Newcastle University

School of Computing Science

Antares: A Scalable, Efficient Platform for Stream, Historic, Combined and Geospatial Querying

Rebecca Simmonds

Submitted for the degree of Doctor of Philosophy in the School of Computing Science, Newcastle University

June 2016

Abstract

Traditional methods for storing and analysing data are proving inadequate for processing "Big Data". This is due to its volume, and the rate at which it is being generated. The limitations of current technologies are further exacerbated by the increased demand for applications which allow users to access and interact with data as soon as it is generated. Near real-time analysis such as this can be partially supported by stream processing systems, however they currently lack the ability to store data for efficient historic processing: many applications require a combination of near real-time and historic data analysis. This thesis investigates this problem, and describes and evaluates a novel approach for addressing it. *Antares* is a layered framework that has been designed to exploit and extend the scalability of NoSQL databases to support low latency querying and high throughput rates for both stream and historic data analysis simultaneously.

Antares began as a company funded project, sponsored by Red Hat the motivation was to identify a new technology which could provide scalable analysis of data, both stream and historic. The motivation for this was to explore new methods for supporting scale and efficiency, for example a layered approach. A layered approach would exploit the scale of historic stores and the speed of in-memory processing. New technologies were investigates to identify current mechanisms and suggest a means of improvement.

Antares supports a layered approach for analysis, the motivation for the platform was to provide scalable, low latency querying of Twitter data for other researchers to help automate analysis. *Antares* needed to provide temporal and spatial analysis of Twitter data using the timestamp and geotag. The approach used Twitter as a use case and derived requirements from social scientists for a broader research project called Tweet My Street.

Many data streaming applications have a location-based aspect, using geospatial data to enhance the functionality they provide. However geospatial data is inherently difficult to process at scale due to its multidimensional nature. To address these difficulties, this thesis proposes *Antares* as a new solution to providing scalable and efficient mechanisms for querying geospatial data. The thesis describes the design of *Antares* and evaluates its performance on a range of scenarios taken from a real social media analytics application. The results show significant performance gains when compared to existing approaches, for particular types of analysis.

The approach is evaluated by executing experiments across *Antares* and similar systems to show the improved results. *Antares* demonstrates a layered approach can be used to improve performance for inserts and searches as well as increasing the ingestion rate of the system.

Declaration

I declare that this thesis is my own work unless otherwise stated. No part of this thesis has previously been submitted for a degree or any other qualification at Newcastle University or any other institution.

Total word count approx 45,000 Rebecca Simmonds

Publications

Portions of the work within this thesis have been documented in the following publications:

Simmonds, R. M., Watson, P., Halliday, J. and Missier, P. (2014). A Platform for Analysing Stream and Historic Data with Efficient and Scalable Design Patterns. *2014 IEEE World Congress on Services*, (Ii), 174-181. doi:10.1109/SERVICES.2014.40

Mearns, G., Simmonds, R., Richardson, R., Turner, M., Watson, P. and Missier, P. (2014). Tweet My Street: A Cross-Disciplinary Collaboration for the Analysis of Local Twitter Data. *Future Internet*, 6(2), 378-396. doi:10.3390/fi6020378

Simmonds, R., Watson, P. and Halliday, J. (2015). Antares: A Scalable, Real-Time, Fault Tolerant Data Store for Spatial Analysis. *2015 IEEE World Congress on Services*, 105-112. doi:10.1109/SERVICES.2015.24

Acknowledgements

Firstly I would like to thank my supervisory team Paul Watson and Paolo Missier for their continued support throughout and for the opportunity to be a part of such a great project.

I would like to give a big thanks to Stephen Dowsland and Mark Turner for their collaboration and contributions to the visualisations and Tweet My Street project - it was a pleasure working as a development team with them.

Tweet My Street would not have been possible without the collaborative work and support of Graeme Mearns and Ranald Richardson - who it was a pleasure to work with - many thanks to them.

I would also like to thank Dawn Branley for her support and collaboration.

I would like to thank the following people for their continued support and for reviewing the thesis: Matthew Forshaw, Jonathan Halliday, Derek Mortimer and Simon Woodman.

I would also like to thank all of my colleagues and friends from Newcastle who have supported me and helped to make this project possible, they are: Michael Bell, Ryan Emerson, Barry Hodgson, Adnan Hussain, Beth Lawry, Stephen Mackenzie, Oonagh Mcgee, Dominic Searson, David Stokes, Jennifer Warrender and Ginny Wooley.

Lastly I would like to acknowledge and give huge thanks to my family especially Alison Simmonds, Ingrid Simmonds and Mark Simmonds for their continued support and strength through the whole process it would not have been possible without them.

Contents

List of Figures

List of Tables

Acronyms

List of Acronyms

API : Application Program Interface

CEP Complex Event Processing

CQL : Cassandra Query Language

DBMS: Database Management System

DRAM: Dynamic Random Access Memory

EPL : Event Processing Language

GFS : Google File System

HDFS: Hadoop Distributed File System

JDBC: Java Database Connectivity technology

JPA : Java Persistence API

JVM :Java Virtual Machine

MBR : Minimum Bounding Box

MTTF: Mean Time To Failure

MVCC: Multi-version Concurrency Control

RDBMS: Relational Database Management System

SQL: Structured Query Language

SSD: Solid State Drive

TTL: Time to Live

1 Introduction

Contents

The emergence of "Big Data", including data from social media, sensors and online resources, has created the need for novel processing and storage technologies. Traditional methods for storing and analysing data have proved inadequate due to the sheer volume of data, and the rate at which it is being generated. Given the real-time nature of many of these data sources the success of an application often depends on the stream of data being analysed very quickly after it is collected. Additionally, with the growth of mobile devices, data often has location tags assigned to it, and the ability to process this in a timely manner is the basis for many applications. For example, direction applications need to respond in near real-time. However, large scale location data can be challenging to process with new scalable technologies due to the rich querying functionality required to query latitude, longitude and metadata (multidimensional data).

Near real-time analysis of these types of datasets can be partially supported by stream processing systems, however many applications require a combination of stream processing and historic data analysis. Stream processing can often be limited by the size of memory available, which can be expensive to increase. Therefore being able to process the stream as it is collected then store it in a database supports a more affordable and scalable solution. This also means that querying the data at a later date is also possible. For example being able to identify an event in real-time, say an earthquake, then querying the historic store later to try to identify the epicentre.

Twitter analytics is used as a generalisable example to drive the design and evaluation of a scalable system that can achieve this (however, *Antares* is capable of processing any temporal and geospatial data). The Twitter firehose emits Tweets at an average rate of 5000 per second [1]. The data used was based on temporal and geotagged information.

Each Tweet is essentially a JSON object made up of a set of tags representing the metadata of the Tweet itself. The data model taken from this metadata and used for *Antares* contains a user, retweet, geotag, timestamp, hashtag and status. **User**: this is who posted the Tweet. **Retweet**: this demonstrates whether the Tweet has been retweeted and if so gives the Tweet a new id, as well as containing the original one. **Geotag**: this describes the location the Tweet was posted using a latitude and a longitude. **Timestamp**: this is when the Tweet was posted. **Hashtag**: this describes the topic of the Tweet and is one word preceded by a hashtag. **Status**: content of the Tweet.

To identify events as they occur, for example, all Tweets with a specific hashtag, requires the ability to query data arriving at the full rate of the Twitter firehose. All Tweets then need to be stored in a database, which must be able to simultaneously run queries against the stored (i.e. historic) data, potentially spanning billions of items. For example, at the current rate, Twitter will see 150 billion generated in 2015, and has witnessed bursts as high as 143,199 per second [1].

The system must also support "combined" queries that use both stream and historic analysis techniques together in order to answer questions such as identifying all Tweets that have been retweeted more than a certain number of times. The combined query searches both stream and historic data in a time period that stretches from the past to the future. One query will be submitted by the user and the system will provide a novel mechanism for transparently translating and executing the correct type of query.

Designing a system that can scale to handle the full arrival rate of events from the Twitter firehose while simultaneously answering user queries against the stream and historic data is challenging in terms of providing reliability and availability. This is challenging due to the rate of ingestion as the system must also enable availability to query simultaneously, which means investigating concurrent solutions. The system must also ensure measures to maintain reliability while supporting availability.

This challenge is made yet more complicated by the importance of geotagged data for processing Twitter and other types of streaming data. Existing solutions use spatial indexing techniques, including tree and hashing structures, to accelerate geospatial queries. However spatial data can still be challenging to process at scale due to its multidimensional nature as each point contains a latitude, longitude and metadata (related to the content posted). The complexity of processing stream, historic and multidimensional data has driven the design and implementation of *Antares* to support scalable queries that range over both time and space.

1.1 Thesis Contributions

This thesis investigates this problem of scalably supporting applications that require a combination of near real-time and historic data analysis, and describes and evaluates a novel approach for addressing it. The approach used within this thesis is a layered one, which allows a scale that a non-layered approach would not due to the volume of data being ingested and timeouts caused by this. Any solution to this problem must provide:

1. A stream processing system for near real-time analysis of the high-velocity incoming data.
2. A historic data store capable of high data ingestion rates, that reliably stores incoming data even while simultaneously executing queries.
3. Scalable, high throughput, low-latency processing of queries over the historic data store for geospatial and temporal data.
4. Investigate the viability of a layered approach to scalable querying of Twitter data.

A survey of different data processing technologies was undertaken to investigate and evaluate a layered approach, which sort to retain the benefits of NoSQL systems and the scale that can be provided by utilising them. So new research was needed to design methods to support low latency querying and high throughput rates for both near real-time and historic data analysis simultaneously.

This thesis addresses these problems and introduces a framework called *Antares*, a scalable system, which processes data-streams, historic data and combines the two mechanisms to provide analysis for multidimensional data e.g. temporal and geospatial. In particular, it provides new methods required to support the high performance querying of geospatial data. *Antares* runs in the Cloud, exploiting its potential for scaling out and elasticity (supporting the simple addition and removal of nodes for a burst of events). The thesis describes the design of *Antares* and evaluates its performance on a range of scenarios taken from a real social media analytics application.

The results show significant performance gains when compared to existing approaches, for particular types of analysis.

The thesis' contributions are therefore:

1. The design, implementation and evaluation of *Antares*, which is used to develop techniques of scalable querying of Twitter data.

2. The design, development and evaluation of optimisations for the high throughput ingestion of data streams - specifically by a historic store to improve the stream processing through the use of a layered approach

3. The design, development and evaluation of a novel in-memory cache to reduce query response time and increase the scalability for querying spatial data.

1.2 User Requirements

The motivation for designing and developing such a system can be demonstrated through the user requirements necessary for a cross disciplinary project called Tweet My Street. This project used *Antares* to analyse geotagged and temporal data to derive data-driven insight from Twitter data.

Antares needed to provide combined querying to support identification of events on the stream, which would support the querying of a much larger stored dataset. The results from the large stored datasets had to return in near real-time so that a combined solution with the stream data was possible. The novel solution provided by *Antares* allows users to write one query for both of these processing techniques rather than two. This abstraction supports easier and more efficient execution of combined queries. Additionally *Antares* ingests the data simultaneously while executing queries. This aims to provide the user the ability to query current data while not dropping other data that could be investigated later. If the user queried an event, for example, a conference, there may be new hashtags that become popular later in the conference. *Antares* supports the storage and querying of that data after the hashtag has been identified, therefore giving the user rich temporal querying.

Antares also supports increased performance geospatial querying. The requirement of the system was to use a simplified append-only model to enable vast improvements in the efficiency of geospatial querying. *Antares* required the ability to zoom in/out of the display area and also to pan around. The system would use its append only qualities to simplify and reduce query execution time. The system was append only due to the nature of the above requirements – all data must be stored to support querying at a later date. This resulted in the huge increase in performance when compared with current commercial systems. *Antares* supports querying of large geospatial datasets, using a map visualisation to enable user interaction with the dataset.

1.3 Dissertation Outline

The scope of the remaining chapters is as follows:

- Chapter 2 outlines the technical aspects of the thesis and terms that will be used in later chapters.
- Chapter 3 discusses the design and implementation of *Antares.* It focuses on comparing related systems. It then describes the low-latency of the combined processing and the optimisations to enable this. This is then evaluated and conclusions drawn from these experiments.
- Chapter 4 then moves on to the extension of *Antares* to cope with large amounts of geospatial data processing. Following the same structure as mentioned: related work, architecture, evaluation and conclusion.
- Chapter 5 describes the application of *Antares* and how it scalably analyses Twitter data in a project called Tweet My Street. Related work is compared, the user interface is described and a critical analysis of the system is given.
- Chapter 6 concludes the thesis, reflecting on how well the objectives were met, and proposing topics for future work.

2 Technical Context

Contents

2.1 Introduction

This chapter introduces technical aspects that are used within the thesis. The aim of the chapter is to give an expanded explanation of the terms and their meaning within the context of this thesis.

2.2 Stream

Stream: a sequence of data elements which are given to the computer in a continuous, real-time flow as the data is produced. It is different to batch processing as each item is produced individually then given to the system to process there is always incoming data (unless the data source has stopped producing data) therefore there is a constant intake of data rather than a batch of data items which need processed in periodic intervals.

2.3 Stream Processing

Stream processing: the ability to analyse data as it is produced.

Ingesting a stream of data with the ability to produce some output which may or not be a stream, set of tuples or tables.

2.4 NoSQL Database

These databases do not use the conventional relational schemas to store the data. These types of database promote denormalisation of the data as they claim such easily scalable systems that extra data would not compromise the execution time of search or write queries. The schema is usually more flexible with resultant trade-offs between consistency, availability and partitioning tolerance.

2.5 Social Media Analytics

Social media includes micorblogs and social networking services such as Facebook, Twitter and YouTube. The analysis of this data using techniques which focus on the structure and terms specific to this media (rather than traditional techniques) is called social media analytics.

2.6 Twitter

Twitter is a social media microblogging platform which allows users to post Tweets, which are 140 characters posted to other users. The social media platform allows users to chose the content they see by following other users, and users have followers. Hashtags are used to express a topic within the Tweet and help to support users searching the social media platform as well as providing popular topics to suggest to users these are called "trending" topics.

2.7 Combined Querying

Within the context of this thesis the term combined querying means one or more queries which use both historic and stream processing. The order of the query, whether it be sequential or parallel does not affect the term *combined querying* so long as both layers (historic and stream) have been used in the processing to produce some output.

3

Architecture

Contents

3.1 Introduction

Antares is a scalable, cloud-based framework designed to support the storage and querying of high-velocity, high-volume Twitter data. *Antares* supports low latency combined, historic, streaming and geospatial processing. *Antares* has been designed and implemented to enable other researchers to use the tooling to analyse Twitter data, and as a result it has successfully supported published research projects [2], [3] and [4]. *Antares* is currently being deployed in a cross disciplinary project called Tweet My Street [3].

Antares uses a layered approach to implement these querying techniques. The layered approach helps to extend and exploit the desirable characteristics of a stream processing system and a historic data store to provide scalable Twitter analysis. *Antares* builds on two existing components – Cassandra and ESPER.

The key features of the *Antares* design are:

1. Database optimisations to enable efficient stream and historic data querying. The resulting low query latency facilitates combining the querying of historic data with stream event processing.

2. Enabling transparent submission of queries - the user should only enter one query whether it be a stream, historic or combined - the system will translate this into the correct query submission and execution.

3. Optimisations, which are needed to ingest high velocity and volume data streams. Consistency can be lowered to provide higher ingestion rates- therefore all data written to the database will be eventually consistent.

4. A distributed scalable caching layer which enables fast, efficient geospatial querying.

5. A set of algorithms which are required to map the caching layer to the storage layer, which enables efficient querying to exploit scalable data storage.

6. Consistency checks for the distributed geospatial cache.

7. A web interface for interaction with and visualisation of large volumes of data.

This section describes the architecture for which *Antares* is based on and extends. *Antares* supports scalable querying of the Twitter firehose. This required high ingestion rates and a "combined" querying mechanism, which would support: stream and historic querying as well as exploiting the low latency to provide a combination of both querying techniques. *Antares* would provide easy querying of Twitter data by combining both querying mechanisms through one query stated by the user. *Antares* takes these queries and translates them into the correct type of query whether that be stream, historic or combined. This chapter argues that *Antares* uses a layered approach to support scalable Twitter analysis which provides the user with low latency querying and a single interface to enter querying parameters for all querying mechanisms. The scale *Antares* provides would not be possible with a non-layered approach as the querying and insertions are distributed throughout the system to allow for greater intake of requests. The motivation for Antares was to

1. provide researchers with a scalable means of analysing Twitter data
2. investigate different technologies to provide scalable Twitter analysis.

A literature review of the technology was undertaken to identify a suitable technology which could scale-out to exploit the advantages of the elasticity of cloud computing. This chapter describes different technologies which were considered for the layered approach within *Antares*. These are then compared with current "combined" systems to show the difference between *Antares* and these. *Antares* and the data model used to design the queries is then described in detail. This refers to the querying mechanisms, query language, mapping and then the evaluation of the system. The evaluation of *Antares* used Twitter data and simulated firehose rates to evaluate the latency of the queries and how the combined querying and writing to the system affects the speed of query responses.

3.2 Related Work

This section reviews technologies which could be used for the layered approach to analysing Twitter data. These technologies are compared and the desirable features of each decided whether to use or not. The stream processing systems are reviewed for scalability to validate if it could be used within the system to achieve feature three.

A set of databases which are regarded or have been described as scalable are also investigated then a database with the desired qualities is used. The last part of the related work looks at systems which have been combined/layered like *Antares* these are reviewed and compared. The related work is used to examine the technology space which currently exists for layered approaches and identify technology which can be extended to provide scalable Twitter querying and analysis. Then compare *Antares* with current systems and identify differences and the improvements made by the system.

3.2.1 Scalable Stream Processing

Stream processing systems are becoming increasingly important with the spread of sensor and mobile technologies. The need for distributed and/or fault tolerant systems is leading to new approaches. Additionally the eruption of Big Data has led to a change in direction for data analysis, with it requiring larger and larger data sets to be processed within restrictive time limits. Therefore this section reviews and evaluates current technologies that provide stream analysis.

A literature review was carried out on different stream processing systems, which reviewed the scalability of the system to ingest and process data and the querying mechanisms that are used by these systems. These were compared and then a suitable stream processing system was chosen to support the scalability of the layered approach taken within *Antares*.

To identify a stream processing system which is capable of ingesting a large amount of data a literature review was undertaken describing and comparing current technologies. These were then used to chose one that would support the layered and scalable approach of *Antares*.

3.2.1.1 Storm

Storm [5], [6] is an open source real-time processing system, which executes continuous queries across stream data to identify events. Storm provides efficient and quick querying for real-time applications. It is a stream processing system that was recently acquired by Twitter and has just become open source.

Streams of data are inputted, processed and then a resultant stream is outputted. It provides complex message processing of stream data, as well as continuous queries. *Antares* supports real-time querying as well, it supports queries which are continuously executed across the data stream being processed. *Antares* also provides low latency query responses.

Storm has the ability to process millions of messages per second by supporting scale-out functionality. It supports distributed RPC, parallel search querying and set operations on large-scale data sets.

Distributed RPC allows queries to be divided and executed in parallel, which supports efficient response times. However Storm shreds data if the ingestion rate becomes too much - *Antares* uses high ingestion rates achieved through asynchronous calls and batch to avoid this problem.

The framework is held completely in-memory and therefore does not provide stand-alone persistent data storage, which is similar to the ESPER layer within *Antares* - Cassandra is used to provide persistence.

Storm uses "topologies" to execute queries similar to map reduce across the stream. These topologies are continuous queries and can be executed for an unbounded amount of time. Joins are not typically supported. *Antares* does not use joins as this technique could decrease performance, denormlisation is used to improve performance.

3.2.1.2 Drools

Drools Fusion is a rule based, forward chaining inference based rule engine and complex event processing system (CEP), which ingests multiple asynchronous streams [7].

Drools supports user-defined sliding windows and is schemaless in order to provide flexibility. Facts are executed by Drools, which are a set of rules.

Queries are constructed of rules and facts, which are used to examine the data. Multiple streams can be ingested and joined. The system is provided by Red Hat and is therefore open source. Multiprocessor hardware supports partitioning. Optional logging is supported to provide persistence as well as pluggable storage through the use of the Java persistence API (JPA). If JPA is used then this supports additional consistency as the system itself already uses transactions.

Antares does not need strict consistency rules - like for example a banking system would to ensure that bank accounts are consistent - a slight delay in consistency for Twitter is acceptable to improve performance.

Drools has the capability to scale to 900,000 facts on a 64bit JVM [8] with reasonable performance.

The system supports elasticity and it is used in the cloud to automate the increase and reduction of resources in a Java EE application.–what java ee application

The system is suited to algorithmic trading, telecom rating, fraud detection, content-based routing, credit approval, insurance and risk assessment. *Antares* does not need the functionality that application like these would require - ACID transactions are useful for such applications but will only affect the performance of Antares with no gain for the low latency queries required by Antares.

Utilising the rules so that they are effective can be hard. It is not very useful for smaller projects where the rules do not change over time and can be more difficult to debug.

3.2.1.3 S4

S4 [9] is a stream processing system which is used to process unbounded data streams. It is distributed and partially fault tolerant, depending on the pluggable storage used.

Each node processes the information and then messages are used to communicate this between nodes. This could cause performance issues while ensuring that each node within the cluster knows who has processed what. This is not a problem when using ESPER in Antares - it provides scalability on one machine. Antares uses a scalable historic store to provide collection of big data, where as the stream processing system is used to provide scalable real-time analysis of current data only - for the Twitter firehose.

It is suited to applications for: click stream analysis, evaluation of online algorithms and marketing campaigns. –how does this compare to antares do the applications have similarities?

S4 does not have a buffering mechanism therefore if there is too much data the excess is lost (data shredding). As there is no buffering it would not be suitable for large Twitter streams. Antares aims to process large scale data therefore loss of data from shredding would not meet the desirable characteristics of the system. ESPER provides a scalable means of ingesting the entire firehose without loss of data.

The system uses modified map-reduce functions to execute queries across the stream. *Antares* supports real-time queries with results being continuously pushed back to the user.

3.2.1.4 System S

System S [10] is a stream processing system which responds to information quickly and can then dynamically change requirements. It continuously analyses data at high rates and rapidly adapts to changing data forms and types. *Antares* focuses on Twitter data (for now) and therefore does not require quick dynamic changes to different inputs - it would be future work - but currently may affect performance and scale.

The system is highly available, heterogeneous and distributed. The system is used for providing security and information confidentiality for shared information by IBM. It executes a process similar to a continuous query that can be executed across multiple streams. Security is not a specified feature of *Antares*, therefore there would be no benefit in trading performance for secure data storage.

The system is used for anomaly detection, telescope data retrieval and analysis, energy trading services, financial services, health monitoring and manufacturing. System S is not an open source product and focuses on security features, which reduces performance.

3.2.1.5 ESPER

ESPER [11] is a CEP system, which ingests stream data and executes continuous queries over these to identify events. It is a standalone system. It supports sliding tumbling and combined time-windows for temporal querying. Events are filtered and analysed and then a response is returned in real-time. An event occurs when a constraint in the continuous query is met.

ESPER has a flexible schema with no constraints. Operations that are supported are: grouping, aggregation, sorting, filtering, merging, splitting or duplicating of event streams. The query language supported is called EPL and is a derivative of SQL, which supports complex querying. The system supports inner and outer joins to combine streams.

A variety of pluggable database storage is available. ESPER is multi-threaded and provides a concurrency-safe iterator, with read-write locking. The system has a performance-focused design, which includes: query strategy analysis and index building.

Antares opted to use ESPER as it is suited to complex computations, high throughput and low latency real-time applications. ESPER also supports complex queries and user-defined time-windows, which means it is suited to performance workloads. *Antares* requires high throughput and low latency querying for handling stream processing.

In ESPER a time-window signifies that an event must happen within that time period for the query to trigger some action. In the case of *Antares* it could trigger the execution of a query to the database or the return of results to the user. This is dependent on the type of query executed by the user, if it is a stream query then the results from a query executed across the stream data are returned. As mentioned some of the combined queries contain a “trigger event”, which means when an event is identified by ESPER a historic query is executed across the database too and the results combined for the user. ESPER was chosen as it can handle large numbers of events, and specify time-windows proficiently.

3.2.1.6 Rainbird

Real-time counting (e.g. URLs and Twitter clicks) was introduced by a commercial system called Rainbird [12]. This uses Cassandra as a persistent store for durability and executes stream processing in the application layer. This is similar to *Antares* using the layered approach and an historic store for its scale to support collection and processing of large scale datasets.

It uses tokenisation for textual analysis on Tweets by counting their occurrence. The system supports write rates of hundreds of thousands per second and a read volume of tens of thousands of reads per second. These queries are all executed with low latency and are approximately under 100 ms. Rainbird supports horizontal scaling of terabytes. However the querying language is limited to only counting operations, therefore it would be impractical to use in the project.

3.2.1.7 Millwheel

Millwheel [13] is a fault tolerant stream processing framework used for Internet scale applications. The system is used widely at Google for low latency data processing. Users specify a directed computational graph and continuous flow of records all while being fault tolerant. Data is sent along the edges and through nodes to process. Any node or edge can fail at any time and the result will still be correct. Millwheel supports high fault tolerance, which is a characteristic which may decrease performance and therefore would not help achieve the stated features.

3.2.1.8 InfoSphere

InfoSphere [14] is IBM's commercial offering for stream analysis. It provides user-defined operations and a high throughput rate up to millions of events per second. Frameworks like Infosphere and Millwheel provide new stream processing technologies however they are not open source and could not be considered for this project.

3.2.1.9 Apache Samza

Apache Samza [15] is a distributed stream processing framework, which uses Kafka for messaging and Hadoop Yarn [16] for fault tolerance. It supports a simple API for access. This is used within Twitter's Heron [17]. It has managed state, snapshotting and restoration to support fault tolerance. For scalability it partitions data to distribute across a cluster and uses Yarn to manage the containers.

3.2.1.10 Heron

Heron [17] provides stream processing at scale and is the replacement for Storm to help support easier debugging and increased scale. Heron was adopted by Twitter on June 4th 2015. It provides a solution for real-time user counts and real-time engagement with Tweets and advertisements. *Antares* provides aggregate functions as well, but it supports a wider functionality of querying.

Heron runs topologies (these are the same topologies as Storm executes – mentioned previously) and allows "backpressure" to adjust the flow of data. Spout backpressure allows the ingest rate to be monitored and modified to ensure that the ingestion rate is always equal to or less than the maximum ingestion capacity of the system. This means that no tuples are dropped and ensures no work is lost. This is different to Storm which drops tuples if it is unable to handle them. Backpressure is used so debugging is easier as you can see when a skew in data has occurred and identify the root cause of the error [17].

A Heron instance does most of the work and runs only a single task, therefore debugging is easier as there is only one task executing. Experiments show that Heron reduces latency and increases throughput.

3.2.1.11 Conclusion

This section has reviewed and identified stream-processing engines that could be used within the *Antares* system. After reviewing the literature it was identified that [12] and [5] do not have the SQL capabilities of ESPER (for more complex querying rather than GET, SET and delete – which is a trade-off for performance) and Drools [7] supports ACID transactions which can decrease performance and is therefore focusing on the wrong characteristics which are necessary for *Antares*. *Antares* is based around scalable and high performing querying and analysis of Twitter data.

Security is the main focus of [10], which is not necessary for *Antares*. Reliability and the ability to ingest high rates of data is one of the major contributions of *Antares* therefore a system like S4 uses messaging to support consistency and shredding to reduce ingestion of data when a node becomes over-saturated this was not suitable for the desired characteristics of Antares - namely the the scalable and performance required (wanted/stated/contributed) by the system.There is a lot of choice, but we chose ESPER due to its support for complex querying, low latency and scale, which is essential for the Twitter analytics applications that *Antares* must support.

The next section identifies different historic store which are regarded as scalable and examines the different characteristics then identifying the correct data store to extend.

3.2.2 Historic Processing

To chose a technology which would support scalabilty, flexibility and features to best allow for a layered and scalable approach to querying and processing Twitter data received as a stream a set of technologies were investigated. This involved identifying different scalable technologies and comparing them to see which would be fit for use within Antares to support the achievement of contribution –blah blah blah.

The historic store should provide a high insertion rate, low latency querying and scale-out functionality. The ability to support large-scale data processing is vital due to the petabytes of data being produced by sources such as social media and sensors. Traditional processing techniques do not typically meet the needs for processing such large datasets. Traditional relational databases do not provide the scale, and batch processing databases do not support timely enough responses to meet the requirements of real-time applications. Therefore new technologies and mechanisms are being explored.

This section reviews different database technologies aimed at solving these problems, by either using a new database approach or by extending traditional RDBMS systems. By reviewing the different technologies available a conclusion about suitable applications and workloads for each is drawn. Each of the databases is described and compared using common features of databases. The features that were compared are shown in the Table 3.1.

Characteristic	Description
Schema	Describes the schema if there is one and how the database structures data held within it
Joins	Whether the data layer supports joins
Open Source	Whether the database source code is freely available
Partition type	Describes how and if the database is partitioned (sharded)
Logging	Does the database log operations?
Locking	Describes the locks used by the database
Storage	Whether the database is in-memory or held on disk
Consistency	Whether the database uses ACID transactions or uses an eventual consistency model
Availability	This describes the probability that the database is operational at a given time
Partition Tolerance	Whether the cluster continues to work even without communication between all the partitions
Performance	This describes the performance of queries executed over the database
Scalability	This describes whether and by how much the database performance can be scaled out

Table 3.1: Database characteristics and descriptions

Before classifying databases in these terms, some of the terminology used in the classification is introduced.

Blanket Statement Removal: Removal of occasional statements (queries) that span the entire database.

CAP Theorem: CAP Theorem (consistency: a read sees all previously completed writes; availability: reads and writes are always successful and partition tolerance: during a network partition, a distributed system must choose either consistency or availability) describes that a system can never provide 100% of all three of these properties at any one time. Therefore there must be a trade-off between properties. For example if a system is very available and partition tolerant then it may implement eventual consistency [18].

Check and Set: Uses version stamps, so that when an object is retrieved, so is the version stamp, and this is then passed to the set method. The system will then verify whether the version number is the latest and allow the update, or, if not, fail it.

Cold Cache: When the cache starts it has no values - is empty - so there is no speed-up advantage to having it.

Consistent Hashing: When a hash table is resized only k/n keys need mapped where $k = numberofkeys$ and $n = numberofslots$, as opposed to all of them.

Chubby: Loosely coupled distributed lock service [19].

Decision Tree: A tree structure, which shows the consequences of a chain of decisions.

Hash Partitioning: A hashing function is used on the key (or another attribute) to calculate the location in the database the data is written to.

Dominant Workloads: Workloads that are major resource consumers.

HQL: Hibernate query language - fully object oriented similar to SQL.

Master-Master: All nodes accept requests and each node will propagate changes if replication is used.

MVCC: This is a concurrency mechanism, which marks old data "out-dated" rather than deleting it. There are multiple versions of data, which allows readers to see the data as it was when it was requested - even if it was updated during the read.

Optimistic Locking: Multiple transactions can compete without interfering with each other. No lock is used while writing and before committing the transaction verifies if any other transaction has modified the data during the read. If there is a conflict the transaction is rolled back.

Pig: High level programming language to abstract from Java.

Quorum: n machines must be available to make a read/write.

Referential partitioning: Used to facilitate joins across a shared-nothing network. It is a method of aggressive hash partitioning that attempts to take into account the foreign key relationships of the tables. During the data load, referential partitioning includes an additional step, which involves joining with the parent table to find the foreign key.

Shared Nothing: Each node is independent of all others; this includes independent memory and disk space. This means there is no contention for resources and that the nodes are self-sufficient.

Sloppy Quorum: All reads and writes are performed on the first n healthy nodes.

Split Execution Environment: Divides the environment that queries are executed in, providing the database layer and the application layer (a map reduce layer).

SSTable: This is a key-value file, which is used to provide persistence.

Vector Clocks: An algorithm for creating a partial ordering of events in a distributed system. Messages hold the state of the sending processes clock.

Workload aware partitioning: By monitoring query patterns and data accesses, it moves data (using graph partitioning) to ensure fewer cross server transactions, reducing the cost and time to query. When the query is analysed this analysis is outputted in the form of a graph. The graph joins with edges which transactions go between which nodes (tuples), these edges are then weighted to show how many transactions occur. This graph is then used to find partitioning that balances the load and decreases the weight of edges.

New database technologies are now outlined using the terms described in Table 3.1, which may be considered for use in *Antares*.

3.2.2.1 Relational Cloud

Relational Cloud: A relational database as a service, which provides strong consistency using ACID transactions. It uses a central coordinator to manage the system for the user and understand different workloads. To manage the workloads the system hosts multiple databases on one server, it then analyses these and moves them about as necessary [20].

Schema: A relational schema.

Query Language: SQL, the system uses a decision tree to identify the correct nodes to search for the data, unless there are a lot of requests then it uses the lookup table.

Joins: A coordinator is used to consolidate the database to reduce the amount of distributed joins, but they are possible. Therefore joins are generally executed within the database on a single node.

Open Source: No – MIT's investigation of cloud based technologies.

Partitioning Type: Workload aware partitioning which means by monitoring query patterns and data accesses, it moves data (using graph partitioning) to ensure less cross-server transactions, reducing the cost and time to query. When the query is analysed this analysis is outputted in the form of a graph. The graph uses edges to join transactions which are executed across different nodes (tuples), these edges are then weighted to show how many transactions occur. This graph is then used to find partitioning that balances the load and decreases the weight of edges. This provides less multi-node transactions as well as load balancing.

Logging: The databases have a combined log, which means they all commit together (group committing).

Locking: Provides locks for the transactions within the system (but tries to reduce multi-node locks).

Storage: Uses a standard DBMS with database servers, supports MySQL, Postgres and JDBC.

Consistency: The database supports strong consistency by using transactions. It uses multi-node transactions and tries to partition the nodes intelligently creating as few multi-node transactions as possible (more costly and larger overhead).

Availability: Replicates data by partition for availability.

Partition Tolerance: The system is a consistent and available system, when there are failures the system will repartition and move data to bring it back online.

Scalability: Relational cloud allows multiple databases on one server. Scalability is achieved through workload aware partitioning, which uses graph partitioning. This limits the scale, but a solution to this is to use blanket statement removal and sampling tuples and transactions, these heuristic methods help scalability. The system has not currently been tested on the cloud, it has however used 128 tables on eight machines and it showed throughput increased with scaling [21].

Performance: The latency is increased because of privacy and security in the coordinator, however this was the application of the database. To support improvements in performance a router is used to decide the node the query should be executed on and the distribution plan (to help load balancing and scalability). The distribution plan provides performance enhancement by improving the flow of the queries.

Fault Tolerance: Failover is supported through the use of replicas, which are handled by the coordinator while it is distributing the workload.

Elasticity: Graph partitioning of data provides near linear elastic scale-out even for complex transactions.

Applications: Due to the improvements in security this database could be used in the public or private cloud. It is also designed for applications that require high stability and elasticity.

Workloads: The database should be used for skewed and standard workflows. Currently it is suited to OLTP and web applications.

Conclusion: This database provides very good security methods (designed and implemented by MIT). CryptDb is used to encrypt the data in the database and then the user can query the encrypted data. Currently the system is in development and various elements need to be put together for it to work on the cloud. Performance is enhanced using its workload-aware approach to querying and distributing transactions and partitions.

Antares features focus on performance, –scalability and high ingestion rates of the Twitter firehose, Relational cloud however focuses on security. The privacy and encryption used in Relational cloud increases latency therefore it is not suited to a system which is focusing on performance enhancements like Antares.

3.2.2.2 HBase

HBase: Apache HBase is a NoSQL column store, which is based on BigTable. To scale-out nodes can be added. The architecture for this is master-slave, however multiple masters are used so there are no single points of failure. HDFS is used as storage and is tightly integrated with Zookeeper. The database was developed by Facebook [22].

Schema: There is no schema.

Query language: B-Trees are used to support quick range queries. Additionally map reduce and a HIVE interface are supported. The Hive interface is used to communicate with the underlying Hadoop architecture.

Joins: Joins are not supported.

Open source: The software is open source.

Partitioning type: The partitioning and distribution are transparent. The data is divided using a key range, these are called blocks, and are distributed to each node in the cluster. When querying, the correct key range is identified using a look-up table, which is denormalised to prevent bottlenecks.

Logging: Any updates to the system are logged to support crash recovery (WAL write ahead log).

Locking: There are no global locks, however locks and transactions are available at row level.

Storage: Updates are written in-memory and then periodically committed onto disk-HDFS.

Consistency: There is no global consistency, however the system does offer row level atomic transactions. Additionally the system supports wide scope user-defined transactions. There are optimistic concurrency control aborts if there is a conflict.

Availability: Highly available, but if a machine goes down there is a short period of unavailability.

Partition Tolerance: Replication over multiple datacenters supports partition tolerance.

Scalability: Parallel batch processing using map-reduce supports scalability.

Performance: Facebook uses the database for real-time messaging by load balancing, random sequential access and compression. It is a high performance system, however this is chosen over durability.

Elasticity: HBase uses a gossip protocol to communicate and a peer-to-peer architecture, which allows the system to add and remove nodes quickly. However this can only be done easily for reads. The data is split into regions; these are saved onto the HDFS. The regions have multiple replicas across different nodes. Region-servers are used to manage different region datacenter nodes. When new data is added this causes the current and newly added data to be split across nodes and replication to be considered. To rebalance this data the HDFS has to be re-balanced and the ownership of the data decided within HBase.

Applications: It is used for Facebook's messaging system. It is used for random real-time read/write access to big data applications that require cloud elasticity.

Workloads: It has been designed for write dominant workloads.

Conclusion: HBase is used for Facebook messaging. The system processes large datasets efficiently and quickly. It is eventually consistent and should be used for systems that are focused on availability.

3.2.2.3 MongoDB

MongoDB: MongoDB is the leading document store for available, partition tolerant systems. The system uses collections and stores data in the common JSON format. The network is a master-slave architecture [23].

Schema: The database has no schema and uses JSON objects.

Query Language: MongoDB supports the execution of map-reduce functions and employs a rich declarative query language. Additionally it supports ad-hoc queries.

Joins: There are no joins supported.

Open source: The system is open source.

Partition type: There is automatic sharding on a user-defined attribute which is transparent.

Logging: OpLog [24] is used, which resides on a local server and the update and modifications are written to this log.

Locking: There are no locks used.

Storage: Memory mapped B-Trees are saved to memory and then flushed to disk. It uses GridFS [25] for storing objects.

Consistency: The system is characteristically eventually consistent, however ACID transactions can be executed on fields.

Availability: The weaker consistency and no refusal of writes even when there are failures in the system supports high availability. Additionally the system replicates the data and each of the nodes storing the replicated data may also answer queries.

Partition Tolerance: The master-slave architecture supports partition tolerance as the slaves with replicated data can answer the query.

Scalability: If the system is partitioned it can support very scalable reads and writes.

Performance: If large map-reduce jobs are executed then execution is carried out in batch. This can result in longer return times, which is dependent on the complexity of the query and the size of the data. However this is to be expected when executing a map-reduce job. By using asynchronous replicas, query performance is improved because the master can send requests out at any time.

Fault tolerance: Replication of the nodes for failover and recovery is automatic. Replication is asynchronous and happens at the shard level. The system recovers from a failure by electing a new master. Server failure while a slave is reading from the log to replicate means that data is lost.

Elasticity: The cluster is elastic and supports load balancing.

Applications: The database is specialised for large datasets and supporting simple scaling to process and store it. The system is currently being used by Craigslist, which uses the database to archive information. Foursquare also uses the database for check-ins as it provides a mechanism for geospatial indexing and small location-based updates.

Workload: Reads and writes are scalable so both workloads are suitable for the database.

Conclusion: MongoDB is a popular document store, which provides a rich querying language as well as being very scalable and available. It provides strong local consistency, which reduces conflicts and stale reads and writes.

3.2.2.4 BigTable

BigTable: BigTable is a column-oriented store, which is one big table consisting of column keys. It uses a distributed storage system, which is partitioned into tablets each of these is a range of row keys. BigTable is a consistent and partition tolerant system sacrificing availability. The architecture is best suited to many smaller sets of data and has a limit of 100 column families with no restriction on the number of columns. It has no schema, automatically load balances and the structure can be modified dynamically [26].

Schema: The system is made of column families but is very flexible. There is one keyspace (the table) and data is denormalised to reduce response time. The schema is saved in Chubby.

Query Language: It allows simple queries and uses SSTables for easy lookup.

Joins: There are no joins allowed.

Open Source: It is not open source.

Partition type: The data is divided into shards using the row key and these are called tablets.

Logging: There is a commit log per tablet, which is stored in GFS (Google file system).

Locking: It uses a distributed locking service called Chubby, which uses a master to serve requests only as long as there is a majority of active servers.

Storage: All data is stored in GFS and SSTables provide persistence of immutable objects for lookup.

Consistency: The system supports strong consistency with ACID semantics and uses one server tablet per data shard. Transactions are available across entities and entity groups – this is limited to five for groups to keep performance up. It uses version control to try to reduce stale data, which can be controlled in two ways. The first is by keeping N copies, the second is by keeping data for a set amount of time.

Availability: Availability is limited as a trade off for strong consistency. If a tablet fails then the data contained in that tablet is unavailable until it is fixed.

Partition Tolerance: Replicas support partition tolerance for node failures.

Scalability: The system is very scalable and petabytes of data can be processed across thousands of commodity servers. Simply adding servers to the cluster can support scaling to different capacities [26].

Performance: Petabytes of data can be processed with high performance. The performance is tunable through the master and the bigger the scale the higher the performance.

Fault tolerance: Replicas are deployed using the GFS, execution is synchronous and the master maintains these. Recovery and error detection is also the responsibility of the master, which is completed by periodically polling for lock state. If the master receives no communication back or the server returns with a lost lock exception then the master attempts to acquire an exclusive lock on that file. If the server cannot acquire a lock then the server file is deleted.

Elasticity: The master supports elasticity by splitting the tablets when the size limit is exceeded and load balancing the database. Performance is sacrificed over durability, therefore performance may be slower but the data is not lost.

Applications: Applications suited to the database are web indexing, Google Earth, Google analytics, Orkut, Personalised search, Writely and Google finance.

Workloads: The system is best suited to large-scale datasets, heavy write loads, throughput-oriented batch processing jobs and latency-sensitive serving workloads.

Conclusion: BigTable is a popular NoSQL database that provides scalability and elasticity. ACID transactions are supported, which sacrifices some of the availability. The database was created by Google to solve scale problems and is used by the company.

3.2.2.5 HadoopDB

HadoopDB: This system creates a hybrid of map reduce and DBMS. The database has a shared-nothing architecture, which can easily be executed across a cluster containing commodity hardware. A cluster consists of multiple independent nodes, which are connected using Hadoop. Originally PostgreSQL was used, but the framework changed to a NoSQL database to increase performance. The system uses HDFS and a map reduce layer. The system uses a database connector for MySQL, Postgres and VectorWise. These allow the SQL queries to execute across the database and transforms the result to a key value pair for the Hadoop layer [27].

Schema: It uses tables, so that a derivative of the SQL language can be used to query the database and so joins, aggregation, selection, etc. are included. The most recent version is using a columnar database, where the database content is stored by column instead of row to improve performance

Query Language: HadoopDB uses Pig and HQL, it has a flexible query interface which uses SQL or map reduce functions. Queries are parallelised using Hadoop, which serves as a coordination layer. To improve performance mostly queries can only execute on a single node, which is easier after the map reduce layer. The map reduce layer is a master-slave architecture, in which the master controls the jobs and slaves track the tasks.

Joins: Allows joins but these are executed in the application layer by Hadoop.

Open Source: HadoopDB is open source, however there is also a commercial version called Hadapt.

Partitioning Type: Referential partitioning, this is used to facilitate joins across a shared-nothing network. It is a method of aggressive hash partitioning that attempts to take into account the foreign key relationships of the tables. During the data load, referential partitioning includes an additional step, which involves joining with the parent table to find the foreign key. This can be extended to an arbitrary number of tables.

Logging: A catalogue stores information about the partitions, datasets in the cluster and replica locations.

Locking: It is assumed that storage will be PostgreSQL or VectorWise, therefore there would be locks used for transactions.

Storage: Uses HSFS as a distributed file system.

Consistency: It is assumed that the framework has strong consistency using ACID transactions as the underlying databases do.

Availability: The framework is highly available across a cluster [27].

Partition Tolerance: If a partition fails then the performance degrades dramatically and can even fail until it is brought back up again.

Scalability: HadoopDB uses map reduce to scale-out at a lower price and is highly scalable by allowing distributed processing of large data sets. Hadoop's ability to schedule tasks supports scaling out to thousands of nodes.

Performance: The framework provides high performance and efficiency by pushing the query processing into the underlying database system. Similar to the parallel database, it provides load balancing, flexibility and extensibility.

Fault Tolerance: HadoopDB achieves fault tolerance by restarting tasks that have failed on other nodes. Its fault tolerant properties are similar to that of Hadoop.

Elasticity: There is a global and local hasher which partitions and load balances the partitions between the nodes in the cluster, which additionally provides a dynamic method for repartitioning data.

Applications: HadoopDB is best suited to cloud and big data applications and has been used for analysis of data within the semantic web for biological data analysis [28].

Workloads: Workloads that are analytical and use structured data are best suited for HadoopDB.

Conclusion: This is a system that continues to be extended and improved to generate a database using Hadoop technologies. It started by using PostgreSQL, but it is now looking into using a columnar DBMS.

3.2.2.6 Infinispan

Infinispan: Infinispan is a NoSQL, data-as-service, key-value store with a data grid architecture. It has a REST interface and is a consistent available system [29].

Schema: Is a key-value store and so has no schema. It uses hibernate OGM (for persistence) which doesn't have any particular schema as well, however JPA does have a class schema.

Query language: It uses map reduce functions and supports querying using hibernate search and Apache Lucene [30].

Joins: There are no joins supported.

Open source: It is open source.

Partitioning Type: It is a data grid for a main memory cache and so does not have partitions, it extends the cloud elasticity to the data layer.

Logging: It uses a log for transactions.

Locking: Locks are used, but with non-blocking algorithms. The acquisition of locks has been improved so that only locks on a single node are used, this prevents deadlocks on multiple nodes which would lead to updates no longer being possible.

Storage: It is stored in-memory and is a distributed cache. It is persistent and provides a very large heap.

Consistency: It provides ACID transactions and is highly concurrent.

Availability: Highly available. Nodes can fail but the system will still work. It has replicas across the network, as well as the ability to persist state to configurable cache stores.

Partition Tolerance: Consistency is chosen over partition tolerance.

Scalability: It is extremely scalable, as data is distributed equally and there is no limit to the size.

Performance: Response times are linear with low latency because it is in-memory.

Fault tolerance: There are replicas across the system, which supports high fault tolerance.

Elasticity: It is an elastic system, which is best suited for elastic data and the cloud.

Applications: Enterprise and cloud applications.

Workloads: Big data applications with data flows, which contain bursts of events that require consistency.

Conclusion: Inifinispan is an in-memory key-value store, which is highly scalable with transactions for consistency. It is elastic, which makes it very suitable for the cloud.

3.2.2.7 Titan

Titan: Titan is a distributed graph database, which can be used in conjunction with other databases. The database supports complex algorithms, which can be executed across the system to provide complex analysis of large datasets [31].

Schema: It uses a graph representation.

Query Language: It uses Gremlin, Frames and Rexter traversal.

Joins: It does not support global graph operations, so it does not provide joins. "Titan is a scalable OLTP graph database focused on handling a large number of concurrent transactions against a single graph" [31].

Open source: The system is open source.

Partition type: The partitioning of the database depends on the database used for storage.

Logging: The system has no logging.

Locking: Locking is user-defined.

Storage: The storage is pluggable and a variety of databases is supported such as Cassandra and HBase.

Consistency: This is again dependent on the database being used for storage.

Availability: Once again this is dependent on the storage used.

Partition Tolerance: Once again this is dependent on the storage used.

Scalability: It is very scalable and capable of supporting 3 billion nodes, 100 million vertices, 10,000 concurrent users and 50 machines [31].

Performance: The database supports concurrent transactions and can execute 49 million transactions in 2.3 hours.

Fault Tolerance: Fault tolerance is tunable and user-defined.

Elasticity: The database is designed for simple addition and removal of nodes.

Applications: It should be used for large-scale data that has complex relations that would be best represented by a graph.

Workloads: Quick reads would be the appropriate workload for the database as it is graph based.

Conclusion: This framework is best suited for extending existing systems' analytical capabilities. The graph can be used for quick reads and analysing complex relations. However the characteristics of the database are largely based on the system that it is extending.

3.2.2.8 SimpleDB

SimpleDB: This is Amazon's key-value store. Each key has a document, which supports multiple indexes. Documents are held in domains, and there is the ability to have domains in domains. The domain indexes are automatically updated [32].

Schema: There is no schema.

Query Language: The querying language is simple, with get, set and delete.

Joins: There are no joins allowed.

Open source: The system has no open source version.

Partitioning type: There is no automatic sharding.

Logging: There are no logs as there are not any transactions. Additionally the system does not use MVCC so there is no mechanism for identifying clashes on the client side.

Locking: There are no locks as the system is eventually consistent and executes concurrent writes.

Storage: The data is stored in commodity servers in a data centre.

Consistency: The system is eventually consistent. Provides a consistent read and an eventual read, however [33] found that there were more stale reads when the consistency was set at "consistent read". Eventual read also has lower latency.

Availability: It is highly available, even if some of the network is down.

Partition Tolerance: Even if there are failures or connectivity issues the system will still be available, therefore the system is partition tolerant.

Scalability: Due to its simplistic approach the system is very scalable.

Performance: Performance is tunable and exploiting its advantages like sorting attributes will support high performance.

Fault tolerance: Fault tolerance management is automatic and replication is asynchronous.

Elasticity: Amazon handles scaling and automatic elasticity.

Applications: The system is used for web and cloud applications.

Workloads: The system is suitable for high write loads that are not bursty.

Conclusion: Amazon SimpleDB is a cloud service, which does not require user management. Amazon provides the database as a software as a service. It provides the user with a lot of support, as well as a simple and easy way of storing data.

3.2.2.9 Neo4j

Neo4j: Neo4j is a leading graph database implemented in Java, which uses node and relationship properties. It provides the user with a REST interface for easy access and is a consistent and available system [34].

Schema: There is no predefined schema so it can be easily evolved.

Query language: It is a graph database so employs traversal and pattern matching.

Joins: The graph traversal is the equivalent of a recursive join.

Open source: The database has a dual license, so there is an open source version and an enterprise version.

Partitioning Type: The database can be partitioned, however this must be done manually.

Logging: Neo4j uses logging for transactions. The slaves log the transaction they have committed, which is then propagated to the master.

Locking: Locks are kept on the nodes being modified when adding, moving or removing data, each of these operations is treated as an ACID transaction.

Storage: The storage is a disk based Java persistence engine and there is an optimised storage manager for storing graphs. It is SSD ready.

Consistency: Neo4j uses ACID transactions for mutable operations and provides strong consistency.

Availability: The enterprise server has a high availability feature, which enables a fault tolerant database architecture, and horizontally scaling read-mostly architecture. It can be made fault tolerant by ensuring the slaves contain an exact replica of the master.

Partition Tolerance: Graph databases are inherently hard to partition, therefore the database is not partition tolerant as there are no partitions. If a section becomes unreachable then the whole system becomes unavailable until the problem is fixed this is so that there are no inconsistent writes.

Scalability: The database scales horizontally to billions of nodes and relationships and allows multiple databases on one server.

Performance: High performance, fast queries through traversals. Frequently outperforms relational back ends.

Fault tolerance: Slaves are used to make replicas of the master nodes to provide fault tolerance, which makes it a very robust system.

Elasticity: New nodes can be added dynamically, and relationships can be created between them.

Applications: The database is used for enterprise computing applications and data with relations. It performs efficiently for graph traversals when querying data, which means it is effective for connected data which would require multiple joins in a relational database.

Workloads: Read heavy workloads and analytical workloads where traversal and connectivity of data is important. External caching is needed for low latency reads.

Conclusion: Graph databases are becoming increasingly popular for use by social media sites to support networking and operations such as “a friend of a friend”. It provides users with ACID transactions and powerful querying capabilities for recording relationships between data.

3.2.2.10 Riak

Riak: Riak is an advanced key-value store, which can be treated like a document store. The system has no master, which means that all of the nodes in the system are equal. The database is distributed and a derivative of Dynamo, which uses a REST interface for connections. The system is networked in a ring and divided into partitions, each of which is managed by a vnode. Each data point is stored with a primary key, which is used for querying, however there are no secondary indexes, the rest of the data is stored in JSON format [35].

Schema: Riak has no schema and there is no set data type as all data is converted to JSON, which also means that any language can be supported.

Query Language: The database supports map reduce. The database does not support secondary indexing, however objects can be linked –the number of these is limited– however this does support range queries. Possible querying operations are limited to get, put and delete, however full text search is optional but can increase latency.

Joins: No joins are supported; they must be executed in the application layer.

Open source: The database is open source.

Partitioning Type: A hash partition is used on the primary key to partition the database. The network ring is divided into equal partitions to support load balancing. A gossip protocol is used to manage which nodes are in charge of which partition.

Logging: Statebox [36] can be used to store write state to increase concurrency.

Locking: The database does not have locks to prevent write conflicts and instead supports high availability by allowing writes to the database at any time. The consistency is kept by using semantic reconciliation. This where the statebox merges differences.

Storage: The storage is in-memory and supports some pluggable memory too.

Consistency: The system is eventually consistent and therefore does not support ACID transactions. The consistency is tunable and dependent on the number of replicas. The consistency can be based on each read supporting different degrees of consistency on each replica. It has optimistic concurrency and uses a MVCC derivative. Vector clocks are used for versioning.

Availability: If a VNode ceases to execute due to a failure then the other nodes execute the failed node's tasks, this provides the user with high availability. Any node within the network can service a request, therefore it is possible the system will always accept a read or write, however it does depend on the consistency level.

Partition Tolerance: Riak is a highly available system and if a partition is down consistency will be sacrificed to still allow writes.

Scalability: Riak is a distributed, horizontally scalable database, which scales to 10s of nodes for the enterprise version.

Performance: Riak can use pluggable engine stores, the recommended one is Bitcask, which supports tunable performance and durability. There is no single point of failure and maintenance can be carried out with a rolling start.

Fault tolerance: Replicas are deployed which support fault tolerance by taking over the tasks of failed nodes. Additionally it supports hinted handoff, so when the node is back online the new data can be written there. There is also no single point of failure. It uses the gossip protocol to identify who is alive.

Elasticity: Riak is elastic; it allows the addition and removal of nodes automatically, as well as providing automatic load balancing. This works well for small loads but is completely unresponsive for larger loads. When a new node is added partition ownership is redistributed and data is transferred immediately.

Applications: This is a scalable database with a REST interface and is therefore suited to web applications. It is fully elastic so can be used in the cloud. It should only be used if consistency isn't the main concern and availability is the priority.

Workload: The system is suited to intensive read and write workloads because of its tunable consistency.

Conclusion: Riak provides a more advanced key-value store, which is scalable and provides a REST interface. Although it is essentially an available partition tolerant system, consistency can be altered and tuned for the application's specific requirements. Storage for the system is also pluggable which supports more flexibility.

3.2.2.11 Conclusion

This literature review identified many different technologies, these were evaluated and one chosen to extend and use within *Antares*.

Relational Cloud provides an SQL solution to scalable processing however the emphasis is on security making a trade off between that and performance. *Antares* does not require high security and instead aims to improve performance.

SimpleDB provides scalable database solutions, however no complex querying is supported and they were therefore not appropriate for *Antares*.

Databases [35] and [22] predominantly use map-reduce and have no secondary indexing, therefore making geospatial querying more complex and not suitable for *Antares*.

HadoopDB was focusing on partitioning the database to support scale-out functionality for the cloud. This did not meet the requirement of high ingestion of large scale data to be stored and processed, of *Antares*.

Graph databases can be notoriously hard to partition therefore Titan and Neo4j were not the optimal solution for the system.

After reviewing all of the different technologies the first decision was to use an open source technology so that experimentation was possible. The database selected was Cassandra due to its scalability. In particular, it supports high write rates, which is important given the rate at which Tweets from the firehose need to be stored. Additionally the data store is easily and simply horizontally scalable with easy indexing for more complex querying. This gives it the potential to provide low latency responses to queries. Cassandra is now examined in more detail.

3.2.2.12 Cassandra: in Detail

In Cassandra each dataset is contained within a keyspace; these are the equivalent of a database in RDBMS. A keyspace contains column families (tables), shown in Figure

3.1, which hold a set of rows. Each row has a unique identifier – a row key. Each row can have a different number of columns; there are static and dynamic column families. Static column families use a static set of names, although the number of these used in each column can vary depending on the data ingested. Dynamic column families store arbitrary column names taken from the data.

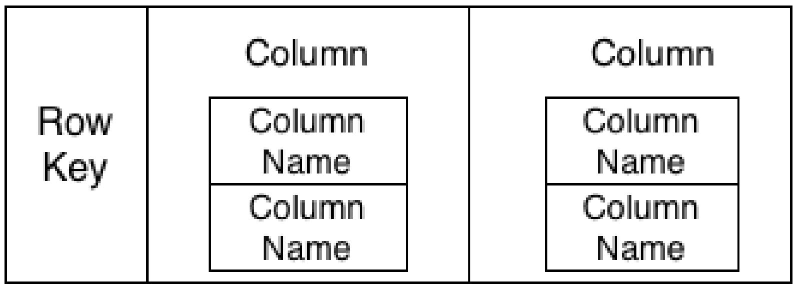


Figure 3.1: A Cassandra column family

A Cassandra cluster is assembled using a ring structure as shown in Figure 3.2. There is no master node and each node is responsible for managing the data stored on that node only. Adding nodes to the cluster supports horizontal scaling. To improve performance, as a node is added, the data is automatically load balanced across the cluster in milliseconds.

Data is partitioned across nodes in Cassandra to improve performance and dependability. How this is done depends on the partitioner used and the replication factor. The default partitioner for version 2.x of Cassandra is a Murmur3Partitioner, which uniformly distributes data across the nodes using the MurmurHash hash values. The row key is then used to distribute column family data across nodes in a cluster.

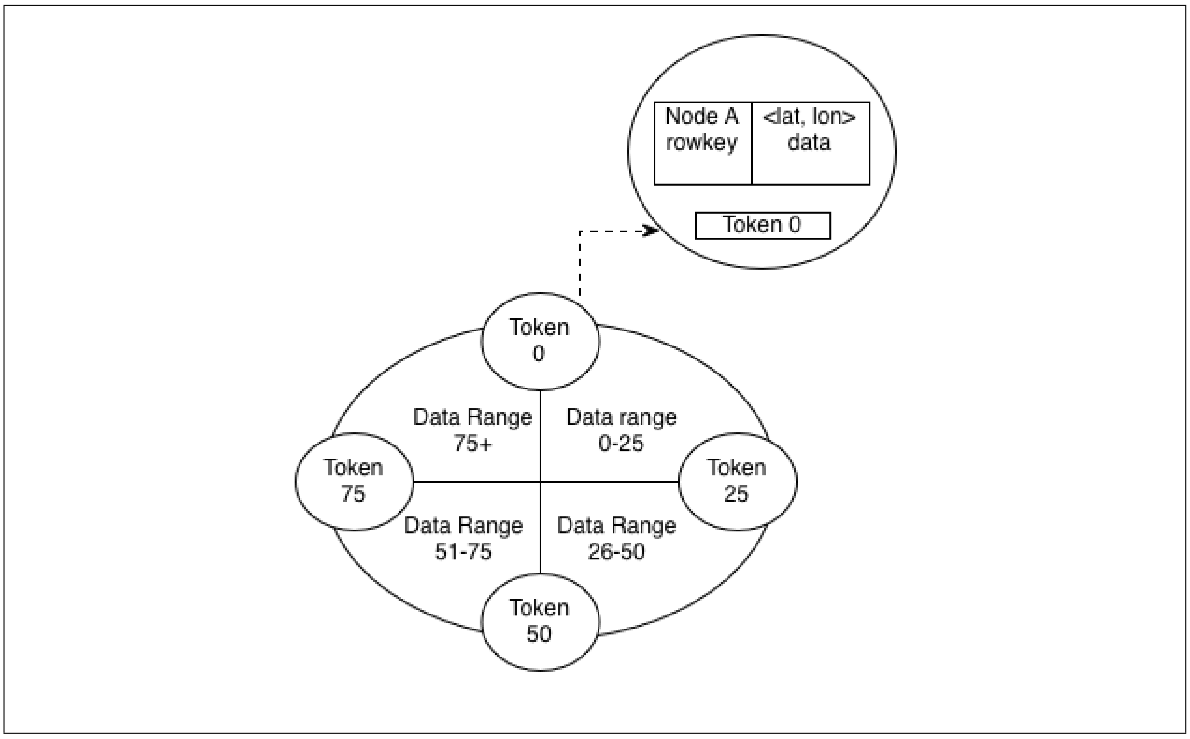


Figure 3.2: A Cassandra cluster

The hash of the row key is calculated and used to give the data a token. The token is used to place the data on a node, as shown in Figure 3.2. By using the row key to partition data the rows are never divided, therefore decreasing the response time of queries as only one node will need to be queried. The set of nodes can be viewed as a ring divided into different ranges. Each hashed key is part of a range within the set of nodes. For the cluster used by *Antares*, the ranges are equal, as each of the machines has the same specification.

Tokens are used to load balance the cluster. Each token informs the node of the range of data it is responsible for. Queries are distributed to any node in the cluster and then directed to the node which owns the data – this can be determined using the token. When an insert is accepted by a node in the cluster the row key is hashed to calculate the token. The token is then used to locate the node the query needs to be executed on. If it is data which needs to be appended to a row currently residing in the store then the request is submitted to the node which owns that data. If it is a new row it can be written directly to the node as rows are not stored sequentially.

A gossip algorithm is used to distribute information to the cluster about additional nodes or failures. The replication strategy used in *Antares* is “simple strategy” – this means that each row is saved on a designated node, and also one node along the ring, moving clockwise. This provides fault tolerance for the cluster by ensuring no single point of failure. The cluster uses the load balancing policy, token-aware balancing, which evenly distributes queries across the cluster. They are then either satisfied by the receiving node or sent to the correct one for execution.

Without the correct design using Cassandra for scalability, fault tolerance and speed becomes inefficient and impractical, limiting data analysis rather than empowering it. *Antares* optimises the database structure for efficient querying. Stream processing and database systems have been reviewed, however the next section identifies current combined querying systems.

3.2.3 Combined Processing

The ability to combine stream and historic processing quickly and efficiently can be a difficult problem to solve. This has been exacerbated by the sudden increase in the velocity and volume of datasets such as those generated by sensors and social media. New mechanisms and technologies are therefore being used to support "hybrid" processing. This section investigates different current systems which use combined processing of voluminous data, which are then compared with *Antares* to describe the differences, similarities and contributions of *Antares*.

3.2.3.1 StreamInsight

StreamInsight analyses event data being streamed in from multiple sources and gains insight through historical data mining [37]. It provides a framework that sits on top of an SQL server and is free to download. A dashboard for users to interact with and gain support is available. The dashboard provides graphical displays of the results. The system has a flexible schema based on the stream data it is collecting, however the underlying database is restricted to SQL relations.

Antares extends Cassandra therefore is not restricted by SQL relations and supports a mechanism for storing data where performance is not limited by relations and joins. The data is denormalised in the database which allows for querying of only one "table"

- this is because an index is created on the data relation and stored in one table. Then the database only needs to read from that table, as opposed to reading from two tables and joining the correct subset of data. Then if another relation is required the table (or index) is created for that too, this duplication is not a problem as the database can be scaled out. The system is made durable by logging operations to an event log.

Streaminsight uses query checkpointing to provide high availability, where as *Antares* uses asynchronous execution of queries for this. It allows the queries to execute in parallel so that the availability is increased as more queries can be accepted rather than waiting for the query to execute before accepting another (synchronously).

Streaminsight has no simple way of scaling-out, although additional nodes can be added there is no suggested mechanism for how the query would be executed over multiple nodes with the relational data. The data may be related to data in tables on another node, this would therefore need to be joined in the application layer. This would decrease performance and increase query response time. Unlike *Antares* which supports the addition of nodes to the cluster even while the cluster is running. The data is then automatically load balanced by Cassandra to support distributed execution for increased performance. Data can be partitioned but it does not support partition tolerance.

It is a high performance system supporting response times of milliseconds. Applications it is suited to are financial trade feeds, operational data from sensor networks, manufacturing equipment and data center monitoring infrastructure. This is a commercial system that uses stream and historical analysis for businesses. It provides analysis of complex events and exploits the advantages of the underlying Microsoft SQL Server [38].

3.2.3.2 Truviso

Truviso [39] is an append-only stream processing system, which uses continuous queries to incrementally update input streams. It uses CQL from the Stamford STREAM project [40]. This is the same query language that *Antares* is used as the base of describing the *Antares* querying language. Truviso is used for counting and aggregate functions, Antares supports temporal and geospatial querying therefore incremental updates are too simplistic for the system.

The system ingests a stream and produces a processed stream as output. The system uses raw and historical streams, the raw is from sensors (real-time) and the historic is from databases, which allows the combination of two. A typical query divides the stream into smaller streams; these are processed and historical data included in the analysis if required. This result is then converted into a result stream. *Antares* uses a similar technique but it is based on the temporal nature of the query where as Truviso divides the streams to help with performance only - *Antares* does include the stream analysis to support better performance as well.

Truviso is used for applications, which are involved in the financial market, monitoring systems, real-time decision support, fraud detection (cross channel fraud) and on-line gaming (detecting malicious behavior and monitoring QoS). The project is not open source, but provides an interesting solution to streaming in conjunction with the SQL database PostgreSQL. These applications take advantage of the transactions used within the system, however *Antares* has no requirement for transactions as it would only decrease performance which is the key characteristic of the *Antares* system.

3.2.3.3 Cloud Dataflow

Cloud Dataflow [41], [42] is a Google product for unified programming; the SDK is available open source. This product was introduced in 2014 by Google and emphasises the importance of combining batch processing and real-time stream analytics. It uses windowing techniques to add temporal querying. This is the same as *Antares*, which also uses time windows, however *Antares* uses the time window to identify which query to execute. Dataflow uses a layered technique as well to provide stream and historic querying, however the time windows specified must be for either historic or stream and different queries must be specified for each. *Antares* supports one querying language for both and uses a query monitor to execute these by identifying the time window the user has specified (past, current, future or both). Antares also supports combined querying which supports triggered and simultaneous querying.

It supports a wide range of data processing scenarios including session analysis, anomaly detection and funnel analysis. *Antares* focuses on Twitter analysis on in this thesis but has the ability to be used for other applications.

The service is fully elastic as the service is managed and deployed in Google's cloud. Therefore resources are added and removed as is required for processing, which is automatic. This is different to *Antares* as nodes must be added and removed manually, however load balancing is automatic and there is no requirement to shut down the cluster for this adjustment. These features mean dataflow and Antares are horizontally scalable.

Fault tolerance is managed by the service as well, and fault tolerant consistent execution is guaranteed regardless of the size of the data or cluster and the complexity of the data. *Antares* supports fault tolerance through re-tries and works with eventual - every write will eventually be correct.

The system is best suited to applications with high-volume computation, workflow synthesis, and extract transform load (ELT). The system provides a managed service for users to use quickly and efficiently.

3.2.3.4 Spark

Spark [43] combines a stack of high-level tools and supports their combined use. The tools are: Spark SQL, MLib, GraphX and Spark Streaming. These tools support streaming and historic analysis and Spark is a tool that can be used to manage and create workflows for these. This is similar to *Antares* which uses a layered approach to provide improved specific features - in this case stream, historic and combined analysis. However *Antares* uses the query monitor to provide a transparent layer to which any of the queries is mapped to the correct querying mechanism.

Spark allows pluggable storage to hold the data before and after processing. It is a high performance system, which executes programs up to 100 times faster than Hadoop MapReduce [44] for in-memory computations, and ten times faster for disk-based analytics. Applications can be written using a variety of languages, exploiting a set of inbuilt operators. It can be deployed in most environments and supports a wide range of data sources. Spark is predominantly a workflow engine which combines sources, so not a store itself. Spark is a similar system to *Antares*, supporting more tooling for analysis, but no transparent input of a variety of different querying techniques.

3.2.3.5 Summingbird

Summingbird [45] is a hybrid system which combines streaming and map reduce functions to execute on Storm and Scalding. It has three modes: batch, stream and hybrid (which uses both). This is the same as *Antares*, however as mentioned with other combined systems it is missing the the transparent submission that *Antares* has.

It sacrifices fault tolerance to provide quicker more efficient responses to eradicate the slow response times of batch and map reduce functions executed in Hadoop. Summingbird is a layer on top of Storm and Hadoop which supports the mapping of both stream and batch processing. As has been mentioned Storm is shreds data if the ingestion rate becomes too high, this is not a feature that *Antares* implements - it uses buffers and has a higher ingestion rate so that data does not need to be shredded, therefore Storm would not be suitable (which is the underlying technology of Summingbird). Storm has also been changed and extended to create Heron which adds to the functionality and is a replacement for Storm - so would not be suitable.

State is updated incrementally because of the large batch jobs that are executed over the data. *Antares* identified the need for faster combined processing of large datasets, however it uses optimisations to index and ingest data using a NoSQL store to allow for low latency querying and efficient combination of stream and historic processing.

3.2.3.6 Lambda

Lambda [46] is an architecture that uses Spark, Kafka [47], Akka [48], Cassandra [49] and Scala [50]. It provides a layered approach like *Antares* to provide support for specific queries. However as has been mentioned there is no transparent submission like other systems, where as *Antares* does have that feature.

Lambda supports a variety of different analytics supporting the user for a wide range of processing, where as *Antares* focuses on providing scalable and low latency querying for Twitter data providing the user with easy and transparent methods of submitting any query through a user interface and the system itself identifies which query to execute.

It supports fast access to historical data on the fly for predictive modelling with real-time data from the stream. Designed to handle massive amounts of data by taking advantage of both batch and stream processing. This is similar to *Antares* which provides stream, historic and combined querying. It uses Spark to combine both types of processing. It supports complex analytics using Kafka for stream processing and Cassandra to store historic data. Cassandra is used within *Antares* too for its scalable and flexible characteristics.

This became an Apache project in 2014 and demonstrates the importance on combining stream and historic data with efficiency.

3.2.3.7 Conclusion

Antares required scalable storage and an efficient ingestion of data. [43], although a high performing system, required additional tools to be used for the above aim and is predominantly a workflow engine, which does not meet the requirements of *Antares*. [45] uses map reduce which requires large batches of data to be analysed for a longer period than was suitable for use within *Antares*. *Antares* aimed to return in near real-time when querying the database to combine data with the stream in a timely manner. [46] and [41] were not available when *Antares* was first designed and have been added to the related work. [51] does not provide the scale necessary for *Antares* without complex partitioning. Therefore it did not make sense to use a tool which is not designed specifically for scaling horizontally unlike Cassandra. [52] is not open source and requires the addition of a database to allow for storage.

Antares utilises open source software, which can quickly and efficiently stream processes and query a database. Therefore it was decided to exploit the scale ESPER and Cassandra to extend the system and provide low latency querying for stream, historic and combined analysis. *Antares* also supports querying of complex data types such as geospatial data. Therefore the next section identifies techniques for processing and scaling out.

Antares provides a quick and efficient mechanism for entering user-defined parameters into an interface for interaction and visualisation. The framework is an optimisation of existing technologies used to overcome slow response times from query execution over large datasets that have been saved to disk.

Truviso decreases performance for transactional characteristics, which does not meet the requirements of *Antares* - which are based around scale and performance. Therefore, it is not a suitable system for use.

Dataflow is a high performance stream and historic analysis system designed and implemented by Google and based in the cloud with elastic features for scaling in and out automatically. However it does not feature the combined querying mechanism of *Antares* or the simplistic transparency of inputting one query for any querying mechanism - stream, historic or combined.

Spark is a similar system to *Antares*, supporting more tooling, however, it has no transparent input of a variety of different querying techniques.

Summingbird did not have the desired features of *Antares* it did not provide buffering and high ingestion rates of *Antares*. It is also based on Storm, which has now been replaced by Heron and implements the shredding technique.

Lambda is very similar to *Antares* and provides a layered approach with a wide selection of tools, however it does not have the ease of use and transparent submission of *Antares*. *Antares* also focuses and supports scalable Twitter analysis with low latency querying and high throughput. This is different to Lambda as it allows the user more freedom to design and build their own system where as Antares has already been designed and implemented to provide large scale ingestion and low latency querying.

Antares provides a transparent means of submitting stream, historic or combined queries through a user interface to support scalable and low latency Twitter analysis. It is different from other layered approaches as it gives the user optimised functionality and high performance with little effort from them. It exploits temporal indexing features to enable large scale Twitter analysis.

3.3 System Architecture

The previous section justified a choice of a combination of ESPER and Cassandra to provide the scalable, low latency and high ingestion rates described in features one-three for requirements of *Antares*. The literature review identifies combined analysis systems and compared them to *Antares*. *Antares* supports a more focused approach on scale and high ingestion rates of temporal and geospatial data than other systems. It supports transparent submission of queries independent of what type of query is being executed. This section describes the architecture which supports then then moving on to the mapping between the architecture and the querying mechanisms.

Antares is a scalable, low latency analysis tool, which provides complex processing of historic and stream data and the ability to combine both. One motivation behind the design and development of the system was to exploit the scalability of NoSQL technologies to ingest large data streams. Scaling out databases so they can support the fast querying of large-scale data can be challenging in a system which is continuously ingesting new data. Having the ability to return queries in near real-time allows the user to combine these results with stream processing results and in a timely manner, supporting insights from both past and current events. The queries executed by *Antares* can be divided into three categories, stream, historic and combined.

Antares is based around two technologies ESPER and Cassandra. Optimisations were designed to remove the inefficiencies for querying both stream and historic data. There is a web interface that accepts queries from the user. *Antares* has a query monitor for interpreting whether a query is stream, historic or combined. This solves the problem of how to query the different technologies. The system accepts stream data for storage and querying simultaneously. When a Tweet is ingested it can be processed in ESPER if a stream query is executing. Each Tweet is also stored in Cassandra to enable historic queries. This is executed concurrently as the queries are executed over the store.

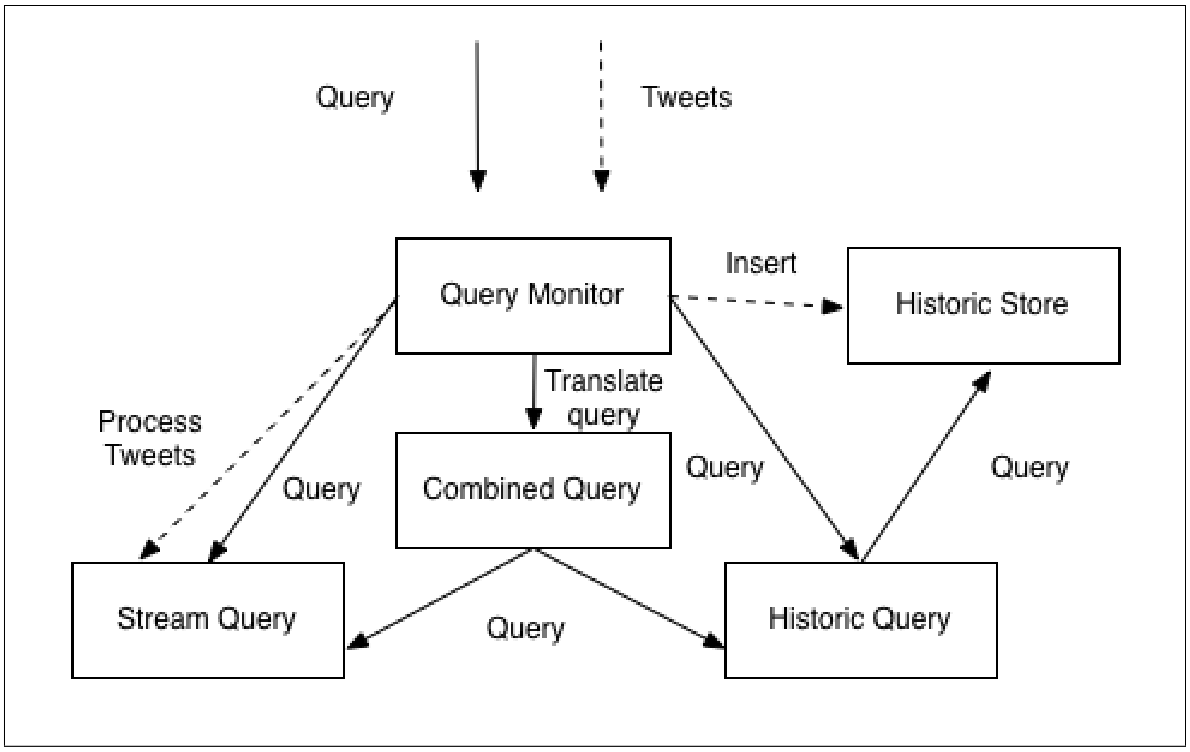


Figure 3.3: *Antares*: abstract architecture

As a query enters *Antares* it is passed to the query monitor shown in Figure 3.3. Here the time parameters are extracted to identify whether a stream, historic or combined query should be submitted. As a Tweet enters *Antares* it is stored in the historic store this is shown with the dotted arrow in Figure 3.3. The Tweet is also passed to the stream query if one is being executed.

3.3.1 Stream Querying

A stream query is executed when a time period starts or ends in the future. This is then sent to the query monitor, which identifies that this is a stream query and sends the query to ESPER. The query monitor uses the parameters set by the user, through the web interface, and the time-window and constructs a query for ESPER. Once the query is sent to ESPER a continuous query is executed over the time-window and results which meet the constraint specified by the query are returned. This is shown in Figure 3.4.

A Java class was designed and implemented to represent all of the values specified in the Tweet data model in Section 3.9. This class enabled the ESPER query engine to use Java objects (the pre-defined Tweet object) to identify different parts of the Tweet meta data and query them. As each Tweet enters *Antares* if a stream query has been executed then the Tweet is passed to ESPER. ESPER then uses the Java class to convert the Tweet into the Tweet Java object. Once this has been done then the Tweet object is inspected for any information that is related to the query constraint. For example when an aggregate query is being executed then if the Tweet meets the query constraint the count is increased.

All continuous query metadata is held in memory by ESPER for each query executing at the time. Once the query constraints are met the result is take out of memory and passed back to the query monitor to be displayed to the user. The size of the dataset is constrained by the amount of RAM. This is why the historic store is used to support scale. It is assumed that the user will execute stream queries more frequently and for shorter periods of time. Where as the database is used for wide-ranging and scalable execution across long time periods.

ESPER supports 10,000 events per second [53]. Therefore it is assumed that for *Antares* this rate is high enough to provide reliability - therefore ESPER will not block or drop messages.

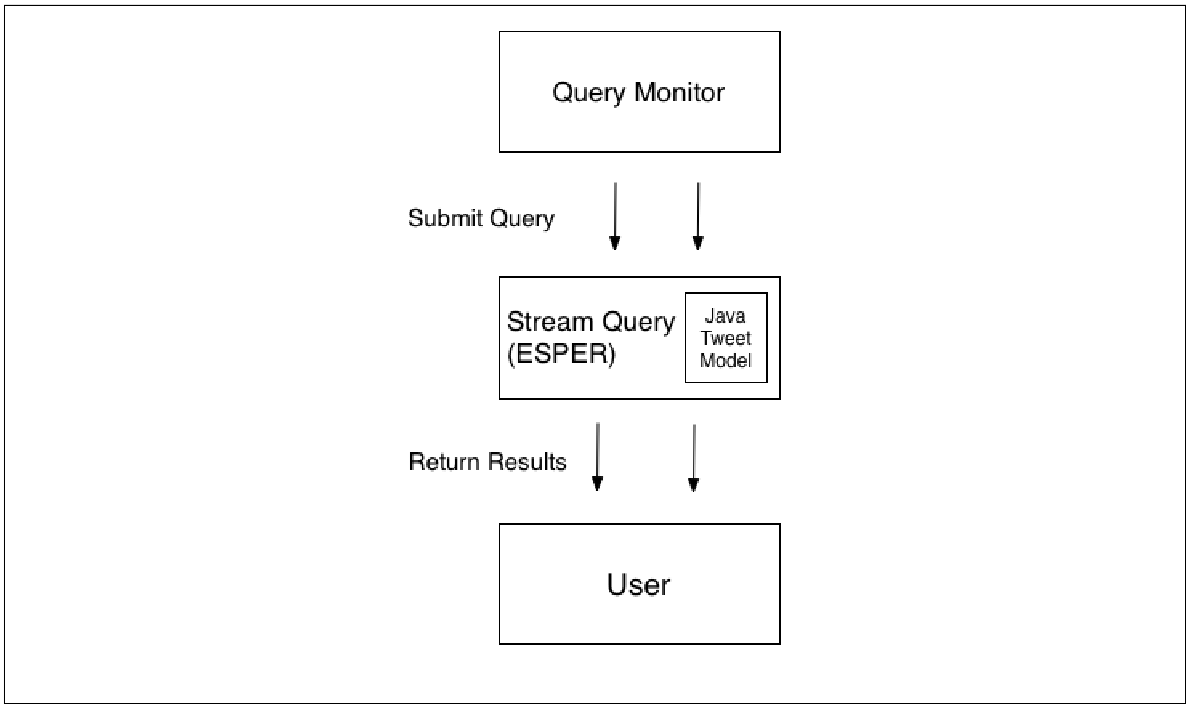


Figure 3.4: *Antares*: stream querying architecture

Now the syntax of the language will be described. Queries can be constructed in the form of:

SELECT $a_1...a_n$ FROM s <WHERE $p>$ || <HAVING $p>$

where $a_1...a_n$ is a projection list, s is a stream of Tweets with a time window which represents a period of time from the current time to a point in the future and p is a predicate.The angular brackets <> denote an optional term such as the $WHERE$ and $HAVING$. One or none of these may be contained in the query. There are restrictions on stream queries, the time window is constrained from the present to any point in the future. The syntax for this language in this thesis is focused on Twitter analysis so must use the data model specified (this is due to the focus of the thesis- to explore scalable Twitter analysis framework solutions). This also means the $FROM$ clause should only contain one stream. The projection list must also not contain aggregate functions other than $COUNT$. The evaluation of the stream results in a stream of the projection list variables that adhere to the predicate.

3.3.2 Historic Querying

A query is historic if both time parameters are specified in the past. The query monitor, which examines the parameters entered by the user through the web interface verifies this. A query is then constructed consisting of the time period and any other parameters specified by the user. The query is then executed by Cassandra, as shown in Figure 3.5, and the results returned to the user.

Now the syntax of historic querying will be described. Queries can be constructed in the form of:

SELECT $a_1...a_n$ FROM h <WHERE $p>$ || <HAVING p>

where $a_1...a_n$ is a projection list, h is a database of stored Tweets, with a time period from the current time to a point in the past and p is a predicate. The angular brackets <> denote that a term is optional, for example, $WHERE$ and $HAVING$. One or none of these can be used in the query.There are restrictions on the historic queries, the time window must start in the past and end no later than the current time. The *FROM* clause is primarily constrained to the schema used in this thesis to improve performance. The projection list must also not contain aggregate functions other than *COUNT*. The evaluation of the historic store results in the projection list variables that adhere to the predicate.

The following pseudo code executes a query to ESPER or Cassandra depending on the timewindow specified by the user. This is done transparently to the user and the Query Monitor determines whether the query should be a stream, historic or combined query. The condition x could be threshold or hashtag, however this pseudo code is used to generalise the process of mapping the queries.

Algorithm 1 Execute Query

```
input:
date starttime # beginning of the time window
date endtime # end of the time window
variables:
date now # currenttime()
date timewindow = endtime - starttime
if timewindow <= now then
   find schema
   execute query over cassandra
   return query result
else if timewindow >= now then
   execute query in ESPER
   return results each time condition is met
else if timewindow > now AND timewindow < now then
   compile EPSER query
   compile cassandra query
   execute query over cassandra and ESPER
   return query results
end if
```

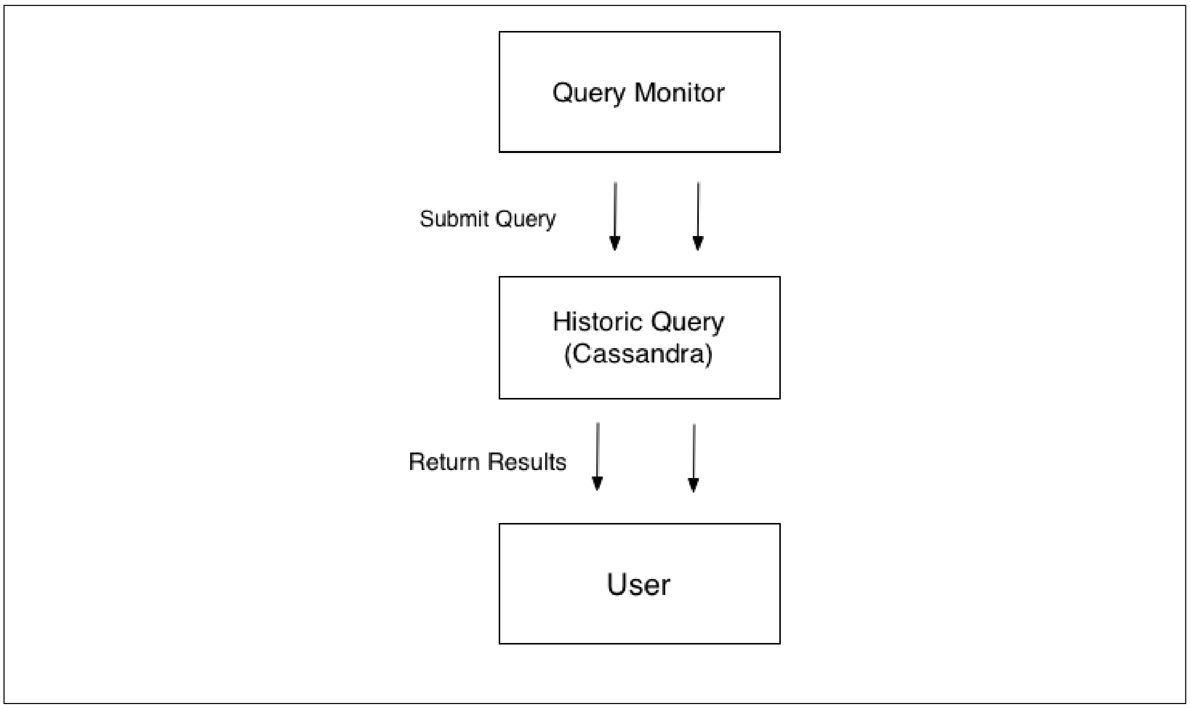


Figure 3.5: *Antares*: historic querying architecture

3.3.3 Combined Querying

Combined queries consist of a time-window which spans the past and the future. The start time will begin in the past and the end time finishes the time-window in the future. The query monitor constructs a stream and historic query, this is done by decomposing the time period into the time in the past (start time until now: the time-window for the historic query) and from now until the future end time (the time period for the stream query). There are two types of combined query.

The first type submits a query to ESPER. ESPER executes a continuous query, processing each Tweet that is ingested from the stream. It uses the same mechanism as previously described to identify whether the query constraint has been met. If it has then an historic query is executed, which is shown in Figure 3.6 where the parameters of the query are passed to the query monitor and then an event is triggered from ESPER to the historic driver which queries Cassandra, this is all then returned to the user.

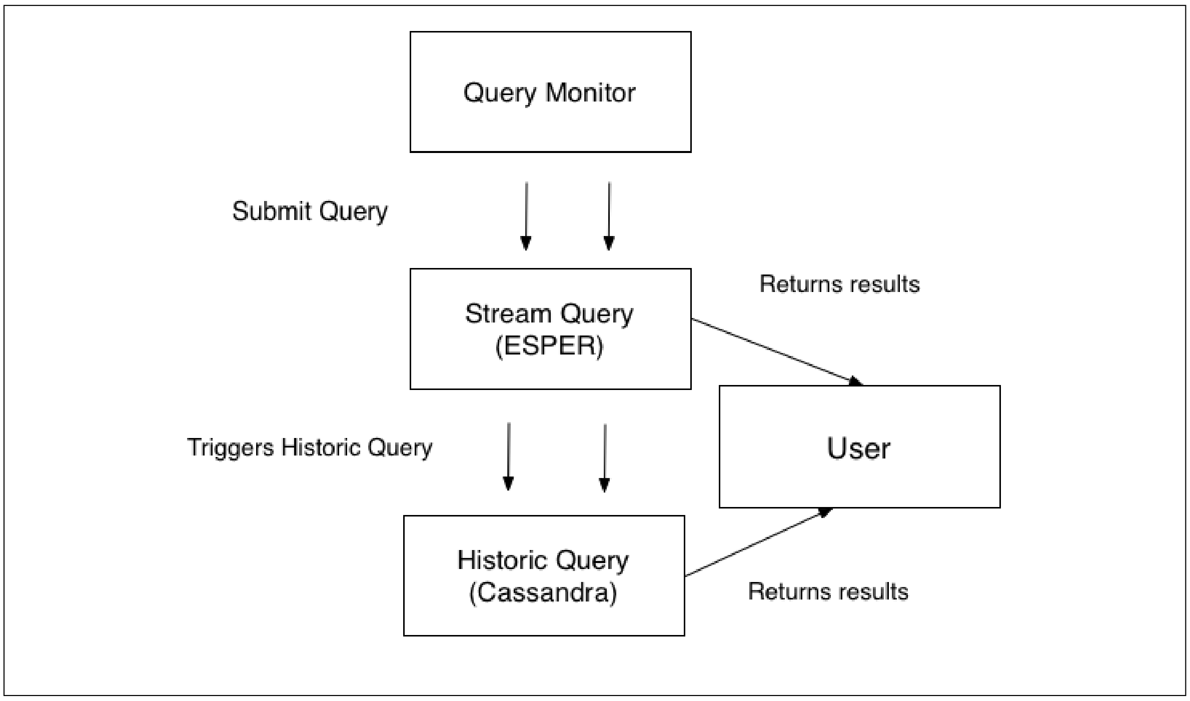


Figure 3.6: Antares: combined querying architecture (type 1)

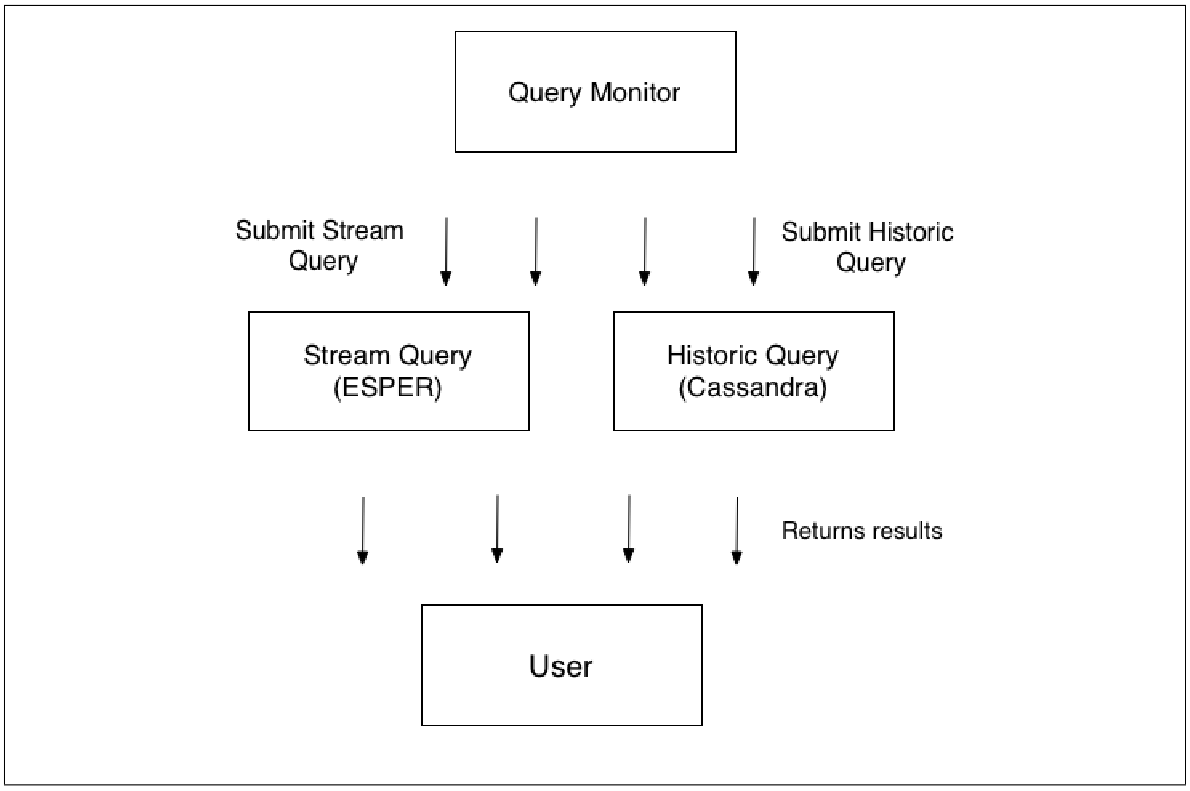


Figure 3.7: Antares: combined querying architecture (type 2)

The second type of combined query executes a stream and a historic query at the same time, as shown in Figure 3.7. The results are then returned to the user. The query monitor receives a query request. The query monitor then identifies whether it is a combined query. If it is then two queries need to be constructed, therefore there is a slight delay in current time for the queries, however both queries will have the same current time - therefore it does not cause any problems. The historic query uses current time as the end time of the query and the stream query uses it as the start time.

The continuous query is executed first to ensure that the current time for the stream starts as quickly as possible. Then the historic query is executed - it is not important how late this is submitted, as all Tweets are stored as they enter *Antares*, therefore there is no loss of data and the query searches the user-defined time-period independent of the submission time. If the stream query fails then the historic query is not executed and the query is re-tried. This method of submission and execution ensures no race conditions as the two queries will always be executed in the correct order.

The stream processing system is required to support the scale of the system and maintain intake of requests. Without ESPER the amount of requests to the database would approximately double, therefore there would be more strain on the database. Once a query is made to the database if it is unsuccessful there is a timeout of 5 seconds then the query is retried. The timeout was chosen to ensure that enough time was given to allow for multiple queries to execute before trying again. If the time was too short, for example, 100 milliseconds then the system would still be overloaded giving it 5 seconds allowed for a long enough time to retry without being too long for a noticeable delay to the system. This would become increasingly more probable as the number of requests increased and then the user may get blocked waiting as the system saturates - this becomes a significant problem when you are trying to write petabytes of data while listening for requests off thousands of users. ESPER allows the stream queries to be executed across another layer of the system, which reduces the load on the database and ensures that the queries can be returned in near real-time. The timeout does not occur so there is no substantial loss to service. Therefore the design decision of including ESPER rather than executing all queries over the database provided higher ingest rates and lower latency queries to support features one and three.

Now the syntax of the combined querying will be described. Queries can be constructed in the form of:

$$\text{SELECT} \quad a_1 ... a_n \quad \text{FROM} \quad sh \quad \text{<WHERE} \quad p > \quad || \quad \text{<HAVING} \quad p >$$

where $a_1 ... a_n$ is a projection list, sh is a stream/database of Tweets, with a time period from a point in the past to a point in the future and p is a predicate. The angular brackets <> denote that the term is optional. In the case of the $WHERE$ and $HAVING$ you can use one or none of them. The combined queries time window is defined as any point in the past or future and determines the way in which the query is executed. The projection list must also not contain aggregate functions other than $COUNT$. The evaluation of the stream/historic store results in the projection list variables that adhere to the predicate.

The query language is mapped to the back end by a series of steps and functions Figure 3.8 shows these functions. The functions are described below - time window and collect. The time window function constructs a query using the time period specified, sending the query to s (stream), h (database) or hs (both). The collect function is a cross product of all of the results after they have been evaluated to return to the user.

```
Time_Window:

if(time >= now)
send to h

else if(time <= now)
send to s

else
send to sh
```

Collect description:

$$\text{SELECTION} \quad h, \quad s => \quad \text{STREAM} \quad hs$$

In Figure 3.8 the user inputs a select query which is based on a time period which starts in the past and ends in the future - this is a combined query. The predicate defined in the example is a hashtag therefore the query will return all Tweets which match this hashtag. Once the query has been submitted the time_window function determines which queries to compile to and which layers the query should be submitted to. As the figure shows the example uses a combined query so a query is compiled for each layer. As can be seen the queries are both selects and contain a status in the projection list and a hashtag is used as the predicate. Once the queries have been executed the results from either layer are collected as a stream of values (or statuses in this example) and returned back to the user.

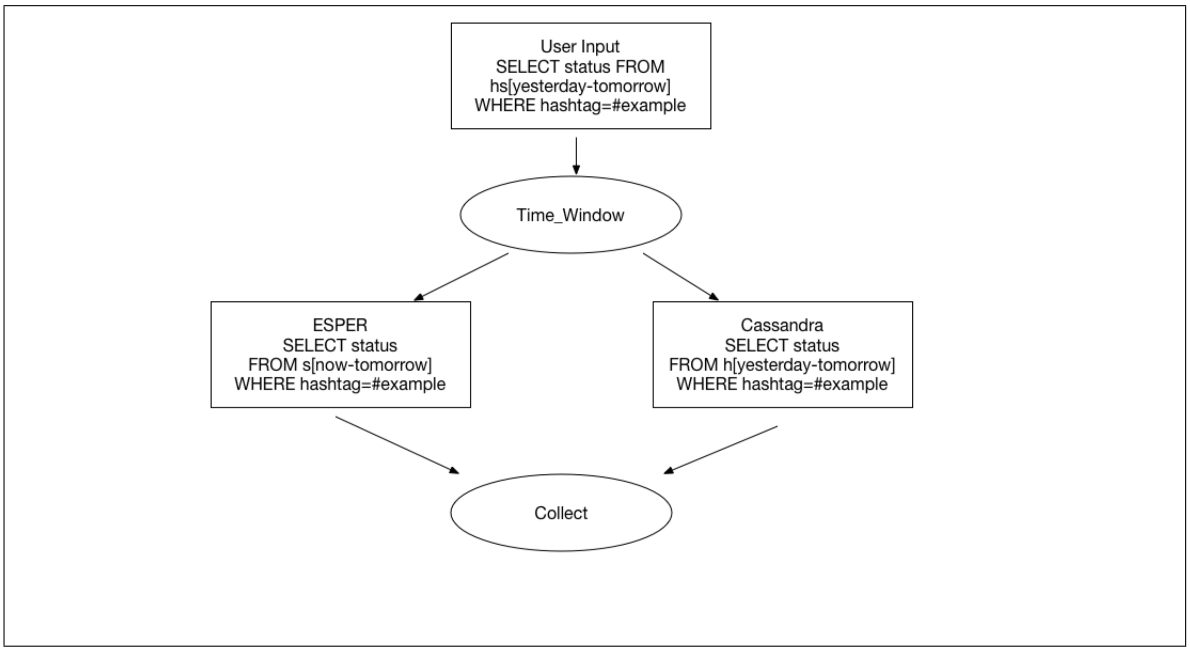


Figure 3.8: Query language mapped to the back-end

3.3.4 Insert Rate

The Twitter firehose can exceed over 500 million tweets per day [1], 5,000 per second, therefore the development of a scalable, available and consistent framework to store all of these can be challenging. Scaling to such large volumes of data can lead to consistency and availability problems if the throughput rate is not high enough. *Antares* implements novel techniques to solve these challenges efficiently.

Cassandra executes queries synchronously, this results in queries being executed serially and the database locking until a request has returned, yielding increased response times because the client is blocked from executing until the lock is released decreasing availability. *Antares* uses the Datastax library to make asynchronous calls to the database, eliminating throughput problems. A higher insert rate is possible as more queries are executed simultaneously.

Executing the queries asynchronously means the database does not block increasing the throughput rate; this resulted in *Antares* being able to support tens of thousands of data insertions per second. Executing the queries asynchronously results in concurrent query execution; however when executing queries in parallel the order they are returned in is not guaranteed. This can cause reduced consistency as there is no guarantee a write will be made successfully as soon as as a Tweet is ingested by *Antares*. However this trade off is acceptable to support the scale-out features of *Antares*.

Antares uses a Java Class called Future, it represents the results of an asynchronous task, in the case of *Antares* the task is a query executed across the database, these are stored in a list. The query is executed and a listener is used to listen for the return of the query, whether it is null (the execution failed) or the results from the query. The list is used as a log of incomplete and unsuccessful query requests, but when a successful event returns the task is removed from the future list. Unsuccessful queries are retried; this ensures eventual database consistency and that all writes will eventually succeed. For example, a query result returns with an exception, there is a timeout of 5 seconds and then the query is retried. When there is a spike in the volume of data being ingested, there may be more requests in contention meaning that there may be be more clashes or time-outs, therefore retries are necessary.

The number of tasks in the list is limited to the protocol depth (approximately 128 requests per node) to ensure no overflow exceptions and that all the data is written to the database. Once this limit is exceeded then the futures are forced to finish, if they are successful they are taken off the list until the size of the list is under the limit, if it returns with an exception then it is retried. However the list will continue to receive requests. This list of futures is only limited by the size of memory. When a future is first in the list it is submitted to the database and executed is possible, if not then it is stored in the server buffer. When the server buffer becomes full queries return with exception and response time increases. This is why *Antares* is configured and tuned to prevent this by limiting the amount of requests - this is shown in section 3.5.

To support the high throughput rate of the asynchronous calls there must be sufficient connections to the database, to maintain these connections there is a database thread pool. This maintains the number of connections and ensures it does not become too large, as if there are too many then the database will block. The database thread pool ensures only 9 connections per machine are ever established at any one time, as defined in the configuration settings.

Another optimisation is to batch query execution. Queries are batched and then submitted to the database asynchronously. This results in parallel execution of queries, thus potentially increasing performance. It also ensures the data flow is high enough so resources are not under utilised.

Antares implements a novel query monitor to translate the one user-defined query into two separate queries if the query is combined or if it is stream/historic then the query monitor passes it to the right module of *Antares*.

Antares implements asynchronous querying and batching to extend Cassandra to support high insertion rates while scaling-out. The next section goes into detail about how the geospatial querying was extended.

3.4 Data Model

The architecture design, motivation and implementation have been described, the mapping between the architecture and the querying is now described.

Antares [2] was designed and implemented to support querying against streaming and historic data for the analysis of Twitter.

A literature review was undertaken to understand the domain area - Section 3.4.1. From this, a comprehensive set of queries were defined to encompass the relations in the Twitter data model - Section 3.4.2.

Its goal was high throughput and low latency. *Antares* supports real-time, historic and combined querying by optimising two technologies: ESPER and Cassandra. *Antares* implements an index structure, which is required to enhance Cassandra and enable low latency querying to support optimised querying for stream, historic and combined processing. This is described in Section 3.4.3.

Streaming data can arrive at high rates and volumes, therefore optimisations are required to allow high-rate ingestion while querying the database and stream simultaneously. These are described in Section 3.4.4.

Antares has been in a research project called Tweet My Street, Section 5.4 describes some of the use cases used for this.

The next section describes each of the technologies and how they were extended within *Antares*.

3.4.1 Case Studies

A literature review was undertaken investigating what topics were being analysed using social media, focusing mainly on Twitter. The questions collated from this were used to see what domains could be used to identify information about world events. The aim was to identify a set of unrelated domains and then use the same generic queries and query patterns to derive information from the Tweets. The three domain areas found by an exhaustive literature review were marketing and advertising, conferences, and emergency responses. From these areas a set of queries were derived.

Marketing and Advertising: The need to understand the popularity and spread of Tweets and topics contained within them is important commercially [54]. For example, [55] highlights some of the important questions that companies want answering. This includes monitoring Tweets with specific keywords, and using retweets to identify the popularity of certain marketing items. Examples of questions that have been identified within this domain are:

1. What is currently trending? [56]
2. Did this hashtag trend yesterday? [54]
3. What retweets will be popular in the next hour? [56]

Conferences: In conferences and presentations, the use of a common hashtag, along with a dedicated Twitter profile, is commonplace. This is used as a method of publicising the event, as well as monitoring attendee and external engagement [57]. Some common questions from this application domain are:

1. What has been posted about this conference? [58]
2. What are people's opinions on this paper? [59]
3. Which aspects of the conference have been the most popular? [60]

Emergency Response: Emergency response encompasses a range of types of incidents, e.g. from natural disasters such as floods and earthquakes, to crimes in your neighbourhood. [61] explains in depth the important role that social media has when responding to emergencies. Key questions that have been asked in this domain are:

1. What information is available about this flood? [62]
2. What Tweets were sent from the city that the earthquake originated from? [63]
3. Which retweets were posted most during the riots? [64]

These are just three examples. The next section describes the Twitter data model and queries for investigating Tweets.

3.4.2 Queries

To enable the querying of Twitter data to answer common questions, a data model was designed to capture the relations between the data types in Twitter, and is illustrated in Figure 3.9.

Antares needed to provide a scalable and efficient means of answering these questions. The questions needed to be wide ranging enough to cover multiple domains while having the ability to represent stream historic and combined queries. So that *Antares* could translate one request into the correct query type to enable simpler use for the user. These issues were addressed by identifying a small core set of generic queries that could answer the questions from the case studies. The queries are now described.

In these descriptions, terms that are used are listed in Table 3.2. Each of the queries can be a stream, historic or combined query; the difference is described in Table 3.3. Type *a* is a historic query – these contain a time window held within the past. Type *b* are combined queries – these cover a time window, which begins in the past and then

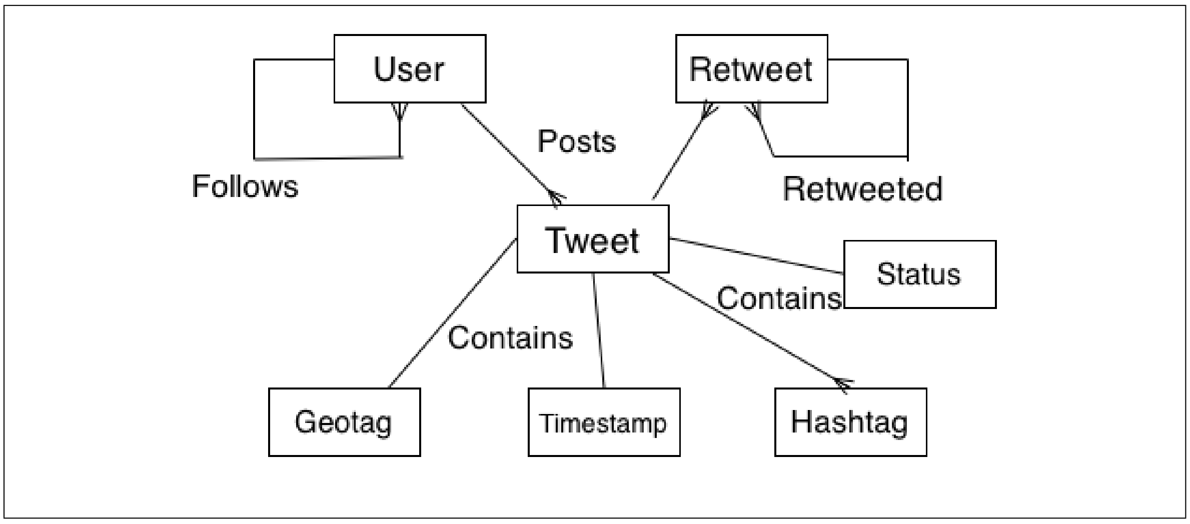


Figure 3.9: Twitter data model

finishes in the future. The last type, *c*, are stream queries – these are executed in a time window in the future. Parameters to be used in the system and passed to the query monitor are declared in the parenthesis next to the query number.

The query language is based on the CQL (continuous query language) discussed in [40]. The query is based on a time window which is specified inside the square brackets. As each of the queries can be either a stream, historic or combined query the store over which the query is executed –whether it be stream, database or both– is represented by the place-holder "Tweets". Therefore you can use the query language to specify a specific query type or use the word Tweets to mean they can be executed across both layers within the system. It uses an SQL-like declaration for condition statements.

Each query is used to describe the relationships in the data model. This is achieved by changing the parameters in the queries to produce different results. Table 3.4 shows the six different parameters used in the queries and in which queries they were used. The data model describes the data and the relations that *Antares* focuses on. The queries provide a comprehensive set of queries to search the data model. Each query defines a relation in the data model.

Trends were a common theme in the questions asked of the use cases. Hashtags are a prominent feature of Tweets; they provide a means of categorising Tweets into themes without very complex textual analysis. They help to group Tweets and trends for users to express their opinions or to find out about topics they are interested in.

Parameters	Description
hashtag	topics in a Tweet prefixed by #
count	a tool used to store the number of times an event has occurred
threshold	an integer that represents a total the count is constrained by
location	a string representing the location of a Tweet
retweet	a ‘Tweet that has been reposted by another user
time	this is a timestamp of when a Tweet was posted
tweetid	a unique number given to the Tweet to identify it

Table 3.2: Parameters and values in the queries

	Type	Lower Threshold (startTime)	Upper Threshold (endTime)
a	Historic	Past	Past
b	Combined	Past	Future
c	Real Time	Future	Future

Table 3.3: Query definitions

Queries	Parameters					
	threshold	startTime	endTime	location	hashtag	tweetid
Q1	x	x	x			
Q2	x	x	x			
Q3		x	x	x	x	
Q4		x	x		x	
Q5		x	x		x	
Q6						x
Q7		x	x			x

Table 3.4: Query parameters

	Questions	Queries
Marketing and Advertising	Question 1	Query One
	Question 2	Query Four
	Question 3	Query Two
Conference	Question 1	Query Five
	Question 2	Query Five
	Question 3	Query One
Emergency Response	Question 1	Query Five
	Question 2	Query Three
	Question 3	Query Seven

Table 3.5: Questions and matching queries

Being able to analyse these can help to support researchers in filtering the amount of information they are analysing and increase the probability of the information being relevant. Therefore Query One identifies trends and returns the resulting hashtags and their count. This query takes three parameters, as can be seen in Figure 3.4. These specify a set of upper and lower time bounds and a threshold limit. The value of threshold specifies how many times a hashtag must occur between the time bounds before it is identified as a trend. The counter is incremented each time a duplicate hashtag in the same time-bound is included in a Tweet. A counter is a Cassandra device which increments automatically.

```
Q1(startTime, endTime, threshold)
SELECT hashtag, counter
FROM Tweets[$startTime-$endTime]
HAVING COUNT(hashtag) > $threshold
```

Another feature of Tweets is the “retweet” function – this allows people to identify more popular Tweets quickly and easily. The retweet function basically re-posts a Tweet that was posted by another user, which results in the Tweet being shown to additional users, increasing the reach of the Tweets. Therefore it can be derived that the more popular a Tweet the more relevant it is to current world events or opinions. Therefore Query Two identifies popular retweeted Tweets. Here the variable threshold provides the minimum number of times a Tweet must have been retweeted. If the retweet count exceeds the threshold then the Tweet is considered a popular retweet and a result returned to the user.

```
Q2(startTime, endTime, threshold)
SELECT  counter, tweetid
FROM Tweets[$startTime-$endTime]
HAVING COUNT(retweet) > $threshold
```

Location is a very important part of the metadata which provides additional information. By looking at the area from which a post originated, additional insights can be derived about marketing locations, the epicentres of natural disasters and much more. Query Three finds all Tweets with a hashtag and location specified by the user in a time-window. The location parameter is a user-defined string. This was used for simplicity and is extended to include latitude and longitude in Section 4.4. This is because the user-defined location is not accurate.

```
Q3(startTime, endTime, location, hashtag)
SELECT tweet
FROM Tweets[$startTime-$endTime]
WHERE tweet.location=$location
AND tweet.hashtag=$hashtag
```

Query Four returns a count of how many times the specified hashtag has appeared in tweets within the time bounds.

```
Q4(startTime, endTime, hashtag)
SELECT COUNT(tweetID)
FROM Tweets[$startTime-$endTime]
WHERE tweet.hashtag = $hashtag
```

Query Five allows users to specify a hashtag of interest to them; the query returns the tweets containing the hashtag within the time-bound.

```
Q5(startTime, endTime,hashtag)
SELECT tweet
FROM Tweets[$startTime-$endTime]
WHERE tweet.hashtag=$hashtag
```

Query Six identifies the number of users that follow the user that posted the Tweet.

```
Q6(tweetID)
SELECT userID as userid
FROM Tweets
```

```
WHERE $tweetID=tweet.id

SELECT noOfFollowers
FROM Tweets
WHERE userID = userid
```

Query Seven returns all the times that a Tweet has been posted or retweeted.

```
Q7(tweetId, starttime, endtime)
SELECT counter
FROM Tweets[$startTime-$endTime]
WHERE $tweetID=tweet.id
```

As mentioned each query describes a relation in the data model listed are the query and the relation it describes.

Query One: multiple *hashtags* are in multiple statuses.

Query Two: users *retwee*t Tweets.

Query Three: Tweets contain a *geotag*.

Query Four: one *hashtag* can be mentioned in multiple Tweets (returns the *count* instead of status).

Query Five: one *hashtag* is mentioned in multiple statuses.

Query Six: *users* have $followers$ (relation follows).

Query Seven: Tweets are posted at a specific point in time.

The *hashtag* has a more complex relationship as it is a many-to-many relationship. Therefore more queries are dedicated to querying this variable.

A set of use cases were used to define a data model for the relations between the data types in Tweets. A set of comprehensive queries for searching the relations within the data model were then defined and used to support an exhaustive search of the domain. The next section describes the database design to enable low latency querying of the data model.

3.4.3 Schema

To support storing the data in a more easily retrievable way, an efficient and novel global indexing system was designed and developed to provide support for the queries mentioned above. One important design decision was to denormalise data to reduce execution times and increase throughput by eliminating the cost of joins for the generic queries.

The two key design challenges for creating a scalable system were:

1. How to efficiently access Tweets by time.
2. How to efficiently perform aggregations.

The database design is now described and shows how these challenges were met. One of the design challenges faced when constructing schemas in a NoSQL store is whether or not to replicate data. Denormalising data can make querying more efficient, by producing indexes specific to that query. Replicating data increases the storage space taken up, however due to the cost of hardware now and the ease of scaling out there is no need not to increase performance by denormalising data.

An example of that is the column family **tweet**, which is indexed by *tweetid* with the *status* being used as the column name. This means other indexes can store just the *tweetid* and then use this index to retrieve the *status* if it is required. As has been shown some queries use *tweetid* for identifying tweets. This indexing technique allows the status to be retrieved with increased efficiency and performance. This also improves write performance as Cassandra can write without reading. This is because the data is not written sequentially, therefore *Antares* does not need to find the correct place for inserting data it can just be written directly. The system takes the *tweetid* and writes to the next space on disk, therefore no read is required beforehand. This also meant no other keys need to be changed and consistency is not an issue.

```
CREATE TABLE tweet (
    tweetid text,
    status text,
    PRIMARY KEY (tweetid))
```

The **hashtag time** column family was designed for Queries One and Four. All Tweets arrive from the firehose with metadata that includes their creation time. For efficient access to data from Tweets created within particular time-bounds, the schema uses time buckets, each of which stores data created within a particular time range. A timestamp is extracted from the metadata of the Tweet and split into two values: day and minute. Day is the primary key and minute the column name. Each column in the row represents a minute within a day (e.g. 0-1399). The value of the day is determined by dividing the timestamp by 86400000 (as this is the number of milliseconds in a day and timestamps are stored in milliseconds).

The granularity of the time buckets was chosen by analysis of typical queries within the case studies, for example, a conference may be held over a week. Therefore a daily analysis of the Tweets about that day - with an hourly granularity - would enable a user to delve into appropriate detail. The schema also takes advantage of Cassandra's composite column keys: this column family uses *minute* and *hashtag* as a composite column key (the `PRIMARY KEY` keywords in the schema introduce the keys: the first value is the row key and the rest form the column name; when the column name consists of multiple columns, it is called a composite column key).

To deal with the challenge of efficiently handling aggregations, a *count* is maintained of the number of times that a *hashtag* has occurred within the period identified by the *day* and *minute*. When a new 'Tweet arrives from the firehose the composite key is used to increment the appropriate counter. Queries One and Four exploit this schema. One performs a range scan over the rows, using the minutes to identify the correct range, which selects the *hashtags* and their *counts*. Once these have been retrieved, the count for each different *hashtag* is totaled and returned to the user if it exceeds the threshold. Query Four instead uses the day and the whole composite key to identify the count of a specific *hashtag* in that time period.

```
CREATE TABLE hashtag_time (
    day bigint,
    minute bigint,
    hashtag text,
    value counter,
    PRIMARY KEY (day, minute, hashtag))
```

The column family **tweet time** follows an identical pattern to **hashtag time**, with *hashtags* replaced by *tweetids*. Queries Two and Seven use the schema to access data based on *tweetid*.

```
CREATE TABLE tweet_time (
    day bigint,
    minute bigint,
    tweetid text,
    value counter,
    PRIMARY KEY (day, minute,tweetid))
```

The column family **user** is indexed using the same pattern as *tweet*, but metadata from the Tweet is replaced by information about the user.

```
CREATE TABLE user (
    userid bigint,
    handle text,
    PRIMARY KEY (userid))
```

Tweet geo is based on the same time bucket structure as the previous column families; it is used to answer Query Three. The column name is a composite column key of *timebucket*, *hashtag* and *location*, allowing for an exact key match to efficiently retrieve the status of the Tweet based on *location* and *hashtag*. It has the same pattern as Query Two. The *location* and *hashtag* from the metadata are checked for a match. *location* is taken as the user-defined string, as previously mentioned, and extended in Section 4.4.

```
CREATE TABLE tweet_geo (
    day bigint,
    minute bigint,
    location text,
    hashtag text,
    status text,
    PRIMARY KEY
    (day, minute, location, hashtag))
```

Hashtag status is indexed by time, it allows Query Five to retrieve a *status* using the *hashtag*. It adds redundant data (*status*) instead of creating a relation between *hashtag* and *status* for more efficient querying.

```
CREATE TABLE hashtag_status (
    day bigint,
    minute bigint,
    hashtag text,
    tweetid text,
    status text,
    PRIMARY KEY
    (day, minute, hashtag))
```

User follower uses the same pattern as **tweet user**, using a *userid* it finds the number of *followers* that was saved at that time. This is an exact key match query and will perform with the same efficiency as the previously mentioned column family.

```
CREATE TABLE tweet_user (
    tweetid text,
    userid bigint,
    PRIMARY KEY (tweetid,userid))

CREATE TABLE user_follower (
    userid bigint,
    follower_count bigint,
    PRIMARY KEY (userid, follower_count))
```

This section describes the database schema, which will be used to support efficient low latency querying, which enables combined querying with the stream processing system. The next section will describe how the queries and the schema are translated into stream processing queries.

3.4.4 Stream Schema

Each of the queries described in Section 3.4.2 can be executed across the stream or the database. Here the execution of queries across the stream is explained. For ESPER to execute a continuous query a Java object was defined to represent a Tweet. The Java object represented the data model and contained attributes included in Figure 3.9, which supported the analysis of the relations described in Section 3.4.2. Therefore each Java object has the attributes: *status*, *hashtag*, number of *followers*, *timestamp*, *geotag*, number of *retweets* and *userid*.

When Query One is executed in ESPER the *hashtags* and their count are stored in an in-memory hashmap. The hashes are the keys and the value is a counter, which is incremented each time the *hashtag* is included in a Tweet. As the queries are time-bounded, the space occupied in the hashmap by a record can be recovered once the query reaches the end of the time-window, and so the memory requirements are limited.

When executing Query Two over the stream it extracts the *retweet* count from each Tweet and identifies whether it exceeds the stated threshold. Therefore it varies from the historic query as there is no aggregate function the value is taken from the live Tweet.

Query Three is similar to Query Two as it has an exact and extracts the *location* and *hashtag* from the Tweet. If the values match the parameters specified by the user then the results are returned to the user. The query is executed over the user-defined time-period, however results are returned each time a constraint is met and as soon as it is met.

An aggregate function is executed across the stream for Query Four, which counts each time a Tweet mentions a *hashtag*. This is a counter held in-memory and each time the *hashtag* is mentioned the counter is increased. The result is returned after the time-period has finished.

Query Five returns the each Tweet in which the user-defined *hashtag* is mentioned. Each time a Tweet contains the *hashtag* then Tweet is returned while the query is executed.

The execution of Query Six varies on the stream when compared to the database. The user specifies a *tweetid* and if it is used at that point in time then the Tweet is returned otherwise the query returns.

Query Seven uses the *tweetid* and if a Tweet is identified as having this *tweetid* then the user object (contained in the Tweet metadata) is accessed and the number of $followers$ extracted. This is then returned to the user, however the query is executed for the full time period in case the Tweet is retweeted. If the Tweet has been retweeted then the *userid* maybe different therefore the number of $followers$ will be different.

Antares provides a novel mechanism for combining both of these querying techniques by executing one query. This section has explicitly described the schema for the stream processing.

3.5 Evaluation

Antares provides low latency, scalable query execution over large volumes of stream data. It has been employed in research projects to allow automated and efficient analysis of Twitter data [3]. This section evaluates its performance: in particular the two key measures: the query response times and data ingest rates of the system.

These experiments described here explore and evaluate the various design choices, extensions and optimisations that support efficient, scalable stream, historic and combined queries.

3.5.1 Ingestion Rate

To evaluate the system, a collection of 10 million Tweets were compiled using the Twitter4J streaming API [65]. High ingestion rates are then replayed, to replicate the Twitter firehose. These can be faced when dealing with high velocity data streams – the data created by the Twitter firehose can average 5,000 Tweets per second [1]. The Tweets were replayed at the highest throughput rate the hardware specification would allow. Twitter4J is a free API, which allows access to Twitter data – it allows different types of collection, with both searching and streaming APIs. The search API allows the user to search for Tweets historically, however it is restricted to 150 queries every 15 minutes. The streaming API limits the number of Tweets collected from the firehose to 1% of all the Tweets at any given point in time.

The experiments described in this section were created to prove the system could scale to support high ingestion rates – allowing, for example, the Twitter firehose to be written to disk without any loss of data.

The experiments were executed in the Amazon cloud, using *m1.large* machines, the configuration for which was: Memory: 7GB, 2 cores, Storage 840GB and network performance: moderate, operating system Ubuntu.

The dataset of Tweets used is 14GB is size however this is simulated to imitate the firehose. The system constraints for memory are kept to a minimum so that the operating system does not cache things and therefore a correct representation of Antares using the database and disk storage is represented. The Tweets are replayed from another machine in the same datacentre as to not include network delays but also not on the same machine so that it does not affect the way the database and the machine use the memory on the machine.

Cassandra has two different methods of storing data. The first is a Memtable, which is in-memory storage. As data enters the Cassandra cluster it is stored to a Memtable. Each Memtable has a tunable storage limit, once this is reached then the column families are flushed to disk. SSTables are used for disk storage. These are immutable data files and each one corresponds to a column family. Additionally there are temporary SSTables, which are merged into one large SSTable periodically and asynchronously. This supports fault tolerance, as there is a non-volatile store of the Memtables even if they have not been flushed to disk. As the data enters the system for the first time it is saved in-memory as this is quicker than writing to disk. Once the amount of data in the Memtable surpasses the storage limit it is written to disk, which takes longer and can reduce the ingest rate. Creating a faster initial ingestion rate does not affect the overall average rate of ingestion.

Each Cassandra node was started in an empty state, with only the indexes described in the database. Two optimisations were designed and implemented to increase ingestion rate: a) batching of queries b) asynchronous execution of queries. Batching queries ensures the flow of data is sufficient to utilise the full capacity of the database. If the flow of data is too low then efficiency is lost as resources are not used and therefore become underutilised. If the flow of data is too high then the database becomes over-saturated which causes queries to fail. Cassandra executes queries synchronously, which can cause a bottleneck. This is a result of sequential execution and callback waiting time. By executing queries asynchronously the ingestion rate of *Antares* can be increased dramatically as queries can be executed in parallel.

Cassandra was designed to support highly scalable writes but the extensions in *Antares* aimed to increase the ingestion rate significantly so that it can handle much higher velocity streaming data.

3.5.2 Batch Experiment

The experiments were designed to evaluate the performance and scalability of the extensions implemented by *Antares*. The first experiment is executed over one Amazon machine and measures the effect that batching the Tweets to modify the data flow has on the ingestion rate of the Cassandra instance.

A batch is a set of insert queries - one query represents one Tweet to be inserted - the batch is sent as a list of these inserts to the database. When this is submitted to the database if there is CPU power available then the inserts will be executed in parallel.

The experiment consisted of one client, –or *m1.large* machine– which sends varying sized batches to the one-node Cassandra cluster to be written to disk. Batches of Tweets are replayed at the highest rate the Amazon machine specification can replicate. The number of Tweets processed within a given time is counted and the Tweet per second rate is calculated by taking the $numberofTweets/numberofseconds$.

Figure 3.10 demonstrates as the batch size is increased from 1 to 200 the number of Tweets ingested increases by over 50%, from 3,047 to 6,895. The number of Tweets ingested begins to tail off after this with a decrease in ingestion rate for a batch of 400 Tweets. This is because the batch has become too big and the number of queries has over saturated the database. This causes queries to time-out (default time-out is 10,000 ms), and therefore another query must be sent to ensure the data is written. This results in an increase in the time taken for queries to execute and therefore the number of Tweets written within that time period is reduced. This demonstrates that batching the Tweets has a direct impact on the ingestion rate and is used to optimise *Antares* to produce the most efficient ingestion rate.

Batch size (Number of Tweets)	Tweets per second
1	3046.8
100	5101.13
200	6895.39
300	6928.61
400	6824.68

Table 3.6: Tweets per second written to *Antares* with increasing batch sizes

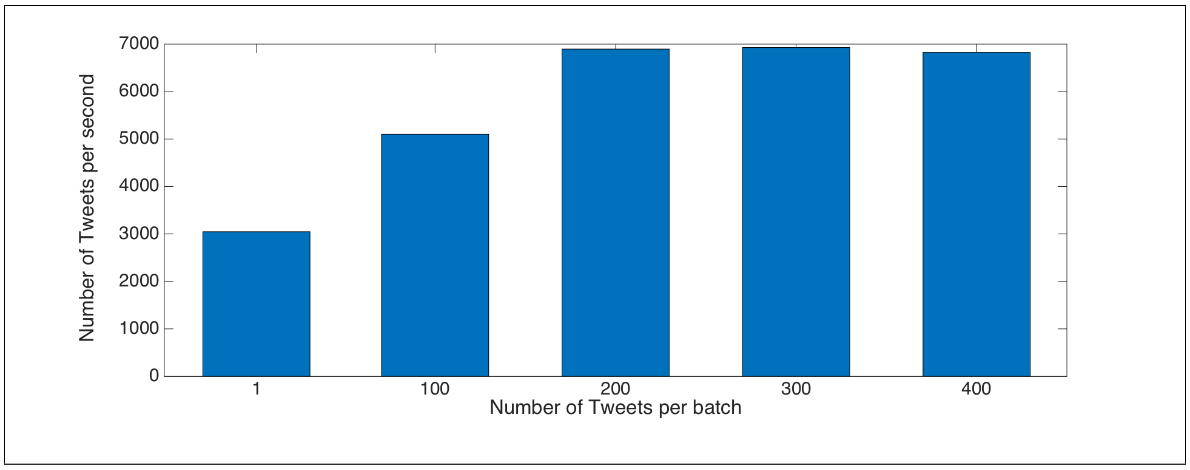


Figure 3.10: Tweets per second written to the *Antares* one-node cluster using increasing batch sizes

3.5.2.1 Comparison of *Antares* and the Synchronous Cluster

The next experiment replays Tweets again, however it varies the number of nodes in the cluster. Each run starts with a completely empty state server. The experiment was executed over a one-node cluster increasing to an eight-node cluster.

Antares is compared with a Cassandra cluster, which is indexed using the same design pattern. However the "synchronous" cluster does not use batching or asynchronous writes to improve performance.

Figure 3.11 demonstrates a comparison between *Antares* and a cluster with no asynchronous queries or batching. As is demonstrated in Figure 3.11, the ingest rate increased linearly and *Antares* has the ability to consume over five times the average Twitter firehose with an eight-node cluster. The four-node cluster supports an ingestion rate of 20,470 Tweets per second, by doubling the number of nodes the ingestion rate is increased to 34,881 Tweets per second. As the average rate is 5,000 Tweets per second a four-node cluster is more than sufficient for ingestion. Using a larger cluster enables *Antares* to cope with spikes in the data flow. Therefore the number of nodes used for the experiments is four as it has the capability of capturing the entire firehose and potential bursts of activity from users or applications.

The difference between clusters is 6,928 Tweets to 208 Tweets on one machine. This difference only increases as the cluster size increases. 34,881 Tweets can be ingested by *Antares* however the synchronous cluster can only ingest 349 Tweets in an eight-node cluster. This is an improvement of approximately 100 times more data ingestion. This demonstrates the efficiency of the optimisations made to *Antares* when compared with a normal Cassandra cluster. 34,881 Tweets is more than five times the average Twitter firehose, which means *Antares* is capable of handling the full firehose but also bursts of events.

The addition of nodes provides greater hardware support, however the results show that the software extensions support higher ingestion rates and therefore less hardware is required for a given ingestion rate. The percentage savings shown in Figure 3.12 demonstrates that Antares improves performance by nearly 100% for a four-node cluster.

Number of Machines	*Antares* (Tweets per second)	Synchronous (Tweets per second)	Percentage Savings (%)
1	6929	208.06	97
2	10768	247.18	97.7
3	20470	283.19	98.61
4	34076	349.09	99.98

Table 3.7: Tweets per second written to *Antares* and the Synchronous System

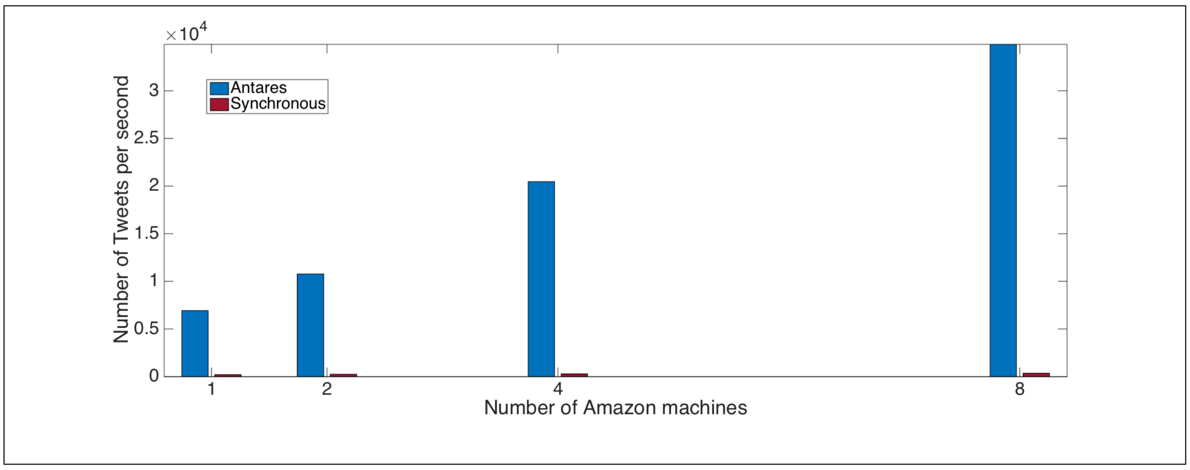


Figure 3.11: Tweets per second written to *Antares* and the Synchronous System

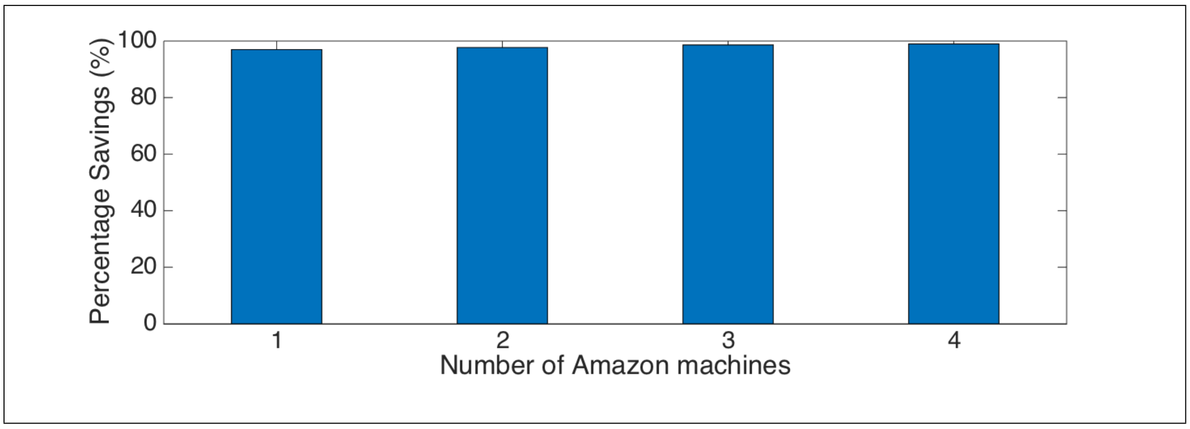


Figure 3.12: Percentage savings for *Antares*

3.5.2.2 Conclusion

Query batching and asynchronous execution has shown improvements in the ingestion rate. Twitter data was used to evaluate *Antares* and show optimisations supporting an ingestion rate of over 30,000 Tweets per second. The batching ensures the resources of *Antares* are used. Asynchronous queries provides a means of query execution that does not block and reduce availability. *Antares* can provide a mechanism of ingesting the entire Twitter firehose without reducing availability.

3.5.3 Read Performance

The previous experiments determined that *Antares* can support an ingestion rate of five times the Twitter firehose. However the ability to search and process this data efficiently and quickly at scale was vital to support the combination of real-time stream analysis and historic processing. This was implemented through the design and development of indexing mechanisms, described in Section 3.4. This section evaluates the execution time for these indexes against a non-indexed system, which includes an evaluation of the effect of increasing query loads on the data store.

To evaluate the historic store, different types of query were executed over the Cassandra cluster and the mean response times plotted against the number of machines used. The first set of experiments takes a workload of 1,000 queries – one of type exact match and one of range query. The system was deployed on the cloud and the same specification Amazon machines as previously mentioned were used.

Antares was deployed over a one-node cluster increasing to a four-node cluster (as this is the optimal cluster size for ingestion rate) to evaluate the effect on query response times with indexing with increasing cluster sizes. These queries were taken from the generic queries described in Section 3.4.1. The exact match query is Query One and the range query is Query Four. These were executed sequentially over the database and the mean response time measured, after each result was recorded an extra *m1.large* machine was added to the cluster.

The data was automatically load balanced by Cassandra and then the queries were executed afterwards to ensure load balancing had no affect on the execution time. Load balancing a cluster can take up to a few seconds and stops calls to the database. The workloads were executed across an *Antares* cluster with indexing to establish efficiency and an unindexed cluster –schema shown in Figure 3.15– to compare and evaluate the optimisations made to *Antares*.

The start and end time parameters were randomly generated values each entered into individual queries. Fifty percent were generated to be within the earlier half of the day, fifty in the later half. Values were placed at the start (earlier in the day) and end (later in the day) of the row. Hashtags were generated for the exact match query, 80% were hashtags that were included in the dataset, 20% were not.

```
Q1(startTime, endTime, threshold)
SELECT hashtag, counter
FROM hashtag_time[$startTime-$endTime]
HAVING COUNT(hashtag) > $threshold
```

Figure 3.13: Range query

```
Q4(startTime, endTime, hashtag)
SELECT COUNT(tweetID)
FROM hashtag_time[$startTime-$endTime]
WHERE tweet.hashtag = $hashtag
```

Figure 3.14: Exact match query

```
CREATE TABLE unindexed(
tweetid text,
hashtag text,
time bigint,
userid text,
PRIMARY KEY(tweetid)
);
```

Figure 3.15: Schema for the unindexed column family

The unindexed cluster stores the information about a Tweet in one column family. Counters cannot be used without indexing the column family, so the values that are collected must also be counted in the application layer.

Before describing the query mechanism for the unindexed cluster an example schema is demonstrated in Figure 3.16 as an example of the constraints for querying a Cassandra cluster. In the table **example** there must be an equals operator on a to use an equality operator such as less than on b. There must also be an equality operator on a and b to be able to use the other operators on c. This is to ensure that the query responses are timely and that searches do not have to span the entire cluster. To query without this pattern, the `ALLOW FILTERING` statement is used.

```
CREATE TABLE example(
a text,
b text,
c text,
value text,
PRIMARY KEY(a,b,c));
```

Figure 3.16: Example schema

The query shown in Figure 3.17 is an exact match using the unindexed schema. It queries for information about a Tweet which contains a hashtag and was posted at a certain time. As the unindexed schema uses the *tweetid* as a primary key –and does not index on hashtag– `ALLOW FILTERING` is used to enable querying on a non-primary key component. Therefore each Tweet must be queried to identify whether it was posted in the time-window and if it contained the hashtag.

```
SELECT *
FROM unindexed
WHERE hashtag=? AND time<? AND time>?
ALLOW FILTERING;
```

Figure 3.17: Unindexed exact match query

The query shown in Figure 3.18 is a range query executed over the unindexed cluster. Again all the Tweets must be searched, returned and then the count totalled.

```
SELECT *
FROM unindexed
WHERE time < value AND time >value
ALLOW FILTERING;
```

Figure 3.18: Unindexed range query

3.5.3.1 Exact Match Experiment

The query from Figure 3.14 was executed across *Antares* increasing the cluster size for each experiment.

Number of Machines	Antares (ms)	Unindexed (ms)	Percentage Savings (%)
1	60.38	10679.84	99.43
2	45.39	14686.33	99.69
3	40.35	18179.95	99.78
4	34.66	32950.1	99.89

Table 3.8: Mean response time for exact match queries executed across *Antares* and an unindexed cluster

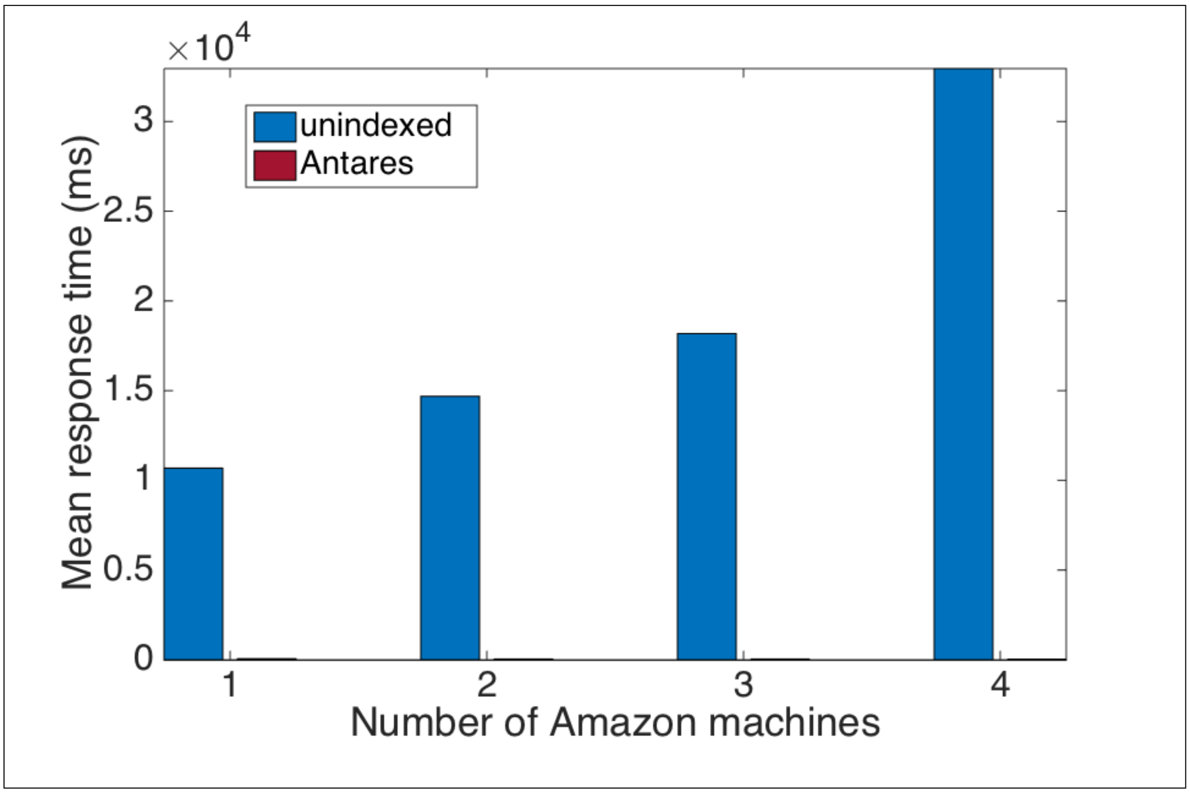


Figure 3.19: Mean response time for exact match queries executed across *Antares* and an unindexed cluster

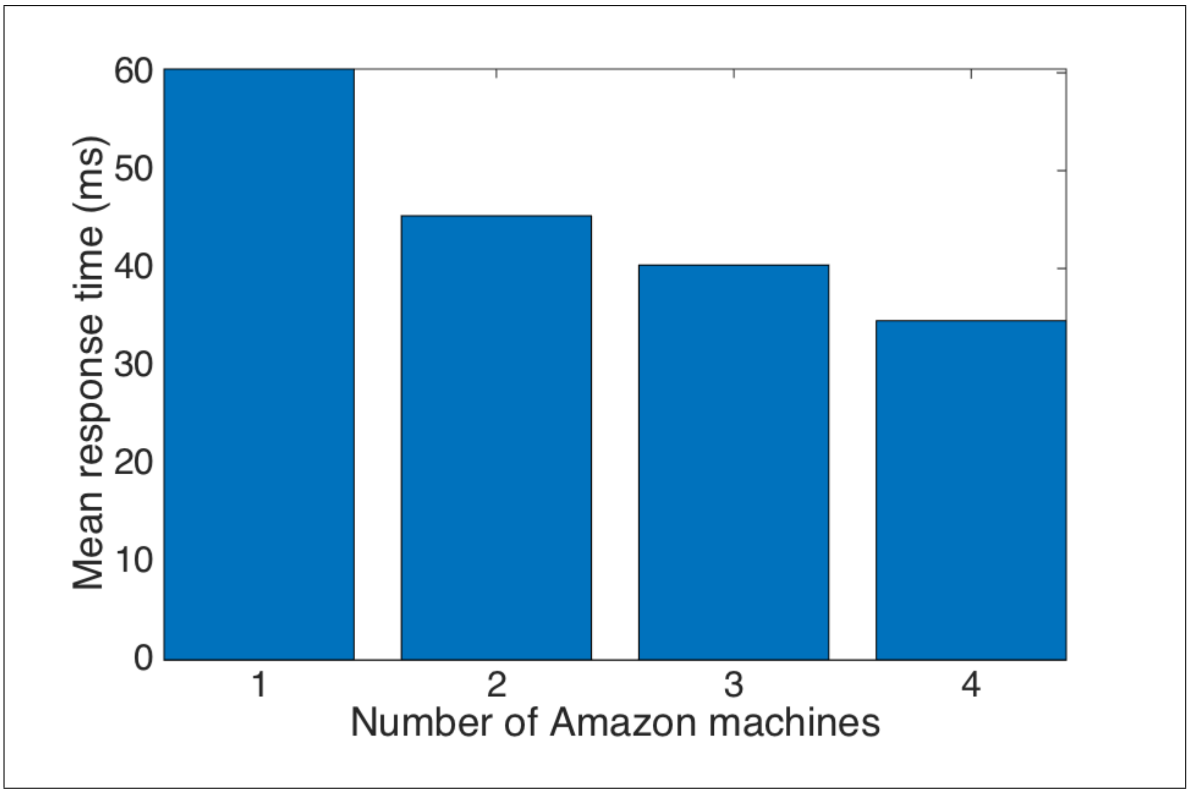


Figure 3.20: Mean response time for exact match executed across *Antares*

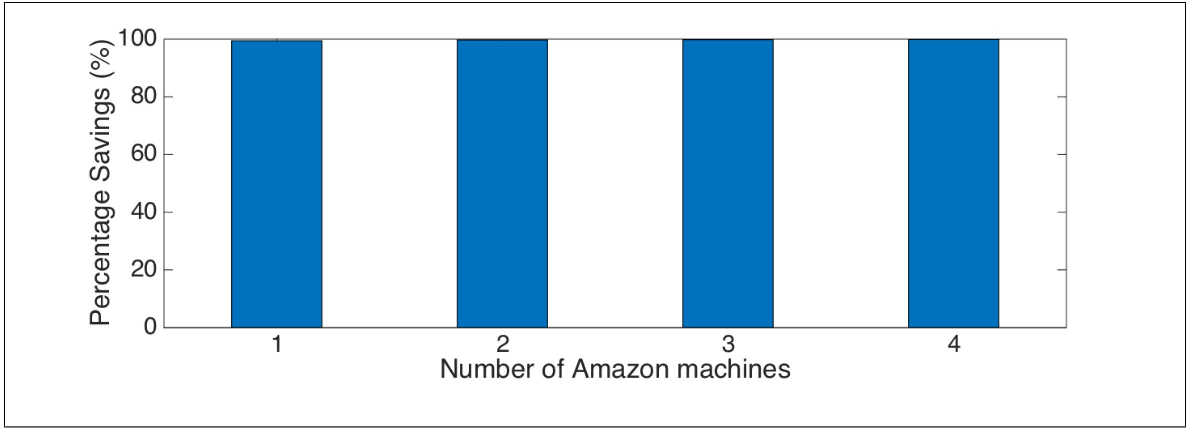


Figure 3.21: Percentage savings for *Antares*

Figure 3.19 shows a graph of the dramatic difference in mean response times comparing *Antares* and an unindexed cluster. For a one-node *Antares* cluster the mean response time of a query is 60 ms, which is very efficient chiefly when compared to the unindexed cluster's mean response time of 10,679 ms. For a four-node *Antares* cluster the mean response time is reduced further to 34 ms. *Antares* ensures this by using unique indexing which guarantees the data will be on one node. Therefore when a query is executed it is accepted by any one node in the cluster and sent on to the node containing the data using a hashed value of the key, as previously explained in Section 3.2.2.12. This reduces execution time when compared with the unindexed cluster as only one hop is required. As hardware is added to the cluster performance increases as shown in Figure 3.20, as there are more resources to accept more queries.

The unindexed cluster of nodes increases the response time when compared with *Antares*. The response time for one node increases to over 10,000 ms when compared with 60 ms for *Antares*. Response times increase within the unindexed cluster as the number of searches is increased. In the unindexed cluster each Tweet is stored separately by their *tweetid*. Therefore any queries executed with constraints other than the *tweetid* require searching the entire dataset. The queries used in this experiment have temporal and textual (*hashtags*) constraints, therefore each Tweet must be queried to identify whether it meets the constraints of the queries. When Tweets are stored in the unindexed cluster each Tweet will be stored in a new row for each hashtag contained in the Tweet. This results in increased searching, therefore n rows maybe be searched just for one Tweet, n depending on the number of hashtags contained in the Tweet.

The addition of nodes increases the response time of the cluster by nearly 3 times when compared with the response time of the one-node cluster. This is because the database has to examine the data held on each node, which introduces network latency, increasing the overall response time. Additionally the performance decreased because of cluster load balancing. When a node is added to the cluster it becomes responsible for a shard of data. The row key is used to partition the data, for an unindexed cluster each Tweet has its own row key, therefore each Tweet is stored on any machine with no order. This increases the response time, as the query has to execute on every machine. There is therefore an improvement of a factor of 1000 for indexed queries compared with an unindexed schema.

Antares scaled linearly and reduced execution times by nearly a half when compared with the unindexed cluster. These results demonstrate that the index is correctly distributing the data so to exploit the parallelism of the cluster. As the number of nodes is increased, the concurrent execution of queries can increase, as there are more available CPUs. As can be seen in Figure 3.21 the saving for query execution time is nearly 100% for all cluster sizes.

3.5.3.2 Range Query Experiment

The range query shown in Figure 3.13 was executed across both types of cluster, the results are shown in Figure 3.22 and Table 3.9. The range query has an increased response time when compared with the exact match query as there are more comparisons made within the row. Additionally more data is returned from a typical range query. The query response time dropped by a factor of 4 when additional machines were added as shown in Figure 3.23. When *Antares* is compared with the unindexed cluster the response time is reduced by a factor of 190 for a four-node cluster. This was because the index exploited the additional hardware resources and parallelism.

The indexed cluster uses a composite key to allow efficient querying not only on the partition key. This means the `ALLOW FILTERING` key word does not need to be used, which degrades performance, as execution is unpredictable. Unpredictable execution can lead to additional disk seeks and increased latency. Figure 3.24 shows the savings that *Antares* supports, the optimisations provide over 90% of savings for each cluster. This demonstrates the improved performance of the querying and provides near real-time results so the results from each query can be combined seamlessly. This allows a user to execute one query, which combines the results. This is a novel approach to combined querying.

Number of Machines	Antares (ms)	Unindexed (ms)	Percentage Savings (%)
1	1041.36	10970.13	90.51
2	610.28	24195.63	97.48
3	595.58	36316.36	98.36
4	255.72	48359.02	99.47

Table 3.9: Mean response time for range queries executed across *Antares* and an unindexed cluster

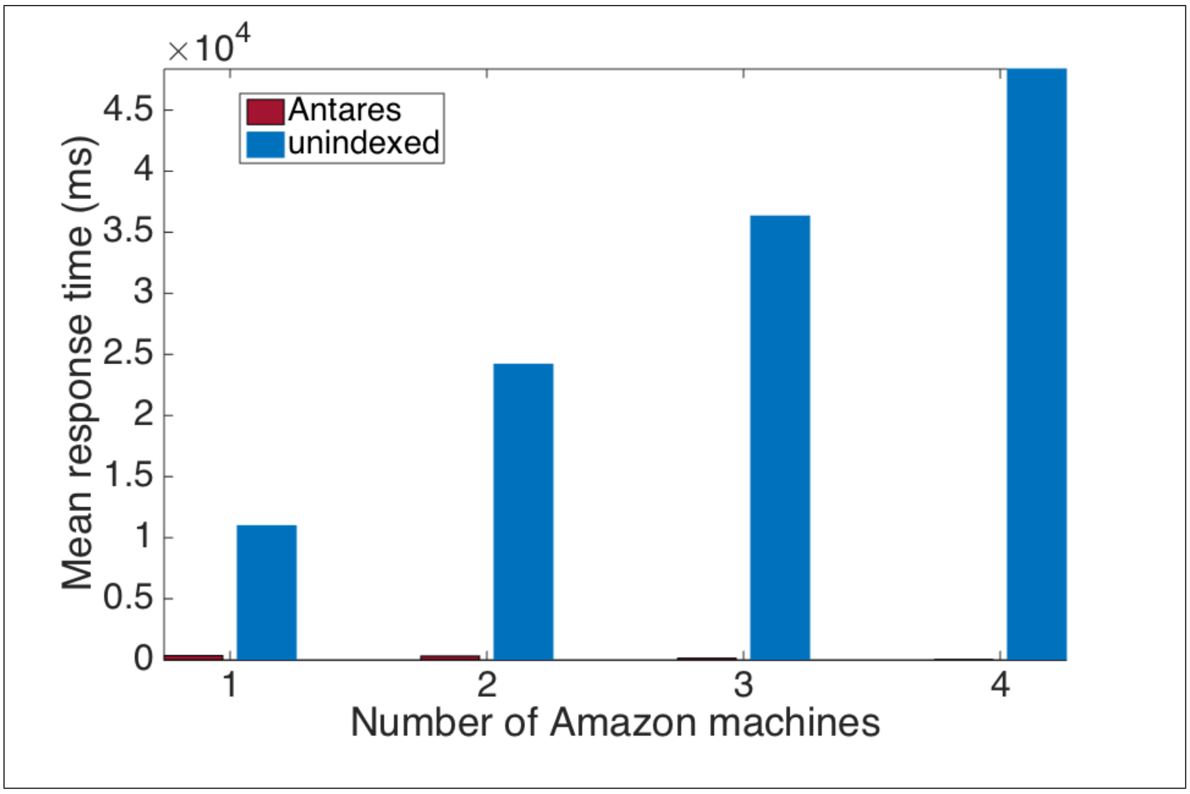


Figure 3.22: Mean response time for range queries executed across *Antares* and an unindexed cluster

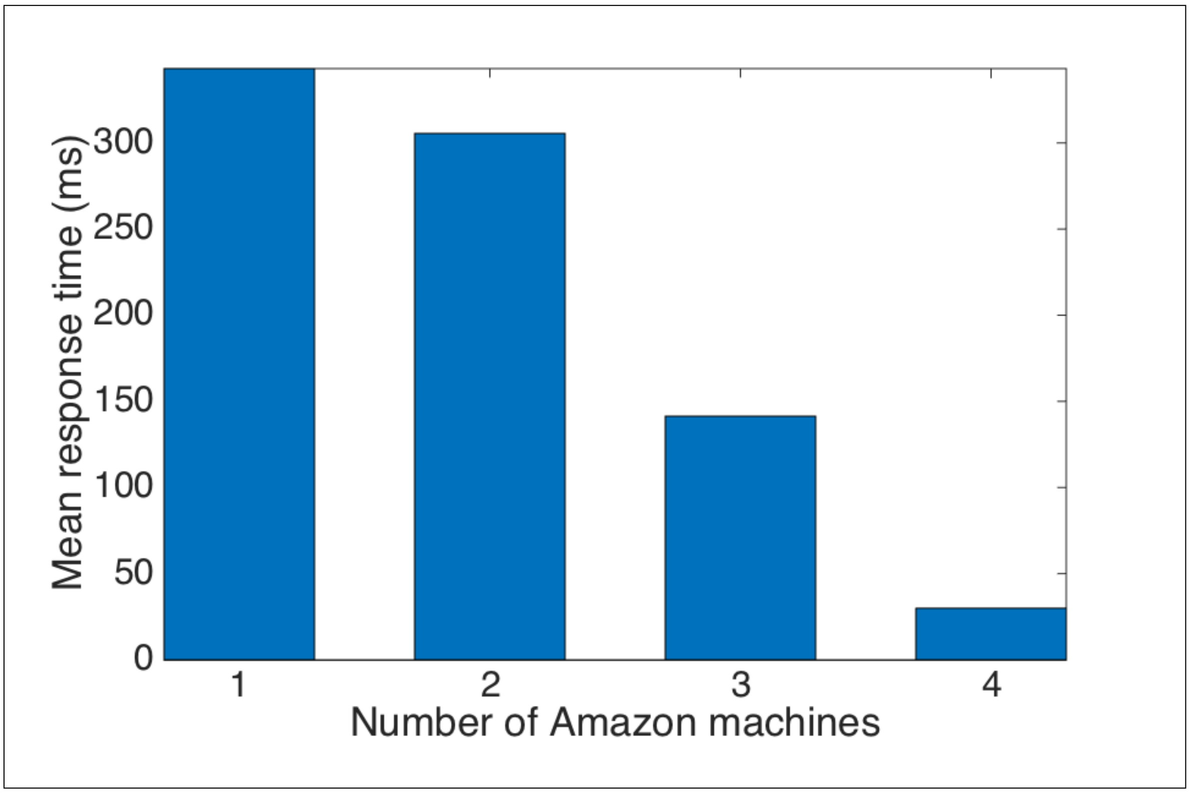


Figure 3.23: Mean response time for range queries executed across *Antares*

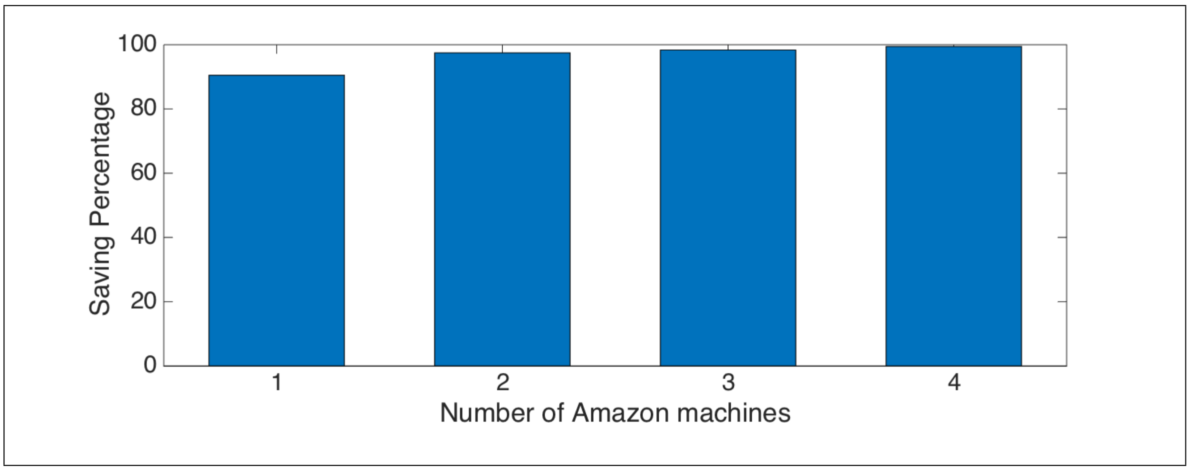


Figure 3.24: Percentage savings of the *Antares* system

The range query and the exact match query had similar response times when executed across the unindexed cluster. This is because the query still has to search through all of the stored Tweets. Once again as nodes were added into the cluster the mean response time decreased on *Antares* by a factor of 4, whereas the response time increased for the unindexed schema by a factor of 4. The unindexed schema increases response time as the row key is the *tweetid*, so each node must be searched. This increases the number of queries and the network time.

3.5.3.3 Stress Experiments

The next experiment evaluates the performance of *Antares* under stress. The experiment executes increasing numbers of concurrent queries, evaluating the query throughput rate. Each experiment is again executed across the same four-node Amazon cluster as previously mentioned. To test scalability, the experiment uses a JMeter [66] plug-in to create threads that generate queries. The cluster was pre-populated with the Twitter dataset before the queries were executed across *Antares*.

The time-window parameters were randomly generated values each entered into individual queries. Fifty percent were generated to be within the earlier half of the day, fifty in the later half. By using values that are randomly ordered reads to the database are not sequential. To return larger amounts of data –to stress the range query– values that placed queries at the start (earlier in the day) and end (later in the day) of the row were used. This also simulates real world values, as the case studies suggest, queries would be written based on events, which do not have to be chronological. A randomly generated set of days was used to represent queries that span more than one day, which requires multiple queries or an `IN` statement. Multiple queries are required as each day is a new row, therefore there is a separate row key to search for each day, as described in Section 3.4.3. Cassandra stores the most recent data to the Memtables, until the commitlog reaches capacity and flushes to SSTables. Therefore if the data is newer it will still be in-memory but if it is older it will have been written to disk. Therefore having a spread of dates measures the system's response time when reads must come from memory and disk.

Hashtags were generated for the exact match query, 80% were hashtags that were included in the dataset, 20% were not. The database behaves differently when no results are found. This is because *Antares* need only read once from disk and as there is no result it returns straight away.

Number of Queries	Mean response time (ms)
0	0
10	99
100	137
200	140
500	141
1000	200
2000	201
3000	206
10000	402

Table 3.10: Mean response time for exact match query with increasing concurrent queries

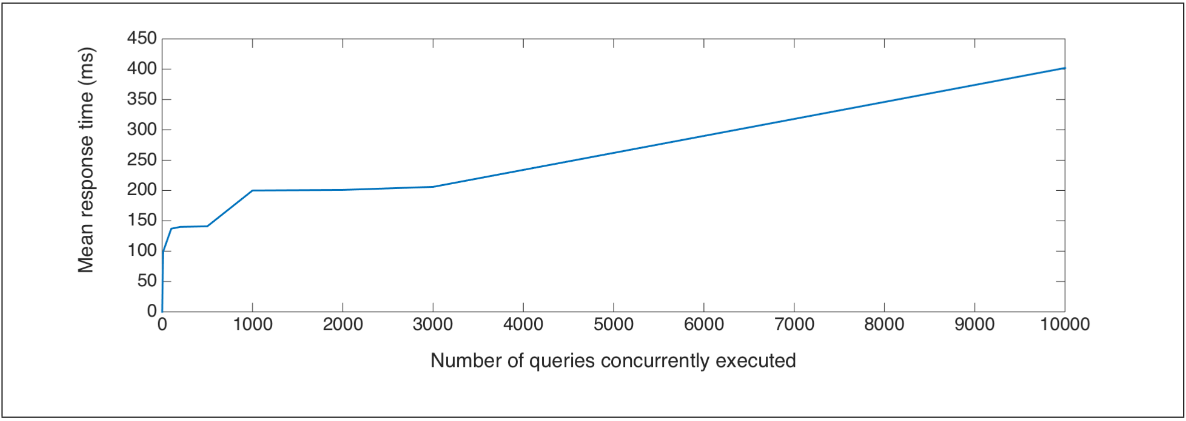


Figure 3.25: Mean response time for exact match query with increasing concurrent queries

Figure 3.25 shows a graph of the average response time for Query Four when executed across the database. *Antares* supports a higher throughput and only starts to increase rapidly around 10,000 queries. *Antares* supports large numbers of users querying the system simultaneously without losing data. Even with 10,000 queries the response times are under 400 milliseconds.

The next experiment used Query One. This is a range query and parameters select all hashtags within a time-bound. The parameters for this query were generated with the same distribution as the previous one. This query differs from Four as it uses the minutes to bind the query to a time range. The queries for this experiment are generated by the same mechanism as previously described. The average response time for Query One is displayed in Table 3.11, this shows 1,000 queries executing with a response time of under 2,000 milliseconds. Figure 3.26 demonstrates the linear scaling of the reads as the number of queries increases. This is because the indexing reduces processing, increasing the efficiency and performance. This query is slower than Four, as a range query queries more data. The experiments were stopped after 10,000 as the longest response time for a user reaches 2 seconds.

Number of Queries	Mean response time (ms)
0	0
10	299
100	383
200	415
500	1072
1000	1891

Table 3.11: Mean response time for Query Four with increasing concurrent queries

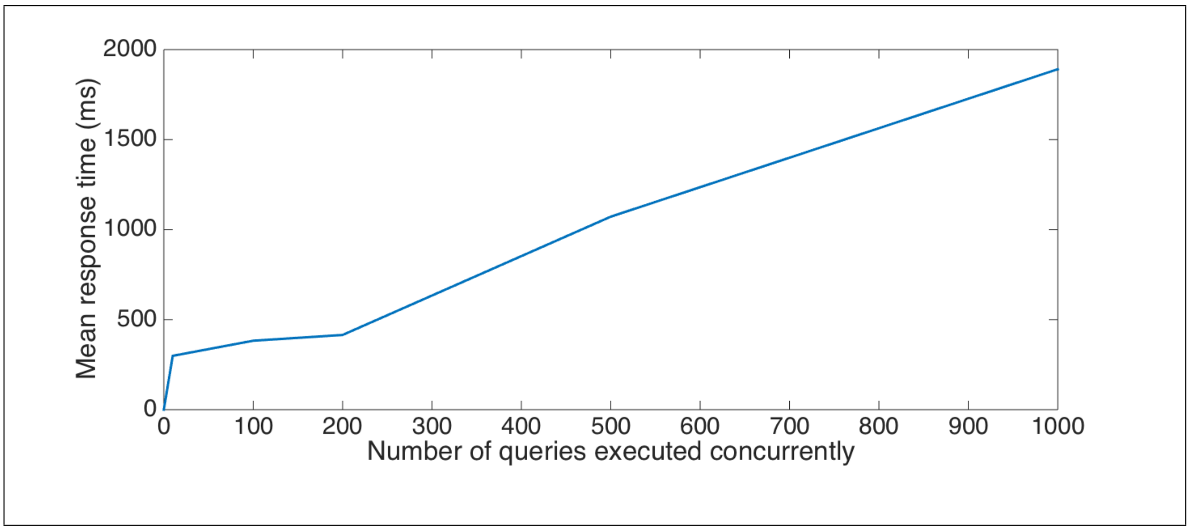


Figure 3.26: Mean response time for Query One with increasing concurrent queries

3.5.3.4 Conclusion

The response time of the queries provided low latency query responses that ensured that stream querying could be combined with the historic store. Cassandra requires index designing and implementing to support low latency querying. Temporal time indexes can help to reduce latency and have a percentage savings of over 90%.

3.5.4 Stream Queries

As described in Section 3.2.1.5 the ESPER complex event processing system was used to process the stream data. The following experiment evaluates whether ESPER can be used to ingest the event stream at the full rate, and whether triggering queries has an effect on performance. Figure 3.27 shows ESPER executing both without a trigger to the historic store and with. As described in Section 3.3, when a user specifies a time period which starts in the past and continues into the future *Antares* recognises this and executes a combined query. For this *Antares* executes a continuous query and once a constraint is met another query is executed across Cassandra.

The first dataset shows the throughput of Tweets, for the exact match query with over 160,000 Tweets in 18 seconds (9,000 per second). This demonstrates ESPER's ability to process the Twitter firehose quickly and efficiently. This is the same for the stream with read load, the dip at the beginning is ESPER reaching a stable state after staring up.

This experiment shows that the CEP system can meet the performance requirements. For each Tweet the hashtag is identified and counted. The hashtags are saved and a count for each stored in-memory and checked until a threshold is exceeded.

Time Elapsed seconds	ESPER	ESPER with read load
0	0	0
2	30100	7379
8	100000	82139
15	142000	119217
20	170000	170000

Table 3.12: Tweets processed per second by ESPER

3.5.4.1 Conclusion

ESPER supports a scalable stream processing system which can ingest the entire firehose without having to shred data. This provides an implementation for the scalable features stated in the introduction.

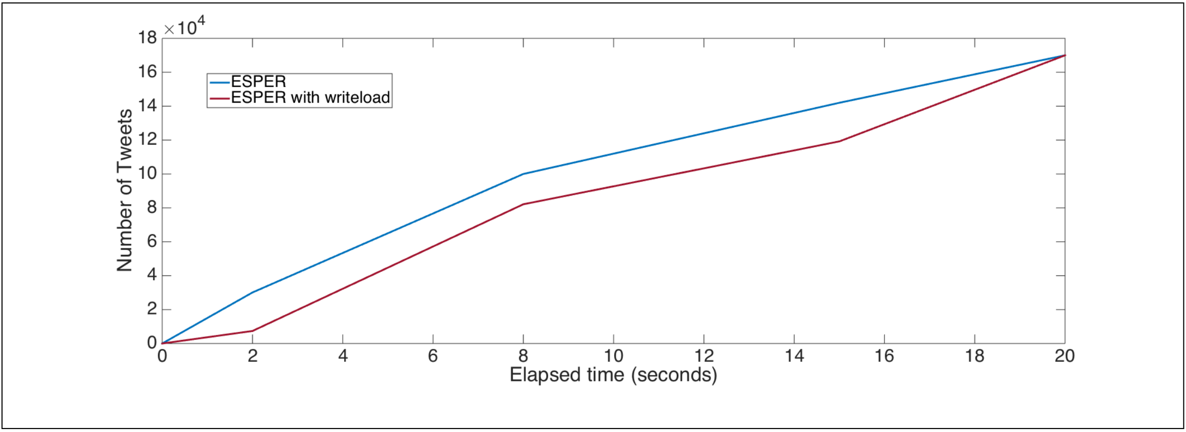


Figure 3.27: Tweets processed per second by ESPER

3.5.5 *Combined Queries*

This section evaluates how *Antares* performs for combined queries (Section 3.3). Combining stream and historic processing can be challenging, as historic queries must return results with low latency so as to return up-to-date results in time to be combined with the stream query being executed. It would be a huge drawback to receive the results after the stream query had returned and another was executing because then the streaming data results may be irrelevant.

For a combined query the time-window specifies a lower time-bound –a time in the past– and the upper time-bound in the future. The query monitor identifies that it is a composite query and sends a query to ESPER (to execute a historic query if the threshold is exceeded). For this combined query, exact match query (Figure 3.13) and range query (Figure 3.14) have been used. As the Tweets are received from the stream, ESPER keeps state about the hashtags, and counts each time they have appeared, until they exceed the threshold specified in the query. If this occurs, the exact match query is executed over the database, by passing the hashtag and the time-window for the historic data. This returns a result specifying whether the hashtag has been mentioned before in that time-bound and selects the contents of Tweets it was contained within. The time-bound is 20 seconds with a threshold of 20 hashtags. As previously shown, the range query executes efficiently in ESPER but whether combining it with a query to the historic store affects its performance is now explored. This is shown in Figure 3.28.

The experiment is executed over the same four-node Amazon cluster as previously described. The collected set of Tweets is again used to simulate the Twitter firehose with a peak load. Each Tweet is then written to the database. While the data is being written, varying numbers of queries are executed across the database. Exact match query was used as it had the most efficient execution and the parameters were kept the same as the previous experiment to simulate an identical read load. Figure 4.13 illustrates the same linear shape as the previous results as the amount of queries increases. The mean response time is under 600 milliseconds for 2,000 queries, eventually reaching 1.5 seconds for 5,000. This illustrates the system's ability to process a heavy simultaneous read and write load.

Number of Queries	Mean response time (ms)
0	0
10	100
100	146
300	197
421	361
916	518
1997	523
5000	1535

Table 3.13: Mean response time for read and write load

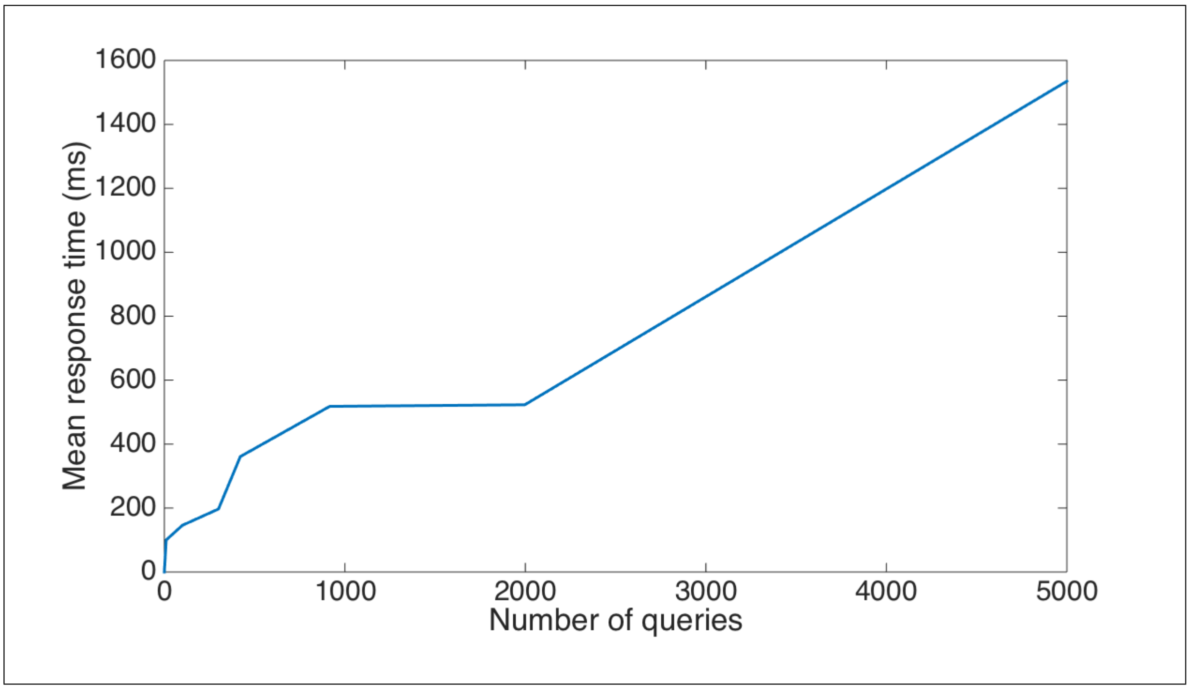


Figure 3.28: Mean response time of query four with increasing concurrent queries while simultaneously inserting to the database

3.5.5.1 Conclusion

Antares supports combined querying while simultaneously inserting data into the historic store. The performance does not decrease significantly and supports collection and processing of large scale data. The mean query response times for searching in the database provides a mechanism for combining the stream processing with the historic processing.

3.6 Conclusion

This section provided a literature review of different technologies, which led to the combination of ESPER and Cassandra being used in *Antares*. These technologies were used to provide a scalable and low latency layered approach to the processing of Twitter data. Cassandra allowed feature one to be implemented using indexing techniques to map the Twitter data to the database using temporal patterns - this reduced the query response time. The evaluation demonstrated the use of temporal indexing supported low latency queries so they could be combined with stream data in a timely manner while simultaneously writing to the database.

The query monitor maps the time window given by a user to the correct query - this is then used to execute the correct query whether it be historic, stream or combined. This satisfies feature two as the user enters a time window without having to know the kind of query they are executing.

The layered approach shown in this chapter supported scalable stream and historic queries by load balancing requests utilising both layers. Requests could be handled by either layer this meant the number of time-outs were reduced. A time-out blocks for 5 seconds, however with a large number of requests this will increase linearly if the number of requests is the same. However, if there are bursts of requests this could increase exponentially, therefore the use of two layers supports the load balancing of requests and reduces the number of re-tries. The layered approach takes advantage of in-memory speed and the scale of disk to provide scalable near real-time querying of Twitter data.

Antares uses batching and asynchronous execution to improve the ingestion rate and support scalability of the system - this achieved feature three as shown in the evaluation section where the ingestion rate was approximately 35,000 for only four machines.

Antares supports an indexing structure for efficient querying, which was derived from a literature review. *Antares* implements a set of comprehensive queries to analyse the relations of a data model defining Twitter. It provides a novel mechanism for combining queries, but allowing the user to transparently enter only one query. Asynchronous queries and batching are used for high insertion rates, extending Cassandra.

The rest of the features are described in the following chapters.

4

Spatial Extensions

Contents

4.1 Introduction

This chapter extends on the previous system architecture design and implementation demonstrating scalable analysis of geospatial data. The previous functionality of *Antares* is improved to help an on going project Tweet My Street. As has been mentioned Tweet My Street is a cross disciplinary project with social scientists who required scalable geospatial analysis. Scalable geospatial analysis is a growing area of research with many trying to identify a solution to support analysis of mobile devices which enable location services and gps devices are used to "tag" a users location. Twitter allows the user to add their location as they post a Tweet. Therefore the analysis and visualisation of this tag in a Tweet can support users in adding context to their analysis. For example if we use the use cases described in the previous chapter, if a company are running an advertising campaign for a new product they can monitor the hype on Twitter by tracking keywords - however if they use location too then they can monitor the location of the ad campaign too, and derive whether people passing the advert are commenting on the product which will show the effect of the advertisement itself.

Antares aimed to provide the user with a scalable means of displaying the data on a browser to allow the user a more automated means of analysis. This chapter will describe the extensions that were designed and implemented to support this. Spatial analysis can be complex due to its multi-dimensional nature, however there are traditional techniques out there. These however do not have the scale of current noSQL solutions. The research identifies whether using a noSQL database with spatial processing techniques can help to improve the scale of processing for spatial data - which is needed due to the large amount of data produced by mobile devices and as they increase in number so will the data. A literature review was undertaken to identify processing techniques and the different noSQL solutions to processing spatial data. A solution was then designed and implemented, which is described. This was then evaluated against different noSQL solutions using a simulated stream of Tweets.

4.2 Related Work

This section describes different geospatial structures which support efficient spatial storage and processing it then leads on to current solutions which use these techniques and noSQL databases to enable scalable geospatial analysis. *Antares* uses a simpler querying model to support a scalable solution for querying Twitter data with low latency query responses. The aim of *Antares* is to provide a scalable map view of Twitter data that maybe panned around, zooming in and out with speed and no limitation on the size of the data. The size of the data should also not hinder the performance. The literature review undertaken reviews current solutions and compares the technologies used.

4.3 Research into Spatial Databases and GIS

This section reviews and evaluates various geospatial processing structures and how these are being extended to exploit the scalable nature of noSQL databases. Spatial data is intrinsically complex to process, which is attributable to its multidimensional nature. Multi-key searches are required to support range queries across such data. There are multiple well-established structures employed for storing and processing spatial data, in this section we review and compare a subset of different indexing structures for geospatial analysis, the most common of these are tree structures.

4.3.1 B-Trees

A B-Tree [67] can be defined as a hierarchical structure of nodes beginning with a root node, which references sub-trees of linked child nodes. These direct searches to the leaf nodes containing data. These structures are employed to store and process hierarchical data. When identifying or inserting the data each level of the tree is traversed, comparing nodes until a leaf node is discovered and returned or inserted. If spatial data is stored using a hierarchical structure an n-dimensional rectangle is used as a bounding box for the objects indexed. Each node will have a maximum number of entries. All leaves appear on the same level, and if m is the maximum number of entries then m=‘m/2 is the minimum number of entries in a node. Each node represents a bounding box, which represents a geographical area. If the data is inserted and the nodes limit is surpassed then it is split into two new leaf nodes. Rebalancing the tree is carried out when it is necessary to ensure the maximum number of levels below the root is the same depth or to within a set limit. The reason for this is to reduce the length of branches therefore reducing the number of traversals and reducing query execution time.

A B+ Tree is an n-ary tree [67]. The root node can be a leaf or a node with child nodes. The number of child nodes is variable. Each node in the tree only holds a key and the data appended to an additional layer after the leaf nodes. The data is appended to the position which has the correct key, so when the tree is traversed for that data it will find the leaf node that points to that piece of data.

4.3.2 R-Trees

A fundamental type of tree is R-Trees [68], these are a dynamic indexing structure. R-Trees are typically height balanced, therefore the main cost incurred when using this structure is rebalancing the tree. This tree structure uses proximity to group clusters of points together for querying. Each node in the tree has a capacity limit; when this is exceeded the node is split and data contained by that parent node is divided between the two new child nodes. A node is split based on the position and closeness of the points contained in the node. There are different methods for this division: Linear – choose far apart points as ends and keep the points that are held within; Random – choose nodes randomly and assign them so that they require the smallest minimum bounding rectangle (MBR) enlargement; Quadratic – choose two nodes so the dead space between them is maximised; Exponential – search all possible groupings.

A variant of the R-Tree is the R+ Tree [68]. This structure completely avoids overlapping bounding boxes, which is achieved by adding objects to multiple buckets, which means the depth of an R+ Tree could be substantially larger than an R-Tree constructed from identical data. R-Trees are a useful data structure for geospatial processing, however when compared with structures like geohashing they are overly complex.

4.3.3 Kd-Tree

Kd-Trees [69] are binary trees which equate the median of all the points stored in the node and then split the node at that location in the bounding box. Kd-Trees also support the simple addition of another dimension, such as time. Nodes split alternatively between each dimension in the structure, therefore each level represents a different dimension in the tree. Kd-Trees are particularly interesting as they can use a third dimension and are easy to implement.

4.3.4 Quad Tree

Another modification of the Kd-Tree is the Quad Tree [70], instead of dividing the buckets of the tree into two each node is split into quadrants. This structure is used for two-dimensional data e.g. "find all the cities which are within 300 miles of Chicago or north of Seattle". Inserting data has the same principle as a binary tree with comparison done at each level to decide into which one it is inserted. Deleting and merging two trees can be difficult. Quad Trees support easy geospatial processing and allow for more fine-grained querying when compared with the Kd-Tree.

4.3.5 Geohashing

Geohashing [71] is a hierarchical spatial data structure, which uses base 32 to encode geospatial data. The more similar the geohash the closer the data points are. It is used as a simple and easy mechanism for storing geospatial data, as the algorithm is simple and partially matching the hashes can give an approximate mechanism for detecting closeness quickly.

4.3.6 Distributed Trees

As the size of datasets increases this has led to the development of distributed structures such as the SD-R Tree [72], which is an in-memory distributed R-Tree. It is an extension of an R-tree – each leaf node stores a subset of indexed data and the height differs at most one. The cluster that holds the distributed R-Tree is accessed by an application on the client. The client accesses the cluster from an image stored on the client. Therefore that image (and the structure of the tree held in that image) can become outdated by splits to the data (these are the same as splits described above for the R-Tree). The cluster sends messages to the client periodically to stop the update on the outdated image. Distributing the tree in this way is more complex than the indexing in *Antares* and unnecessary to improve query execution time. The messages may increase wait time and with only one client this will cause a bottleneck when ingesting the data.

A P-Tree [73] is a distributed B+ Tree and is an index structure for distributed range queries. This maintains sections of semi-independent B+ Trees at each peer. There is no primary copy replication. Each key is viewed as the smallest key to each peer, and then each one is responsible for looking after the left-most point in the tree from root to leaf. Each peer relies on a subnet of its peers to complete the tree. The P-Tree is a distributed tree but lookups can be expensive if the entire path of the tree is required, as many nodes may need searched.

The Oct-Tree [74] is a three-dimensional, distributed variant of the Quad Tree. It employs a geohashing algorithm to identify the node containing the requested points. A disadvantage of using this tree is that the root can become overloaded.

4.3.7 NoSQL system and Geospatial Processing

Social media, sensors and portable devices generate large volumes of spatial data; this has led to a deviation from traditional processing techniques. NoSQL databases support scalable, flexible and fault tolerant solutions for processing data. However they do not have well-established spatial querying techniques as do RDBMS. Therefore new mechanisms are required to combine spatial querying and distributed fault tolerant NoSQL systems. Some NoSQL based approaches are now described.

Cassandra is a column-oriented database, which provides support for spatial querying by using a framework called Solr [75]. Solr uses a textual search system called Lucene [30] for indexing the data and employs Cassandra as the data store. A Prefix Tree is used to index the spatial data; the tree divides the world into a grid. Each of these grids are further decomposed into smaller grid cells as the depth of the tree increases.

The Solr nodes distribution techniques do not efficiently exploit Cassandra's scale-out capabilities: each node in the cluster has its own tree, which maintains its own data. Therefore, any query being executed across a multi-node cluster has to search each node, which increases query response time.

Neo4j [34] the graph database, supports spatial querying using an R-Tree. It uses this structure for low latency searching of geometric shapes and topology operations.

MD-HBase [76] enables multidimensional spatial querying using the NoSQL database HBase. It linearises co-ordinates by using bit interleaving and the z ordered curve algorithm (which maintains locality) to index the data. This is referred to as the index layer and references the key used to query the database. MD-HBase builds two standard indexes Kd-Tree and Quad Tree. MD-Hbase can handle hundreds of thousands of inserts per second on a modest 16-node cluster and support response times as low as 100 ms [76]. It uses linearisation techniques to transform multidimensional points into one-dimensional data points. Additionally a novel naming schema called longest common prefix naming is used. A look-up array is stored and used for searches. It is similar to a Prefix Hash Tree. Concurrent read while splitting may be inconsistent, [76] have left consistency as additional work.

MongoDb [23] employs spatial querying; it uses geohashing to represent different quadrants of the region. This database suits sparse spatial point data because of its flexible schema.

Hadoop GIS [77] is a high performance spatial data warehousing system, which uses map reduce to process large datasets. It supports multiple spatial queries on map reduce through the use of spatial partitioning, a customisable spatial query engine RESQUE [78], implicit parallel spatial query execution of map reduce and effective methods for changing query results by handling boundary objects. It supports Hive for declarative queries. Spatial querying involves geometric computations, which can be very compute intensive. Datasets are broken down into tiles and then processed in parallel. This is achieved by using spatial partitioning – when tiles are too dense the process is executed again. Real-time spatial query engine supports generic querying, simple parallelism and leveraging existing query mechanisms. It builds tile based indexes on the fly, supporting dynamic indexing.

GISQF [79] is a system for spatial analysis, which extends Hadoop to implement map-reduce for spatial querying of GDELT [80] datasets. The system produces more efficient response times than Hadoop.

Another Hadoop solution is CLoST [81] which is a scalable framework used for spatial-temporal data. Its main objective is to avoid scanning a whole dataset, it is based on hierarchical partitions that use core attributes to efficiently parallel process range scans. *Antares* provides scalable spatial analysis but with real-time responses, unlike the batch map reduce functions of Hadoop based systems.

Another spatial NoSQL database is HbaseSpatial [82]. This is another system that extends HBase. HBaseSpatial has evaluated the system against MongoDB and MySQL. This system uses a static grid index for querying, but it uses vectors for searching, inserting, and identifying whether the latitude and longitude of the point data is contained within a grid.

MHB-Tree [83] combines spatial querying with a NoSQL data OrientDB. It uses a B-Tree based structure to process real-time insertion of spatial data. It stores the data using a geohash, and is evaluated against MySQL.

4.3.8 Conclusion

The approaches described here use more complex querying mechanisms, however this sacrifices the scale and performance that *Antares* requires. *Antares* exploits the scalability of a noSQL database to increase ingestion and search sizes with low latency, which is achieved by using a simpler querying model used to display the data in a view port. The system is designed to ingest and process large scale datasets, therefore there is no requirement to delete data this also improves the performance of *Antares* and differentiates it from other systems described in this literature review. This thesis aims to exploit the scalability, fault tolerance and flexibility of NoSQL technology and combine it with spatial querying techniques to develop novel algorithms and an indexing structure to support low latency querying that out performs existing systems.

4.4 Geospatial Querying

Geospatial data is inherently difficult to process due to its two dimensional nature – point data consists of a latitude and longitude. For example, querying for all points contained within a rectangle area on a map could require comparing each point held within the database. Being able to reduce the amount of points queried is beneficial and will reduce response time of queries. Traditional spatial querying techniques use tree structures to reduce the size of data queried to reduce response time. With the increase in dataset volume and the need for faster and more efficient processing, geospatial processing has seen the rise of a new challenge when processing data.

Antares supports a simple querying model for retrieving points contained within a rectangle [4]. This simplicity supports dramatic improvements in querying response times as will be further explained in this section. This query model is only possible because the system is append-only. The aim of *Antares* was to insert and retrieve large-scale data. No modification functions were required as the aim was to collect all data to support its retrieval at any point in the future.

Antares provides a point in rectangle querying model to support the querying of Twitter data in a view port for social scientists research. This has been driven by the cross-disciplinary project Tweet My Street where large scale datasets have been used to analyse different use cases. The data in these datasets provided geo-tags which if displayed on a map provide a richer analysis technique. The aim of *Antares* was to provide scalable and low latency querying of this dataset - no matter how many points were contained within the rectangle. Therefore the implementation of *Antares* supports this functionality and provides the user with a scalable and low latency mechanism for querying geospatial Twitter data.

Antares supports scalable geospatial querying through a user interface. The main aim of *Antares* was to provide functionality for the user to be able to zoom in and out quickly in a browser using the viewport (the user's visible area of the web page) as the region being queried.

Antares aimed to use the scalability of Cassandra and geospatial querying techniques to provide a scalable and efficient means of achieving this. Datastax uses Solr (with

Cassandra), as mentioned in Section 4.3.6, for geospatial querying. However it is limited in performance.

Solr allows users to query geospatial regions but with high latency and timeouts for larger regions (investigated in more detail in Section 4.6.2). Therefore *Antares* aimed to reduce query response time and additionally support large region querying using Cassandra and enable the user to quickly and efficiently pan around a browser querying the data. The problems with Solr are now described alongside the design of a new system to produce the required performance.

4.4.1 Solr Overview

Cassandra is a column-oriented database, which provides support for spatial querying by using Solr [75]. This is a search server built on Lucene Core. However, the response time for retrieving large areas of data points is too long, this is shown in Section 4.6.2. Query response times must be near real-time to support quick zoom in/out functions. This is overcome by the new approach described in this section. This uses an in-memory cache and novel algorithms to significantly improve performance and scalability.

Solr has three different types for spatial data "LatLon", "PointType" and the "Recursive Prefix Tree Field Type" (RPT) [84]. LatLon represents a two-dimensional point of latitude and longitude, based of the spherical earth model. The latitude and longitude are saved as separate numbers within the LatLon construct. PointType is the same however it does its calculations using the flat earth model. RPT uses the Prefix Tree for storage and searching. Geospatial types are added to an XML file to register with Solr that it is being used. Solr uses bbox, which generates a box using a specified co-ordinate and a distance. The distance is then used to create the region being queried from that point.

Solr uses a prefix tree to index the spatial data. This tree employs 'Geohashing to search the tree using prefixes. It is an ordered dynamic tree structure, which uses keys of type string, as shown in Figure 4.1.

Unlike a binary tree, no node holds the id of another. Instead, the position in the

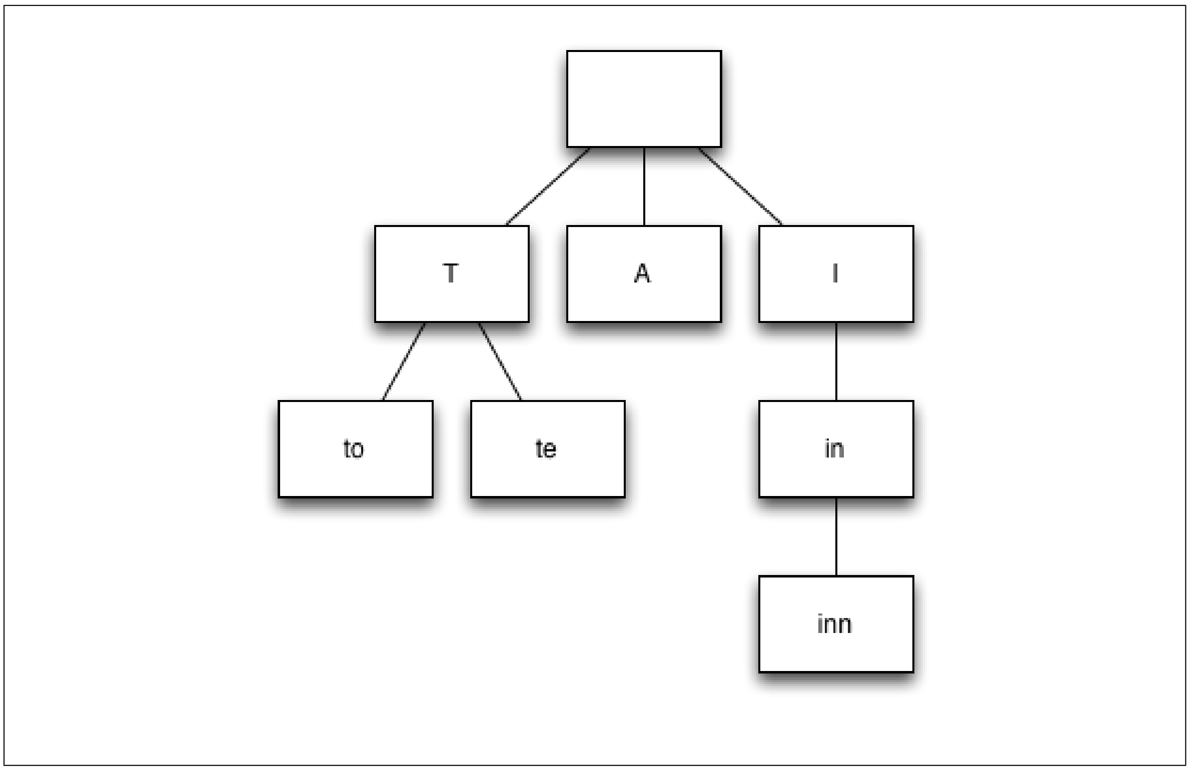


Figure 4.1: Prefix Tree

tree defines the key it is associated with. All descendant nodes have a common string prefix associated with the parent node, with the exception of the root node, which is an empty string. The values of each node are not associated with every node, only the leaves and nodes with a key in a prefix of that node's key. The Solr tree divides the world into grid cells as described for other tree structures. Each grid cell is then further decomposed into smaller grid cells. The larger ones being level one and the smaller ones level two, this continues for n levels dependent on the size of the dataset.

Each grid is given a prefix as a key using a Geohash – in Figure 4.1 the first node in level one has a prefix of "T". As nodes are split the prefix is extended – as can be seen in Figure 4.1 the child nodes of "T" become "to" and "te".

When a query is executed to search for a region of points, the Lucene filter is employed to identify which nodes to search for and which to disregard. Once these have been identified, each Geohash is compared to the user's query and matching points are returned.

As nodes are added to the cluster a tree is created for each one. The structure is then responsible for maintaining data on that node only, so adding more nodes provides additional storage but does not utilise the added CPU power by executing commands in parallel. Therefore, any query being executed across a multi-node cluster has to search each node, which increases query response time. This results in increased delays to response time and limits the querying power of Solr.

Lucene is not an eventually consistent system and this is why the data cannot be distributed. If the data were distributed across the cluster and a global index was used then the system would have to implement a global lock. Therefore each time a read or write was executed on the system each node would block until the query returned, therefore yielding a increase in response time.

4.4.2 Dynamic Index Layer

This section describes the investigation of a dynamic grid to help support scalable and near real-time spatial querying of Twitter data. The design decisions, structures and algorithms employed to support low latency querying will be discussed

Antares employs an indexing layer, which uses a cache to store pointers to geospatial data in Cassandra. This enables efficient querying of the spatial data to reduce response times. The indexing layer constructs and executes queries using the cache; this ensures that Cassandra does not know anything about the structure of the cache. This enables efficient querying and takes advantage of Cassandra's column family schema for indexing data.

A common performance problem when viewing a selection of points on a map can be the volume of data being returned to the browser. Typical static grid methods will return all the points in each grid that has been specified. If a view port intersects multiple grids then the number of points is multiplied by the number of intersecting grids even if they are not all shown in the view port. This can be exeacerbated by the density of the number of points in the grid as a static grid method would have no means of dividing the grids if they became highly populated. This can increase query time and even make browser viewing impossible if the time-out is too high.

It was believed that to reduce the number of points outside the viewport being returned a dynamic grid would help and therefore reduce query time. By implementing a dynamic grid structure more densely populated areas can be split into smaller grids, supporting more specific and fine-grained querying. Limiting the number of points in a grid would also improve performance and ensure that response times are reduced as there would be a maximum on the number of data points that can potentially be returned at any point. Therefore a dynamic grid indexing structure would support the aim of zooming in and out and panning around.

The cache used by *Antares* is a tree structure. It improves spatial querying performance by extending Cassandra and leveraging its scalable characteristics. The index layer is used to manage and construct queries to the database to ensure low latency querying. *Antares* supports spatial querying by employing algorithms, which are executed across different tree structures.

The index layer has different user-defined configuration parameters that can be tuned; the height of the tree can be specified to avoid tree nodes representing distinctly small geographic areas (e.g. GPS is only accurate to four meters). The capacity of each tree node is also configurable, providing fine-grained or broader tree nodes.

Figure 4.2 demonstrates the distributed nature of *Antares* – both the client nodes and the Cassandra nodes are distributed to increase efficiency and scalability. A client node is the server holding the tree structure and not a client holding a browser. As can be seen in Figure 4.2 the clients ingest data, execute queries and the indexing layer is also stored in-memory here. Storing the structure in-memory means that traversing the tree takes very little time. Each of the clients ingests and accepts queries simultaneously. The queries to the Cassandra cluster can be accepted by any node that is available.

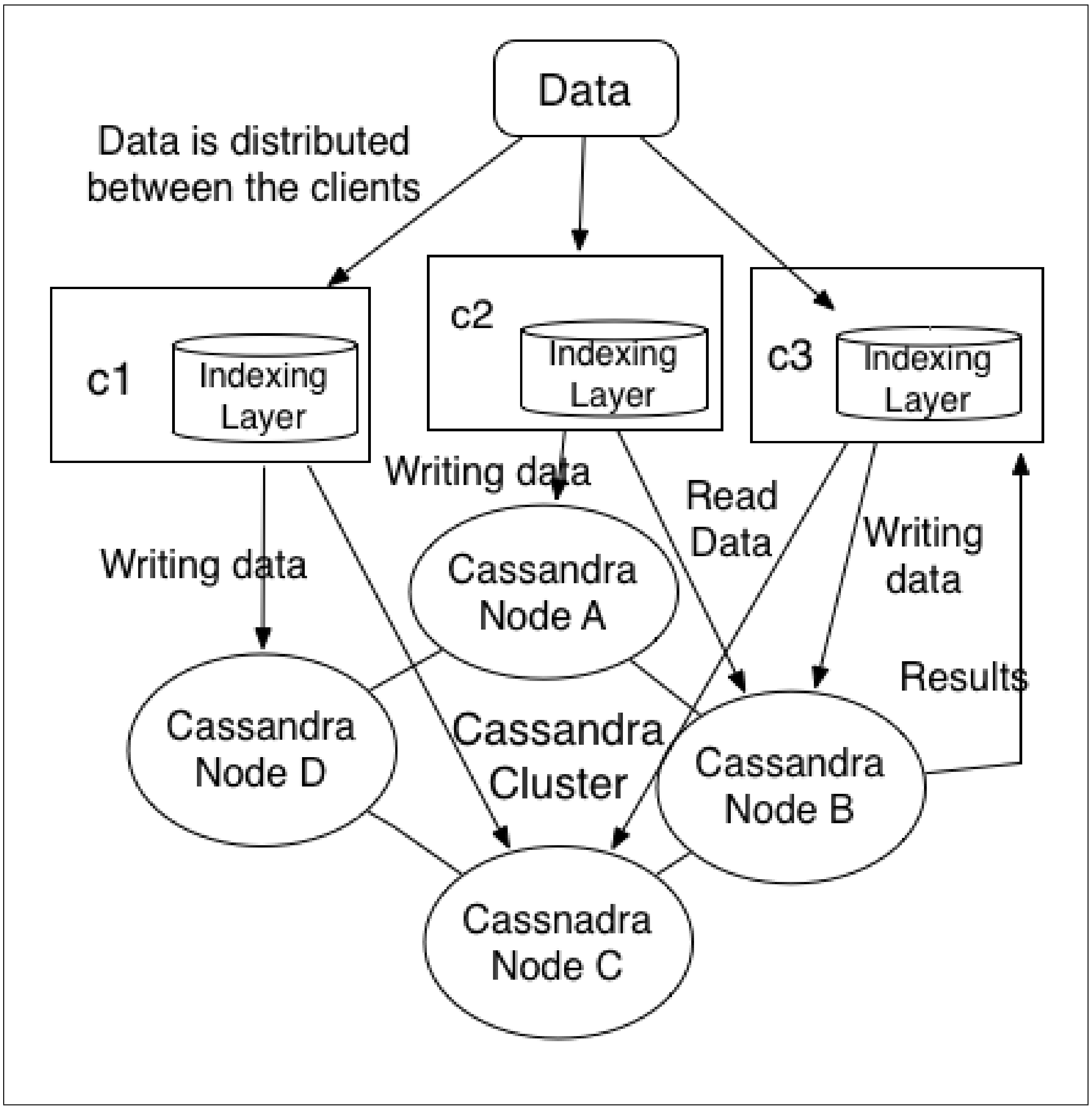


Figure 4.2: *Antares*: geospatial architecture

Figure 4.3 shows the abstract layers of the structure of the geospatial querying for *Antares*. Each time new data is received by a client node this is sent to the indexing layer. Here an algorithm is executed to traverse the tree and identify the correct tree node to insert the data into. A query is then constructed and sent to the database to write the data. When a query is submitted to the client node this is also passed onto the indexing layer. However the indexing layer executes a search algorithm, which traverses the tree to identify the tree nodes, which must be queried. Then a query is constructed and submitted to the database to retrieve the data.

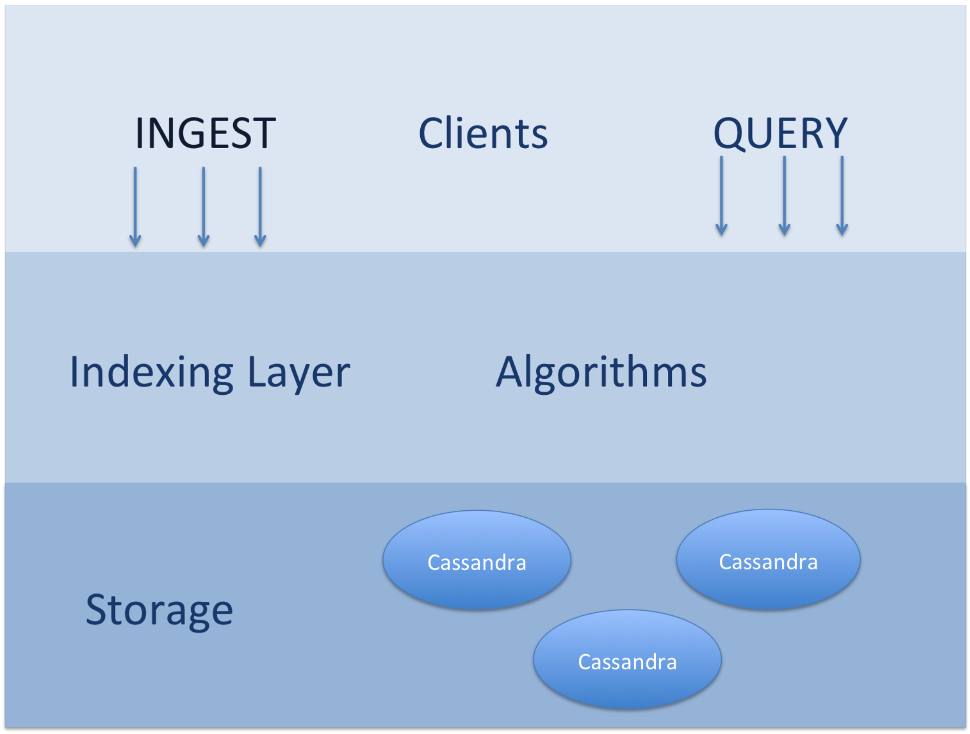


Figure 4.3: *Antares*: abstract geospatial architecture

Antares supports the interchange of different structures but employs the same algorithms over them. The system takes advantage of well-established tree querying techniques but removes some of the more “traditional” storage and processing techniques employed when considering tree processing. This is possible due to the immutable nature of the data used, as it is append only. The requirements of the tree also change due to its distributed nature, as distributed locking can cause problems and deadlocks; to avoid these, algorithms were designed to avoid the need for locks, and as a result, most operations can be done concurrently.

The indexing layer is held on a client node and the tree is copied across each client node in the cluster, which decreases the chance of bottlenecks and supports scalability by increasing the number of nodes ingesting data. All spatial queries utilise this layer, rather than accessing Cassandra directly, as shown in Figure 4.2.

The tree currently supports three operations, one to add data into the tree and then split the tree nodes if they have reached capacity, a search algorithm for point data and a range algorithm.

Each node in the tree represents a geographic region on a map as demonstrated in Figure 4.4. The first grid is the whole world then as the number of Tweets stored increases the grid is decomposed into smaller grid cells. Figure 4.5 shows the division of the grid cells using a Quad-Tree. The grid is divided into four new cells for this tree structure, so node B transfers data into nodes F, G, H and I. Nodes F, G, H and I are the child nodes of node B. Each node in a tree structure has a maximum number of entries (in this case Tweets) that it can consume until new child nodes are created and the data is “split” between them.

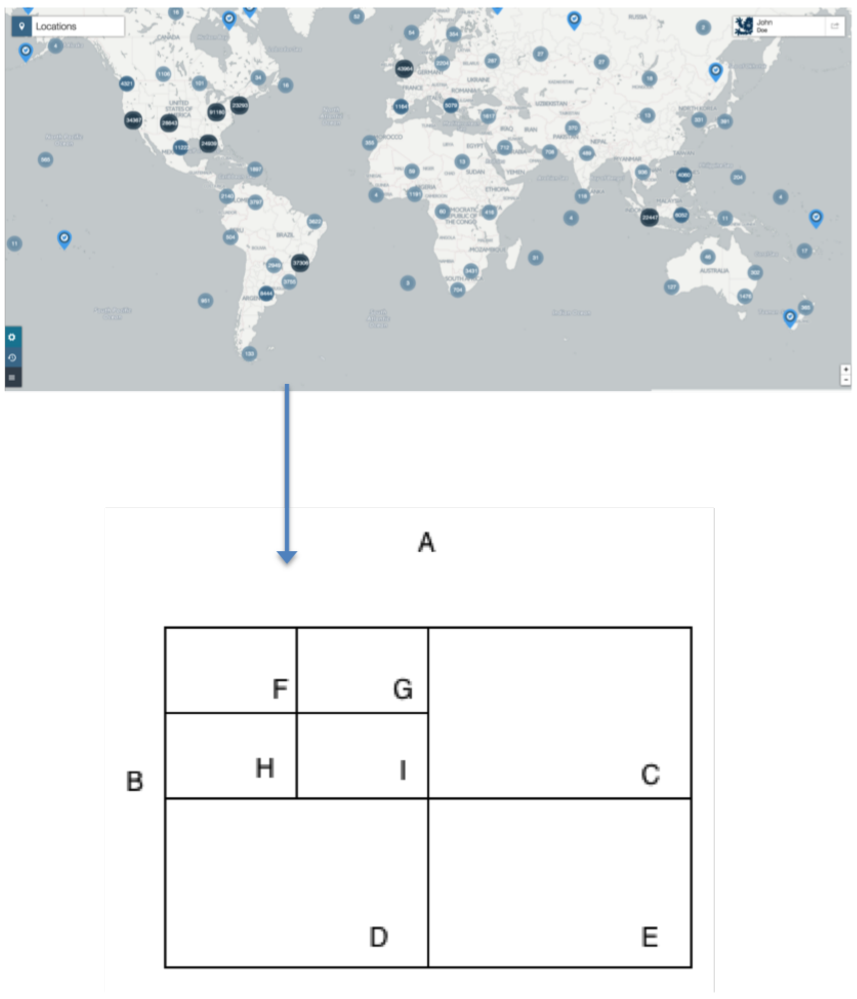


Figure 4.4: The mapping between the world and grid regions which represent tree nodes

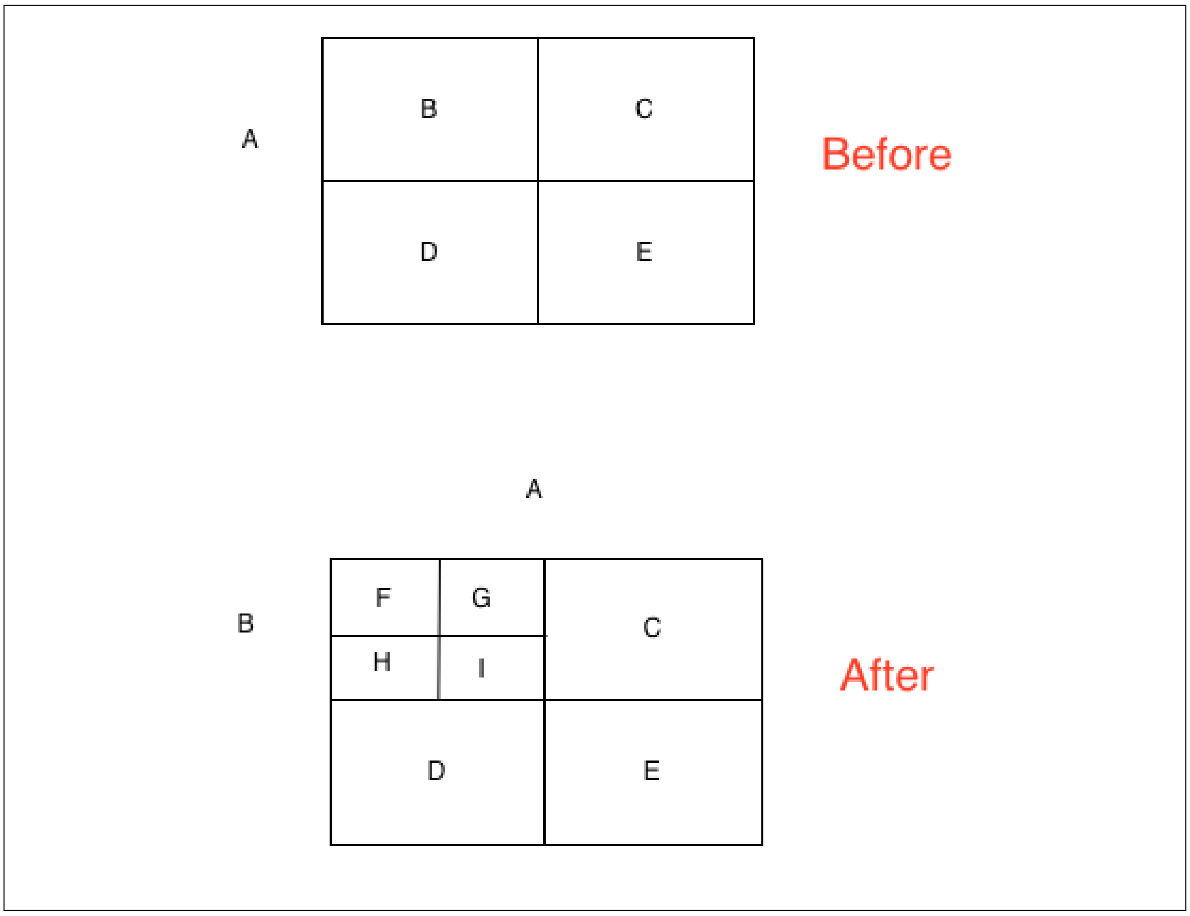


Figure 4.5: Node B being split

Figure 4.6 shows a Quad-Tree which is composed of the grid cells shown in Figure 4.5. Each node in the tree represents a region of the world. Figure 4.6 shows the root node A to the bottom level of the tree with child nodes F, G, H and I. As a grid is split new nodes are created and the data divided between them. By looking at Figure 4.5 and Figure 4.6 it is clear to see that the data is divided by region and not size and pushed to the new child nodes. This means the tree grows dynamically by data-intense locations.

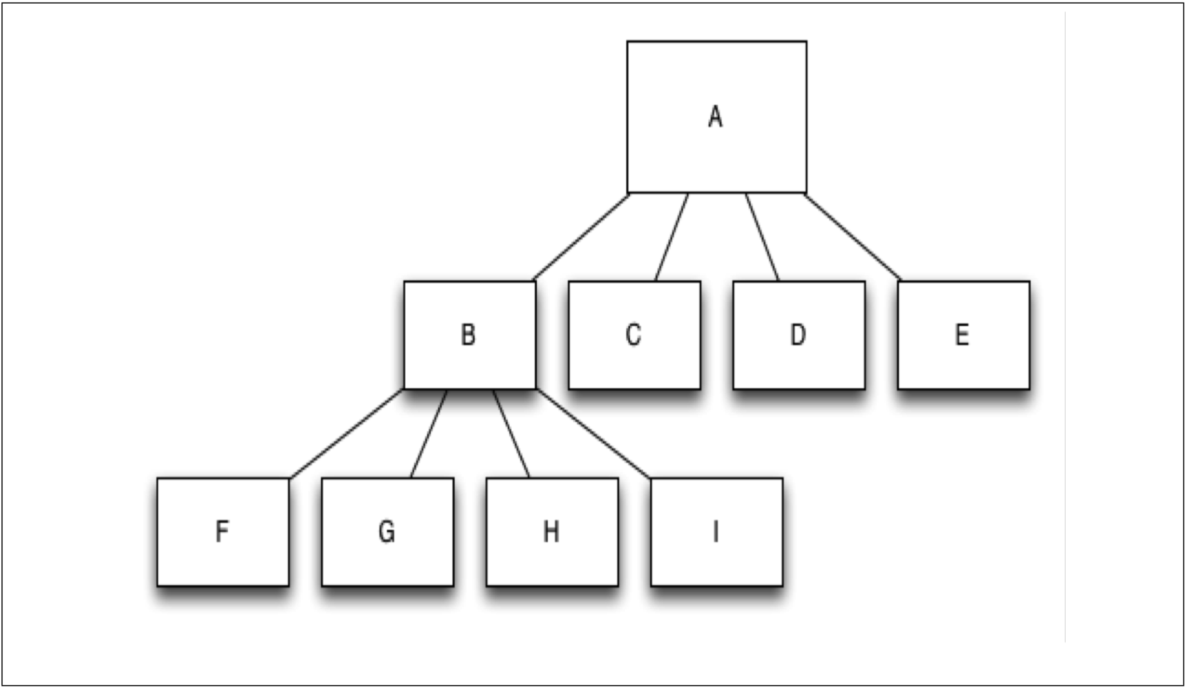


Figure 4.6: Quad-Tree mapped from the grid

Figure 4.7 demonstrates the mapping between the in-memory tree structure and the Cassandra cluster. Each tree node is written to the database non-sequentially; this results in faster writes and larger ingestion rates. This is because the write queries can be accepted by any Cassandra node and therefore wait time is reduced.

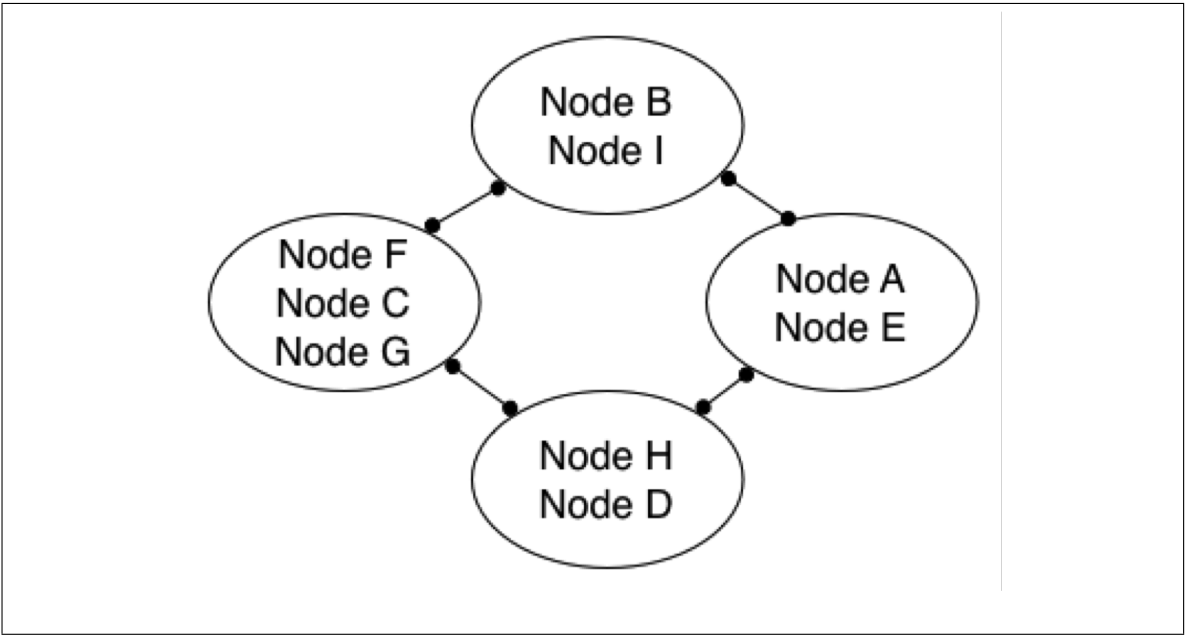


Figure 4.7: The grid changing for a split

Each tree node contains data about the region it maps to in the grid as shown in

Figure 4.8. This includes:

Id: This is a unique identifier for the data.

Number of children: this is the number of children the node owns and is used for consistency checks – mentioned later in Section 4.5

Entries: This is the number of data items already stored in the node.

Capacity: This is the limit on how much information the tree node can store before it needs to be split.

Bounding box: identifies the region the tree node owns.

Figure 4.8 demonstrates all the information held by each node within the tree. The id is used as a key in the database and is used to signify the record that should be read or written. As the id is the key this ensures that all records for one node are held on the same Cassandra node. This is because a key is the identifier for a row and rows by default are always held on the same Cassandra node.

When a new node is added to the cluster, because a node is full, it does not affect the indexing layer. This is because the cluster will automatically rebalance and the id (rowkey) is hashed to represent the new node it is now stored on. However during this time the cluster is not available for querying.

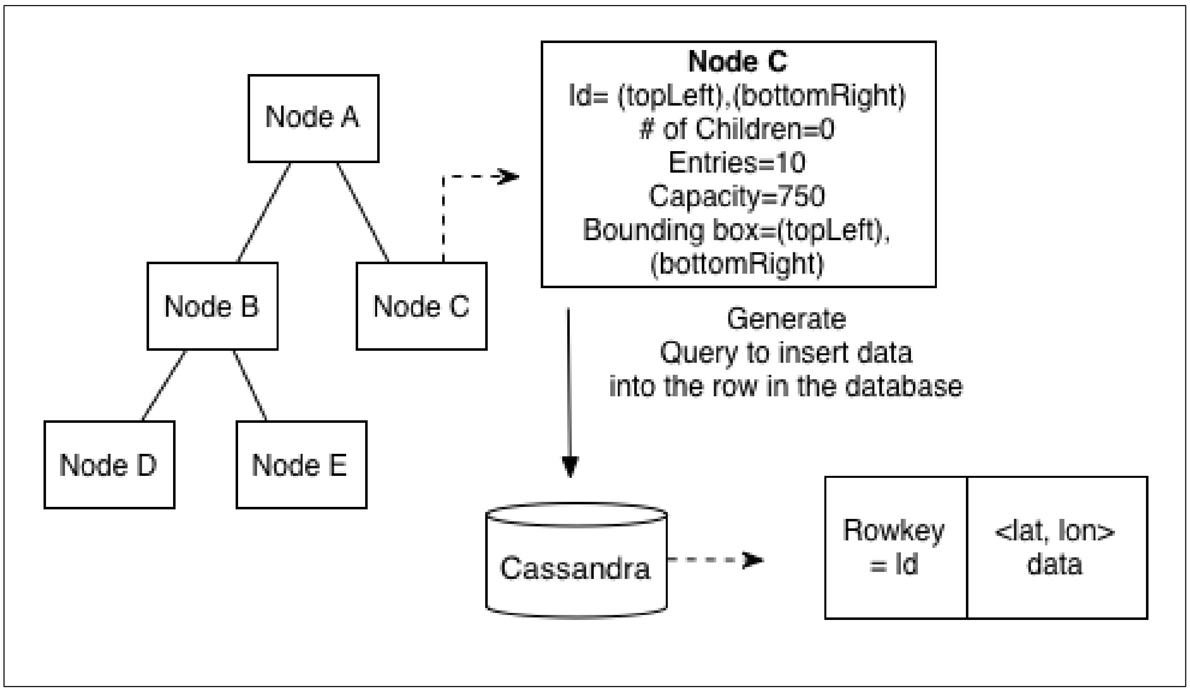


Figure 4.8: The indexing layer converted to Cassandra schema

The data contained within each tree node changes as data is ingested. Once the tree has been traversed and the correct tree node identified for insertion the number of entries is increased. The indexing layer also validates whether the number of entries has exceeded the capacity of the tree node. If the tree node has exceeded the capacity it is then split and new child nodes are generated. The number of children field is then increased.

When a query is accepted by the client to display a set of points on a viewport this can intersect multiple tree nodes (grids). This is because the viewport may intersect more than one tree node, therefore a point could be held in any of the tree nodes that are intersected by that viewport. This is shown in Figure 4.9, where the red grid is the viewport and it covers grids B, C, G, H, L and M. Therefore the client will send this request to the indexing layer where the search algorithm will be executed. This traverses the tree for the intersected grids and constructs a query containing the correct tree node ids. The query is then submitted to Cassandra to retrieve the points.

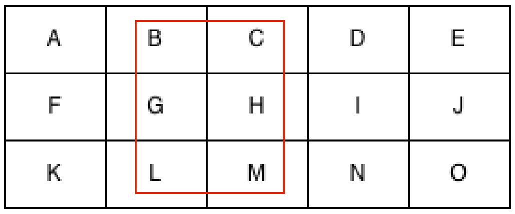


Figure 4.9: A viewport intersecting multiple grids

4.4.3 An Unbalanced Tree

The tree has no balancing algorithm. This reduces query execution time by removing the blocking time that would have been required to balance the tree. This was only possible due to the type of data being used, which meant an append only system could be implemented. With the new "Big Data" mentality, most use-cases don't require the ability to delete and modify data. The collection and processing of all data ever received at any time in the future is the new challenge.

Balancing a tree structure can support lower latency querying as fewer levels will be queried as shown in Figure 4.10. Fewer levels being queried results in fewer disk seeks as not as much data needs to be searched. However this is only an advantage for trees with a large number of levels and that are not distributed.

The tree structures used in *Antares* have a limited height as GPS are only accurate to four meters therefore there is never any need to divide grids/nodes any further after that meaning that the tree does not get any deeper. Data is just entered into the leaf nodes from there on. Therefore it is advantage to remove the cost of balancing the tree and having some branches which are longer as they will not increase query response time to the database.

For example a Kd-Tree of the whole world and having been split to the maximum depth, the height would be 15,000. So each degree is approximately 111 km (111.2 km for the latitude), therefore 4 m is equivalent to 0.036 degrees (rounded to the nearest two significant numbers). A world grid would be constructed of a 10,000 degrees x 5000 degrees therefore if each grid was split to 0.036 there would be 15,000 levels in the tree. Therefore searching in-memory, down one branch is not data intensive and is done quickly. Then only the leaves are added to the query to be sent to the database. This is because the database will still query the same number of nodes, whether it is balanced or not. The cache search executed maybe a few milliseconds longer (worst case) but it will be negligible.

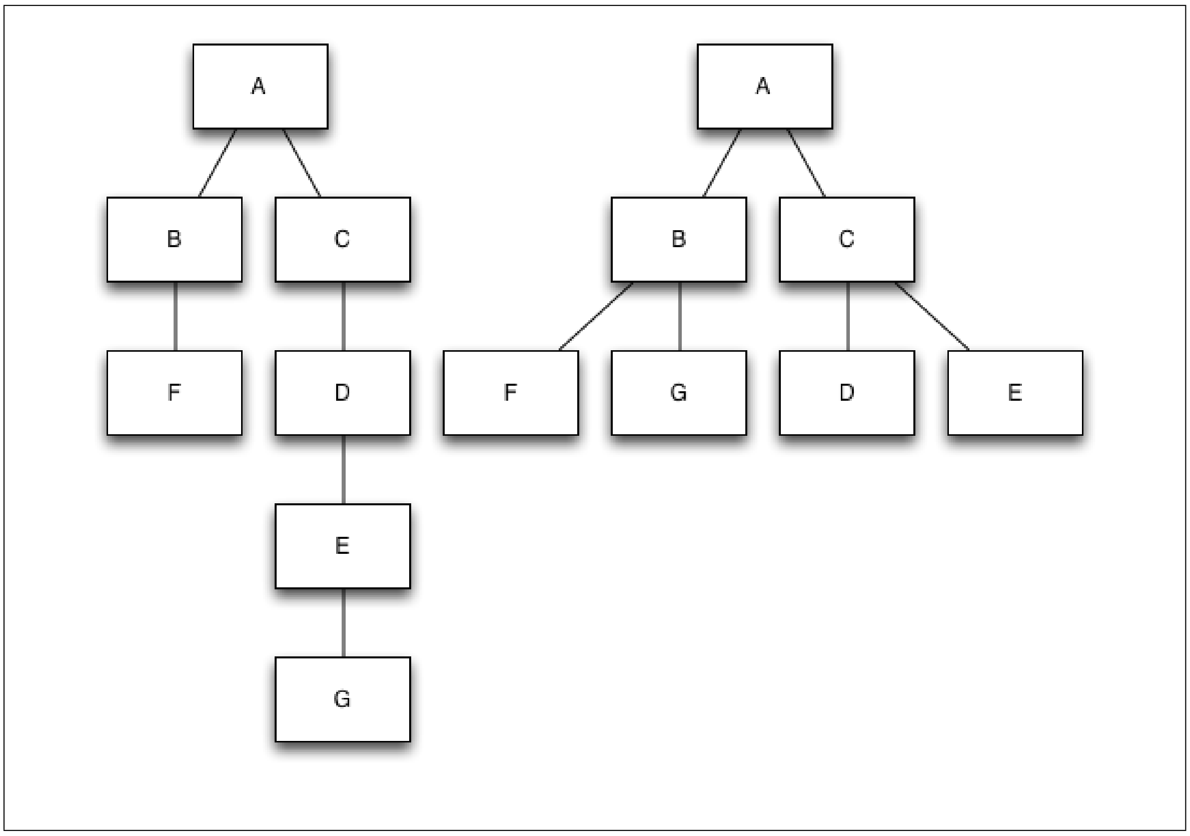


Figure 4.10: An unbalanced tree being balanced

To reduce the chance of bottlenecks the tree is copied and distributed across a number of clients as shown in Section 4.4.2, therefore keeping it unbalanced helps to reduce execution time of queries and increases ingestion rate. If the tree were balanced a global lock would be required blocking all queries and ingestion until the tree was balanced. This would therefore reduce ingestion rates and increase query response times.

Antares exploits the eventually consistent nature of Cassandra to yield increased availability and avoid data loss. This is done by always accepting writes and then checking and updating consistency eventually using a time-out – this is explained in detail in Section 4.5. Consistency can only be eventual because the data is immutable and only the pointer to the data location changes.

A typical disadvantage of using tree structures is query hot spots. This is where positions in the tree such as the root node become query heavy. This increases waiting times for queries to start executing and increases query response times. *Antares* avoids these hot spots by exploiting the distributed nature of Cassandra. The divides are based on write hot spots rather than load. This is because each node is distributed across the cluster non-sequentially, therefore the reads will also be distributed.

Each query to a tree node uses one CPU. Therefore queries can be executed in parallel if there is more than one CPU. Consequently as there are only a fixed number of CPUs the fewer queries executed (less tree nodes searched) the lower the response time as the execution will have a higher parallelism. The parallelism of the query is directly dependent on the number of CPUs. Once each CPU is executing a query then the queries wait to be executed.

Antares was designed to exploit this advantage. As data is entered into the tree it is split and pushed down the tree to the leaf nodes. Therefore the root node and tree nodes which are not children do not need to be queried. This is demonstrated in Figure 4.11 here the nodes marked with crosses are not included in the query submitted to the database.

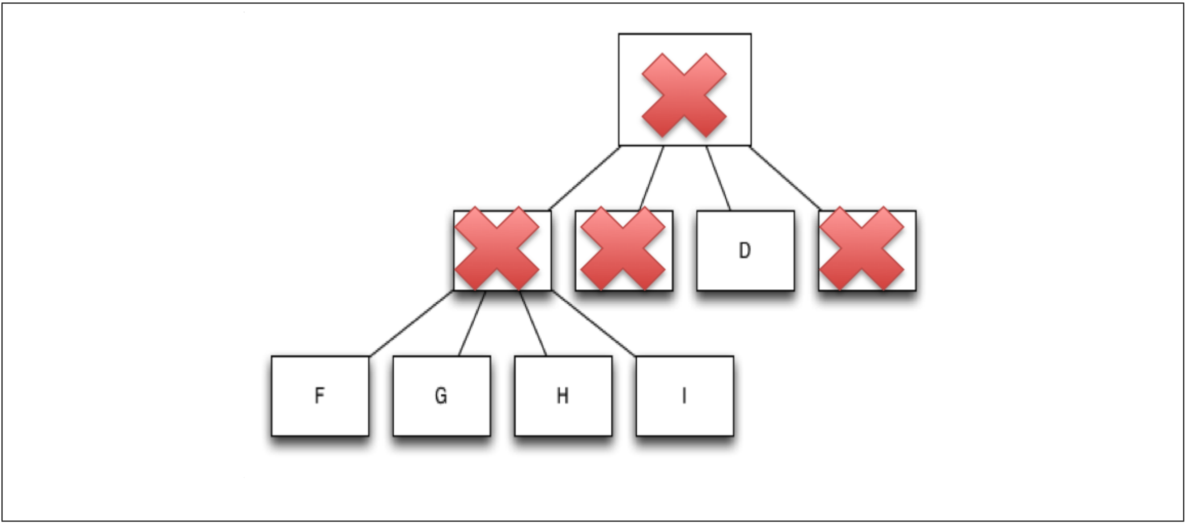


Figure 4.11: Nodes which are not queried as they hold no data

Therefore there are less queries to execute and the response time is reduced.

Another advantage *Antares* has is that all of the tree nodes are not written sequentially to the Cassandra cluster. This results in queries being distributed across the cluster. As a result there is not a single point (e.g. the root node) where queries are concentrated, which prevents hotspots. Potential consistency issues and solutions are described in detail in Section 4.5.

The three structures evaluated in Section 4.6.2 are a Kd-Tree, Quad-Tree and a Geohash structure.

4.4.4 Kd-Tree

After deciding that a tree structure would be optimal for append only geospatial data, as there is no lock required, the tree will not need to be balanced and is used for holding keys not the data itself. The first structure implemented in *Antares* was a Kd-Tree [69], which offers a simple tree structure to execute the algorithms over. A Kd-Tree is a derivative of a binary tree, therefore each parent has two child nodes. Each node contains the number of data entry points added, the number of children and their ids, the capacity of the node and the node id.

Each node also contains a bounding box. A bounding box is the region (or grid) that a node owns, and is used to dictate where spatial data is stored in Cassandra – the node id is the key used to access the data. The id, constructed from the top left and bottom right co-ordinates as shown in Figure 4.12, is used to determine if a point is contained in the bounding box's geographic region. Therefore given a world map, the area of this geographical region is described by using the co-ordinates of the top left corner and the co-ordinates of the bottom right corner, this region is a "grid". This "world grid", shown in Figure 4.12 represents the root node's bounding box. When the tree is first instantiated it contains only one node (the root node), as this represents the "world grid" all points are contained in the bounding box. This results in all data being stored to the data store with the root node id. Once the capacity of the root node is exceeded the bucket and node are "split".

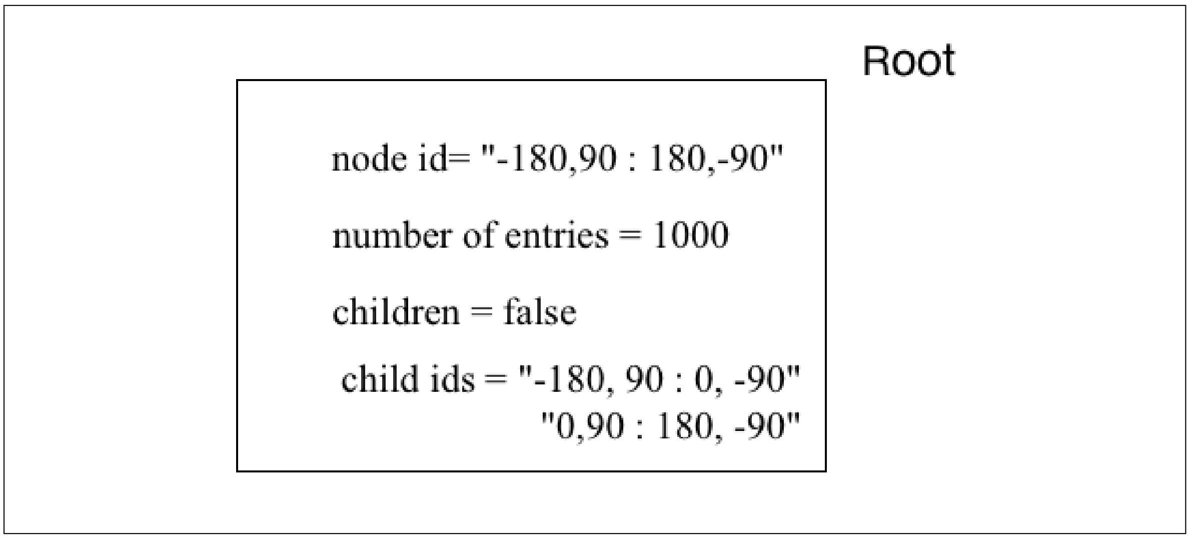


Figure 4.12: A node and its values

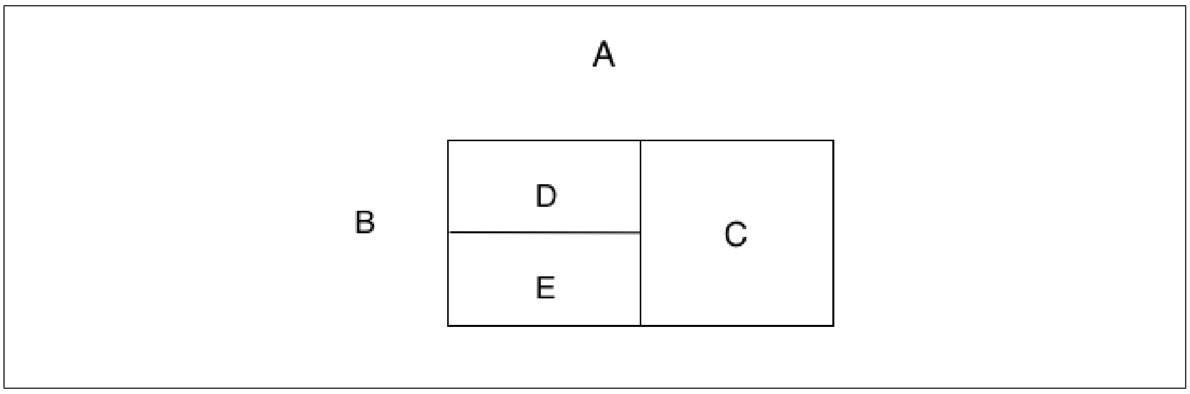


Figure 4.13: Grid represents a Kd-Tree

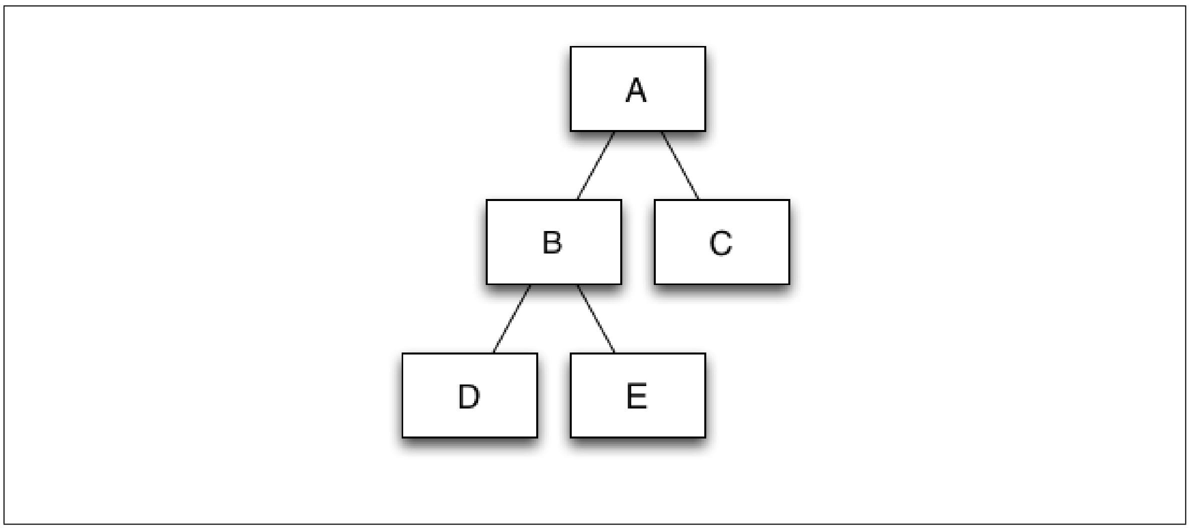


Figure 4.14: Kd-Tree

Figure 4.13 shows the grid of a Kd-Tree, node A is the original root node that has now been split twice, first into node A and B and then node B has been split into nodes D and E – this is because the node capacity was exceeded. The new nodes now own the data in that grid, not the parent. Figure 4.14 shows the Kd-Tree which is mapped from the grid shown in Figure 4.13.

4.4.5 Quad-Tree

The second structure implemented into the tree was the Quad-Tree [71]. This differs from the Kd-Tree as each grid is split into quadrants, resulting in the parent node having four child nodes. As shown in Figure 4.15 the root node A again contains all of the data until the capacity is exceeded. Once this happens the grid is split into quadrants, these are nodes B, C, D and E. In this figure node B is also split as it has exceeded the node capacity. This is then mapped to the tree shown in Figure 4.16.

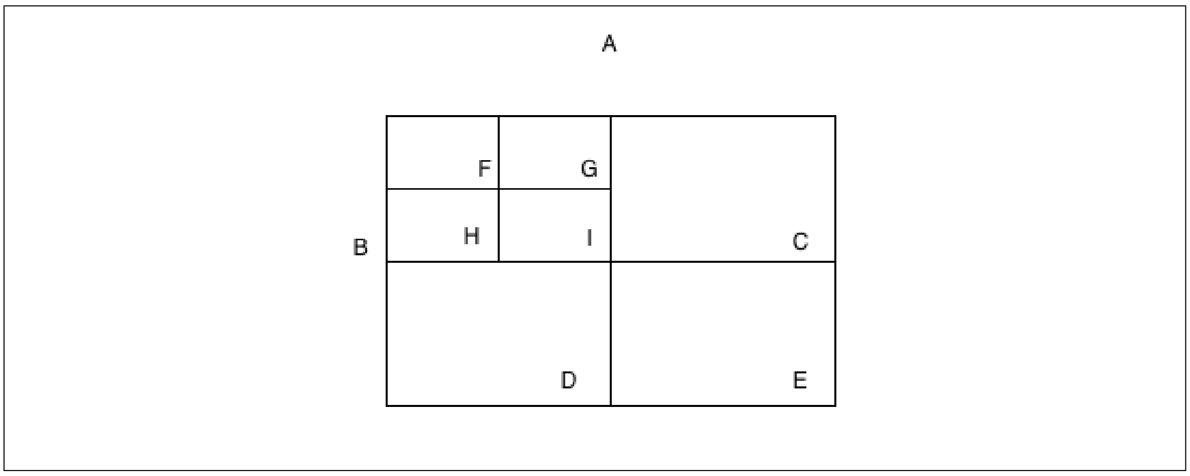


Figure 4.15: Grid representation of a Quad-Tree

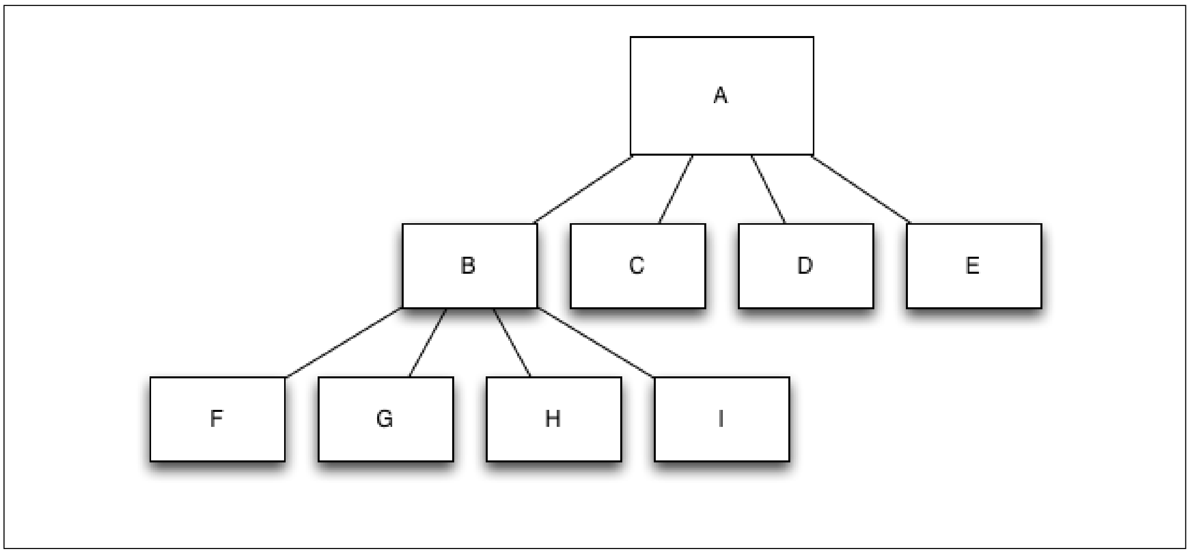


Figure 4.16: Quad-Tree

4.4.6 Geohashing

The third structure is a geohashing algorithm [71], which is essentially a tree structure where parents have 32 children. It uses base32 to encode the latitude and longitude taken from the input data. This involves recursively splitting the node into 32 grids, which represent each of the child nodes. This means there are very fine-grained grids and the tree's depth is much shorter than the others. The tree starts with the same root node as the other two, with a grid covering the whole world, the parent node is split into 32 children. This continues as the data is ingested and stops when splitting the grid would become irrelevant (i.e. when the size of the grid becomes smaller than the accuracy of the device, for example, GPS is accurate to 4 meters). The structure is usually used for proximity queries as the data is stored predominately by locality. Therefore if you execute a proximity search then data points are saved on the disk closer together, therefore there is no requirement for large scans and it reduces the query response time.

4.4.7 Schema

A schema was designed, shown below, and developed to store the geospatial data into Cassandra. It uses the traditional column family model, as Cassandra is unaware of the indexing layer. The row is made up of a rowkey, lat (latitude), lon (longitude) and value. The column name can also optionally include time, which adds a third dimension to the data. The composite column of latitude, longitude and optionally time provides a secondary index for querying, which is referred to as wide rows. The value is a JSON object of the data point content.

```
CREATE TABLE geo(
lat text,
lon text,
time text, --optional
value text,
PRIMARY KEY (lat, lon, (*time)))

*time is an optional attribute if it is not
mentioned then it is not a primary key
```

The schema was modified from the previous **tweet geo**. It was extended to include latitude and longitude to support more complex querying.

The rowkey is a special unique key which identifies the row (the equivalent to a grid in the ”world grid”, i.e. a tree node). The tree uses this rowkey as a pointer to the row in the database. The tree only holds this pointer not the actual data - that is stored in the database.

4.4.8 Query Mapping

The architecture maps queries from the tree structure in-memory to the Cassandra database without the database knowing anything about the indexing layer. The advantage of using this layered approach is searching for index keys can be done in memory and the database is only required for the retrieval of the data itself.

This section discusses the mechanisms used for this and demonstrates how the original user entered parameters are converted into a query to be executed across Cassandra.

The user chooses a viewport in the browser to display the points contained within that rectangle. The top left co-ordinates and bottom right co-ordinates are given to the indexing layer, these are then used to search the tree using the range algorithm described in the next section. These searches are executed in-memory over a tree structure held on a different “client-server”, which is not held on the Cassandra cluster.

Once the correct node or nodes are found which hold data contained in the user-specified grid then the keys from each of these nodes is used to compile a query to be executed over Cassandra. Single key querying is used as it improves the performance of Cassandra, which is optimised for single index queries. If multiple keys are found (i.e. the viewport crosses more than one region) then the additional queries are added to the query using the *IN* keyword. The *IN* keyword is used as it allows the queries to be executed as if they were a set of different queries and in parallel, therefore response time will be reduced. The keys are taken and entered into the following queries to be executed over Cassandra.

```
Q1(topLeft, bottomRight)
SELECT * from geo
WHERE key = topLeft+bottomRight

Q2(topLeft, bottomRight, starttime, endtime)
SELECT * from geo
WHERE key = topLeft+bottomRight and time > starttime and time < endtime
```

These are executed across Cassandra and then this is returned and displayed in the browser-rest used to enable this

4.4.9 Spatial Algorithms

Sets of novel algorithms were implemented for searching and inserting data into Cassandra. Data was collected from streams and inserted efficiently and quickly into the database using the novel cache and algorithms designed and developed for *Antares.*

There is a basic set of algorithms, which are used to search the data to provide efficient low latency response times so that a viewport can be rendered in a browser for a user to analyse. The algorithm's principal constraint was to return in a few milliseconds to enable quick and simple rendering of data points at the client.

4.4.9.1 Insertion Algorithm

The Insertion Algorithm, as shown in Algorithm 2, searches the tree to identify the correct node to insert the data into Cassandra. This is done by recursively traversing the tree for the node containing the minimum-bounding box. The data is inserted into the data store (Cassandra) using the co-ordinates from the minimum-bounding box as a row key.

Algorithm 2 Insert

```
int $capacity$ # limit of points in a node
geo $point$ # an object made of a latitude and a longitude
geo $region$ # the bounding box which represents a geographic area on a map
# a key is the identifier for a record in the database made of a top left coordinate
and bottom right coordinate
Object $node$ # contains all keys for that region
int $maxHeight$ this is the maximum height of the tree
int $height$ this is the height of the tree at runtime
int $numberOfNodes$ # this is the number of nodes contained in the tree-the struc-
ture the nodes are contained in
int $numberOfPoints$ # the amount of data that a node contains at runtime
int $capacity$ # the limit of the number of key entries per node
list $a_1 ... a_n$ # a list of predicates
predicate $p$ # this is the query predicate
database $h$ # this is the historic store where tweets have been stored
# tweet $x$ is Twitter data which contains a latitude, longitude
# and optionally time - the data is take from a stream of tweets $s$
input: tweet $x$
if numberOfNodes<1 then
  $node$ = new node
end if
if $x.point$ contained in $node$.region then
  if $node.numberOfPoints < capacity$ then
    INSERT INTO h VALUES (x)
    return boolean $true$
  else if $height < maxHeight$ then
    Split($node$, $data$)
  else
     INSERT INTO h VALUES (x)
  end if
else
  Insert(x)
end if
```

The next algorithm described is the Split Algorithm displayed in Algorithm 3. Algorithm 3 requires a node to split the data being entered and the number of new nodes to instantiate. This example uses the Kd-Tree structure, for simplicity. When a bucket exceeds the capacity, the node's bounding box is split into two new bounding boxes and two new nodes containing these. The grid is divided by alternating between lat-

Algorithm 3 Split

```
  # This algorithm splits a node into two nodes and points from the original node are
  inserted into the new nodes. The points are then removed form the original node
Require: node, data
  Object node1
  Object node2
  geo point # contains a latitude and longitude
  int count # keeps a count of each search through the tree to split the bounding
  boxes alternatively
  geo region # geographical area represented by a top left co-ordinate and a bottom
  right co-ordinate
  Object key # represents a row in the database and is the top left co-ordinate and
  bottom right co-ordinate
  if count % 2 == 0 then
    # latitude halved and two new regions created
    region.latitude/2
    node1.region = region1
    node2.region = region2
  else
    # longitude halved and two new regions created
    region.longitude/2
    node1.region = region1
    node2.region = region2
    if point contained in node1.region then
      database is updated to ensure:
      node.key = node1.key
      return true
    else
      database is updated to ensure:
      node.key = node2.key
      return true
    end if
  end if
  delete points from node
```

itude and longitude. The data must then be moved from the parent node to the two new child nodes.

Comparing the bounding box co-ordinates with the row keys of these child nodes allows the data to be inserted into the matching grid. After that, the data that was contained in the parent node is deleted. As this is done queries are still accepted. Each client node can access the data at any point or split the data at any point, which can cause consistency issues. How this is dealt with is described in Section 4.5.

4.4.9.2 Point and Range Algorithms

Two algorithms for searching the tree were also implemented – these can be seen in Algorithm 4, Point Query, used to find any data associated with a specific point (represented by a latitude and longitude) in the database. When it is executed it traverses the tree to identify the minimum-bounding box. Once this has been identified a query using the bounding box co-ordinates and the point's latitude and longitude is sent to Cassandra. Any node can accept the query then the correct node, which holds the data, is sent the query by checking the token.

Algorithm 5 is called Range Query and allows any four points to represent a region on a map: the query then returns all the points in this area. This is more complex than a point query as it requires querying multiple buckets to find the points from a user-defined grid, which may intersect many or few of the grids defined in the tree. The index is traversed to identify the path of the tree, which contains the minimum-bounding boxes that intersect the user-defined grid. Once identified the id of the nodes containing the points are used to query Cassandra. As described earlier the data points in the tree are pushed down to the leaf nodes. Once the path of the intersecting grids has been calculated, nodes which do not contain data can be pruned from the path. Therefore the root node and other inner nodes can be pruned as the data they contained has been pushed down the tree to the leaf nodes. When a query is decomposed in Cassandra the parallelism is dependent on the number of CPUs to execute each query parameter, therefore if there are less parameters then this exploits the database's parallelism and response times are decreased.

Algorithm 4 Point Query

```
geo point # contains a latitude and longitude
geo region # geographical area represented by a top left co-ordinate and a bottom right co-ordinate
Object node1
Object id each node id corresponds to a row id in the database
int capacity # the limit of the number of key entries per node
int numberOfPoints # the amount of data that a node contains at runtime
Object Tweet # this is a Tweet which contains a status, timestamp, co-ordinate and user information
geo tweet.point # this is a point (latitude and longitude) contained in a tweet
input: point
if point contained in node1.region AND node1.numberOfPoints <capacity then
   SELECT $Tweet$ FROM hs WHERE key=node.id and tweet.point = point
  return Tweet
else
  node1 +1 # move to next level of the tree
end if
```

Algorithm 5 Range Query

```
Object node1
geo region # geographical area represented by a top left co-ordinate and a bottom right co-ordinate
int capacity # the limit of the number of key entries per node
Object key # a key is the identifier for a record in the database made of a top left coordinate and bottom right coordinate
input: region
if region intersects node1.region AND node1.numberOfPoints <capacity then
   SELECT $Tweet$ FROM hs WHERE key=node.id
  return list Tweets
else
  node1 +1 # move to next level of the tree
end if
return Tweets
```

4.4.10 Extension to Querying Model

Antares uses a simplified querying model to support scalable and low latency querying of geospatial and temporal Twitter data. Extensions to the query model could support a wider range of querying in this section geospatial querying options implemented by the NoSQL database MongoDB are discussed and how the system would be extended to include these described.

A radius query or near by query could be implemented to support the function of finding nearest neighbours or nearby places. The current indexing layer and database would still be untilised for scalability and low latency. When a query was executed it would return the grids that intersect the radius. Once this is returned then an in-memory search would remove the points which were not contained within the radius. This can be calculated using this formula $(x_i - x)^2 + (y_i - y)^2$ if this is less than the $radius^2$ then it is within the circle, if it is equal it is on the edge of the circle and if it greater it is not in the circle. The performance would decrease but as the processing is handled in-memory which would speed the processing up. The database will also still provide low latency and scalable querying as the structure has not changed.

Another function would be to allow polygon querying, this would be implemented in the same way only the equation for calculating a circle would not be used. polygon shapes could also be introduced in the same way using the same indexing structure.

To add deletion or modification to the index would be complex and would reduce performance unnecessarily as the aim of the system itself was to provide scalable storage of Twitter data so that it could be replayed and processed. As *Antares* provides scalable and low latency querying there is no need to delete data and if it is modified it can be re-added and the timestamp would provide a means of deciding which is newer. The requirement was to allow all data within a given time bound to be processed therefore the data should not be removed or the aim is not met.

4.5 Consistency

As previously mentioned *Antares* uses a cache for storing keys (pointers to the geospatial data), querying them and constructing queries. This is distributed over clients to stop bottlenecks, which can lead to consistency issues. This section addresses each of these issues and the solutions that are implemented by *Antares*. The following scenarios cover all use cases when using this simplified querying model for reads and writes and because the system is append-only.

4.5.1 Scenarios

This section describes different consistency scenarios and the solutions implemented by *Antares*.

Read:

The first set of scenarios covers consistency challenges while retrieving data.

Scenario One:

Problem: Client 2 owns a leaf node (Node B in Figure 4.17) that has exceeded the limit; therefore the client splits the node and starts writing to the child nodes. However only Client 2 executes a split function, the rest of the clients do not. Then because Client 1's cache is stale, it writes to Node B. Then Client 2 reads from Node D. In this scenario the stale cache has resulted in new data being written to an old source, therefore not all data related to this query is read and becomes lost in nodes that are no longer queried.

Solution: When Client 2 reads from Node D, it also reads from Node B to ensure, if there is left over data, *Antares* reads all data that has been written to Cassandra. This ensures that even if other clients are stale this does not affect the system and data is read from old caches too.

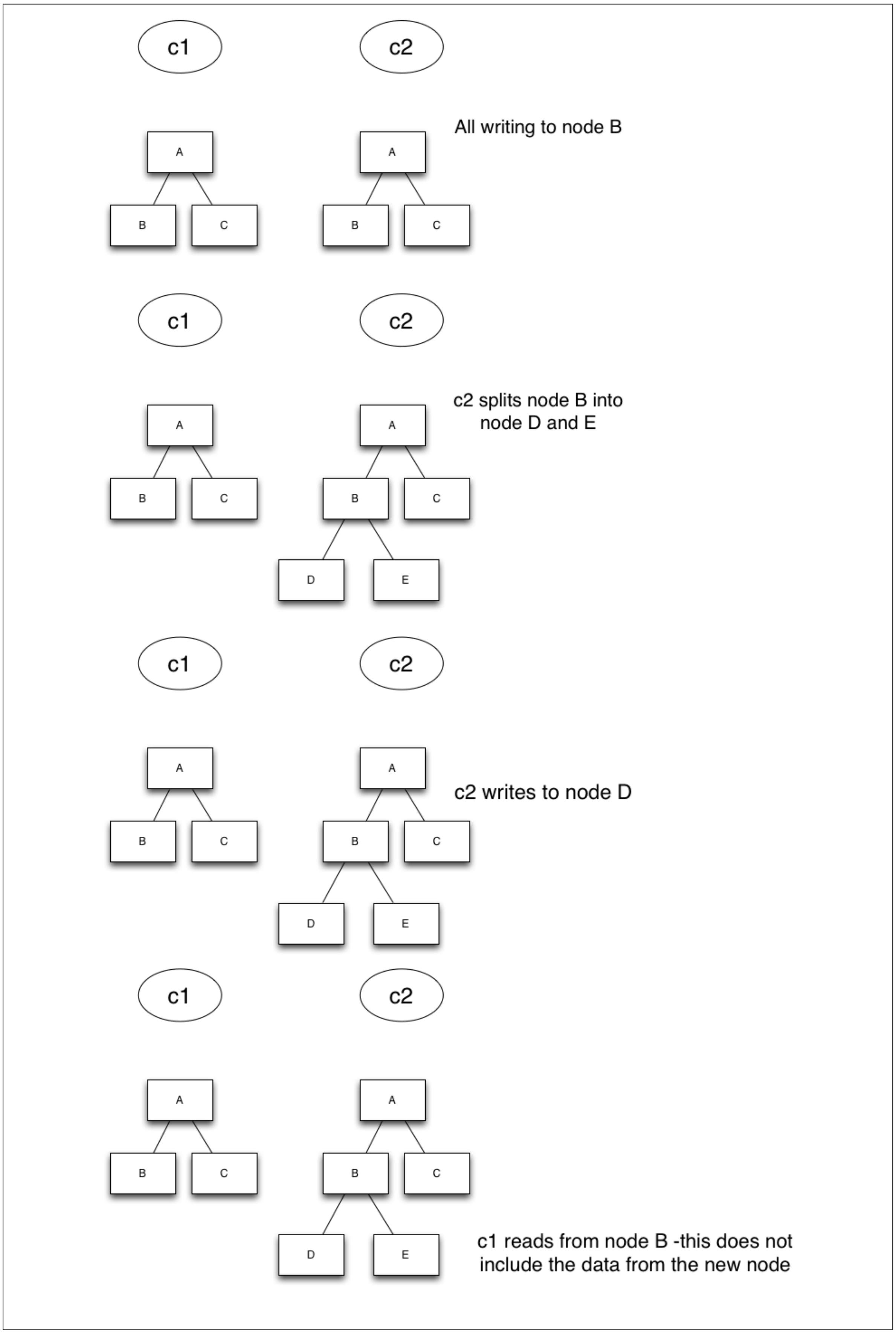

c1
c2
A
B
C
All writing to node B
c2 splits node B into node D and E
D
E
c2 writes to node D
c1 reads from node B -this does not include the data from the new node

Figure 4.17: Scenario One

Scenario Two:

Problem: Client 2 owns a leaf node (Node B shown in Figure 4.18) that has exceeded the limit; therefore the client splits the node and starts writing to the child nodes. However only Client 2 executes a split function, the rest of the clients do not. However this time Client 2 writes to Node D, then Client 1 reads from Node B. This is the same problem: the queries will not return all of the data, as some of the caches are stale. However this time the problem is with the downward path of the tree.

Solution: As Client 1 does not know that Node D exists it doesn't query the new node in Scenario Two. However if Client 1 also queries potential children nodes, then it would be able to return the data, but how does it do this when it doesn't know it exists? As *Antares* splits the grids at each level of the tree, the key for the node can be calculated at every level. The potential height of the tree is known from the user-defined configuration. Therefore, the child nodes' co-ordinates can be calculated by splitting the nodes using the Split Algorithm. This means a new node's id can be calculated and queried by the indexing layer without it having to know if it has been created yet.

Antares also knows the potential height of the tree and therefore the amount of potential leaf nodes. The potential new node id is worked out and added to the query, if it returns data then the client knows its cache is stale. Therefore a split is performed on the node, then another query is sent out to check if the newly split node is a child, if it returns true, do nothing more, if it returns false then another split is executed. As shown in Figure 4.18 Client 1 would need to split Node B to become consistent. This continues until the client is no longer stale, the trade off for this consistency is the extra queries to the database. However all the data will be returned whether the cache is stale or not. The system also has no global locking therefore it does not block while this is being executed. Only one client will block and all other clients will still accept data and queries.

Write:

The next scenarios describe consistency problems while writing data.

Scenario Three:

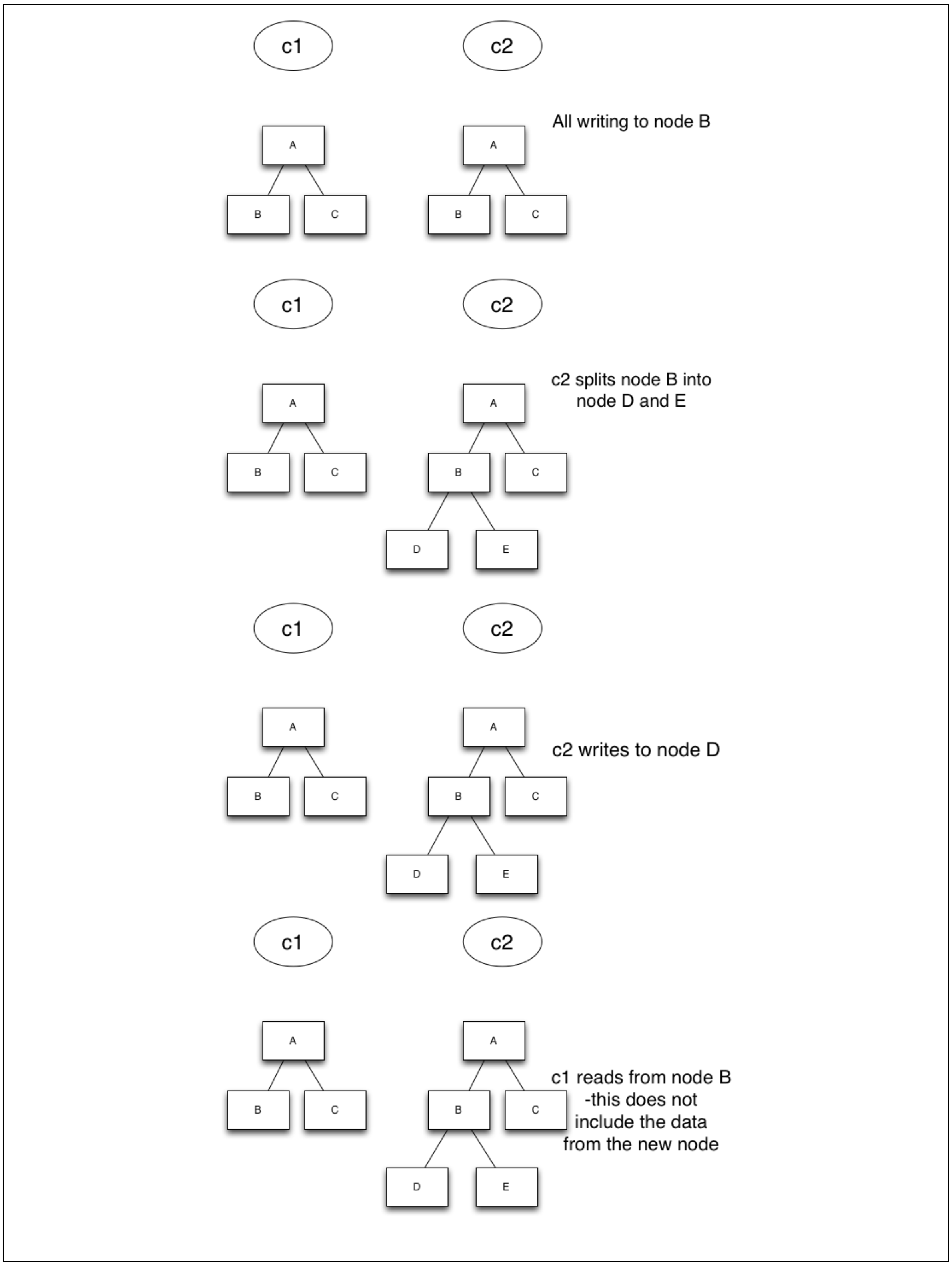

c1
c2
A
B
C
All writing to node B
c2 splits node B into node D and E
D
E
c2 writes to node D
c1 reads from node B -this does not include the data from the new node

Figure 4.18: Scenario Two

Problem: As points are written to *Antares* the index can become inconsistent because each client is a standalone machine and responsible for its own data. If the client has a node which has exceeded its limit then the node is split. This however does not mean that the node is split on any other client, as demonstrated in Figure 4.19. If the client had a heavy write load and no reads came through then there would never be a check to see if the cache was stale or not and the clients could all be out of sync by varying levels in the tree.

Solution: *Antares* employs a time-out in which the cache may be stale, once this is exceeded a query is sent to the database to ensure the current leaf node is a child node, if this is true then the node is split. Every node executes this. This ensures that all stale caches will become eventually consistent and the scenario described cannot happen.

In Scenario Three if Client 2 had timed out then the node id of Nodes B and C would be queried to identify if the database had a true boolean value for “is a leaf node”. In this scenario the database would return false for Node B –as Clients 1 and 3 have split– and true for Node C. Therefore Client 2 would execute the Split Algorithm on Node B pushing all of the data to the new Nodes D and E and returning Client 2 to a consistent state. Queries to Client 2 will be blocked as the split is executed but can be accepted by other clients. The client will also continue to ask the database “am I a leaf node?” until that branch returns true.

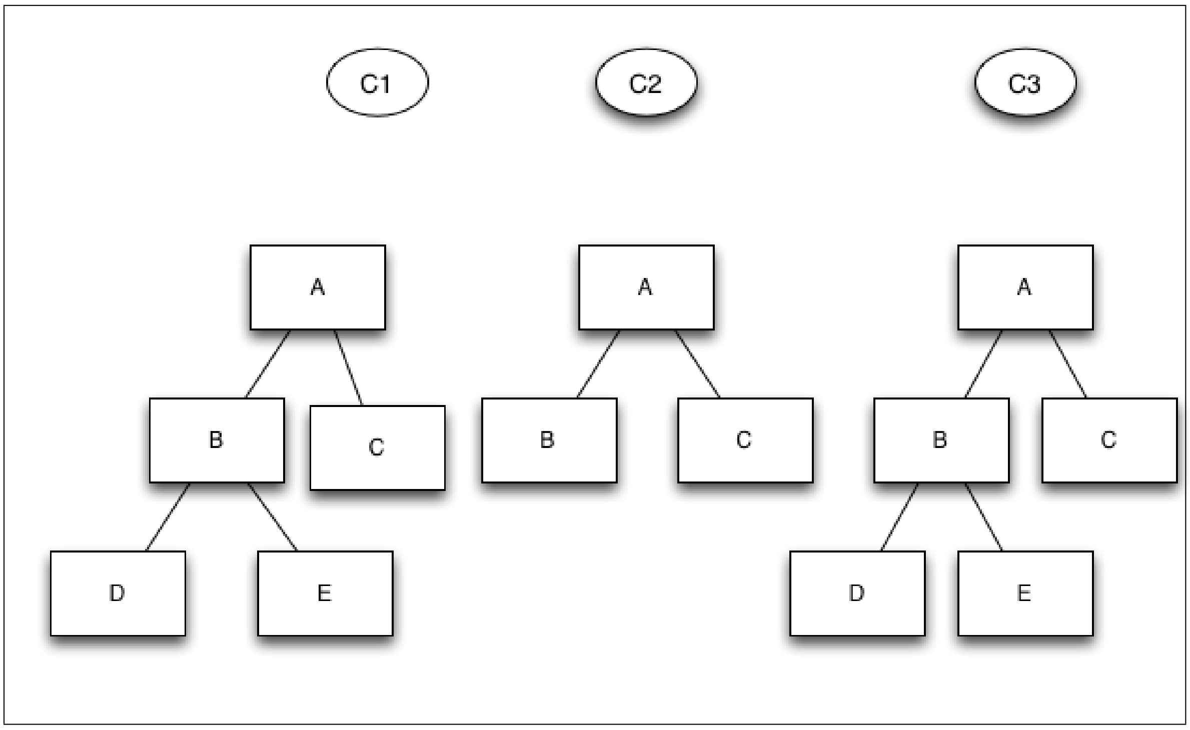


Figure 4.19: Scenario Three

Scenario Four:

Problem: If Client 2 has written to Node C during the timeout then there is data written to the wrong node up the branch. Therefore that data is written in the wrong tree node and is now lost to the user as the tree grows.

Solution: To solve the problem described above when the time-out is exceeded all data is pushed down the tree to the leaf nodes, ensuring the trees are consistent across the client nodes. Client 2 realises it is inconsistent by querying the database, it then splits Node C into Node D and Node E, and all extra data is pushed down the tree. If the limit is exceeded then the tree is split again into Node F and G. This is done recursively until the tree has reached the child nodes. Each of these adds a round trip to the database but at the price of the consistency as each time the leaf is split there is another query to the database to check that it is consistent.

A query is executed that selects all data points with Node D's id and then these are split into Node E, the number of data points in Node E is then checked and if it exceeds the limit the node is split into two new nodes, this is executed recursively until all nodes have a number of data points under the configured node capacity, as shown in Figure 4.20.

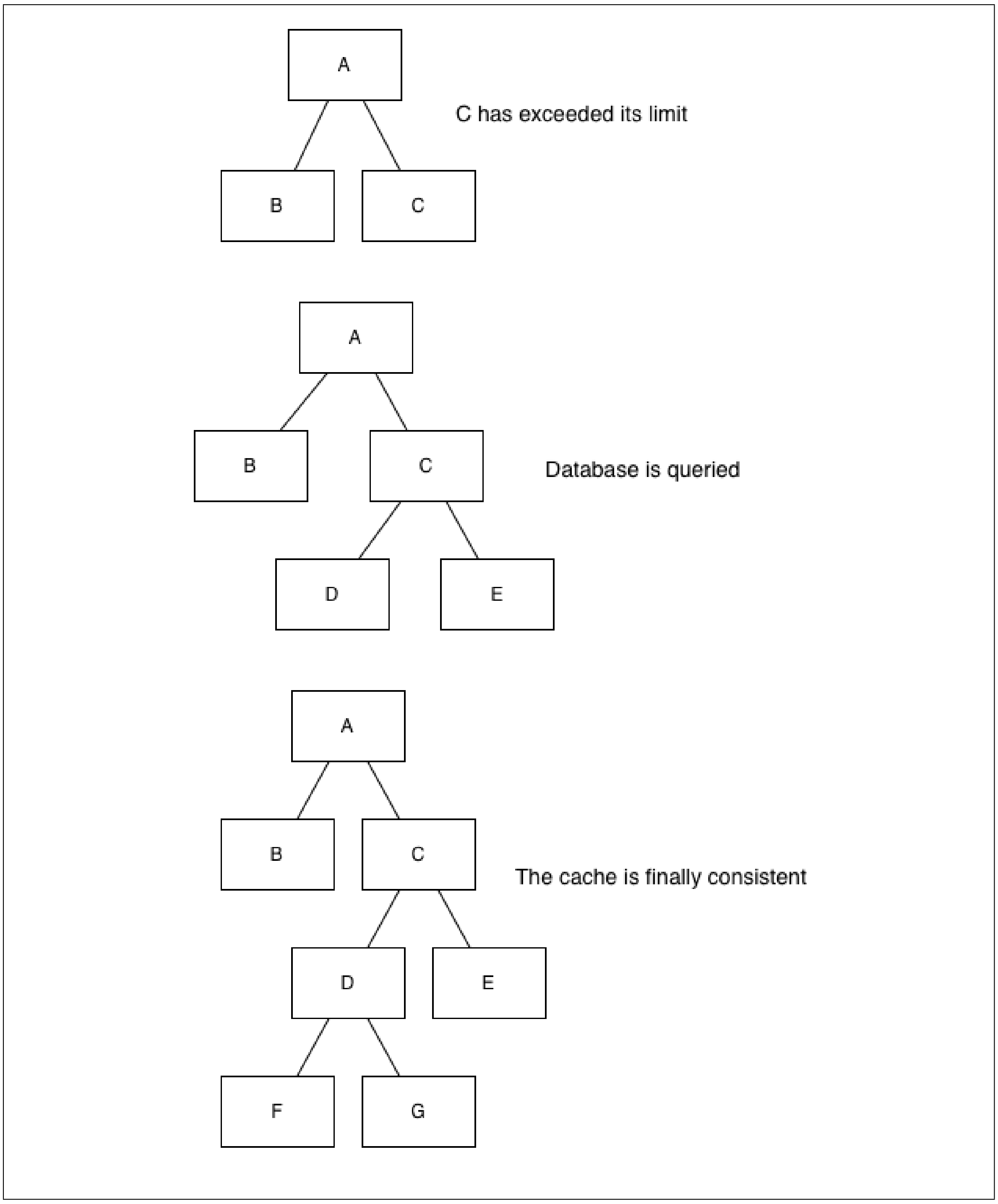


Figure 4.20: Scenario Four

4.6 Evaluation

The next section of this chapter evaluates the new geospatial querying mechanism designed and deployed in *Antares*. The motivation was to support the geospatial querying research being undertaken by the social scientists. This led to a novel geospatial querying mechanism designed to exploit the scalability of NoSQL technology. The next experiments demonstrate the scalability of *Antares* and how efficient the indexing is.

4.6.1 Read Performance

This experiment was designed to show the efficiency and scalability of the new mechanisms to support searching for geospatial data to display within a browser.

Managing the markers in a viewport was troublesome and caused many problems. The original solution would return response times that were so long the request would time-out. This was addressed by the design of a new tree structure and algorithms described in Section 4.4.

To evaluate the new geospatial mechanisms, increasingly large geographic areas were queried. The area was increased to replicate the zoom out functionality required for the user to analyse the data. The first experiment was executed on one machine to prove response times for *Antares* were quick enough to render the map without a browser time-out. The second set of experiments evaluated the scalability of *Antares* by increasing the size of the cluster.

The first experiment shown in Figure 4.21 used a one-node cluster of the previously defined Amazon cloud machines. Twitter data was written to the database so the queries were executed across a pre-populated database to evaluate the search capabilities of the cache and algorithms. For this experiment, the client was not distributed as distributing the client has little affect on reads. However multiple levels of the tree were still queried as if they had been distributed (as described in more detail in the previous Section 4.4.9).

As demonstrated in Table 4.1 all of the results are returned in under 27 milliseconds for a Kd-Tree implementation. This has dramatically improved response times and has provided fast enough querying to visualise the data for the user to analyse when compared with the Solr implementation for geospatial querying with Cassandra. This is demonstrated in Figure 4.21, however as the difference in mean response times is so large Figure 4.22 demonstrates mean response times for between 100-1,000 km^2.

–percent

The experiment ranged from 100 km^2 to 2,000 km^2 – the area was not extended after this as Solr could not query a larger area without timing out. This demonstrates the capability of *Antares* to support large area searches while zooming in, out and panning. The optimisations reduce the response time by a factor of 10 for smaller areas and approximately a factor of 5,000 for larger areas. This is a vast increase in performance, this was achieved by simplifying the query model. *Antares* is required to store all data so it may be queried at a later date, therefore it is an append only system. This allows *Antares* to out-perform other systems such as Solr as the querying model is simpler, which supports more efficient writes. This is how such a large gain in performance is achieved.

km^2	Solr	*Antares*: Kd-Tree	Percentage Saving (%)
100	69.8	6.9	90.11
250	102.44	15.16	85.2
500	259.96	17.5	93.26
1000	570.32	31.8	94.42
1500	78729.36	23.02	99.97
2000	130813.12	26.48	99.98

Table 4.1: Mean response time (ms) for Solr compared with *Antares* for increasingly large areas (km^2)

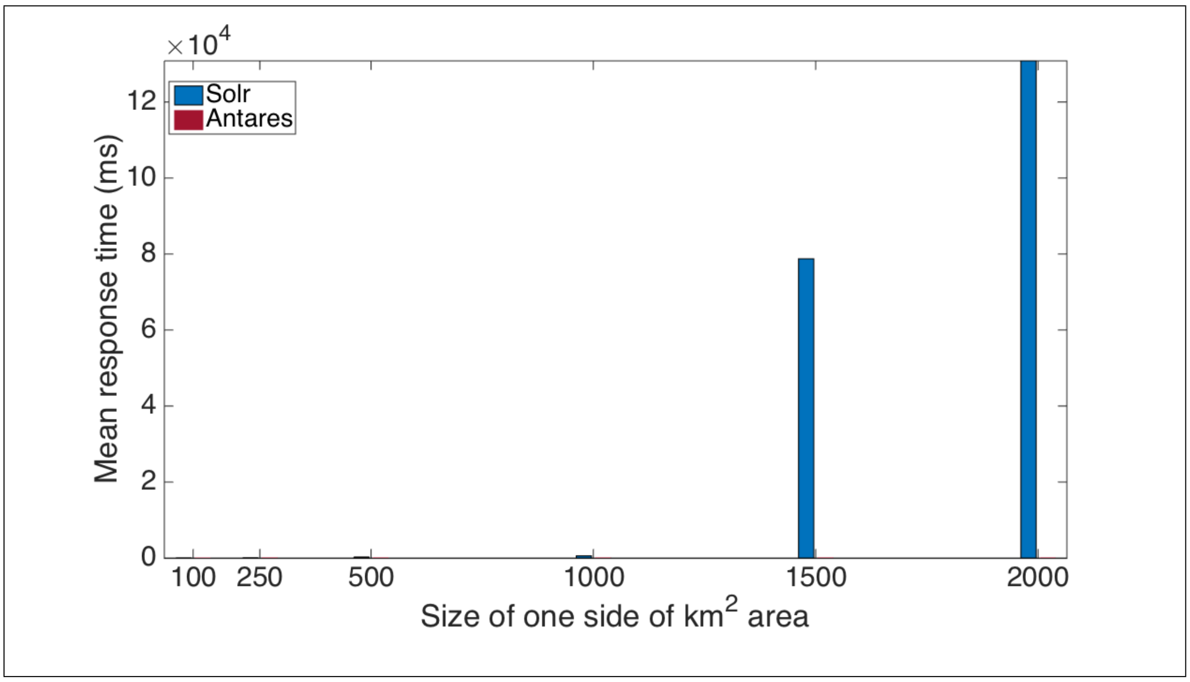


Figure 4.21: Mean response times for geospatial querying (up to 2000 km^2)

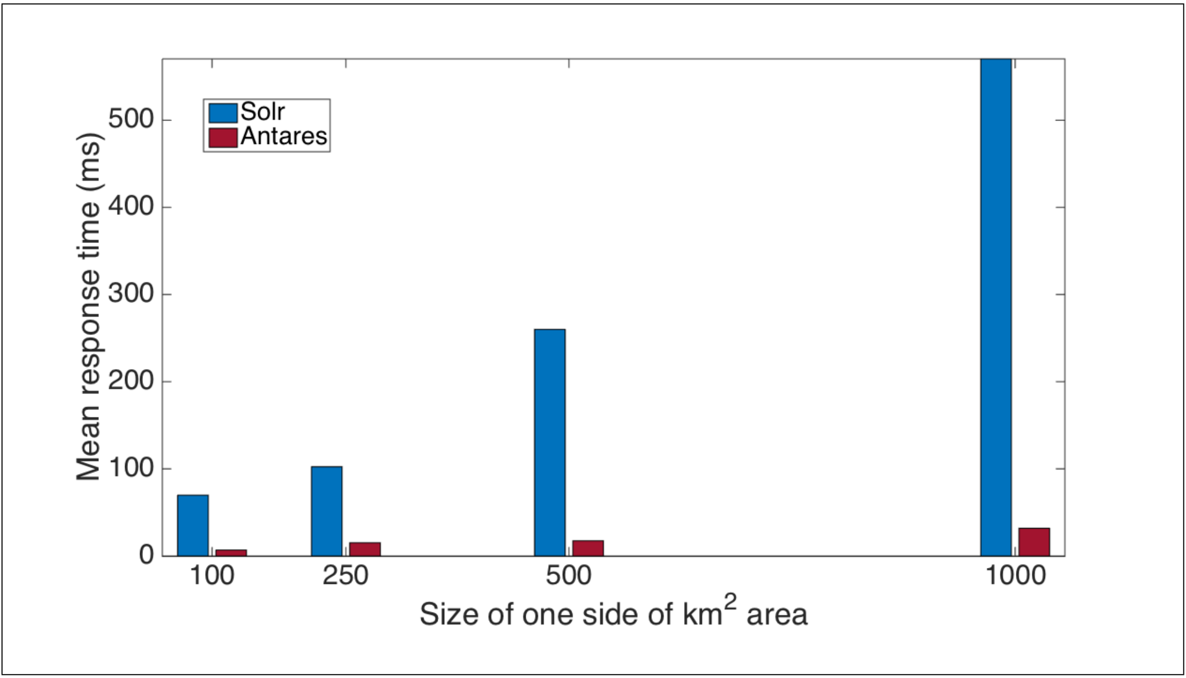


Figure 4.22: Mean response times for for geospatial quering (up to 1000 km^2)

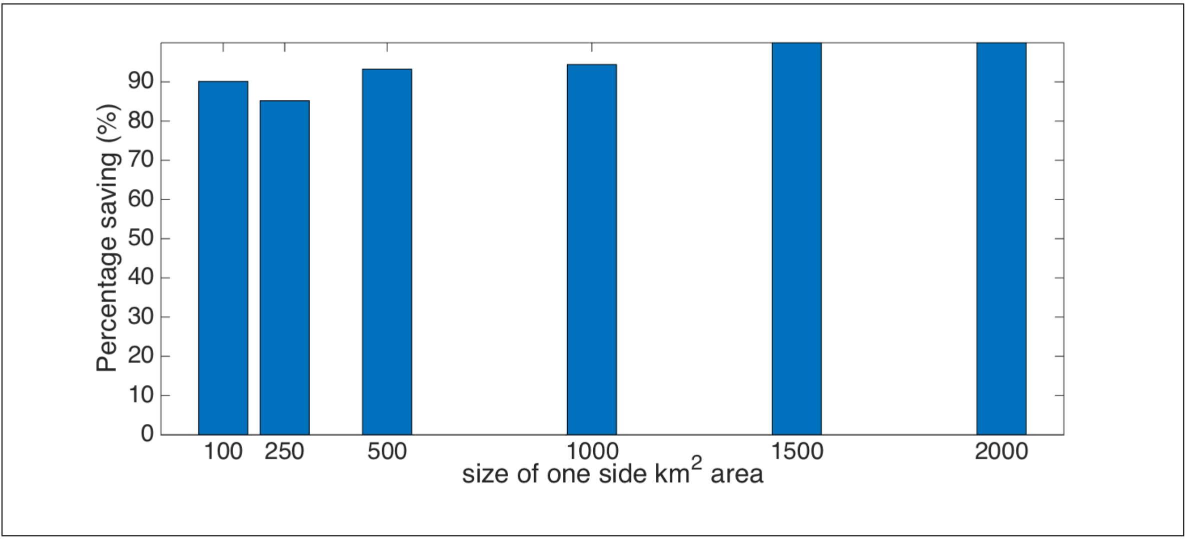


Figure 4.23: Percentage saving *Antares*

The second experiment was executed across the Amazon cluster, and after each experiment a node was added to the cluster. The cluster was already populated and the experiments were used to evaluate searches across *Antares*. As the response times were so small the scale tests were executed using 1,000 km^2 areas. For each experiment an additional node was added to the cluster after the other was finished. The scale tests were executed across *Antares* clusters which implemented the Kd-Tree, Quad-Tree and Geohashing. The experiments in Figure 4.24 show the increase in nodes provides scalable and quicker response times for querying. The percentage saving in Figure 4.23 shows an improvement of over 85% for each cluster, which demonstrates the huge improvements made by *Antares*.

number of machines	*Antares*: Kd-Tree	*Antares*: Quad-Tree	*Antares*: Geohashing
1	31.80	3.56	2.00
2	8.55	1.21	2.13
3	4.61	1.50	1.54
4	8.55	1.16	1.40

Table 4.2: Mean response time (ms) for *Antares* as the number of nodes increases

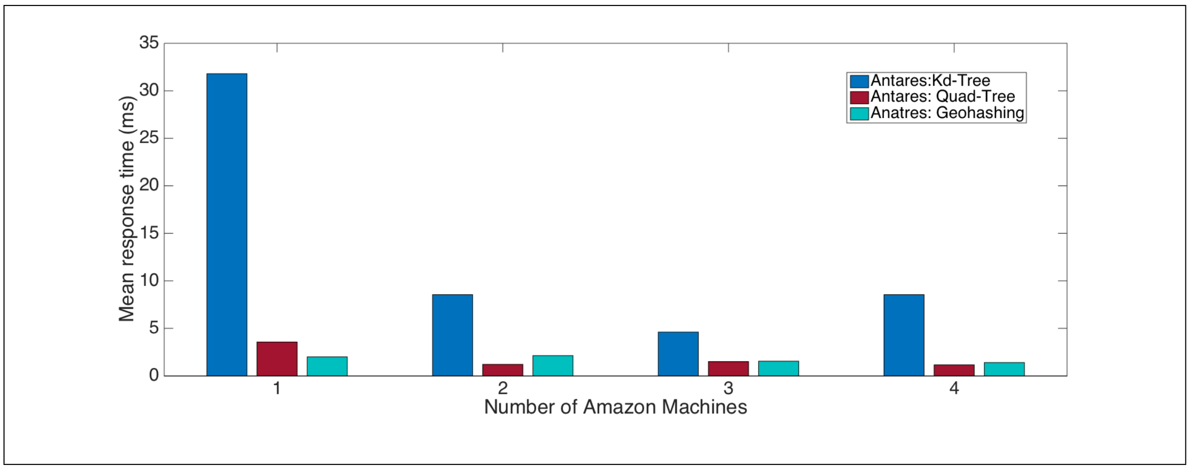


Figure 4.24: Mean response times for map queries as the number of nodes increases

4.6.2 Scale Experiments Comparing Different Structures in Antares with Different Systems

In this section *Antares* was deployed on the cloud and its performance evaluated by comparing it with current NoSQL spatial querying frameworks, Solr and MD-Hbase. Solr uses Lucene on top of Cassandra to index spatial data. MD-Hbase is an extension of Hbase. A spatial dataset of 10 million geotagged Tweets was inserted into these NoSQL databases in preparation for experiments that compare the read performance.

Solr uses a Prefix Tree to index the spatial data. This tree employs geohashing to search the tree using prefixes. It is an ordered dynamic tree structure, which uses keys of type string. Described in greater detail in Section 4.4.1 As nodes are added to the cluster a tree is created for each one. The structure is then responsible for maintaining data on that node only. The queries were also executed across an MD-Hbase cluster (as previously mentioned in Chapter 3). This spatial querying system uses an in-memory structure also, but uses a z ordered curve for querying the data.

The specification of the machines used for this experiment are as follows: 15 Gb ram, 4 cores, 4 * 420 GB of storage, 64 bit and 1000 Mbps I/O performance. Each of the nodes in the cluster was pre-populated before the experiments were executed across the cluster.

The experiments compared the performance of the *Antares* system with Solr and MD-Hbase. The experiments were executed over an increasing number of machines. The query area is also increased and the response time measured.

The first experiment deploys a single node cluster for Solr, Antares and MD-Hbase. For each of the experiments the Tweets were replayed and written to the database, so that each node was pre-populated with spatial data. Each query was then executed specifying a geographical region. The region was increased in size for each execution and the response time recorded. The system executed the queries across a Kd-Tree, Quad Tree and Geohashing. The full set of results is demonstrated in Table 4.3. The table contains n/a for some of the MD-HBase values; this is because the queries only accepted integer values not doubles. A degree is approximately equal to 111 km and MD-HBase uses co-ordinates to specify the region being queried. Therefore the experiments could only be executed in approximately 100 km increases.

km^2	MD-Hbase	*Antares*: Kd-tree	*Antares*: Quad Tree	*Antares*: Geohashing	Solr
50	n/a	6.96	5.05	3.08	35.28
100	103.95	6.36	2.3	3.12	34.76
150	n/a	6.95	2.30	3.375	38.72
200	99.5	6.17	2.44	2.4	42.04
250	n/a	15.16	2.6	4	52.36
300	n/a	12.18	1.72	3.72	61.96
350	125.1	14.52	1.52	1.71	72
400	n/a	13.13	2.84	3.48	62.88
450	223.4	14.32	6.14	3.04	99.24
500	n/a	17.6	1.52	2.72	80.48
550	365.55	18.05	1.52	5.5	139.76
600	n/a	13.67	3.55	2.5	130.32
650	448.9	14.57	1.83	2.04	155.04
700	n/a	14.16	1.52	1.68	187.96
750	n/a	12.55	2	1.72	224.64
800	417.8	13.045	2.32	2	251.56
850	n/a	19.25	3.28	1.6	304.6
900	463.05	19	4.08	1.44	333.68
950	n/a	19.88	5.4	2.92	368.52
1000	413.2	31.8	3.56	2	458.6

Table 4.3: Mean response times (ms) for each framework across a one-node cluster

As shown in Figure 4.25, a one-node cluster employing the Kd-Tree decreases the response time for a small region query by a factor of 6 when compared with Solr. The response time is decreased by a factor of 14 for a query region of 1,000 km^2. The performance of *Antares* improves by over 50% for a larger region, because Solr searches additional buckets when compared with *Antares*. When searching the bucket for a user-defined set of points the Lucene filter must then compare each geohash to the required geographical area to identify whether the point is contained within it. *Antares* uses the search algorithm to prune the tree path (number of buckets queried) so fewer buckets are queried in the data store, which results in fewer disk seeks and quicker response times. As the nodes in Solr's Prefix Tree are filled the hash becomes longer and Lucene has to do more textual comparisons for each point it is trying to identify.

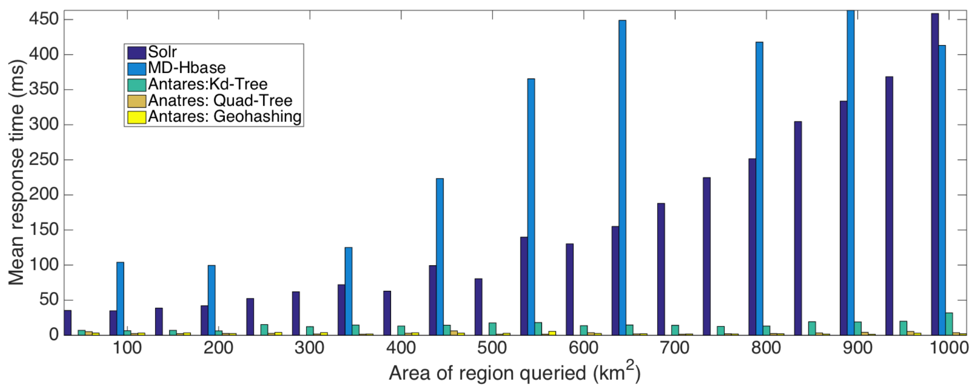


Figure 4.25: Mean response time (ms) for geospatial queries executed across different systems on a one-node cluster

As can be seen in Figure 4.25, MD-Hbase's response time is over a factor of 10 greater than *Antares* when using the least efficient structure (Kd-Tree) on one node. MD-Hbase's response time was approximately as long as Solr's for large regions, but increased by a third for smaller regions on the one-node cluster. This is because the indexing layer for MD-Hbase has to first be queried to return a starting point for the query. It then uses this key to execute the query across the database and return the results. This additional disk seek costs time and can be seen to increase response time in the results.

The next experiment deployed a two-node cluster for each of the frameworks as shown in Figure 4.26. Additional nodes decreased the response time for both *Antares* and MD-Hbase, however it increased Solr's response time. *Antares* improved response times by a factor of approximately 300 when querying a large region and a factor of approximately 80 when querying a smaller region compared with Solr, as is demonstrated in Table 4.4. Solr's response time increases by over double for a small region and over 100ms when compared with the one-node cluster.

km^2	MD-Hbase	*Antares*: Kd-tree	*Antares*: Quad Tree	*Antares*: Geohashing	Solr
50	n/a	5.63	1.13	1.24	84.2
100	108	4.25	1.54	1.5	67.08
150	n/a	4.22	1.04	1.58	62.52
200	136	4.88	1.29	1.20	66.76
250	n/a	3.48	1.76	1.08	69.36
300	n/a	4.22	1.17	1.48	69.72
350	111	3.65	1.56	1.5	73
400	n/a	3.42	1.28	1.38	63.52
450	231	3.43	3.21	1.13	70.44
500	n/a	4.91	1.32	1.16	173.08
550	240	3.08	1.24	1.42	282.44
600	n/a	3.48	1.5	1.29	239.84
650	243	5	1.64	1.38	255.04
700	n/a	3.17	1.4	1.42	252.68
750	n/a	3.54	1	1.32	239.68
800	389	3.38	0.96	1.16	491.2
850	n/a	5.09	1.42	1.36	519.84
900	391	5	1.08	1.38	585.80
950	n/a	5.13	0.96	1.28	566.68
1000	391	8.55	1.21	2.13	608.68

Table 4.4: Mean response times (ms) for each framework across a two-node cluster

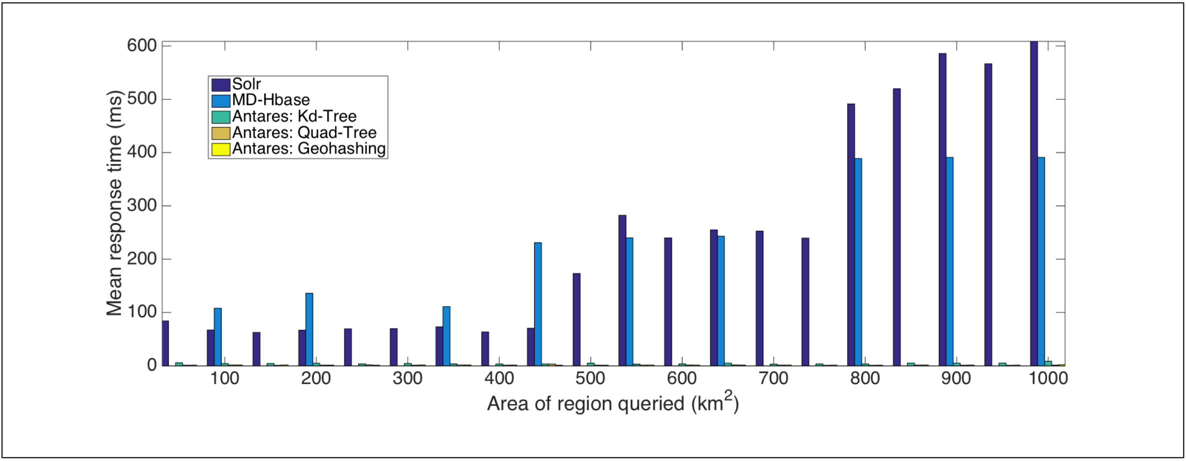


Figure 4.26: Mean response time (ms) for geospatial queries executed across different systems on a two-node cluster

As is shown in Figure 4.27 and Figure 4.28, adding a third and fourth node keeps the response times of *Antares* at between 3-5 ms but only increased the response time of Solr. This is due to the distribution of the indexing structure over the servers – it has to query each node separately. The system described here takes advantage of the distribution and the extra CPU power, executing the searches in parallel for each bucket and returning the values. For a three-node cluster this system is a factor of 9,280 faster than Solr and for four nodes it is a factor of 15,774 faster.

km^2	MD-Hbase	*Antares*: Kd-tree	*Antares*: Quad Tree	*Antares*: Geohashing	Solr
50	n/a	5.79	1.92	1.92	5569.08
100	68	3.88	1.46	1.75	5016.08
150	n/a	3.70	1.42	1.58	5027.92
200	49	4.38	1.83	1.79	5475.68
250	n/a	4.46	1.96	1.8	6002.80
300	n/a	4.21	1.5	1.68	6032.64
350	52	3.54	1.46	1.63	5962.56
400	n/a	4	1.46	1.79	5949.88
450	201	3.3	1.29	1.54	7369.88
500	n/a	3.67	2.29	1.68	12346.24
550	188	4.39	1.88	1.58	24758.80
600	n/a	3.83	1.63	1.64	22175.52
650	219	3.39	1.58	1.54	22567.80
700	n/a	3.58	1.63	1.71	23424.92
750	n/a	4.00	2.79	1.88	24738.80
800	254	3.74	1.29	1.8	46077.92
850	n/a	5.63	1.33	1.6	56477.00
900	282	4.46	1.5	1.42	61770.96
950	n/a	5.57	1.46	1.50	64109.56
1000	243	4.61	1.5	1.54	78871.88

Table 4.5: Mean response times (ms) for each framework across a three-node cluster

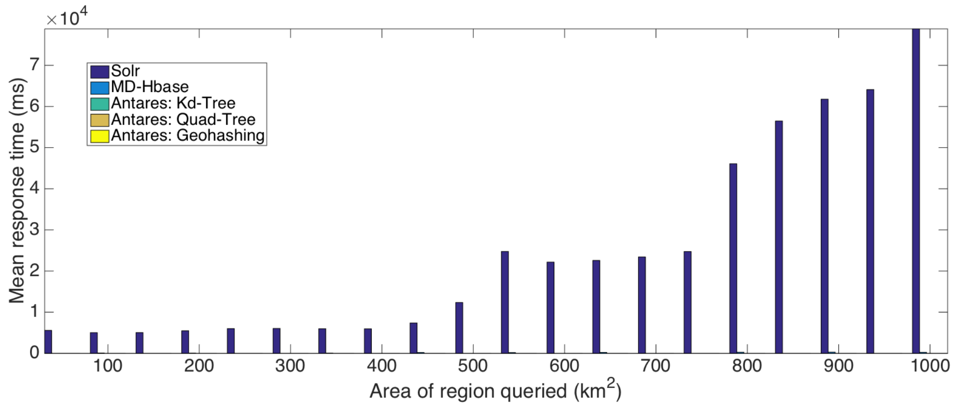


Figure 4.27: Mean response time (ms) for geospatial queries executed across different systems on a three-node cluster

km^2	MD-Hbase	*Antares*: Kd-tree	*Antares*: Quad Tree	*Antares*: Geohashing	Solr
50	n/a	4.54	1.54	1.46	5569.08
100	25	5.047	1.58	1.4	5016.08
150	n/a	3.25	1.5	1.36	5027.92
200	68	4.52	1.33	1.6	5475.68
250	n/a	3.5	1.24	1.72	6002.8
300	n/a	4.71	1.33	1.79	6032.64
350	61	6.38	1.38	1.63	5962.56
400	n/a	3.75	1.32	1.75	5949.88
450	84	3.63	1.33	1.33	7369.88
500	n/a	5.25	1.58	1.64	12346.24
550	138	6.13	1.32	1.46	24758.8
600	n/a	3.79	1.29	1.80	22175.52
650	256	5	1.46	1.56	22567.8
700	n/a	3.17	1.52	1.58	23424.92
750	n/a	3.54	1.58	1.44	24738.8
800	246	3.38	1.40	1.64	46077.92
850	n/a	5.09	1.32	1.63	56477
900	255	5	1.29	1.72	61770.96
950	n/a	5.13	1.48	1.6	64109.56
1000	275	8.55	1.16	1.4	78871.88

Table 4.6: Mean response times (ms) for each framework across a four-node cluster

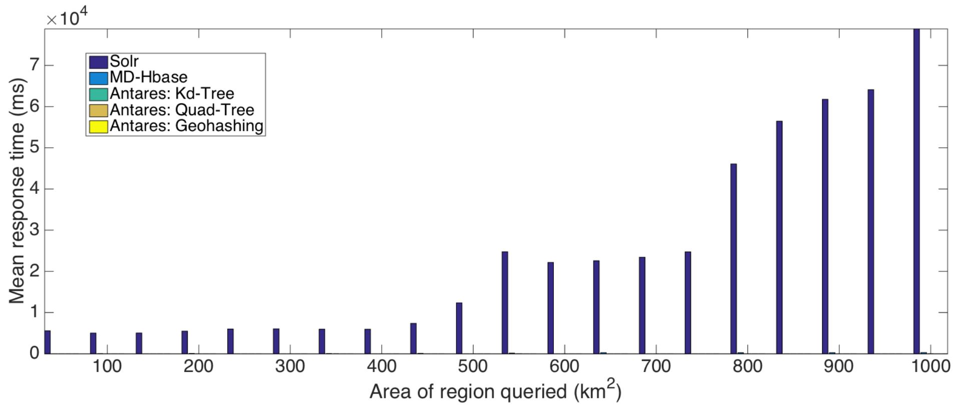


Figure 4.28: Mean response time (ms) for geospatial queries executed across different systems on a four-node cluster

The four-node Hbase cluster reduces the difference between the Kd-Tree and MD-Hbase to just over a factor of 6 for small region searches. The Quad-Tree improved the response time of *Antares* when compared with the Kd-Tree chiefly for larger regions: this is because the structure allows more fine-grained querying, therefore the path will be shorter and fewer buckets are queried. *Antares* queries fewer buckets by employing pruning techniques in the search algorithm, which reduces the response time. This is true for the Geohashing as well, which reduced response time down to little over a millisecond. For each of the frameworks the response time decreases as additional hardware is added to the cluster, except for Solr as previously mentioned. When compared with Solr the performance of MD-Hbase is similar if not slightly worse for smaller regions, when executed on a one-node cluster. However once hardware is added and MD-Hbase takes advantage of the parallelism that is available with more CPU power then the results diverge. Solr performs increasingly poorly as nodes are added and is approximately a factor of 290 slower than MD-Hbase, the full set of results is demonstrated in Tables 4.5 and 4.6.

4.6.3 Writes

The geospatial data also needs to be written to the database. In this next section the efficiency and scalability of this is evaluated. The cluster was as previously described, and contained four nodes. It was expected that when the number of clients was increased, the performance would increase. Therefore additional clients were added to the system to evaluate if and by how much the performance increased. This is achieved by adding a new node to generate and send queries, which also stores a local instance of the tree. By adding clients it is expected that performance will increase as the rate of Tweets being streamed to Cassandra can be increased. Therefore adding more clients should remove the bottleneck.

As can be seen in Figure 4.29 the number of writes increases linearly. Therefore adding clients does scale up the number of writes and prevents a bottleneck. Additionally the consistency checks mentioned in Section 4.5 do not affect the performance. This is due to an increase in resources, which increases the data flow to the server. *Antares* takes advantage of the increased data flow by using all available resources, therefore increasing efficiency and allowing the addition of hardware to increase ingestion linearly. The consistency checks do not affect the performance as when the client stops taking requests to become consistent, the other client nodes can keep writing to and reading from the database.

number of machines	Writes per second
1	7339.704969
2	13409.23856
3	19628.13933
4	26342.15256
5	33443.591

Table 4.7: Mean response time (ms) for *Antares* as the number of nodes increases

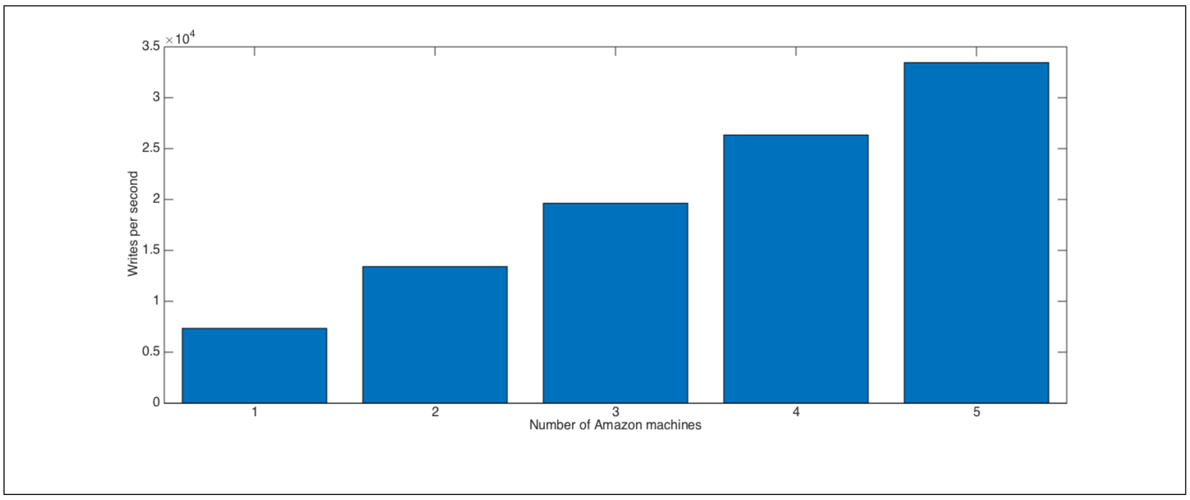


Figure 4.29: Number of inserts per second as the number of nodes is increased

4.6.4 Conclusion

The geospatial extension that *Antares* uses provides a scalable and near real-time mechanism for querying spatial data. The simplified model allows for the queries to return with reduced response times when compared with other NoSQL geospatial systems. *Antares* meets the requirement of near real-time view port querying and has been used in the Tweet My Street project. This however, is only possible because of the append only style of the system and the requirement that it be scalable to allow for data capture to be continual with no time to live limitations. The chapter has also discussed ways in which this could be modified to provide further functionality in the future.

4.7 Conclusion

This chapter has described the spatial extensions of *Antares*. These extensions support the scalable and low latency analysis of spatial and temporal data. The querying model has been simplified to support large scale data viewing and processing. This was the requirement of *Antares* and only possible due to its append only behaviour. The system does make a trade off by providing a simpler querying model however suggestions of how this can be improved were given earlier in the chapter. The scale of the data displayed and processed is the advantage *Antares* gains over other systems which use noSQL for geospatial querying. *Antares* also supported a large reduction in query execution time providing low latency analysis of the data which supports the real-time querying requirement of the system. *Anatres* additionally accounts for consistency problems and uses mechanisms to ensure that the system is eventually consistent. This chapter has demonstrated the scale and low latency querying of the *Antares* cache for geospatial querying which is the third contribution described in this thesis.

5
Application to Twitter

Contents

5.1 Introduction

A layered approach to analysing temporal and spatial Twitter data using a system called *Antares* has been described in earlier chapters. *Antares* uses streaming, historic and combined querying mechanisms to analyse Twitter data. It provides scalable ingestion rates of the Twitter firehose and provides low latency querying for temporal and spatial data extending on current streaming and historic processing technologies. This layered approach supports the scale and low query responses times required to handle high volume and velocity stream data. *Antares* has been used within a cross-disciplinary project called Tweet My Street to analyse Twitter data collected around different events. This chapter discusses the different technologies and mechanisms used to process social media data and compares these approaches with *Antares.* The user interface which displays the Twitter data for user analysis is then described and the projects that have used *Antares* explained to demonstrate how the querying mechanisms were derived and used in the real world.

5.2 Social Media Analysis

Social media generates vast amounts of data per second and these datasets can be used to derive useful insights. This section investigates different uses for social media and then focuses on Twitter analytics. *Antares* uses Twitter analytics to evaluate the system and for research projects, therefore establishing current technologies and research supports identifying improvements in this area.

YouTube is popular for its video content. [85] looks at how content affects the popularity of the video. It can also be used to support the improvement of health issues. Facebook is being used to help arthritis sufferers, by trying to promote educational programmes [86]. Social media is even used to monitor the outbreak of war and civil unrest [87].

The emergence of cloud computing has provided cost-effective, pay-as-you-go capacity to process big data economically. It has encouraged the exploration and analysis of datasets collected from the Internet, which has led to research investigating social media. Twitter is a huge producer of data, which is available by connecting to the firehose. Research investigating Twitter analysis has previously concentrated on various components of the Tweet itself: [39] rates news sites by counting their URL mentions, [88] counts the number of followers that a user has, exploring celebrities and their followers, [89] focuses on defining trends by the co-occurrence of words within Tweets and [60] describes a mechanism for determining popular messages on Twitter so they can be forwarded to people who are following fewer users. Other components of Tweet metadata that are being investigated are gender and location [90]. Twitter can be used as an opinion-sharing network, to establish on-line tensions, with [91] using sentiment analysis to track racism in football. It uses Cosmos [92], which is a system that uses sentiment analysis and machine learning to detect tensions. As an opinion-sharing network Twitter does not guarantee reliability of the content of its messages; [93] questions this by asking how Tweets mirrored the real-life results of the NBA playoffs. Exploring the content of Tweets through analysis of hashtags allows topics and trends to be identified [94]. Timely and consistent information is of high importance in emergency events, so low-latency querying and near real-time results are vital, which is shown in [61].

Stream analysis can be used to detect events such as [95] and [6]. TEDAS mentioned in [96] also uses stream data, to rank Tweets and predict locations in real-time. This paper [97], researched data collected from Twitter around the Egyptian and Tunisian revolutions. It has shown that it is an additional source of information to the news about on going events around the world. A public radio presenter, Andy Carvin, used Twitter to make sense of different information about the Arab spring uprisings and to link demonstrations and the people involved.

Bollen *et al* [98] delve into sentiment analysis to discover how that information can be used to predict the stock market. Investigating how Twitter feeds correlate with the Dow Jones Industrial Average, they found that changes in public sentiment can be tracked from the content of large scale Twitter feeds using relatively simple text processing and that changes in the sentiment were responsive to a number of socio-cultural drivers. Data from social media is not simply background noise there are strong correlations and relationships with activities, actions and issues in the offline world.

The availability of geo-located social media and rise of mobile usage is presenting both an opportunity and a challenge in terms of the analysis of the data produced. Geo-tagged Twitter data gives an understanding of what may be occurring in a given geographic location at a given time, providing a dialogue of what is happening within communities on a day to day basis or during significant events. [99] found that analysing geographic data over time can give interesting insight into events such as natural disasters like Hurricane Irene or public disorder events like the London Riots of 2011. Tweets were also found to be useful in gauging the severity of the situation on the ground.

There is recognition of the quantity of social media data available and the value it has in supporting research in the social sciences. Analysis of said data has proven to be possible and congruent with events and systems in the offline world. There is still scope for improved access, tooling and scalability to aid exploration of Twitter data in order to better understand the data and aid further research.

A theoretical paper that discussed the idea of a Microblogs Data Management System is introduced in [60], this discusses the requirement of a management system which is tuned and can control microblogging data. It discusses the importance of temporal, spatial and keyword searches - these are the characteristics which *Antares* bases its queries. The temporal aspect of the querying is defined as the most important and should be included in all queries - this is the approach *Antares* uses to ensure recency of data and to decrease the result set size to support quicker querying and relevant data. The architecture itself would use a query engine and memory indexer - providing a combination of in-memory querying and historic querying to support lower latency querying. The system must be able to support high ingestion rates so that data is not lost from the high volume and velocity streams of microblogging sites. *Antares* agrees with this and implements a feature to allow for this. The design introduces a memory indexer to provide a solution which will monitor the hits to memory and the usage of the data stored in memory to optimise querying. *Antares* uses a different approach layering the system to use a stream processing system for immediate processing and low latency database queries for historic and wider ranges of querying. The paper also alludes to the most prominent queries being top k to reduce the query time and enable scalable processing - as the data sets produced from microblogging sites are so vast. *Antares* uses its scalable nature to return the entire result set a user requests bounded by time, space or keywords - this is because of its scalable nature a feature requirement of the system. The Microblogs Data Management System is similar to Antares is some ways however is more focused on optimising queries in memory and with smaller result sets, *Antares* instead focuses on scaling Twitter analysis so that query result sets are not constrained but do return with enough low latency to support the combination of historic and stream processing. Both systems however show a design for high ingestion rates, which *Antares* supports and has been proven to support this by evaluation using a simulated stream of Twitter data.

There are other applications that collect and analyse Twitter data, a selection of the most important of these is discussed below.

1. Collaborative On-line Social Media Observatory (COSMOS): A product of an Economic and Social Research Council (ESRC) strategic "Big Data" investment that seeks to bring together researchers from different areas to explore the different dimensions of social media data [92]. COSMOS is an example of an application which is designed for academic users. It is presented in both a desktop application and also an on-line version that benefits from scalability. The desktop application does not have the ability to scale but does provide a suite of visualisation and data exploration tools. Visualisation is customisable and provides textual/trend analysis, geographical mapping, relationship mapping and other more generic charting tools. Data sources on the desktop edition are limited to imported historical data and data taken from the Twitter Search API [100]. COSMOS makes use of other contextual data such as a persistent connection to the UK police API. Use of the Hadoop environment allows the application to scale, whilst storing the tweets in MongoDB. Instantaneous searches are available over the MongoDB datasets, however this is done through the use of in-memory indexes that are costly in terms of resources, with 24GB RAM required to search across one month of Twitter Search API output.

2. TweeQL and Twitinfo: Twitinfo is a Twitter analysis web platform which sits on top of TweeQL a Twitter querying language based on Twitter to support processing of unstructured Twitter data into structured output for underlying systems. Twitinfo processes geospatial, temporal and keyword data. It has a novel algorithm for detecting peaks in the Twitter data which is displayed on a rate graph. The query language itself allows the expressive nature of submitting queries about anyone of these key features. These techniques are similar to *Antares*, however they do not have the scalability of *Antares* or the combination of both historic and stream processing - TweeQL is for stream processing. Twitinfo does not store data into a store so data cannot be replyed or analysed, which is different to *Antares* which allows that functionality. Instead the focus is on the querying language and its mapping from Tweets to structured data - the query language is more complex then *Antares* however both systems have different requirements. *Antares* supports scalable ingestion and combined querying and Twitinfo and TweeQL provide a querying language for mapping a Twitter stream to structured output. The interfaces have similar functionality to iterations of *Antares* including, the rate of Tweets graph, the map of Tweets (however *Antares* builds on this information rather than just geospatial analysis) and the list of Tweets - this was used for specific hashtags. Twitinfo has the added feature of peaks and sentiment analysis. The system also uses a third party geospatial co-oridnates finding software where as *Antares* uses its own scalable and low latency algorithms and caching mechanism to map this from in-memory to the database.

3. SensePlace2: Developed at Penn State University, SensePlace2 is a temporal geospatial analysis tool for visualising Tweets. The visualisation techniques are different to *Antares*, which provides clustering and a time/location graph view.

   "geovisual analytics application which forages place-time-attribute based information from the Twitterverse and supports crisis management through visually-enabled sensemaking of the information derived. SensePlace2 integrates computational methods for capturing, storing, and indexing tweets with visual query and analysis methods" [101].

4. Geosocial Gauge: Developed by an interdisciplinary team at George Mason University, Geosocial Gauge [102] is a research platform that seeks to make use of social media data to provide insight into such areas as use of language on Twitter during natural disasters, and virtual boundaries that exist across traditional boundaries. Geosocial Gauge was created within or in consultation with a multi-disciplinary team to guide iterative development to produce an open tool for research on a wide range of topics.

5. Sentiment Viz: An online sentiment analysis tool created at North Carolina State University. Sentiment is measured using a sentiment dictionary to measure sentiment for the evaluated Tweets. Tweets are pulled directly from Twitter for a user's chosen keyword [103].

6. Radian6: A commercial offering from SalesForce marketed at companies who want to monitor and manage their on-line brand and direct their customer service and on-line marketing efforts [104].

7. Twitonomy: Another web based tool that allows monitoring of individual accounts, lists, hashtags or keywords. The focus is on the individuals and businesses who want to manage their online presence on Twitter. Broader search tools are available; however they appear to suffer high latency when returning queries and provide only basic visualisation [105].

8. Tweet Ping: A web based tool that displays tweets as they are posted in real-time around the world. Tweet Ping is less an interactive analysis tool and more a visualisation tool that shows the global distribution of Tweets as they occur [106]. Tweet Ping provides what can be seen as a shallow view into data provided by social media. Despite this it provides an engaging visual that is useful for attracting interest and attention to a particular subject area. Tweet Ping provides visualisation but lacks the interactivity to provide meaningful analysis. Likewise the application is able to handle a stream of Twitter data but does not show potential to scale beyond the Twitter Search API or to other social media sources.

5.2.1 Conclusion

Much of the literature focuses on application specific querying of Twitter data, unlike *Antares*, which provides non-application specific querying. The queries enable a core set of questions, which can be used in different application areas. *Antares* is solely an on-line tool so there is always access to the scalability of the cloud unlike systems like COSMOS. In comparison, *Antares* is able to quickly return queries from a six-week dataset taken from the Twitter firehose while consuming less than 3GB memory.

Radian6 and Twitonomy are just two examples of applications that are designed for individuals and businesses to analyse social media interactions to better understand how the public view them and their brand. This type of tool also allows users to make decisions on how best to direct their on-line marketing and social media efforts and as a result see better returns on investment. Radian 6 and Twitonomy provide useful analytics and visualisations, these include analysis over time, metrics regarding users and their influence and engagement with other users. Mapping of Tweets is also available. Twitonomy has limited scope in terms of high volume analysis and also there is little interaction available in the visualisation. The search tools also appear to be making use of the Twitter Search API rather than any high volume source such as the Twitter firehose.

Applications in this category can be adapted through usage and feedback; however the visualisations do not have the level of interactivity that would provide a useful research tool. *Antares* has the ability to provide this functionality, given its array of visualisation tools.

Antares focuses on scalability and low latency querying unlike the tools mentioned in this section. *Antares* aimed to improve the ingestion rate, query execution time and scalability of these systems. *Antares* provides visualisations to support the analysis of Twitter data for the user, providing spatial and temporal displays of the data for the user to process, this is described in the next section.

5.3 User Interface

To enable non-expert users to analyse data there is a web interface which was designed and developed with colleagues. This is used to enter parameters to queries and support the visualisation of the data. When data from the web interface arrives at the query monitor, it is analysed and turned into queries against the streaming data and/or the historic data depending on the time-bounds.

In Figure 5.1 the web interface's map feature is shown. The interface allows the user to zoom in and out and pan around a world map. The Tweets are displayed on the map as clusters (the circled numeric values) this is the number of Tweets in that area. This method is used to make the display less cluttered for users and enable easier analysis. By clicking on one of these clusters the Tweets for the area are displayed and the user can then click on the Tweet marker shown on the right hand side of the figure with the Twitter bird as the symbol. Once the Tweet marker has been clicked the content of the Tweet is displayed in a modal – this is shown in Figure 5.2. From there the user can access information about the user, the place, nearby Tweets and events. This is provided by using third party APIs.

A graph of the rate of Tweets is also displayed on the bottom of Figure 5.2. The graph displays the rate of Tweets over time for that viewport. It enables the user to replay Tweets as they were posted. By clicking on the play button the Tweets for each day are displayed, growing in number and displaying the distribution of Tweets in the area covered by the viewport. Additionally specific days can be chosen to display on the map and the playback can be paused. It can be used to predict future peaks or explore historic peaks (e.g. for marketing it would be useful to know when to put an advert on television or social media). This uses the tree structure but indexes on time to add an extra dimension to the schema for querying.

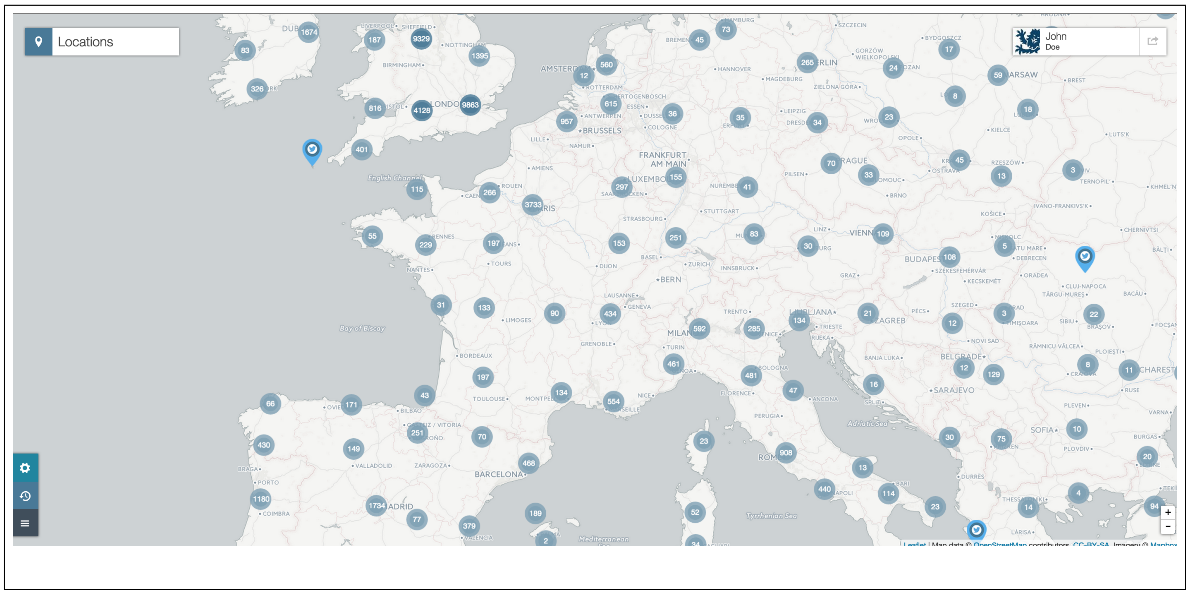
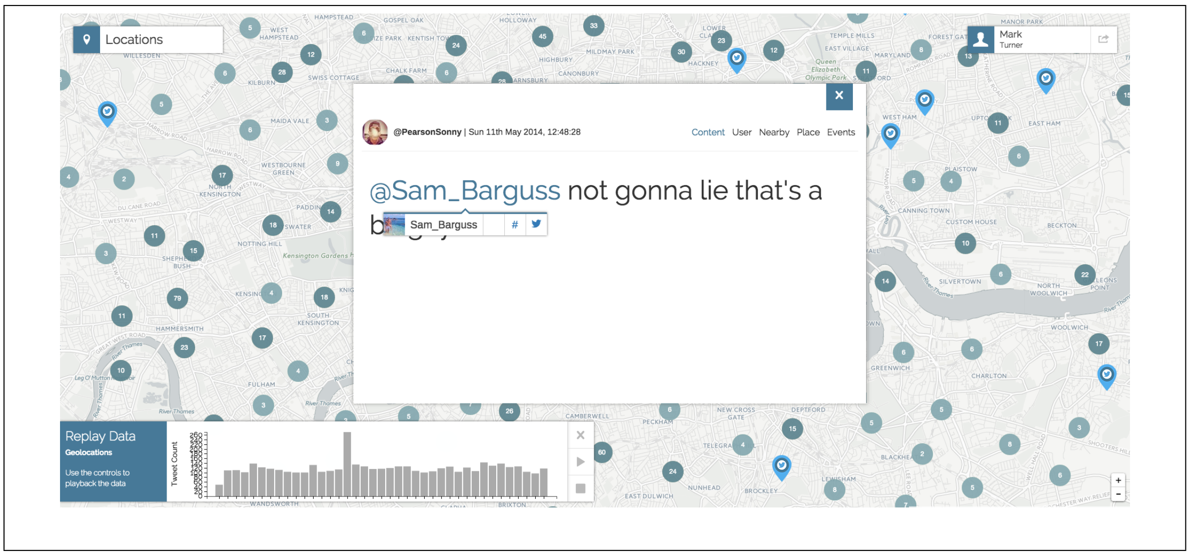


Figure 5.1: *Antares*: map



@Sam_Barguss not gonna lie that's a

Figure 5.2: *Antares*: map with modal

Figure 5.3 displays the word cloud implemented in *Antares*. Here you can see there are multiple hashtags displayed in the viewport. These hashtags have all co-occurred in the same Tweet as the hashtag the user clicked on to display the word cloud. The larger the hashtag the more often it has been posted together with the hashtag selected. This allows users to identify what topics are connected and can support further investigation of a topic using wider search terms.

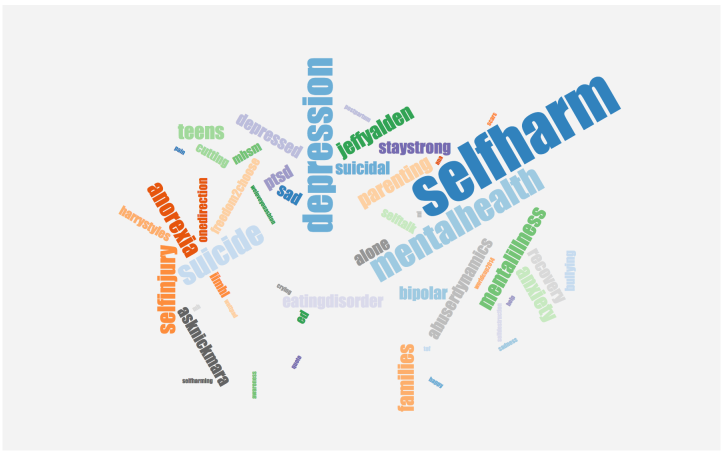


Figure 5.3: *Antares*: word cloud

5.4 Tweet My Street

Once *Antares* had been designed and developed it was deployed in a cross-disciplinary project called Tweet My Street [3], which is a collaboration of social geographers and computer scientists. The project drew on the interest in processing big data and how it can be used to provide insights into social, economic and political data.

The queries derived from earlier drive the web UI to support social scientists in answering "how are more deprived areas using Twitter?" and "can social media endanger people in homophobic countries?"

Antares was used to visualise geographical data by placing a marker on a map to represent where Tweets were posted. The map also combines data about world events that happened within that time period, to compare whether people were interested in world events and if that was having an effect on people's opinions. It uses the location to highlight the five closest Tweets to challenge whether the location is directly involved in similar topics and themes in that area.

Through the collaboration an iterative approach has been employed for the design and development of the system, using user feedback to modify *Antares*. It has supported research and analysis for humanities-based projects.

Antares has been deployed and used in different case studies for the Tweet My Street project, to enable interactive visualisations and near real-time processing of Twitter data. Data has been collected from the full firehose and the comprehensive set of queries have been applied for the case studies.

5.4.1 West End of Newcastle

The West End is one of the more deprived urban areas in Newcastle, which would suggest the inhabitants may not have the technology or means to access and use social media. This case study aimed to identify whether or not this was true, and if not then what information could be derived about the area from social media posts.

The study collected a set of Tweets from the area for a four-week period. *Antares* was employed to visualise the rate, as shown in Figure 5.4, of the Tweets on an hourly basis. This demonstrated that there was a high number of people tweeting from the area, even though, according to Leetaru et al, only 1.6% of people have the functionality turned on [107]. It was discovered that people within the area would often economise to buy technology, which facilitates access to social media, emphasising how important the platform and accessing it is to people.

Further analysis of the data led to findings about the ethnicity of the area, which corresponded to its demographic. It was also established that people were using the platform to express concerns about the local community, as can be seen in Figure 5.5, where a user mentions the lack of heating within a council residence.

The link between transportation and social media posts can also be identified, shown in Figure 5.6. Therefore collection of more real-time data would give an idea of traffic delays and service quality. This study demonstrated how even a smaller collection of Twitter data can be employed to analyse local communities.

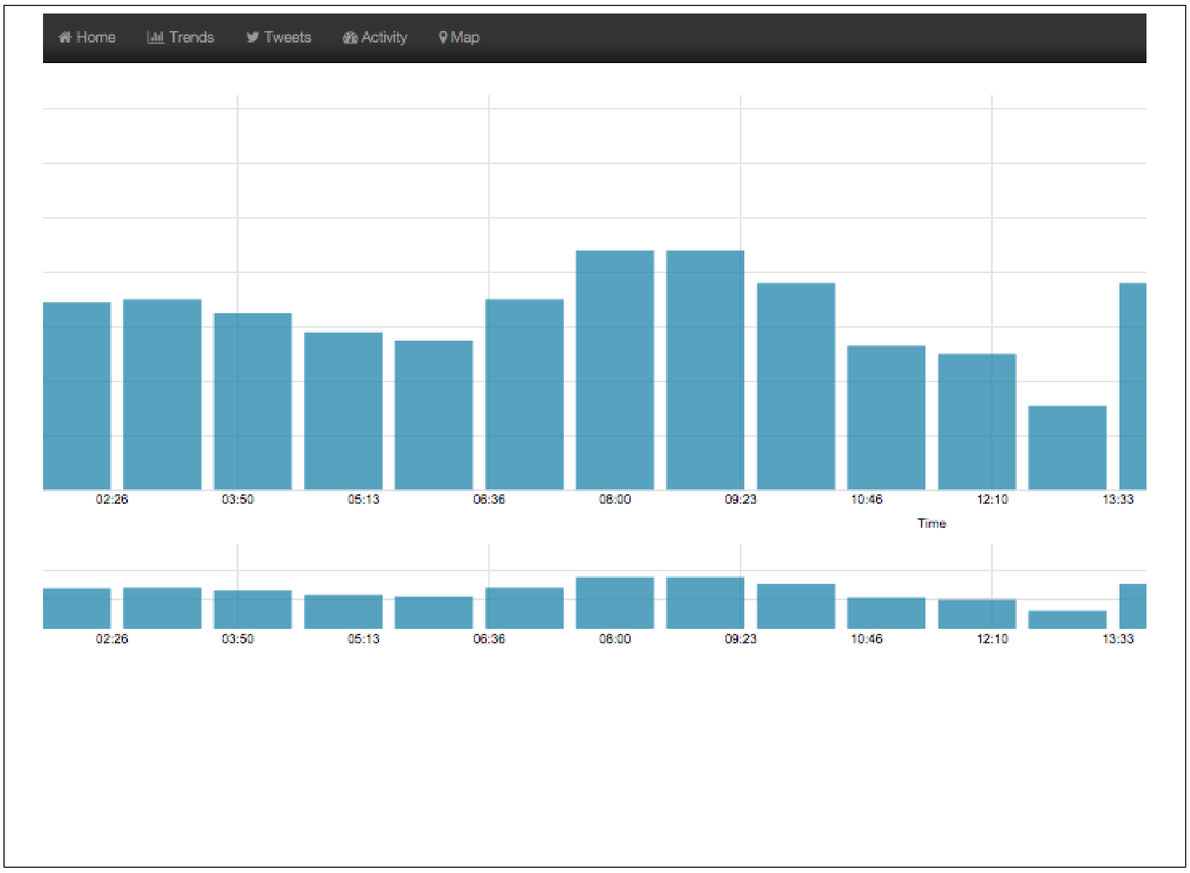
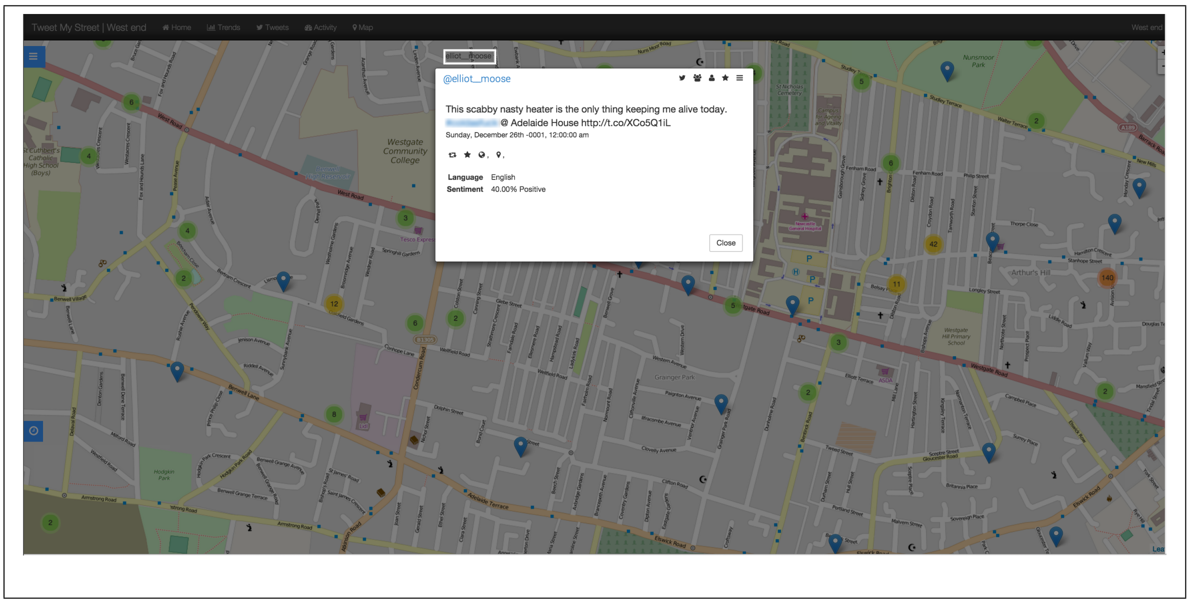


Figure 5.4: *Antares*: activity graph showing the rate of tweets in the West End of Newcastle

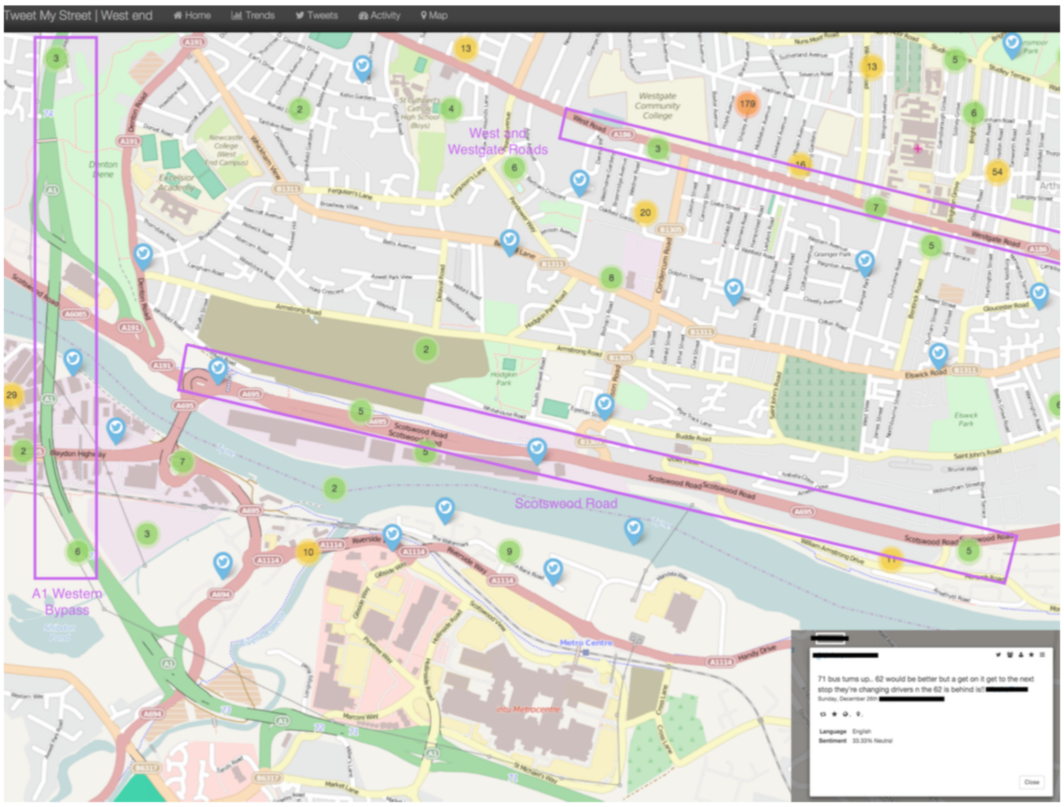


@elliot__moose

This scabby nasty heater is the only thing keeping me alive today. [redacted] @ Adelaide House http://t.co/XCo5Q1iL
Sunday, December 26th -0001, 12:00:00 am

Language	English
Sentiment	40.00% Positive

Figure 5.5: *Antares*: map showing community opinion



[redacted]

[redacted]

71 bus turns up... 62 would be better but a get on it get to the next stop they're changing drivers n the 62 is behind is!! [redacted]
Sunday, December 26th [redacted]

Language English
Sentiment 33.33% Neutral

Figure 5.6: *Antares*: obvious traffic routes

5.4.2 International Day Against Homophobia and Transphobia

Tweet My Street also analysed a collection of ten million tweets. This is a much larger dataset and has been collected at a global level, using a list of hashtags to filter the data instead. The collection was based on the International Day Against Homophobia and Transphobia (IDAHOT), with data being collected for the period three weeks before and after the event. The aim of this study is to support the campaign by analysing the reaction on Twitter and provide insights into the tensions and opinions around the day. *Antares* was used to analyse the data with the aim of relaying this information to the campaigners themselves.

The study focused on African countries which are deemed the "worst places to be homosexual" (WPTBH). The study focused on Saudi Arabia, Sudan, Mauritius, Somalia and Nigeria. The greatest number of Tweets were posted in Nigeria; 99% were in English and 82.89% were posted by males. Not one of the Tweets contained a reference to IDAHOT, therefore demonstrating little impact from the western campaign to WPTBH placed in the global south – this was a noted observation across all the countries mentioned. 11.8% of the data was used to “name” or “out” people suggesting there is a real danger attached to social media in such areas. The information will go towards helping and supporting the campaign to grow and to help places such as the WPTBH.

5.4.3 Psychology

The system is being used to establish whether self-harm cases are on the increase. The study aims to demonstrate whether content from the Internet and specifically social media provides a platform for people to connect and support each other and provides an additional type of support group. It is thought the increase in self-harm may just be the increase of awareness about the topic through platforms such as social media.

Antares collected a dataset containing self-harm hashtags. An example within the dataset is the strong co-occurrence of “one direction”, “harry styles” and “depression”, which was because of One Direction fans not being able to go to the show. The data showed a large amount of people tweeting about self-harm and eating disorders. During only a 24 hour period over 4,000 Tweets were collected. Analysis of the data revealed a positive element to the sharing of self-harm content on social media, with supportive posts being shared more than content encouraging self-harm behaviour. Some examples of positive posts included those offering support to others, or raising awareness of self-harm in the general population. Self-harm dedicated blogs (i.e., those where the users solely blogged about self-harm and no other topics) were in the minority, and those that did exist tended to be of a positive, pro-recovery nature (rather than encouraging self-harm). Overall, suggesting that social media may play a positive role for the majority of users sharing SH related posts.

5.4.4 Conclusion

Antares has provided social scientists with a visualisation tool that enables then to explore and analyse large Twitter data sets in a scalable and timely manner. This provides them with a tooling to collect large data sets and then explore different use cases around events, which would have previously been executed manually. The system itself has shown ingestion rates beyond the size of the average firehose and queries that support the combination of historic and streaming analysis. The spatial functionality supports low latency querying which is an improvement on current technologies allowing visualisations of large sets of data to be mapped. As can be seen from the related work many systems focus on specific querying techniques or specific application areas. *Antares* provides a non-application specific mechanism for scalably processing and ingesting Twitter data. It extends the layered approach of historic and streaming data to support greater scalability and combine different temporal aspects of analysis - reprocessing past events, processing current and future and combining the two. *Antares* covers a specific domain area which was led by the research undertaken by the social scientists, however this does not hinder its performance and it uses this subset of questions to form a model and basis to prove the scalability of the system and how it can be used to derive insights for researchers.

6 Conclusion

The aim of this thesis was to explore and identify possible solutions to support efficient and near real-time querying of large volume stream and stored Twitter data. The solution would support stream, historic, combined and geospatial analysis with the aim of producing low-latency responses when querying large-scale datasets, even as data was being ingested at high velocity. The motivation came from work in a multi-disciplinary project focused on Twitter analytics.

An extensive literature review identified opportunities for improvement for the combined processing of stream and stored data, especially large geospatial data. *Antares* explores how to build on the strengths of noSQL databases to meet these requirements. This research is timely given the recent explosion in streaming data from social media and other sources such as sensors. *Antares* improved on current techniques for geospatial analysis, focusing on improving performance and increasing scalability. The evaluation demonstrated that *Antares* supports simpler geospatial querying that is a factor of 1000 times faster in some cases. The querying functionality could be extended, however for the purpose of quick view port analysis in a browser *Antares* provides a new contribution for querying geospatial data using an in-memory cache and noSQL database.

It was identified that improvements could be made to exploit the scale of noSQL databases with the efficiency of traditional geospatial querying techniques. These could be achieved by adapting them to support novel techniques for querying new datasets such as Twitter. *Antares* supports a novel caching mechanism for efficient geospatial data analysis, which was evaluated and shown to out perform the current state of the art. This enabled users of *Antares* to interactively explore large temporal and geospatial datasets quickly and efficiently something not possible without *Antares*. For example *Antares* was evaluated against Solr for geospatial queries, and shown to be 15,000 times faster for the range of experiments in Chapter **??**.

A layered approach was explored and how this would affect performance and scalability. The layered approach supported scalability to enable larger ingest rates - supporting a stream and historic layer meant that processing could be undertaken in both. Requests made to the system used both layers, which balanced the load of requests to each. This reduced the number of time-outs, therefore reducing query execution time. The time-out is 5 seconds, however with large volumes of data this would have just kept increasing and would have reduced the ingestion rate of the system. The ingestion rate would have been reduced as the system would have blocked due to the wait held by the time-out. The layered approach supports higher ingest rates as the stream processing layer can ingest data and the historic store can ingest data, therefore load balancing.

The layered approach also provides real-time analysis, using the speed of in-memory, and scale of historic stores (disk storage), which is cheaper and more durable, while helping to expand and scale-out the system by adding new nodes.

As demonstrated in Chapter **??**, the design of *Antares* achieved its aims: it is capable of executing tens of thousands of queries concurrently, and in some cases its novel mechanisms halved response times.

The first contribution was made by the analysis of use cases to derive the design and implementation of a set of queries to provide functionality to *Antares* for querying Twitter data - this was then used to support the evaluation of scalability and low latency querying as well as using the tooling to support research of large Twitter datasets. These queries were derived from a data model described in Section 3.4. Implementation of these and an index to optimise data retrieval supported the transparent combination of stream and historic analysis, with percentage savings nearly reaching 100%. Allowing one query to be submitted with *Antares* translating whether stream, historic or a combination of both needed to be executed.

Antares uses indexing techniques to improve the performance of query execution by distributing the data to enable scale-out capability. However the datasets also need to be ingested at high rates to accommodate large sized data streams. *Antares* makes the second contribution by optimising data ingestion using asynchronous batching. For example an eight-node cluster can cope with over 30,000 Tweets per second. This supports scalable Twitter ingestion for analysis which can ingest the average Twitter firehose for five times more data.

The Split Algorithm was designed so that there is no global locking required in the system, this reduces execution time and exploits the asynchronous features added into *Antares*. As *Antares* is an append only system there is no need to rebalance the tree.

This design decision also supported quicker reads and writes as there was no need to hold a global lock on the system when it was being rebalanced.

Antares was used within a cross-disciplinary project called Tweet My Street. This collaboration resulted in use of real user requirements that had to be met. This allowed the system to be evaluated as a research tool by social geographers, while simultaneously generating interesting computing problems. It also meant *Antares* underwent rigorous user testing, which caught errors, and generated new requirements that had not been originally thought of. This helped *Antares* in its aim to provide insights into interesting events by analysing Twitter data. It meant that the key aims of the system like low response times became even more crucial, and it became obvious if they were not good enough for the users.

Future work that could be used to continue research in the area would be to identify a means for making ESPER distributed. ESPER currently only executes on a standalone machine – if it could be distributed over many machines its scale would be very powerful. *Antares* could also be extended to notify people of different events that have been identified for a particular user. A query could be executed over a prolonged period and then when an event is identified on the stream and/or the historic the user would be notified of an event. The next step for this would be to allow a user to reply to this notification to either set off another query execution or to request more information.

Antares is a scalable platform utilised in a cross-disciplinary project where it provides researchers with an advanced analytical tool. The system improves on response times for combined, historic and stream querying. Additionally, *Antares* supports geospatial querying with efficient response times, and results show that it outperformed the current state of the art for this. It has exploited the scale of NoSQL databases to support geospatial querying at scale and so support a rich, interactive user-interface. Designing *Antares* has provided an insight into the technical aspects of NoSQL databases and social media analysis. It is now forming the basis of future work to extend its capabilities and further improve its performance.

References

[1] I. L. Stats, "Twitter Usage Statistics," 2014. [Online]. Available: http://www.internetlivestats.com/twitter-statistics/

[2] R. Simmonds, P. Watson, J. Halliday, and P. Missier, "A Platform for Analysing Stream and Historic Data with Efficient and Scalable Design Patterns," *2014 IEEE World Congress on Services*, no. Ii, pp. 174–181, 2014.

[3] G. Mearns, R. Simmonds, R. Richardson, M. Turner, P. Watson, and P. Missier, "Tweet My Street: A Cross-Disciplinary Collaboration for the Analysis of Local Twitter Data," *Future Internet*, vol. 6, no. 2, pp. 378–396, 2014.

[4] R. Simmonds, P. Watson, and J. Halliday, "Antares: A Scalable, Real-Time, Fault Tolerant Data Store for Spatial Analysis," *2015 IEEE World Congress on Services*, pp. 105–112, 2015.

[5] A. Toshniwal, J. Donham, N. Bhagat, S. Mittal, D. Ryaboy, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, and M. Fu, "Storm@twitter," *Proceedings of the 2014 ACM SIGMOD international conference on Management of data - SIGMOD '14*, pp. 147–156, 2014.

[6] R. McCreadie, C. Macdonald, I. Ounis, M. Osborne, and S. Petrovic, "Scalable distributed event detection for Twitter," *2013 IEEE International Conference on Big Data*, pp. 543–549, October 2013.

[7] Red Hat, "Drools Fusion." [Online]. Available: http://docs.jboss.org/drools/release/6.2.0.CR1/drools-docs/html/DroolsComplexEventProcessingChapter.html

[8] R. Hat, "Red Hat Magazine -Rules and Drools Rundown." [Online]. Available: http://magazine.redhat.com/2008/07/17/rules-and-drools-rundown/

[9] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed stream computing platform," in *Proceedings of the 2010 IEEE International Conference*

on Data Mining Workshops, ser. ICDMW '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 170–177.

[10] IBM, "System S Stream Processing." [Online]. Available: http://researcher.watson.ibm.com/researcher/view_group_subpage.php?id=2534

[11] Codehaus, "Esper - Complex Event Processing," 2014. [Online]. Available: http://esper.codehaus.org/

[12] K. Weil, "Rainbird : Real-time Analytics @ Twitter," *Presented at: Twitter Conference*, 2011.

[13] T. Akidu, A. Balikov, K. Bekiroglu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, S. Whittle, T. Akidau, and K. BekiroÄ§lu, "MillWheel: Fault-Tolerant Stream Processing at Internet Scale," *Proceedings of the the VLDB Endowment*, vol. 6, no. 11, pp. 734–746, 2013.

[14] IBM, "IBM InfoSphere Software - Information Integration, Data Warehouse and Master Data Management." [Online]. Available: http://www-01.ibm.com/software/uk/data/infosphere/

[15] Apache, "Samza." [Online]. Available: http://samza.apache.org/

[16] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, ser. SOCC '13. New York, NY, USA: ACM, 2013, pp. 5:1–5:16.

[17] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja, "Twitter Heron: Stream Processing at Scale," *ACM SIGMOD Record*, 2015.

[18] J. Browne, "Brewer's CAP theorem," *J. Browne blog*, 2009.

[19] M. Burrows, "The Chubby lock service for loosely-coupled distributed systems," *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation SE - OSDI '06*, pp. 335–350, 2006.

[20] MIT, "relationalcloud," 2013. [Online]. Available: http://relationalcloud.com/index.php?title=Main_Page

[21] C. Curino, E. P. Jones, R. A. Popa, N. Malviya, E. Wu, S. Madden, H. Balakrishnan, and N. Zeldovich, "Relational cloud: A database-as-a-service for the cloud," *Journal: 5th Biennial Conference on Innovative Data Systems Research, CIDR 2011*, 2011.

[22] Apache, "HBase," 2014. [Online]. Available: http://hbase.apache.org/

[23] MongDb, "MongoDB Architecture Guide," 2013. [Online]. Available: http://www.mongodb.com/mongodb-architecture

[24] MongoDB Manual 3, "Replica Set Oplog." [Online]. Available: http://docs.mongodb.org/manual/core/replica-set-oplog/

[25] MongoDb, "GridFS." [Online]. Available: http://docs.mongodb.org/manual/core/gridfs/

[26] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. a. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable," *ACM Transactions on Computer Systems*, vol. 26, no. 2, 2008.

[27] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin, "HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for analytical workloads," *Proceedings of the VLDB Endowment*, vol. 2, no. 1, pp. 922–933, 2009.

[28] *SIGMOD '10: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data.* New York, NY, USA: ACM, 2010.

[29] Red Hat, "infinispan," 2015. [Online]. Available: http://infinispan.org/

[30] Lucene, "Apache Lucene," 2013. [Online]. Available: http://lucene.apache.org/

[31] S. Jouili and V. Vansteenberghe, "An empirical comparison of graph databases," *Proceedings - SocialCom/PASSAT/BigData/EconCom/BioMedCom 2013*, pp. 708–715, 2013.

[32] Amazon, "AWS | Amazon SimpleDB âĂŞ Simple Database Service," 2015. [Online]. Available: http://aws.amazon.com/es/simpledb/#pricing

[33] H. Wada, A. Fekete, L. Zhao, K. Lee, and A. Liu, "Data Consistency Properties and the Trade-offs in Commercial Cloud Storage: the Consumers' Perspective," in *CIDR*, 2011, pp. 134–143. [Online]. Available: http://www.cidrdb.org/cidr2011/program.html http://www.cidrdb.org/cidr2011/Talks/CIDR11_Wada.ppt

[34] Neo4j, "Neo4j Graph Database." [Online]. Available: Http://neo4j.com/

[35] B. T. Inc., "Riak | Basho Technologies," 2015. [Online]. Available: http://basho.com/riak/

[36] Mochi Media, "mochi_statebox_riak." [Online]. Available: https://github.com/mochi/statebox_riak

[37] S. J. Kazemitabar, U. Demiryurek, M. Ali, A. Akdogan, and C. Shahabi, "Geospatial Stream Query Processing using Microsoft SQL Server StreamInsight," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 1537–1540, 2010.

[38] Microsoft, "SQL Server," pp. 1–10, 2014. [Online]. Available: http://www.microsoft.com/en-gb/server-cloud/products/sql-server/

[39] M. Grinev, "Analytics for the Real-Time Web," *System*, pp. 1391–1394, 2011.

[40] A. Arasu, S. Babu, and J. Widom, "The cql continuous query language: Semantic foundations and query execution," *The VLDB Journal*, vol. 15, no. 2, pp. 121–142, Jun. 2006.

[41] Google Inc, "Google Cloud Platform," 2014. [Online]. Available: https://cloud.google.com/

[42] PCWorld, "Google takes on real-time big data analysis with new cloud services." [Online]. Available: http://www.pcworld.com/article/2911112/google-takes-on-realtime-big-data-analysis-with-new-cloud-services.html

[43] InfoWorld, "Databricks takes on Google streaming analysis with Spark." [Online]. Available: http://www.infoworld.com/article/2607830/open-source-software/databricks-takes-on-google-streaming-analysis-with-spark.html

[44] J. Dittrich and J.-a. Quian, "Efficient Big Data Processing in Hadoop MapReduce," *Proceedings of the VLDB Endowment*, vol. 5, no. 12, pp. 2014–2015, 2012.

[45] O. Boykin, S. Ritchie, I. O'Connell, and J. Lin, "Summingbird: A framework for integrating batch and online mapreduce computations," *Proc. VLDB Endow.*, vol. 7, no. 13, pp. 1441–1451, Aug. 2014.

[46] H. Edelson, "Lambda Architecture with Spark Streaming, Kafka, Cassandra, Akka, Scala." [Online]. Available: http://www.slideshare.net/helenaedelson/lambda-architecture-with-spark-streaming-kafka-cassandra-akka-scala

[47] N. Garg, "Apache Kafka," 2013. [Online]. Available: http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:No+Title#0\nhttp:

[48] Typesafe Inc, "Akka." [Online]. Available: http://akka.io/

[49] Datastax, "Performance improvements in Cassandra 1.2." [Online]. Available: http://www.datastax.com/dev/blog/performance-improvements-in-cassandra-1-2?utm_source=NoSQL+Weekly+Newsletter&utm_campaign=900dff577b-NoSQL_Weekly_Issue_107_December_13_2012&utm_medium=email

[50] Scala, "The Scala Programming Language," 2011. [Online]. Available: http://www.scala-lang.org/

[51] R. Schindlauer, "Scalability Patterns," 2010. [Online]. Available: https://social.technet.microsoft.com/Forums/windowsserver/en-US/be14f0ed-5066-4d86-a001-c5e6ba4d708c/scalability-patterns?forum=streaminsight

[52] "Cisco Completes Acquisition of Lightwire," 2012. [Online]. Available: http://newsroom.cisco.com/press-release-content?type=webcontent&articleId=744182

[53] Michael Hwang, "Stream Processing Using ESPER," 2015. [Online]. Available: https://www.hakkalabs.co/articles/stream-processing-using-esper

[54] H. Kwak, C. Lee, H. Park, and S. Moon, "What is twitter, a social network or a news media?" in *Proceedings of the 19th International Conference on World Wide Web*, ser. WWW '10. New York, NY, USA: ACM, 2010, pp. 591–600.

[55] D. Fisher, "Interactions with Big Data Analytics population by running controlled," *Interactions*, pp. 50–59, 1983.

[56] M. Mathioudakis and N. Koudas, "Twittermonitor: Trend detection over the twitter stream," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '10. New York, NY, USA: ACM, 2010, pp. 1155–1158.

[57] Twitter, "CS Conferences," 2013. [Online]. Available: https://twitter.com/CS_Conferences

[58] D. S. Shiffman, "Twitter as a tool for conservation education and outreach: what scientific conferences can do to promote live-tweeting," *Journal of Environmental Studies and Sciences*, vol. 2, no. 3, pp. 257–262, Jul. 2012.

[59] D. R. A. McKendrick, "Smartphones, Twitter and New Learning Opportunities at anaesthetic Conferences," *Anaesthesia*, vol. 67, no. 4, pp. 437–8, Apr. 2012.

[60] L. Hong and B. D. Davison, "Predicting Popular Messages in Twitter," *ReCALL*, pp. 57–58, 2011.

[61] A. B. P. Guide, "Social Media in an Emergency," *Opus*, March 2012.

[62] S. Doan, B. H. Vo, and N. Collier, "An analysis of twitter messages in the 2011 tohoku earthquake," *CoRR*, vol. abs/1109.1618, 2011.

[63] S. Nagar, A. Seth, and A. Joshi, "Characterization of social media response to natural disasters," *Proceedings of the 21st international conference companion on World Wide Web - WWW '12 Companion*, p. 671, 2012.

[64] P. J. R. Alan Rusbridger, *Reading the Riots*, guardian ed. Guardian, 2012.

[65] Twitter4j, "Twitter4J - A Java library for the Twitter API," 2014. [Online]. Available: http://twitter4j.org/en/index.html

[66] Mishail, "CqlJmeter." [Online]. Available: https://github.com/Mishail/CqlJmeter

[67] D. Comer, "The ubiquitous B-tree," *ACM Computing Surveys*, vol. 11, pp. 121–137, 1979.

[68] A. Guttman, "R-trees: A Dynamic Index Structure for Spatial Searching," *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data - SIGMOD '84*, pp. 47–57, 1984.

[69] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, vol. 18, pp. 509–517, 1975.

[70] R. A. Finkel and J. L. Bentley, "Quad trees a data structure for retrieval on composite keys," *Acta Inf.*, vol. 4, no. 1, pp. 1–9, Mar. 1974.

[71] J. Liu, H. Li, Y. Gao, Y. Hao, and D. Jiang, "A geohash-based index for spatial data management in distributed memory," in *Geoinformatics (GeoInformatics), 2014 22nd International Conference on*, June 2014, pp. 1–4.

[72] C. Du Mouza, W. Litwin, and P. Rigaux, "SD-Rtree: A scalable distributed Rtree," *Proceedings - International Conference on Data Engineering*, pp. 296–305, 2007.

[73] A. Crainiceanu, P. Linga, J. Gehrke, and J. Shanmugasundaram, "Querying peer-to-peer networks using P-trees," *Proceedings of the 7th International Workshop on the Web and Databases colocated with ACM SIGMOD/PODS 2004 - WebDB '04*, p. 25, 2004.

[74] P. Kalnis, "Distributed Spatial Databases," in *Encyclopedia of Database Systems*. Springer, 2009, pp. 920–925.

[75] A. Rafalovitch, *Instant Apache Solr for Indexing Data How-to*. Packt Publishing, 2013.

[76] S. Nishimura, S. Das, D. Agrawal, and A. E. Abbadi, “Md-hbase: A scalable multi-dimensional data infrastructure for location aware services,” in *Mobile Data Management (MDM), 2011 12th IEEE International Conference on*, vol. 1. IEEE, 2011, pp. 7–16.

[77] A. Aji, F. Wang, H. Vo, R. Lee, Q. Liu, X. Zhang, and J. Saltz, “Hadoop-GIS: A High Performance Spatial Data Warehousing System over MapReduce.” *Proceedings of the VLDB Endowment International Conference on Very Large Data Bases*, vol. 6, no. 11, pp. 1009–1020, 2013.

[78] D.-M. Chang, “RESQUE: A resource based simulation system for construction process planning.” *Dissertation Abstracts International Part B: Science and Engineering[DISS. ABST. INT. PT. B- SCI. & ENG.],*, vol. 47, no. 10, 1987.

[79] K. M. Al-naami, “GISQF : An Efficient Spatial Query Processing System,” *2014 IEEE International Conference on Cloud Computing*, 2014.

[80] K. Leetaru and P. A. Schrodt, “GDELT: Global data on events, location, and tone, 1979–2012,” in *ISA Annual Convention*, vol. 2, 2013, p. 4.

[81] H. Tan, W. Luo, and L. M. Ni, “Clost: a hadoop-based storage system for big spatio-temporal data analytics,” in *Proceedings of the 21st ACM international conference on Information and knowledge management.* ACM, 2012, pp. 2139–2143.

[82] N. Zhang, G. Zheng, H. Chen, J. Chen, and X. Chen, “HBaseSpatial: A Scalable Spatial Data Storage Based on HBase,” *2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications*, pp. 644–651, 2014.

[83] Y. Li, G. Kim, L. Wen, and H. Bae, *Ubiquitous Information Technologies and Applications.* Dordrecht: Springer Netherlands, 2013, ch. MHB-Tree: A Distributed Spatial Index Method for Document Based NoSQL Database System.

[84] A. S. Foundation, “Field Types Included with Solr.” [Online]. Available: https://cwiki.apache.org/confluence/display/solr/Field+Types+Included+with+Solr

[85] F. Figueiredo, J. M. Almeida, F. Benevenuto, and K. P. Gummadi, "Does content determine information popularity in social media?: A case study of youtube videos' content and their popularity," in *Proceedings of the 32Nd Annual ACM Conference on Human Factors in Computing Systems*, ser. CHI '14. New York, NY, USA: ACM, 2014, pp. 979–982.

[86] B. S. Brosseau. L, Wells. GA, "Orthopedics News Article People getting a grip on arthritis II An innovative strategy to implement clinical practice guidelines for rheumatoid arthritis and osteoarthritis patients through Facebook," *Health Education Journal*, 2013.

[87] J. Xu, T.-C. Lu, R. Compton, and D. Allen, *Social Computing, Behavioral-Cultural Modeling and Prediction: 7th International Conference, SBP 2014, Washington, DC, USA, April 1-4, 2014. Proceedings.* Cham: Springer International Publishing, 2014, ch. Civil Unrest Prediction: A Tumblr-Based Exploration, pp. 403–411.

[88] B. Meeder, B. Karrer, C. Borgs, R. Ravi, and J. Chayes, "We Know Who You Followed Last Summer : Inferring Social Link Creation Times In Twitter," *Focus*, 2012.

[89] J. Pöschko, "Exploring twitter hashtags," *CoRR*, vol. abs/1111.6553, 2011.

[90] A. P. Burnap, W. Housley, J. Morgan, L. Sloan, N. Avis, A. Edwards, O. Rana, M. Williams, and P. Burnap, "Working Paper 153 : Social Media Analysis , Twitter and the London Olympics (A Research Note) Social Media Analysis , Twitter and the London Olympics 2012," 2012, to be published.

[91] P. Burnap, O. F. Rana, N. Avis, M. Williams, W. Housley, A. Edwards, J. Morgan, and L. Sloan, "Detecting tension in online communities with computational Twitter analysis," *Technological Forecasting and Social Change*, May 2013.

[92] P. Burnap, O. Rana, and M. Williams, "COSMOS: Towards an integrated and scalable service for analysing social media on demand," *International Journal of Parallel, Emergent and Distributed Systems*, no. May, pp. 37–41, 2014.

[93] E. Baucom, A. Sanjari, X. Liu, and M. Chen, “Mirroring the real world in social media: Twitter, geolocation, and sentiment analysis,” in *Proceedings of the 2013 International Workshop on Mining Unstructured Big Data Using Natural Language Processing*, ser. UnstructureNLP ’13. New York, NY, USA: ACM, 2013, pp. 61–68.

[94] T. a. Small, “What the Hashtag?” *Information, Communication & Society*, vol. 14, no. 6, pp. 872–895, September 2011.

[95] X. Zhou and L. Chen, “Event detection over twitter social media streams,” *The VLDB Journal*, vol. 23, no. 3, pp. 381–400, Jul. 2013.

[96] R. Li, K. H. Lei, R. Khadiwala, and K. C.-C. Chang, “TEDAS: A Twitter-based Event Detection and Analysis System,” *2012 IEEE 28th International Conference on Data Engineering*, pp. 1273–1276, Apr. 2012.

[97] A. Hermida, S. C. Lewis, and R. Zamith, “Sourcing the Arab Spring: A Case Study of Andy Carvin’s Sources on Twitter During the Tunisian and Egyptian Revolutions,” *Journal of Computer-Mediated Communication*, vol. 19, no. 3, pp. 479–499, Apr. 2014.

[98] J. Bollen, H. Mao, and X. Zeng, “Twitter mood predicts the stock market,” *Journal of Computational Science*, vol. 2, no. 1, pp. 1–8, Mar. 2011.

[99] D. Thom, H. Bosch, S. Koch, M. Worner, and T. Ertl, “Spatiotemporal anomaly detection through visual analysis of geolocated twitter messages,” in *Visualization Symposium (PacificVis), 2012 IEEE Pacific*, Feb 2012, pp. 41–48.

[100] T. Developers, “The Search API.” [Online]. Available: https://dev.twitter.com/rest/public/search

[101] G. Center, “SensePlace2,” 2015. [Online]. Available: http://www.geovista.psu.edu/SensePlace2/

[102] G. M. University, “GeoSocial Gauge Where Innovation Is Tradition,” 2014. [Online]. Available: http://geosocial.gmu.edu/

[103] "TweetViz Twitter Visualiser." [Online]. Available: tweetviz.com

[104] SalesForce, "Radian6." [Online]. Available: http://www.salesforcemarketingcloud.com/

[105] Bitly, "Twitonomy_ Twitter #analytics and much more," 2014. [Online]. Available: http://twitonomy.com/

[106] Lightstream, "Tweetping," 2015. [Online]. Available: http://tweetping.net/

[107] K. H. Leetaru, S. Wang, G. Cao, A. Padmanabhan, and E. Shook, "Mapping the global Twitter heartbeat: The geography of Twitter," *First Monday*, vol. 18, 2013.