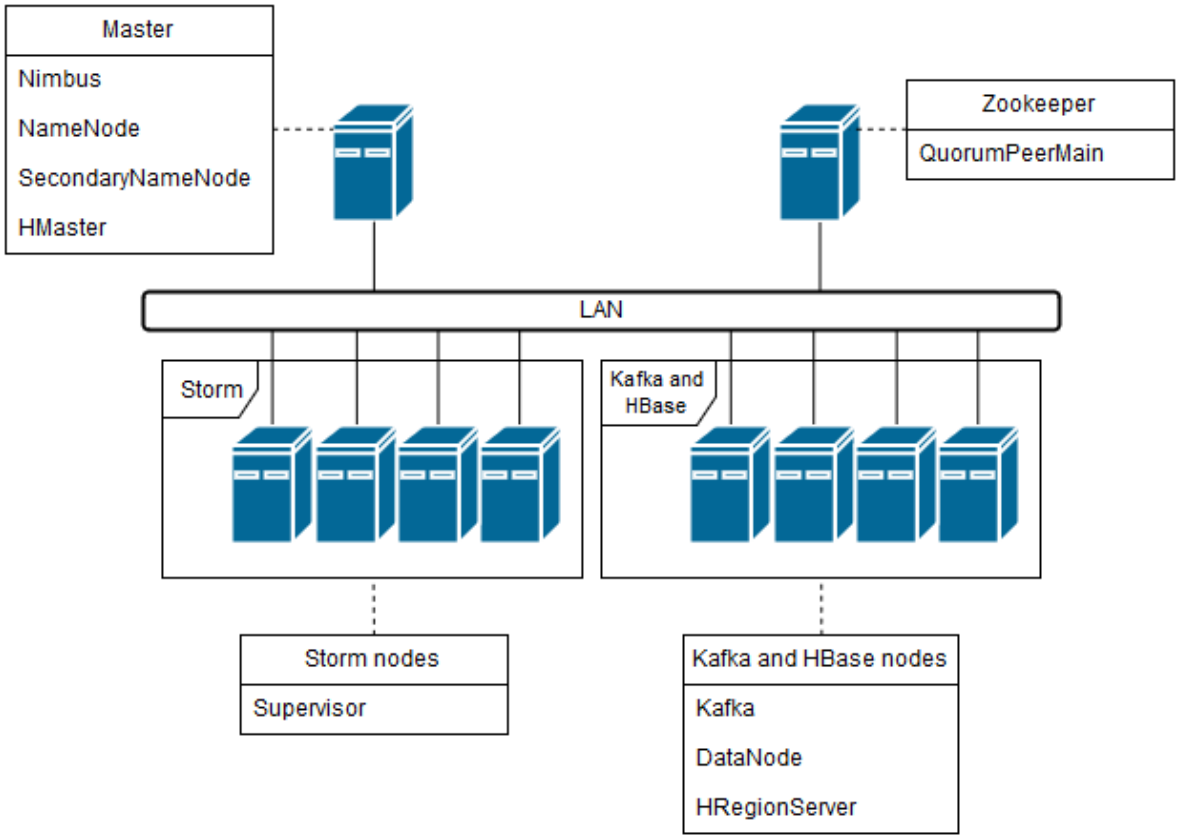

Figure 5.1: Cluster deployment diagram

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

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 [5]. *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

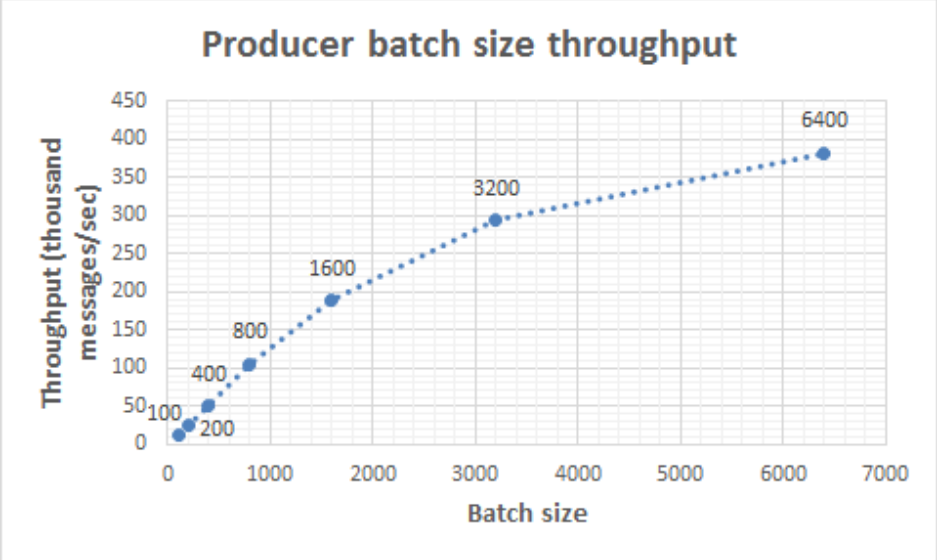

Figure 5.2: Producer batch size effect on topic throughput

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

Kafka *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 in 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 use 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.

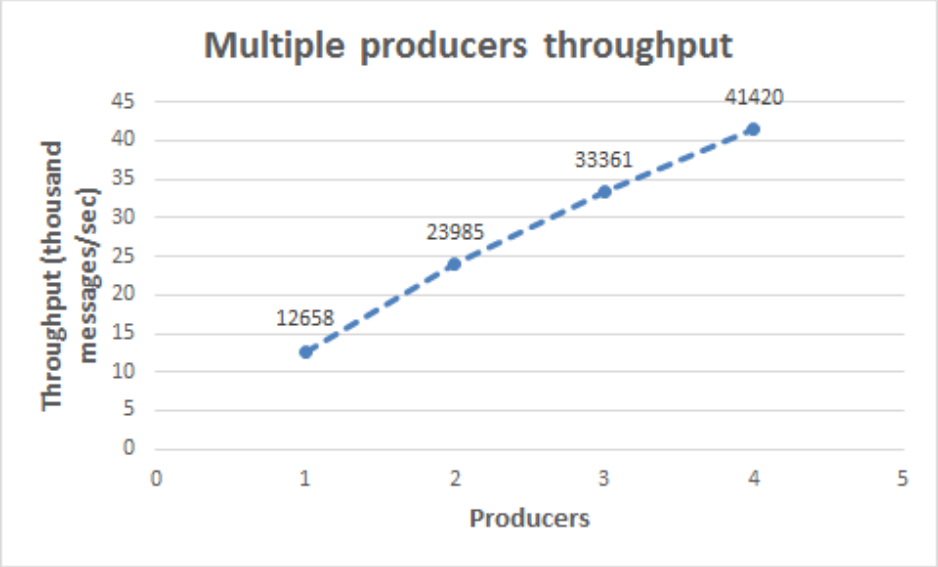

Figure 5.3: Topic throughput scalability with the number of producers

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 partitions is adjusted according to the number of the brokers.

As we can see in Figure 5.4 topic throughput scales almost linearly with Kafka cluster size.

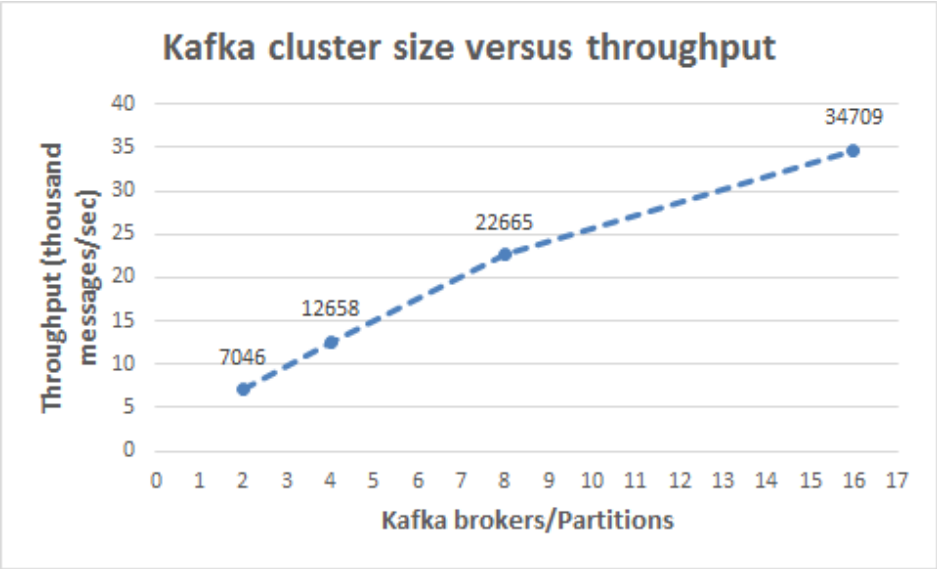

Figure 5.4: Topic throughput scalability with Kafka cluster size

5.4 Storm Performance Tuning

The Storm metrics for the following experiments were extracted from the Storm Web UI, which provides information on the running Storm topologies. The CPU metrics were extracted from the Ganglia monitoring system [13] running on the VMs of the evaluation cluster.

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 the 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 tuning 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 appear to be larger than 1. The parallelism of the Kafka Spout is always 4 to match the number of the topic's partitions.

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 for the rest of the benchmarks.

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 processors 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 expected since the IP to DNS Bolt and the Phoenix Blot perform reads and writes respectively to HBase tables.

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

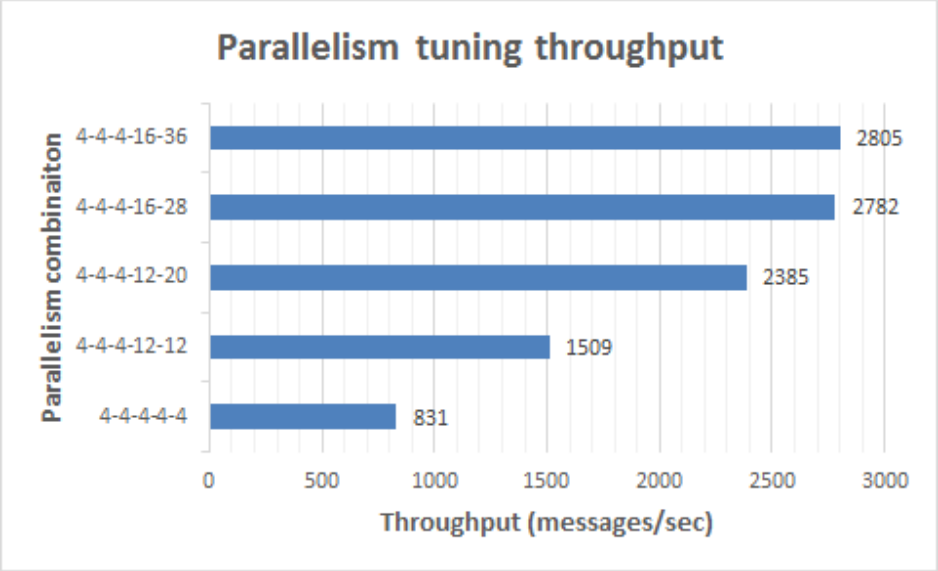

Figure 5.5: Topology throughput during parallelism tuning experiments

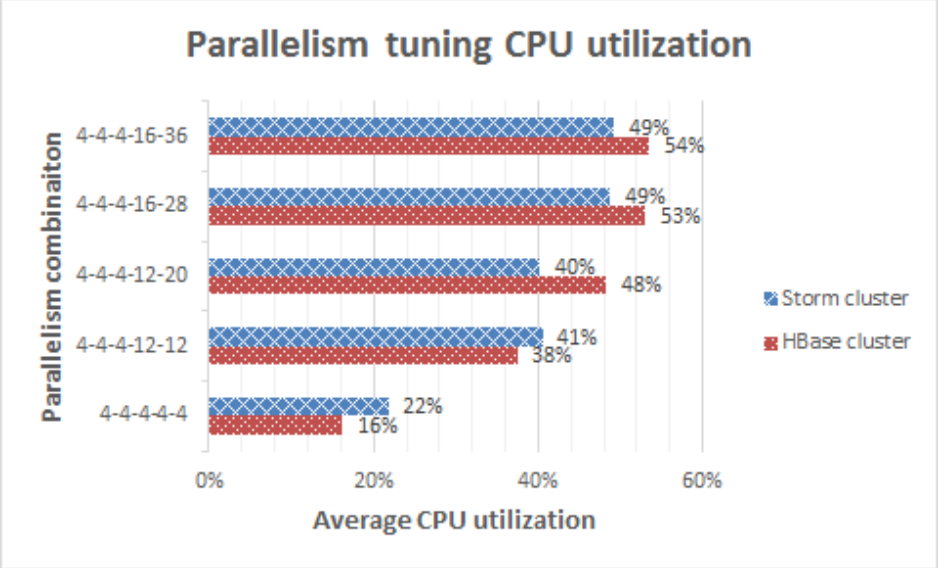

Figure 5.6: Average CPU utilization for the Storm and HBase clusters during parallelism tuning experiments

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 in 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 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

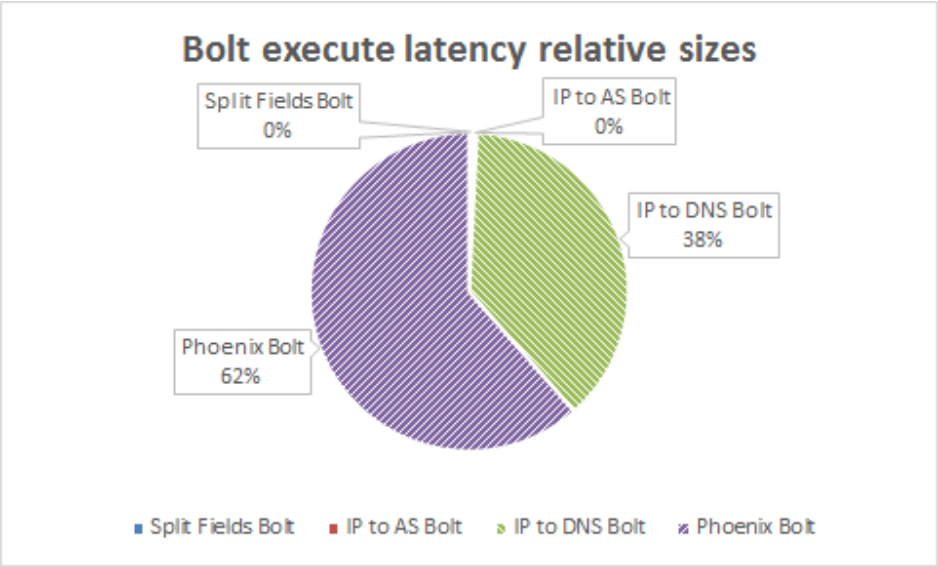

Figure 5.7: Relative sizes of execute latencies for the bolts of the topology

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. Total system latency is measured at **1.161 sec** on average at maximum 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 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 by eliminating write hostspotting. Whereas all the write requests were 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 utilization of the cluster's resources, leading to higher topology throughput.

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.

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

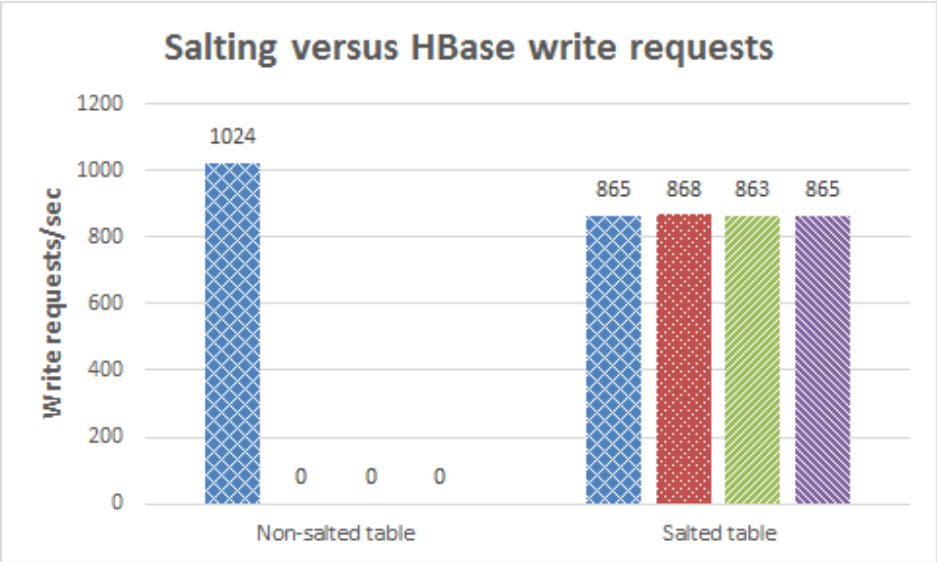

Figure 5.8: Salting effect on HBase write request distribution

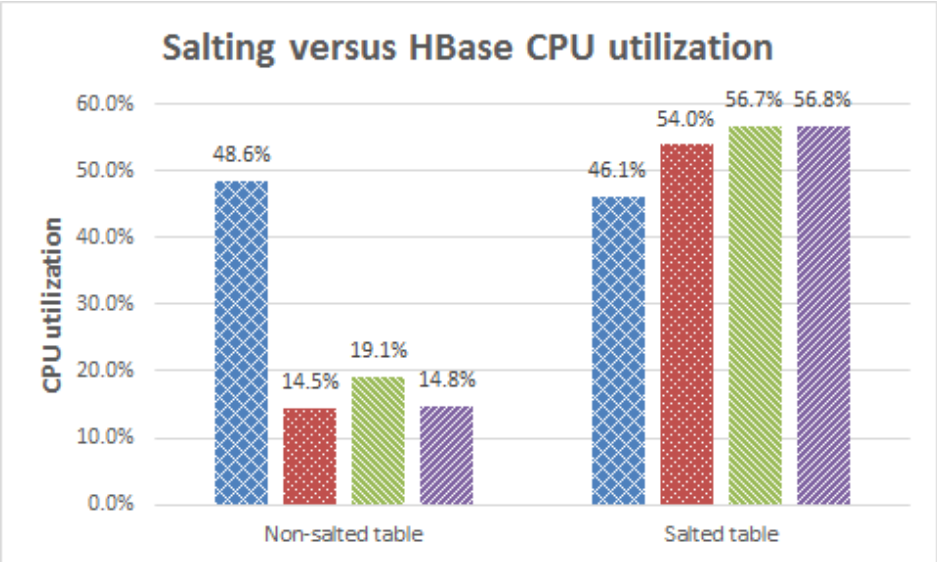

Figure 5.9: Salting effect on HBase cluster CPU utilization

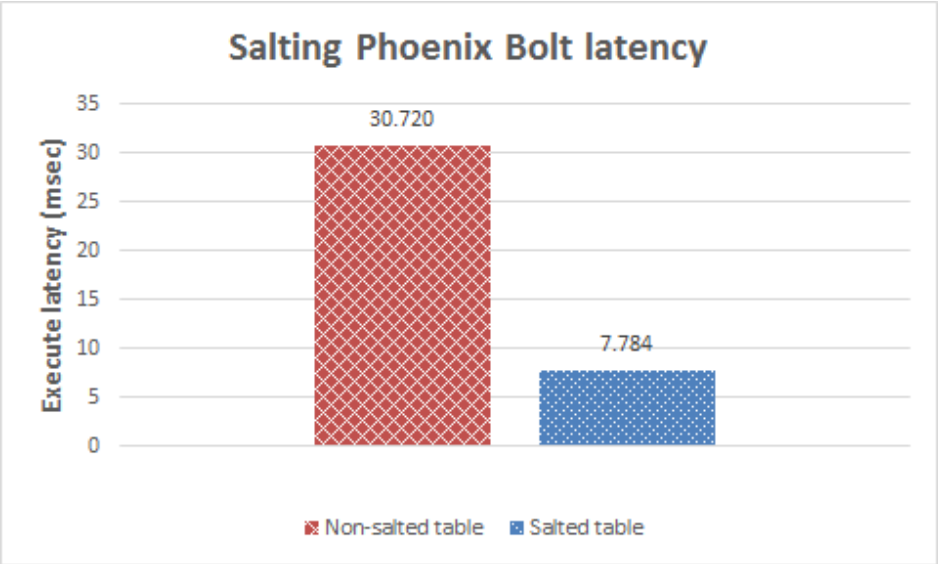

Figure 5.10: Salting effect on Phoenix bolt latency

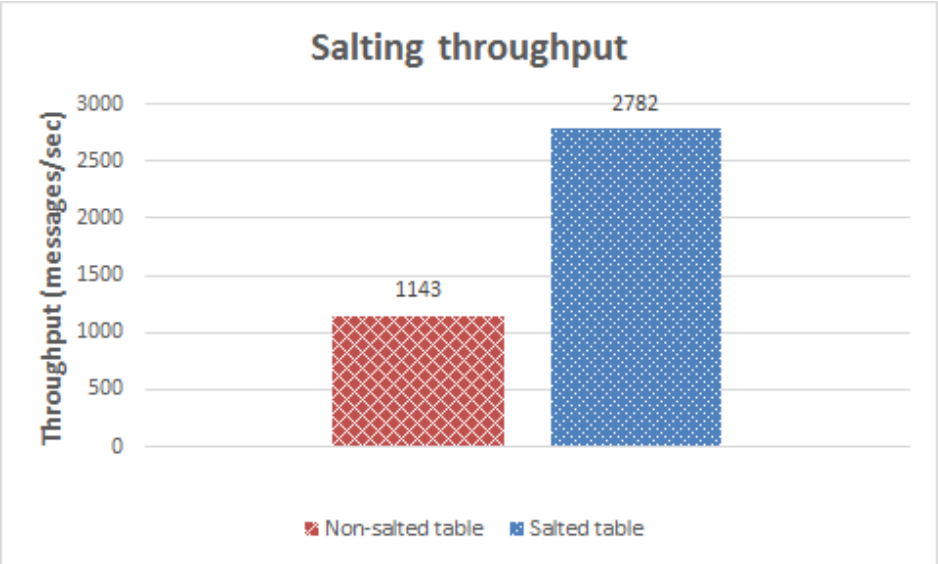

Figure 5.11: Salting effect on topology throughput

5.5 Storm Scalability

To evaluate the scalability of the topology with Storm and HBase cluster size, we measure the topology 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 partitions for the topic, the component parallelism in the topology and the number of salt buckets for the table. After any change to the size HBase cluster we distribute the `rdns` table evenly among the RegionServers and compact it for data locality.

The topology throughput scalability with Storm and HBase cluster size can be seen in Figure 5.12. The average CPU utilization for the Storm and HBase clusters during the scalability experiments is presented in Figure 5.13.

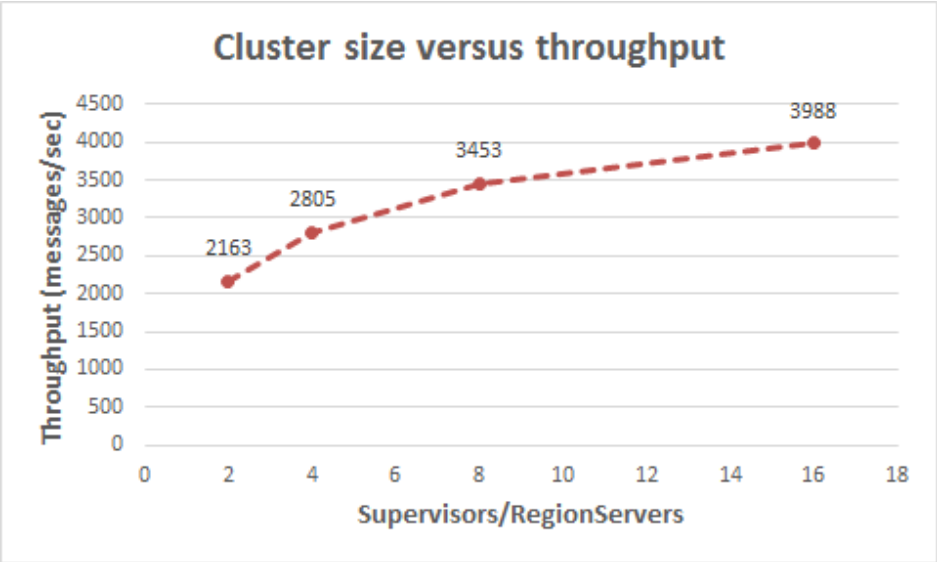

Figure 5.12: Topology throughput scalability with Storm and HBase cluster size

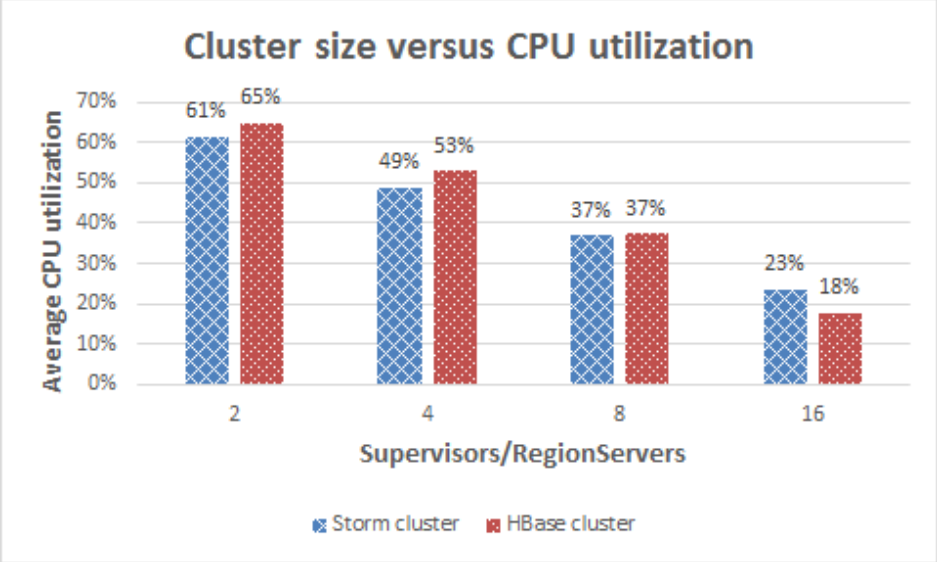

Figure 5.13: Average CPU utilization during scalability experiments

We notice that the topology throughput does not scale linearly with Storm and HBase cluster size and the processors are underused for larger cluster sizes. This indicates that the evaluation cluster setup is suffering from disk I/O saturation. As we increase the nodes for each cluster by adding more VMs, the underlying OpenStack cluster infrastructure remains the same, thus the aggregate disk I/O throughput does not increase proportionally with cluster size. This explains the diminishing increases in throughput as we increase the cluster size. If the aggregate disk I/O throughput was increasing according to the cluster size, for example by assigning every node with a dedicated disk, then the topology throughput would then scale linearly with cluster size.

5.6 HBase and Phoenix Performance Tuning

The comparison basis of the following benchmarks is our final Phoenix table, after all optimizations are applied. 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:

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

The *count* query iterates over the rows of the default column family. This query is useful to measure read performance without any additional calculations.

```
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 in this table ordered by the number of exchanged packets. We also perform the *topN DNS* alternative on some benchmarks, however this query 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 during the calculations and thus 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 disabled and afterwards enabled. 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.

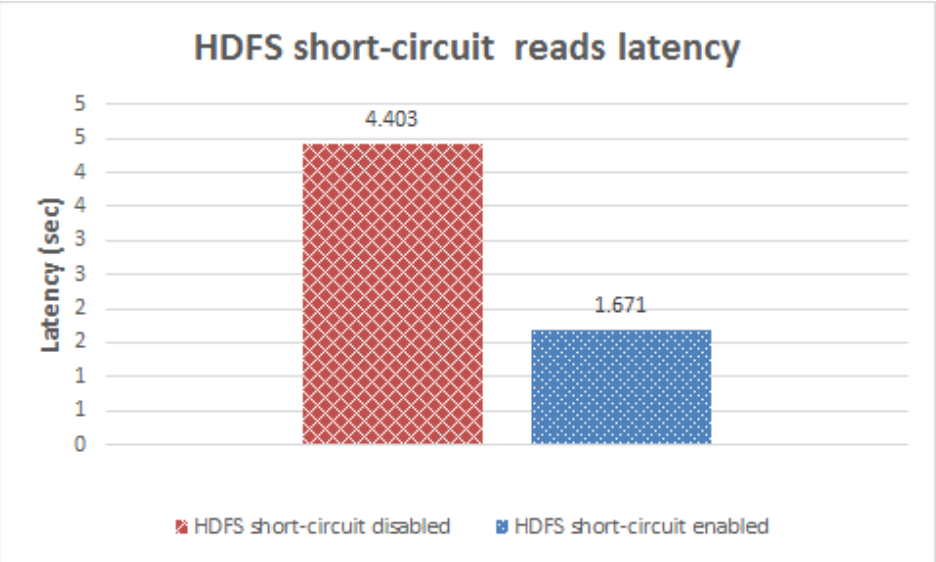

Figure 5.14: Enabling HDFS short-circuit for local reads effect on count query latency

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 performance hit for decompression, decoding or both when reading the cached data.

In our experiment we compare on-disk size and in-cache query latency for the 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.

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, as seen in Figure 5.16. 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.

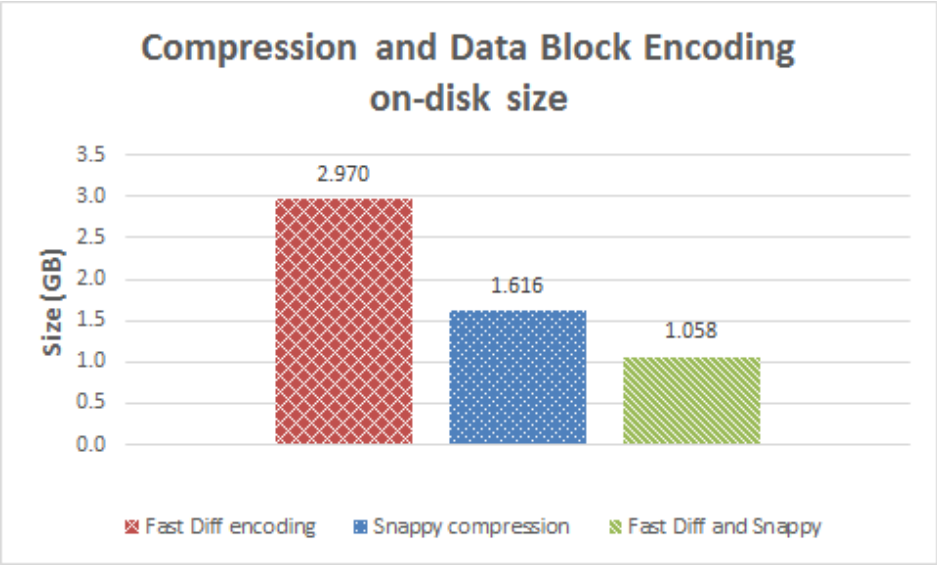

Figure 5.15: Compression and data block encoding effect on the on-disk size of a 10 million row table

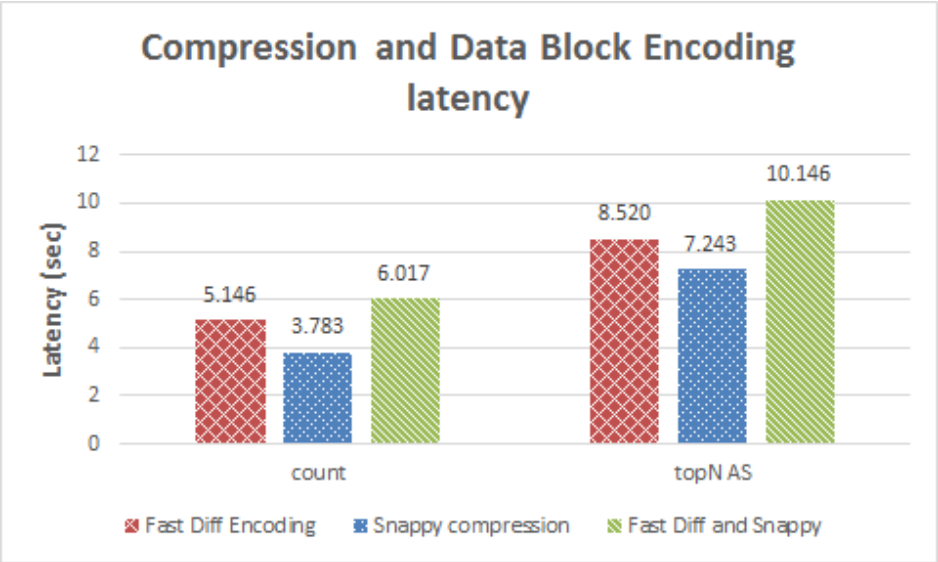

Figure 5.16: Compression and data block encoding effect on query latency

5.6.3 Number of Column Families

We compare query performance between our final table, that includes three column families (d, 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.

The total size of the final table is divided between the three column families, with the percentages shown in Figure 5.17. We measure the total query latency, including data transfer time to the Block-Cache, for queries over 1 million rows on the aforementioned tables and present the performance boost that multiple column families offer in Figure 5.18.

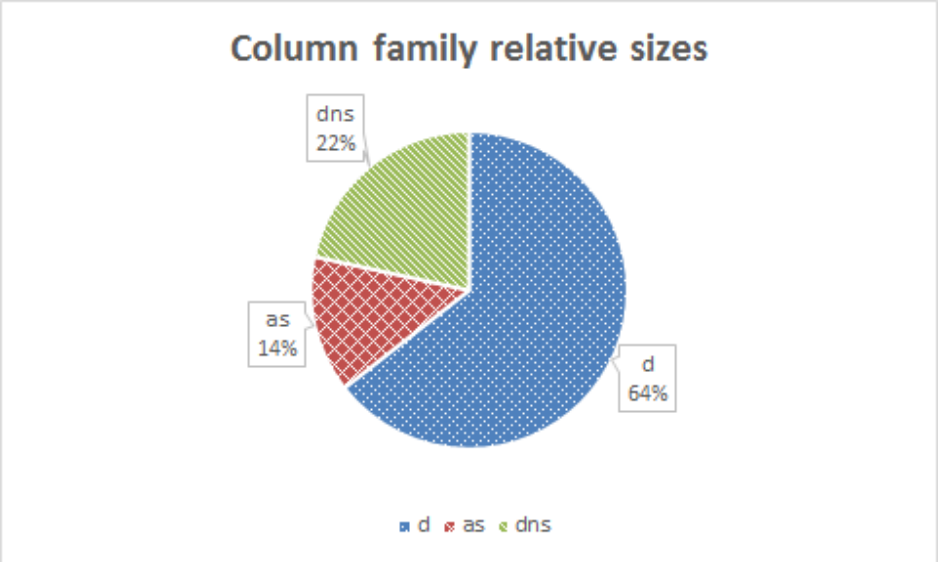

Figure 5.17: Relative sizes of the three column families of the table

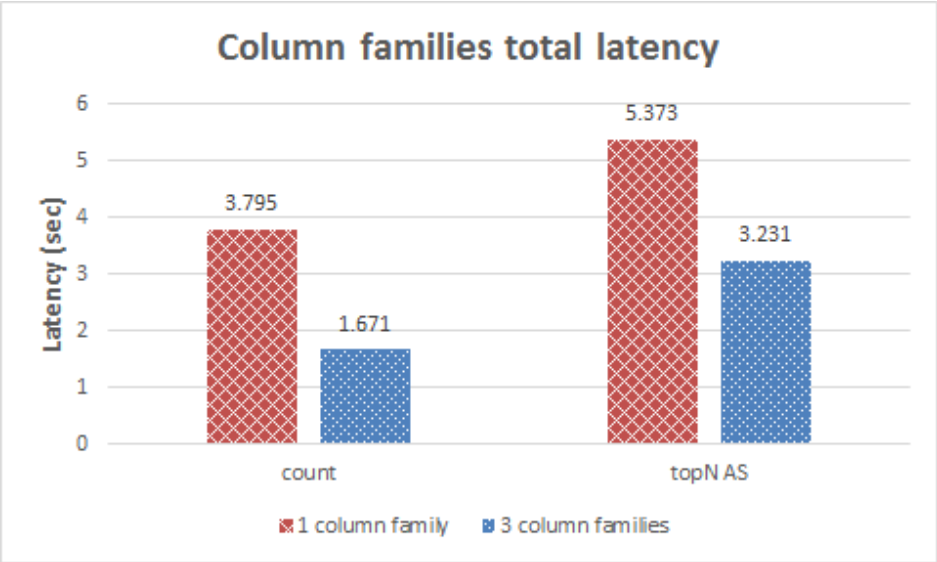

Figure 5.18: Total query latency for tables with different column family setups

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 at 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 latency. Salting speeds up count and topN AS queries by **68%**, as we can see in Figure 5.19.

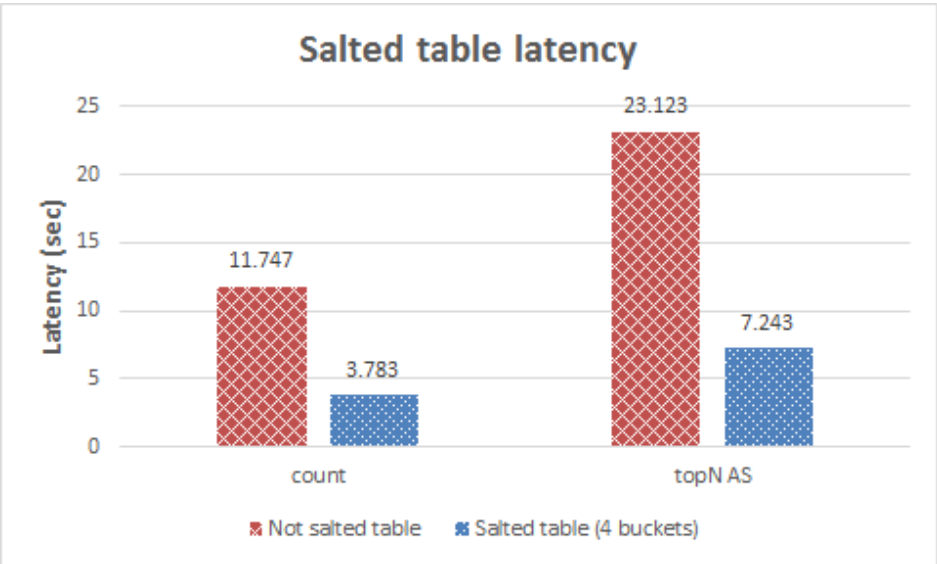

Figure 5.19: Salting effect on query latency

5.7 HBase and Phoenix Scalability

5.7.1 Table Rows

To evaluate the query latency scalability of our table with the size of the data included in the table, we measure the query latency for tables with different numbers of rows. Figures 5.20 and 5.21 show that the query latency scalability with the size of the data is close to linear.

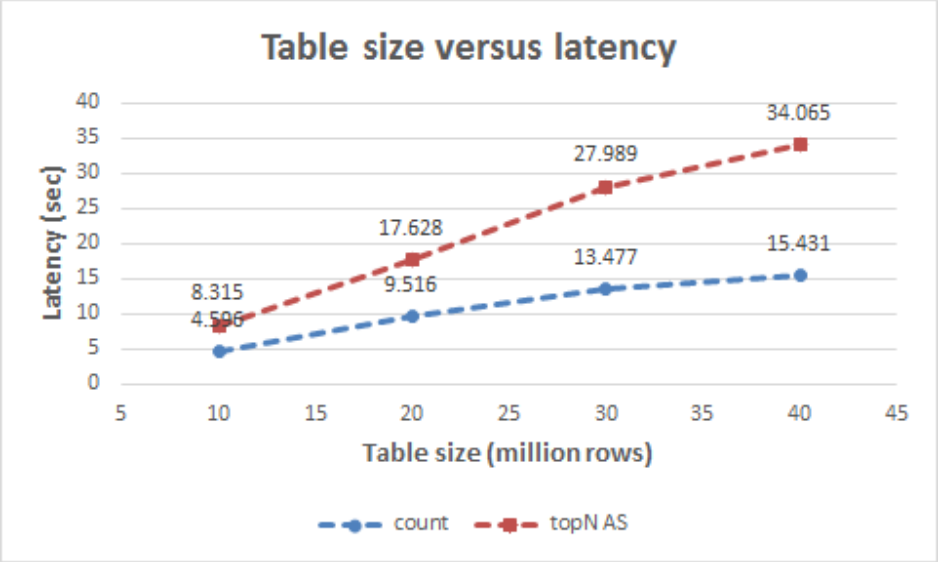

Figure 5.20: Count and topN AS query latency scalability with table size

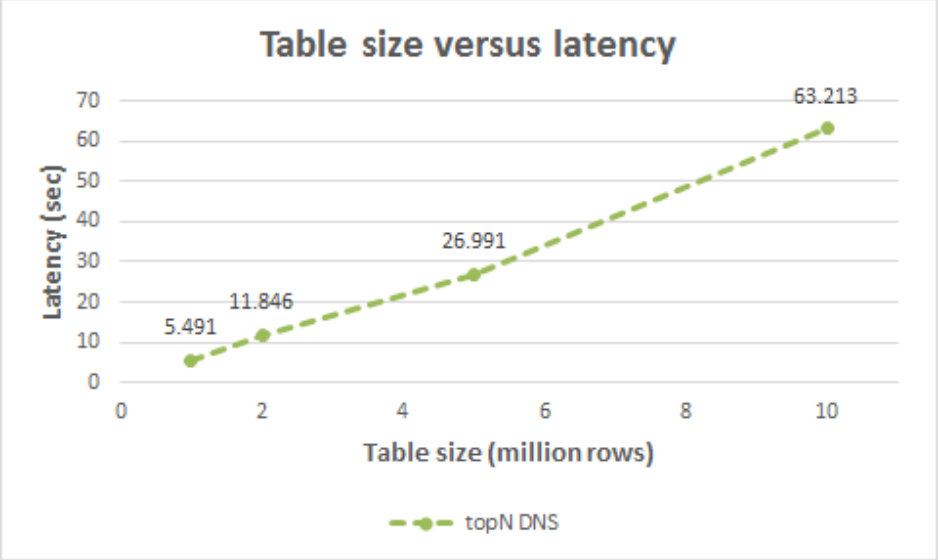

Figure 5.21: TopN DNS query latency scalability with table size

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. The results of this experiment are presented in Figure 5.22.

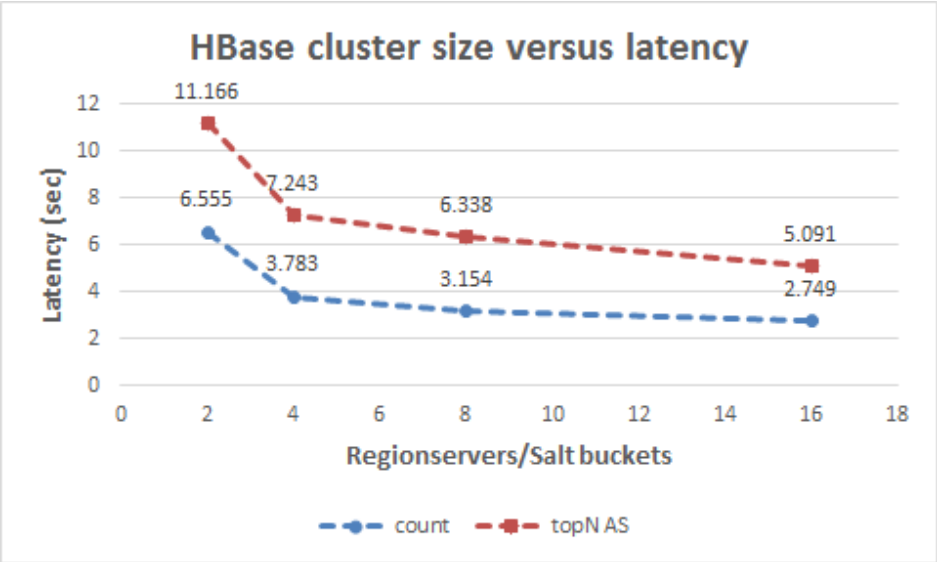

Figure 5.22: Query latency scalability with HBase cluster size

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 5.23 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.

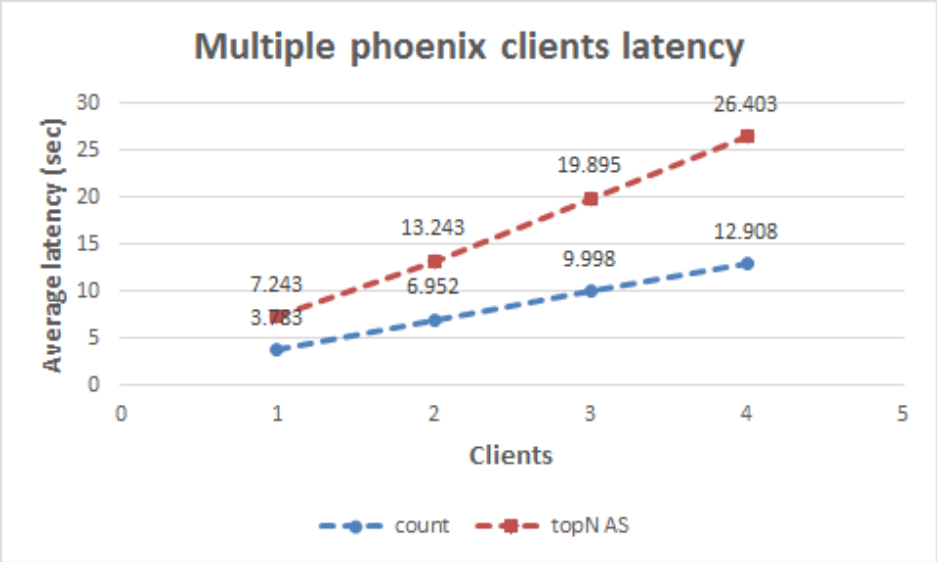

Figure 5.23: Query latency scalability with the number of Phoenix clients

Chapter 6

Conclusion

6.1 Concluding Remarks

This thesis deals with the design and implementation of a distributed system that allows the execution of low latency SQL queries that join a real-time data stream and an external dataset. The use case for which we implement this system is the execution of topN SQL queries that join a real-time network data stream, generated by sampling IXP traffic, and external datasets containing Autonomous System and DNS information.

To achieve low query latency, we implemented a Storm topology that reads the data stream from a Kafka topic, performs the join in real time and stores the denormalized data stream at a Phoenix table in HBase. This allows all subsequent queries to be performed without the need to compute the join on query time. The system's scalability and fault tolerance are ensured by using Kafka, Storm and HBase for its implementation. Storm also provides extensibility to the system, allowing us to easily add more external datasets of any size that are joined with the network data stream.

We also applied a combination of optimizations to the HBase cluster and the Phoenix table that further reduce query latency. More specifically, we use multiple column families for the Phoenix table to reduce the data cached during each query. We enabled HDFS short-circuit for faster local reads. To increase read and write performance, we also enabled compression and salting on the Phoenix table and disabled data block encoding. In addition to that, we disabled BlockCache on the Reverse DNS table, to allow the Phoenix table to fully take advantage of the cache.

Finally, we evaluated the performance of the system using a cluster of VMs. We recorded and analyzed the performance for every component of the system, including the Kafka topic, the Storm topology and the Phoenix table, while tuning the system and applying the aforementioned optimizations. The results demonstrated that our system can process packets with a satisfactory throughput, with a low total system latency and allows queries to be executed with low execute latency. We also experimented with the system's scalability with the cluster size, however in our evaluation setup it did not demonstrate linear scaling for large cluster sizes, because the aggregate disk I/O throughput was not increasing proportionally with cluster size.

6.2 Future Work

Regarding future work that can evolve our system, we propose the following:

- Properly evaluate the system's scalability using a cluster of physical nodes, each one assigned with a dedicated disk.
- Compare the Storm topology of our system to implementations in other distributed stream processing frameworks, such as Storm Trident [25], Spark Streaming [40] and Samza [7]. Storm Trident and Spark Streaming are batching the data stream to achieve higher throughput.

- Compare Phoenix to other low latency SQL-on-HBase querying engines, such as Apache Drill [34] and Spark SQL [27].

Bibliography

[1] An In-Depth Look at the HBase Architecture. http://www.mapr.com/blog/in-depth-look-hbase-architecture.

[2] Apache Hadoop. http://hadoop.apache.org.

[3] Apache HBase. http://hbase.apache.org.

[4] Apache HBase Reference Guide. http://hbase.apache.org/book.html.

[5] Apache Kafka Documentation. http://kafka.apache.org/documentation.html.

[6] Apache Phoenix. http://phoenix.apache.org.

[7] Apache Samza. http://samza.apache.org.

[8] Apache Storm. http://storm.apache.org.

[9] Apache Zookeeper. http://zookeeper.apache.org.

[10] Cisco Visual Networking Index: Forecast and Methodology, 2014-2019 White Paper. http://www.cisco.com/c/en/us/solutions/collateral/service-provider/ip-ngn-ip-next-generation-network/white_paper_c11-481360.html.

[11] Cisco Visual Networking Index Predicts IP Traffic to Triple from 2014-2019; Growth Drivers Include Increasing Mobile Access, Demand for Video Services. http://newsroom.cisco.com/press-release-content?articleId=1644203.

[12] Class TreeMap. http://docs.oracle.com/javase/7/docs/api/java/util/TreeMap.html.

[13] Ganglia Monitoring System. http://ganglia.info.

[14] GeoLite Database. http://dev.maxmind.com/geoip/legacy/geolite.

[15] GR-IX. http://www.gr-ix.gr.

[16] HBaseWD: Avoid RegionServer Hotspotting Despite Sequential Keys. http://blog.sematext.com/2012/04/09/hbasewd-avoid-regionserver-hotspotting-despite-writing-records-with-sequential-keys.

[17] HDFS Short-Circuit Local Reads. http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/ShortCircuitLocalReads.html.

[18] How Improved Short-Circuit Local Reads Bring Better Performance and Security to Hadoop. http://blog.cloudera.com/blog/2013/08/how-improved-short-circuit-local-reads-bring-better-performance-and-security-to-hadoop.

[19] Rapid7 Reverse DNS. http://scans.io/study/sonar.rdns.

[20] Salted Tables. http://phoenix.apache.org/salted.html.

[21] sFlow. http://www.sflow.org.

[22] sflowtool. http://github.com/skamithi/sflowtool.

[23] Snappy, a fast compressor/decompressor. http://google.github.io/snappy.

[24] The Bro Network Security Monitor. http://www.bro.org.

[25] Trident Tutorial. http://storm.apache.org/documentation/Trident-tutorial.html.

[26] Understanding the Parallelism of a Storm Topology. http://storm.apache.org/documentation/Understanding-the-parallelism-of-a-Storm-topology.html.

[27] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. Spark SQL: Relational data processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1383--1394. ACM, 2015.

[28] Arian Bar, Pedro Casas, Lukasz Golab, and Alessandro Finamore. DBStream: an online aggregation, filtering and processing system for network traffic monitoring. In *Wireless Communications and Mobile Computing Conference (IWCMC), 2014 International*, pages 611--616. IEEE, 2014.

[29] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422--426, 1970.

[30] Dhruba Borthakur. HDFS Architecture Guide. 2008.

[31] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[32] Nikolaos Chatzis, Georgios Smaragdakis, Jan Böttger, Thomas Krenc, and Anja Feldmann. On the benefits of using a large IXP as an Internet vantage point. In *Proceedings of the 2013 conference on Internet measurement conference*, pages 333--346. ACM, 2013.

[33] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *ACM SIGOPS operating systems review*, volume 37, pages 29--43. ACM, 2003.

[34] Michael Hausenblas and Jacques Nadeau. Apache drill: interactive ad-hoc analysis at scale. *Big Data*, 1(2):100--104, 2013.

[35] Anand Padmanabha Iyer, Li Erran Li, and Ion Stoica. CellIQ: Real-Time Cellular Network Analytics at Scale.

[36] Jay Kreps, Neha Narkhede, Jun Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, pages 1--7, 2011.

[37] Dimitrios Sarlis, Nikolaos Papailiou, Ioannis Konstantinou, Georgios Smaragdakis, and Nectarios Koziris. Datix: A System for Scalable Network Analytics. *ACM SIGCOMM Computer Communication Review*, 45(5):21--28, 2015.

[38] Douglas Schales, Xin Hu, Jiyong Jang, Reiner Sailer, Marc Stoecklin, and Ting Wang. FCCE: Highly Scalable Distributed Feature Collection and Correlation Engine for Low Latency Big Data Analytics.

[39] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. Storm@ twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 147--156. ACM, 2014.

[40] Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, and Ion Stoica. Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing*, pages 10--10. USENIX Association, 2012.