Tracking Events at Magnet.me

M.T. Dwars H.L.D. Nijessen R. Wieman

July 10, 2014

TU Delft Supervisor:	Dr. G. Gousios
Magnet.me Supervisor:	Ir. A. Nederlof
BSc Project Coordinator:	Dr.ir. F.F.J. Hermans

Abstract

A lot of interesting information can be extracted from the way users interact with a website. The regular way of extracting this information is through Google Analytics. Unfortunately, Google Analytics has two limitations. First, because Google Analytics uses a *pageview* as unit of measurement, it is unable to describe all user interactions. For example, tracking users on different devices is hard. Second, the reports generated by Google Analytics are not tailored for a specific business. For example, Google Analytics' support for key performance indicators is very limited.

We investigated how to build a system that is capable of providing these deeper insights. An *event* is used as the unit of measurement to provide the flexibility that is required to track all user interactions. We designed the system to cope with the data volume required by Magnet.me. This is done by using MongoDB as storage engine. We have shown that our system satisfies our requirements by performing performance tests.

Contents

1 Introduction

More and more companies would like to know what actions their users perform on their website. Especially when the website constitutes their core business. This is also the case for Magnet.me. They currently lack a system to track and analyze the behavior of users on their website, but they want to know exactly what users do on their website.

The purpose of this report is to present a solution for tracking every user interaction on the website of Magnet.me. We present a model for representing interactions of a user with the website. Additionally, we explain how these models are stored and queried. Finally, we provide a way to deploy the system.

This report is structured as follows. First we discuss the problem more in-depth in chapter 2. Chapter 3 contains the research of evaluating several database systems. In chapter 4 the application is presented and its performance is discussed in chapter 5. The deployment of the application is presented in chapter 6. The process of building the application, the workflow and our reflection are discussed in chapter 7. Finally, chapter 8 concludes the report and provides recommendations for future work.

1.1 About Magnet.me

Magnet.me is a start-up founded in 2011. In September 2012, their website that brings students and potential employers together went live. On this website, students can create a profile. The most important part of that profile is the curriculum vitae, which contains education, working experiences, and extracurricular activities. Companies can promote themselves by creating a 'minisite', describing what they do and who they are. The minisite commonly contains a company description (including photos and videos), a list of vacancies, and a list of internships. Additionally, recruiters of those companies can specify their criteria for potential employees. Magnet.me then facilitates the matching process between students and potential employers by using advanced algorithms. Currently, more than 15,000 students and over 400 employers use Magnet.me.

2 Problem Analysis

Magnet.me currently handles approximately 800 users (8000 page requests) on their website per day, with 100 users (800 page requests) during the peak hour (see Appendix B). The problem is that Magnet.me has very limited options for tracking these users; they currently only use Google Analytics[1]. However, Google Analytics has some restrictions:

- It is built around the concept of page views. As a consequence, it is hard to track interactions that matter to Magnet.me.
- It is not possible to incorporate custom back-end events, such as audit trails or server deployments.
- It lacks the possibility to query events together, which is crucial for knowing exactly what students did from the moment they register on the site until they find a job using Magnet.me.

Therefore Magnet.me wants to have their own system that is able to track the actions of users on their website, in such a way that useful reports based on these user interactions can be generated. This system should overcome the above mentioned restrictions. The information this system is going to produce can be very useful for (for example) Magnet.me's account managers or company's recruiters. See Appendix A for the original project proposal by Magnet.me.

2.1 Use Cases

To futher elaborate on the problem, we describe four possible use cases. Though our main goal is to build the infrastructure for tracking user interactions, these use cases are meant to give an idea of how the system is going to be used.

1. Imagine a student registers in the second year of his bachelor education. He creates his CV and visits some mini sites. He finally receives his bachelor's degree and continues to master's level education; he therefore updates his CV at Magnet.me. After finishing his master's education he finds a job at Grolsch using Magnet.me. The key question now is: How did he find that job? For example, did he also look at jobs at Amstel or Heineken? Or did he perhaps look at companies in a completely different branch, such as Unilever? Did he read testimonials, or did he just look at the company description? Did he change his grade point averages (GPAs) to get more company requests?
2. Apart from the very user/company specific questions mentioned above, it is also interesting to generate aggregated statistics. Examples: Which studies generate the biggest job opportunities? How often do students change their GPAs?
3. Whenever a new version of the website gets deployed, the possibility exists that (minor) errors occur. This might result in an unexpected change of certain user actions. It can be nice to see this in the reports generated by the system. Example: A failing save controller of the CV page might result in an increase of save actions, because the users are going to retry to save their CV.
4. Magnet.me makes use of a chatbot called Magneto[2]. Whenever something noteworthy happens, this chatbot should broadcast that event to the employees of Magnet.me. Example: shouting out loud when the 1000th user registers on the website.

[1]`http://www.google.com/analytics/`

[2]`https://github.com/Magnetme/magneto`

2.2 IT Infrastructure

We strive to make adoption of our application as easy as possible. We investigated the current Magnet.me IT infrastructure, because using technologies that are already used within the company would greatly improve adoption.

The Magnet.me website uses AngularJS[3], a JavaScript framework for building dynamic web applications. The backend is built on top of RESTEasy[4]. RESTEasy is an implementation of the JAX-RS[5] specification. This specification prescribes an API for RESTful Web Services. PostgreSQL[6] is used as data storage. Elasticsearch[7] is used for full-text search. Changes to the data in PostgreSQL get propagated to Elasticsearch. Additionally, they plan to deploy Logstash[8] (with Elasticsearch and Kibana[9]) on short term.

2.3 Requirements

From interviews with the company contact we received several restrictions. The product must comply with several non-functional and functional requirements. We will list these requirements in this section.

2.3.1 Non-Functional Requirements

Non-functional requirements describe the requirements that are related to the operation and deployment of the system. For this system, these are:

1. The system must be implemented in Java (version 7 or 8). Additionally, Project Lombok[10] and Google Guice[11] are mandatory.
2. The third party components for the system must be open source and may not be GPL licensed.
3. The system must be horizontally scalable.
4. Communication with the system must be provided through a REST API.
5. Automated software tests (unit tests, integration tests and performance regression tests) must be performed.
6. The system must be shipped in a Docker[12] container.
7. The system must be deployable on a Linux server running Ubuntu 12.04 (Precise Pangolin)[13].
8. The system must be resilient; it should replicate data to multiple nodes in order to minimize the risk of losing data.
9. The system must accept data from multiple servers.

[3]https://angularjs.org/
[4]http://resteasy.jboss.org/
[5]https://jax-rs-spec.java.net/
[6]http://www.postgresql.org/
[7]http://www.elasticsearch.org/
[8]http://logstash.net/
[9]http://www.elasticsearch.org/overview/kibana/
[10]http://projectlombok.org/
[11]https://code.google.com/p/google-guice/
[12]http://www.docker.com/
[13]http://releases.ubuntu.com/12.04/

2.3.2 Functional Requirements

Functional requirements describe the functionality and behavior of the system, i.e. what the system should be capable of. These are:

10. The system does not precompute metrics, but stores raw data instead and allows to compute arbitrary aggregations on that data.
11. The system should be able to store at least the following events:
    - Actor X of group Y **modifies** entry Z
      "Recruiter(1) of organization(2) modifies CompanyPage(1)"
    - Actor X **visits** subject Y
      "Student(1) visits CompanyPage(3)"
    - Actor X **encounters javascript error** Y
      "Student(1) got error('cannot read property of undefined')"
12. The system must provide several insights, of which at least:
    - click paths through application.
    - entity log (audit trail of e.g. vacancy).

Additionally, the system must be flexible. It should be possible to adapt the system to answer different kinds of questions.

2.4 Existing Solutions

To better understand the problem, we investigated several existing analytics solutions that address the same kind of problem. Each of these solutions below differ slightly and none of them perfectly fits our requirements. However, they provide a good overview of the industry and what is currently possible. The following Software as a Service (SaaS) systems come closest to what we require:

Keen.io Keen.io offers an analytics backend to developers through a REST API. Developers can push events into the system and request certain metrics, such as counts, sums, averages, timeseries and funnels. Keen.io does not provide a general dashboard, so developers have to write their own.

Calq.io Calq.io provides custom analytics for mobile and web applications. They provide several endpoints for tracking different types of events. Additionally, a developer can make use of some predefined properties that Calq.io can interpret for easy analytics. Additionally, they provide an easy dashboard that allows a user to build custom queries.

Mixpanel.com Mixpanel is less generic than Keen.io and Calq.io. It focuses more on funnel analysis and uses predefined concepts, such as visits, users and revenues. They provide a generic dashboard that allows a user to create funnels, timeseries and retention analysis.

3 Evaluating Databases

We want to store various *events* and perform an online analysis on these events. An event represents an action that has taken place at a certain point in time. It is a structured piece of data and it usually contains some properties for more in-depth analysis. An event can be generated by different actors, such as a user, developer, admin or server.

The structure of an event may change over time. This allows for a more exploratory style of data analysis. The dynamic nature of events also requires fairly general query capabilities. For example, the system needs to be able to perform aggregate queries, queries on a unique key and queries on some arbitrary fields. As is common in analytics systems, the number of writes will be much higher than the number of reads. Furthermore, the nature of an event allows us to store them in an append-only fashion; updating an event will rarely be necessary.

In this chapter we will evaluate several storage technologies. We first list our requirements in section 3.1. In section 3.2 we analyse various data storage systems based on these requirements. Finally, in section 3.3 we decide which data store is best suited for this use case.

3.1 Requirements

We identified the following list of requirements that the database should comply with:

1. The database should perform well on schema-less events. An event is structured data, however, the structure of an event may change over time.
2. The database should provide ad hoc aggregation capabilities (e.g. storing pre-computed metrics and serving them when queries arrive is not allowed)
3. The database should be able to handle 'general' search queries efficiently. It should be able to search for specific records by a unique key, search for records by some general key, search for records on an arbitrary field, search for records within a certain range, and so on. Therefore the database should support some kind of indexing.
4. The data should be highly available; for example, by replicating the data.
5. The database should be able to scale horizontally.

Although it is not an official requirement, we favor databases that have proven themselves in a production environment. In general, we have limited our search to the better known databases.

3.1.1 Scale Impression

The system should be able to handle a high number of writes and store a large amount of data. The database needs to cope with far more writes than reads. In this section, we will try to quantify these these statements. In order to do so, we have taken the current situation as a starting point. On an average day, Magnet.me receives approximately 1,000 unique visits. Together, these visitors generate approximately 10,000 page views.

Number of events

We want to keep close track of users. Most of the events will be initiated by an action taken by a user, such as a page visit. We have arbitrarily estimated that the number of events will be the tenfold of the number of page views. Thus, in the current situation we expect 100,000 events per day, the majority of which will arrive between 08:00 and 22:00.

Size of events

Each event will incorporate maybe a dozen fields. These fields will contain fairly short information. A page request event, for example, might incorporate fields for the timestamp, user id, request URI and referer. We have chosen 1 kilobyte as the length of an event, though this is a *very* rough estimate.

Using these numbers, we currently need to store approximately $100,000 \cdot 1\text{kB} \approx 100\text{MB}$ of data each day. These calculations of course disregard database management overhead (such as indices), but then again we are only trying to give an impression. Furthermore, if those $100,000$ events are uniformly distributed over the period from 8:00 to 22:00, we will have to process 2.0 events per second. We already know that load is not uniformly distributed, so the peak load will be much higher. Additionally, due to the expected growth, these numbers will only increase. Taking this growth perspective into account, we will design the system to cope with a tenfold of this data.

3.2 Storage Technology

During the orientation phase we have identified several types of storage technologies. These technologies can be characterized as either relational storage or NoSQL storage. Additionally, we have looked into the Hadoop ecosystem. Based on the requirements in section 3.1 we have made a selection among different types of storage technologies for thorough investigation.

The most commonly used software solution for big data projects is Hadoop. We therefore started with investigating the Apache Hadoop project[1] (currently version 2.2.0), including the Hadoop Distributed File System (HDFS) and Hadoop MapReduce. Furthermore, we have looked into several Hadoop-related projects, such as Hive[2], Pig[3] and Spark[4]. We found out that Hadoop was originally created to store massive amounts of data (e.g. 'web scale data'). Additionally, its MapReduce programming model shows that it was originally created to perform batch processing. Hadoop is optimized for high throughput, not for low latency. Solutions built on the HDFS generally do not support indexing. As a result, all data needs to be processed to answer even the simplest queries. Though Hadoop is a good fit for some use cases, it does not comply with the need to perform online queries.

Afterwards we continued our research by looking into relational databases. However, we soon discovered that they lack some important properties. First of all, we want to store a variety of events and the structure of an event may change any time in the future. This may result in many columns and many changes in the table structure, which is unpractical. Secondly, relational databases are generally hard to scale horizontally because of the pessimistic concurrency schemes.

We ended our research with NoSQL databases. These differ from relational databases in their data model and are commonly classified into three groups:

[1]`http://hadoop.apache.org/`
[2]`http://hive.apache.org/`
[3]`http://pig.apache.org/`
[4]`http://spark.apache.org/`

Key-value stores Data is stored data in the form of a key-value pair, where the key is unique and the value is a blob (unstructured, unanalyzed data).

Document stores Like key-value stores, data is stored as a key-value pair. However, the value can now be stored structured and can be queried. The key of such a 'document' usually gets generated by the data store itself.

Column-family stores Data is stored in rows, with the key of that row referring to data in multiple columns. These columns may vary per row, but the common columns are called column families. The data in these column families is often accessed together.

Key-value stores do not provide the required query possibilities, because the content in the value cannot be queried efficiently (as it is not being analysed by the DBMS). However, document stores and column family are both candidates.

We have limited ourselves to the following storage technologies. While selecting these technologies, we have tried to cover a broad spectrum of storage types while keeping the requirements in mind. Each of these technologies will be explained in the following sections:

- MongoDB
- Druid
- Elasticsearch
- Cassandra

3.2.1 MongoDB

MongoDB[5] (currently version 2.6) is a document-oriented database that stores Binary JSON (BSON) documents in a schema-less manner. It allows explicit indexing and supports indexes on fields and subfields. Documents are stored in *collections* which can be seen as the equivalent of a *table* in a RDBMS.

Query interface

MongoDB uses its own binary protocol over TCP to communicate [34]. There are a libraries available in a number of programming languages including Java [33]. JavaScript Object Notation (JSON) is used to express a query.

Each query retrieves documents per collection. Filtering is supported by using boolean queries on fields, subdocuments and arrays. Aggregation is supported through *Aggregation Pipelines* or by using MapReduce [26]. The first performs better, because it runs as native code. The latter performs relatively bad, because it is processed using an embedded JavaScript engine. Furthermore the conversion between the internally used binary format and JSON, being used by the JavaScript engine, causes overhead [4]. Both can use indexes to filter data that does not have to be aggregated [32].

Concurrency Control

MongoDB uses a readers-write lock [31]. When a read lock exists, all reads may take place. However, when a write lock exists, a single write operation holds the lock exclusively, and no other read or write operations may share the lock. MongoDB implements locks on a per-database basis. Collection and even document locking will be implemented in the near future.

[5] `http://www.mongodb.org/`

Architecture

MongoDB uses the concept of sharding to scale horizontally. Sharding is supported through the configuration of sharded clusters [37]. Each sharded cluster has the following members:

Shard A shard stores data. Data can be spread over multiple shards.

Query router A query router interfaces with client applications and directs operations to the appropriate shard or shards. The query router processes and targets operations to shards and then returns results to the clients. A sharded cluster can contain more than one query router to divide the client request load. A client sends requests to one query router. Most sharded cluster have many query routers.

Config server store the clusters metadata. This data contains a mapping of the clusters data set to the shards. The query router uses this metadata to target operations to specific shards. Production sharded clusters have exactly 3 config servers.

A schematic overview of the sharding architecture can be seen in figure 3.1

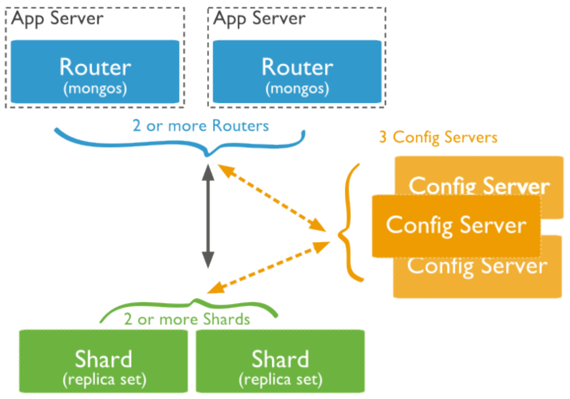


Figure 3.1: Schematic overview of sharding in MongoDB [37]

Sharding partitions a collections data by the shard key. MongoDB uses either range based partitioning and hash based partitioning.

Using range based partitioning documents with *close* shard key values are likely to be in the same chunk, and therefore on the same shard. This makes range based queries fast. However, range based partitioning can result in an uneven distribution of data, which may negate some of the benefits of sharding. For example, if the shard key is a linearly increasing field, such as time, then all requests for a given time range will map to the same chunk, and thus the same shard. In this situation, a small set of shards may receive the majority of requests and the system would not scale very well.

Hash based partitioning, by contrast, ensures an even distribution of data at the expense of efficient range queries. Hashed key values results in random distribution of data across chunks and therefore shards. But random distribution makes it more likely that a range query on the shard key will not be able to target a few shards but would more likely query every shard in order to return a result.

Replication and Consistency

MongoDB support replication through *replica sets* [36]. A replica set is a group of MongoDB instances that provides redundancy and high availability, which can have the following members:

Primary The primary receives all write operations which it records to its operation log. A primary can also receive read operations. Each replication set has only one primary.

Secondary A secondary maintains a copy of the primary's data set. It does so by applying the operations from the primary's operation log to its own data set asynchronously. Secondaries cannot receive writes but can receive reads. Each replica can have one or more secondaries.

A schematic overview of the replication architecture can be seen in figure 3.2

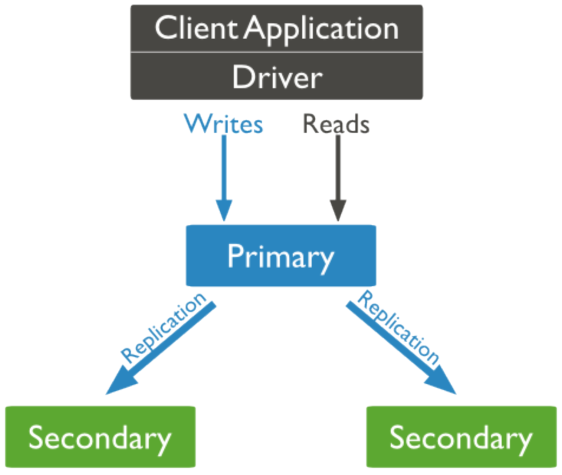


Figure 3.2: Schematic overview of replication in MongoDB [36]

When a primary becomes unavailable the remaining secondaries elect a new primary. If an even number of secondaries is being used an arbiter should be added to the replica set. An arbiter doesn't store data. Its only purpose is to act as a tie-breaker in elections. Replica sets can have a maximum of 12 nodes and a maximum of 7 voting secondaries.

MongoDB is strongly consistent by default when you are only reading from primaries. On the other hand it is possible to configure MongoDB to be able to read from secondaries. In this case MongoDB is eventually consistent [5].

3.2.2 Druid

Druid[6] (currently version 0.6.105) is a column-oriented datastore which is built for real-time exploratory analytics on large datasets. It is designed with a focus on 100% uptime regardless of e.g. machine failure or code deployment. Furthermore it supports realtime ingestion which means that data is immediately available for querying. Druid makes the assumption that written data is immutable. This is based on the fact that Druid is being used to store events, which are per definition immutable [21].

Query interface

Druid provides a REST API in which JSON is used to express the query. It supports the following query types [17]:

- Group by

[6]`http://druid.io/`

- Search
- Segment metadata
- Time boundary
- Timeseries
- TopN

Group by is used for grouping by a certain field and aggregating data. Search queries provide for flexible matching (case sensitivity, fragment search). Segment metadata queries return metadata per segment (a collection of records representing a time interval). Time boundary queries return the earliest and latest data points of a data set. Timeseries queries group data by a certain timeperiod like a day, a month or a second. A topN query can be thought of as an group by query over a single field with a certain ordering. TopN queries are faster than group by queries.

Concurrency Control

Druid uses multiversion concurrency control (MVCC) [21]. This means that each client trying to read will see a snapshot of the system at a particular instant in time. Changes done by a write will not be seen by other users until the write has been completed.

Architecture

Druid has the following five different node types [16]:

Historical Handles storage and querying on hystorical data.

Realtime Listens to a datastream and insert and makes the data immediately available for querying. Aged data is pushed from Realtime nodes to Historical nodes.

Coordinator Monitors the grouping of historical nodes to ensure that data is available, replicated and in a generally "optimal" configuration.

Broker Receives queries from clients and forwards them to Historical and Realtime nodes. Afterwards it merges the results and returns it to the client.

Indexer Is used to load batch and realtime data into the system.

Druid has three external dependencies [21]:

- Zookeeper to register all nodes and which data can be found on which nodes.
- A MySQL instance for storing metadata about the data in the system.
- Something that is called *deep storage*' which is being used to save *historical data*. For example: S3[7], HDFS or Cassandra can be used as deep storage.

An overview of the total architecture can be found in figure 3.3.

Replication and Consistency

Druid supports replication in two ways. Data on Realtime nodes can be replicated by letting multiple Realtime nodes listen to the same event stream. This creates a replication of events. Coordinator nodes may tell different Historical nodes to load a copy of the same data. Historical nodes load data

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

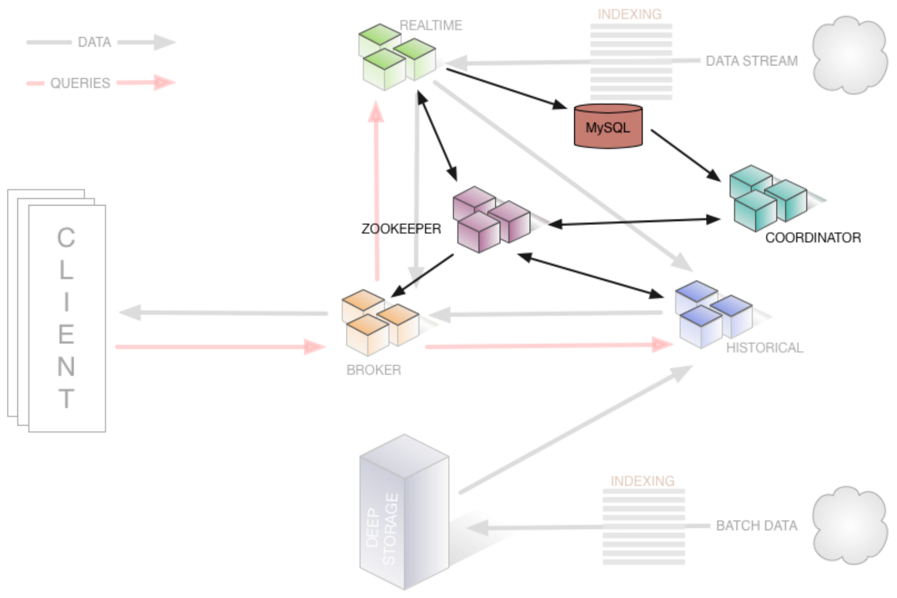


Figure 3.3: Schematic overview of the architecture of Druid [16]

from deep storage. Deep storage is shared storage, so when a Historical node fails another Historical node can load the "lost" data from deep storage [21].

Druid is fullly read-consistent, because data is saved as immutable chunks. The tradeoff is that Druid has limited semantics for write and update operations [21].

3.2.3 Elasticsearch

Elasticsearch[8] (currently version 1.1.1) is a distributed document store powered by Apache Lucene[9]. Its name gives the impression that it is solely a search engine, but it has been used as primary storage for some use cases [2]. Using Elasticsearch as primary storage, however, brings along some limitations. For example, Elasticsearch lacks robust backup functionality. Furthermore, recovery from a corrupt state (power failure, out of disk space) is not a straightforward process [3].

In order to discuss Elasticsearch, we need to introduce some terminology. Documents in Elasticsearch are stored in an index. This is can be compared to storing rows in a table in the relational model. Each document in Elasticsearch has a type and each type has a mapping. A mapping can be compared to a schema definition in the relational model. A mapping defines fields and their data types. When a document is added, Elasticsearch dynamically generates a mapping. In doing so, it guesses the types of the fields. By default, every field in a document is indexed. The kind of index that is maintained depends on the field type [24].

Elasticsearch is impressively easy to setup. When a node is added to the cluster it uses multicast to discover other nodes. The nodes in a cluster together elect a master node. This easy setup makes it possible to run Elasticsearch on a laptop with just a few megabytes of data, or scale out to hundreds of servers and petabytes of data [18].

[8]`http://www.elasticsearch.org/`
[9]`https://lucene.apache.org/core/`

Netflix used Elasticsearch to create a system that is similar to what we are trying to achieve. They created a system to log more than 1.5 million events per second during peak hours. These events consist of log messages, user activity records, system operational data, or any arbitrary data that their systems need to collect for business, product, and operational analysis. Though the infrastructure they use is different, it illustrates that Elasticsearch is able to handle massive amounts of data [7].

Query interface

Elasticsearch provides a REST API and clients for several programming languages. Furthermore, there is a Java client available that bypasses the REST interface [6]. Both interfaces support a flexible query language named 'Query DSL'. When using Query DSL, a query is specified using a JSON request body [24]. This Query DSL supports filtering, (full-text) searching, boolean queries and aggregations among others.

Elasticsearch is fundamentally flat and promotes denormalizing your data. However, it provides some support for managing relations using the concepts of *nested documents* and *parent-child relationships*. Nested documents are treated as individual components but are stored together with the outer documents (on the same shard) to improve read performance. Parent-child relationships provide a looser coupling in which you are more free to update or delete child documents. The disadvantage of parent-child relationships is that they are slightly less performant [19].

Concurrency Control

Elasticsearch uses optimistic concurrency control [20]. This method assumes that conflicts are unlikely to happen and does not block operations from being attempted. However, if the underlying data has been modified between reading and writing, the update will fail. It is then up to the application to decide how it should resolve the conflict.

Events, by their append-only nature, do not require a pessimistic consistency scheme. An optimistic locking strategy suits our needs much better, as an optimistic locking strategy is easier to scale horizontally and sacrifices consistency guarantees for better write performance.

Architecture

Each document in Elasticsearch is logically stored in an index, whereas physically the document lives inside a shard. When an index is created, the number of shards and shard replicas is defined. Replication provides redundancy in case of a node failure. Furthermore, replication can be used to speed up retrieval queries. A hash function is used to determine to which shard a document belongs. As a consequence, the number of shards for a given index can not be changed after the index is created. Elasticsearch makes sure the shards are evenly distributed over the cluster. In case a node fails, shards are automatically reorganized.

One node in the cluster is elected to be the master node. The master node is in charge of managing cluster-wide changes like creating or deleting an index, or adding or removing a node from the cluster. The master node does not need to be involved in document level changes or searches, which means that having just one master node will not become a bottleneck as traffic grows. Any node can become the master [24].

Every node in Elasticsearch knows where each document lives and can forward the request to the node that holds the data we are interested in. If we direct a request at node A, then that node will be responsible for dispatching the request, gathering the response and returning it to the client.

Replication and Consistency

By default, after writing, the primary shard will wait for successful responses from the replica shards before returning. Optionally, Elasticsearch can be configured to perform the replication asynchronously. That is, a return is sent as soon as the request has been executed on the primary shard.

3.2.4 Cassandra

Apache Cassandra[10] (currently version 2.0.7) is a column-oriented store, originally developed by Facebook and open sourced in 2008 [10]. It uses the BigTable [13] data model, but employs Amazon's Dynamo [14] scheme for data distribution and clustering [25].

Cassandra excels at high-volume real-time transaction processing. It always appends new data, which makes writes very fast, but the disadvantage is the need to compact the space later [22].

A common use case for Cassandra is real-time analytics. However, its query abilities are rather inflexible. Arbitrary aggregates are not possible. A lot of power comes from precomputed aggregates. This provides very fast lookups, but is inflexible and hard to change. Furthermore, the user can only filter on previously indexed fields.

Query Interface

Cassandra has its own query language, the Cassandra Query Language (CQL) [9]. This query language is comparable to the Structured Query Language (SQL), as many functions of SQL are also available in CQL. The big advantage of this is that the learning curve for querying the data store will not be very steep. There are various ways of interfacing with the database using various programming languages; libraries are provided by Datastax[11].

Concurrency Control

Cassandra is able to handle high-volume writes because it avoids the overhead of many concurrency control schemes. It does not implement any locking or multi-version concurrency control scheme. Instead, it is up to the application developer to handle concurrent writes. The application should provide a timestamp which is used to determine which write is the newest.

Architecture

Cassandra handles big data workloads over multiple nodes, without a single point of failure [8]. They implement a peer-to-peer distributed system in order to do so. In that system, all nodes are the same and the data is distributed between those nodes.

It is possible to connect to any node in any data center to perform query operations using CQL. Read/write requests can be handled by any node; the node receiving the request acts as a coordinator for that request, meaning it determines which nodes should actually handle the request. Each node has a Partitioner; this is basically a hash function, which determines where the first copy of data is put. Additionally, each node has a Snitch, which determines which data centers and racks are written to and read from [12]. It is responsible for knowing the location of nodes within the network topology and distributing replicas by grouping machines into data centers and racks [8].

[10]`http://cassandra.apache.org/`
[11]`https://github.com/datastax`

Replication and Consistency

Cassandra implements two replica strategies: one for a single data center and a more advanced one for replicating over multiple data centers [11]. The basic idea is that a user-defined number of replicas will be created of every row in the database. The replicas are distributed over the various nodes of a cluster by using a partitioner. There is no primary or master replica; every replica is equally important. When using the advanced strategy for multiple data centers, you can define how many replicas you want in *every* data center.

Cassandra implements eventual consistency, complemented with *tunable* consistency. This means the client application is allowed to decide how consistent the requested data should be, for any read or write operation. The client can thus specify whether it prefers fast response times or accurate data. Additionally, it is also possible to configure consistency on a cluster or data center.

3.3 Conclusion

Comparing databases is hard, especially in the NoSQL realm. Most databases are under heavy development and undergo a lot of functionality changes. As a consequence, they are sometimes poorly documented. Finally, a lot of claims about database performance are not quantified or outdated.

	RDBMS	Cassandra	Hadoop	Druid	ElasticSearch	MongoDB
Indexes	++	++	- -	++	++	++
Schemaless	- -	+	++	+	+	++
Arbitrary aggregates	++	- -	++	+	+	++
Horizontal scaling	- -	++	++	++	++	+
Documentation	++	-	-	- -	- -	++
Setup complexity	++	++	-	- -	++	-
Data model complexity	++	- -	++	-	+	++

Table 3.1: Comparison of databases/datastores. Properties above the line are strict requirements whereas below the line are desirable properties.

In table 3.1 the databases and datastores are evaluated on several key properties. An RDBMS is not schemaless and does not scale horizontally (requirement 5), so it is not an option. Furthermore, Cassandra does not fulfill our requirements because of the lack of support for arbitrary aggregates (requirement 2). Hadoop is not the right solution, because it does not support indexes (requirement 3) so efficient queries are not possible. Druid, Elasticsearch and MongoDB satisfy all of our requirements. Druid, however, has a lot of external dependencies which makes the setup very complex. Elasticsearch, on the other hand, is very easy to setup. MongoDB is less scalable than Elasticsearch because of its "centralized" primary - secondary structure, but we think that it scales more than enough to be used in the Magnet.me setup. Besides that it has great support for aggregations using MapReduce and the Aggregation Pipeline. During intial tests with Elasticsearch we found out that Elasticsearch is really hard to work with because the functionality and the internals are badly documented. In particular the Java driver is not documented at all. Furthermore, the query language is very complex and it turned out that using aggregations only resulted in approximations. We therefore decided to choose MongoDB as backing datastore.

4 The Application: Dolphin

In this section we discuss Dolphin, the developed application. We first describe its functionality, including an overview of the supported operations and methods to obtain insights. Afterwards, we dive into the architecture of the system and the implementation of the discussed insights. Next, we describe how we guarantee the quality of the product. Finally, we briefly describe how MongoDB should be configured to meet the scalability and availability requirements.

4.1 Functionality

As mentioned in the research, we store user interactions as events. These events do not contain any domain knowledge, because this knowledge is already captured by the Magnet.me application. The properties of an event type are dynamic and change over time. However, an event always contains a timestamp and information about the actor. The actor is the initiator of the event, for example a user, developer, admin or server. Additionally, every event is stored in a collection. A collection contains multiple events of the same type. JSON is used to describe an event, because it perfectly captures its dynamic nature.

Listing 4.1: Example of an event (formatted as JSON)

```json
{
  "timestamp" : "2014-07-17T10:08:33Z",
  "actor" : {
    "id" : "user-1",
    "groupId" : "group-2"
  },
  "loadtime" : 200
}
```

An example of such an event can be seen in listing 4.1. This event contains all mandatory properties (`actor` and `timestamp`). Additionally, the event contains a `loadtime` property. A developer can add properties like these to an event at will. When storing this event, Dolphin validates whether the mandatory are actually present.

As defined in requirement 11, there are several types of events: visited, modified, and JavaScript error. All these events occur on an entity. Therefore, `eventId` and `eventType` are mandatory fields on these events. Consequently, Dolphin also performs validation on these fields.

In order to gain insights based on the persisted events, Dolphin provides several operations. These operations follow from the expected use cases (see section 2.1) and are categorized as metrics, time series and funnels:

- **Metrics** are simple operations that can be applied to the data. We have included the following five operations: average, count, maximum, minimum, and sum. These operations are based on a field (except for the count operation). Furthermore, it is possible to group the data based on a field. Finally, a filter can be applied to narrow the results down. For example, using the average operation, one can compute the average load time (field) per user (group) for the last week (filter).

- A **time series** is a series of metrics for a given granularity. The following granularities are supported: second, minute, hour, day, month, and year. For example, using the average metric, one can compute the average load time per user for the last week *per day* (granularity).
- Something totally different is a **funnel**. A funnel consists of a sequence of steps that the actor performs. For example, an actor might register, login and post a comment. Funnel analysis gives insight into the number of (distinct) actors that finish each step and how many of them proceed to the next step. It provides answers to questions such as: How many users completed all registration steps? How many users that registered also logged in?

4.2 Architecture

Dolphin uses many dependencies of which a full list (including version numbers and URLs) can be found in Appendix C. One of these dependencies is Dropwizard, a framework that allows you to easily create RESTful web applications in Java. Dropwizard uses embedded Jetty as servlet container, Jersey for REST (Jersey implements JAX-RS) and Jackson for parsing JSON strings.

All REST functionality is put in resources that are being registered in Jersey when Dolphin gets started. Resources are basically the endpoints of the REST API. The resources use their corresponding Data Access Objects (DAOs) to store and retrieve objects from the persistence layer. Entities are being transformed from and to models.

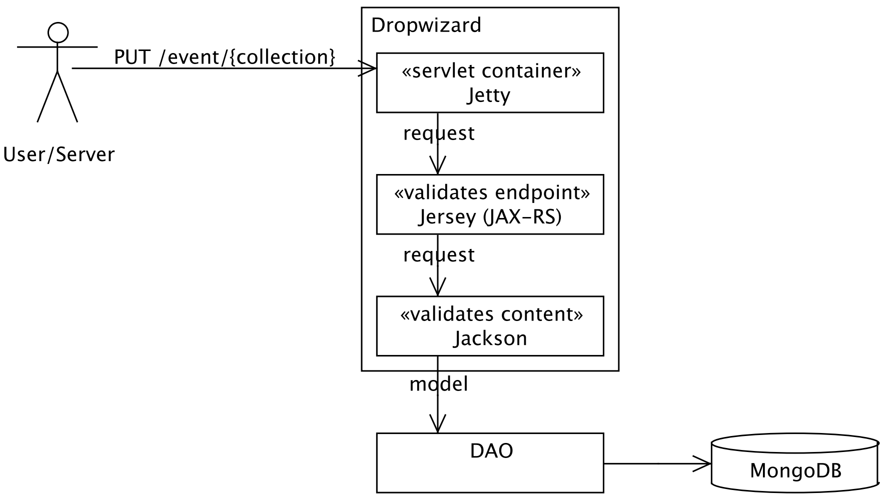

Dropwizard
User/Server
PUT /event/{collection}
«servlet container» Jetty
request
«validates endpoint» Jersey (JAX-RS)
request
«validates content» Jackson
model
DAO
MongoDB

Figure 4.1: Flow of a request

To insert an event into a collection, the client (e.g. a browser, webserver, or mobile application) makes an HTTP PUT request to the corresponding API endpoint. The endpoint depends on the collection to which the event needs to be added. Jetty, the servlet container, accepts the request and passes it on to Jersey. Jersey, an implementation of the JAX-RS specification, performs validation on the request-level. For example, if the URI or HTTP method are not supported an error is returned. If no error occured, the request body is passed to Jackson to convert the JSON to Java objects. In doing so, Jackson implicitly validates the body of the request. If the body was invalid an error is returned. Otherwise, depending on the resource, different actions are taken. For example, to insert a new event, the domain model gets converted to a MongoDB `DBObject` that is inserted into the MongoDB collection. Figure 4.1 summarizes this process.

4.2.1 Domain Models

As mentioned before, the JSON in the body of a request gets converted into a Plain Old Java Object (POJO). Requests to the generic event endpoints consume and produce `Event` POJOs. All explicitly defined event types have their own endpoint and can therefore be translated into their corresponding POJOs (`VisitedEvent`, `ModifiedEvent`, and `JavaScriptErrorEvent`). The actor is also represented as a POJO. Figure 4.2 shows the relationship of these classes.

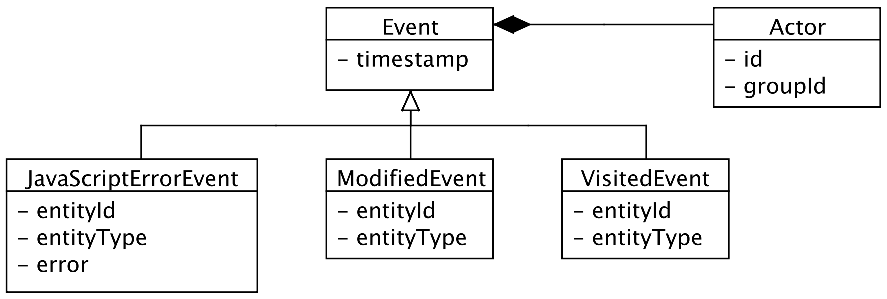


Figure 4.2: Class diagram of special events

4.2.2 Communication

Communication with the system happens through a REST API (requirement 4). The REST API consumes and produces data in JSON format. An example request for inserting an event is shown in listing 4.2. Appendix D contains the full documentation for this API.

Listing 4.2: Example of an HTTP PUT request for inserting an event

```
PUT /event/example HTTP/1.1
Host: localhost
Content-Type: application/json

{
  "timestamp" : "2014-07-17T10:08:33Z",
  "actor" : {
    "id" : "user-1",
    "groupId" : "group-2"
  },
  "loadtime" : 200
}
```

Listing 4.3 shows the response from the request in listing 4.2. Note the `Location` header that contains the Uniform Resource Identifier (URI) of the newly created event.

Listing 4.3: Example of an HTTP response when inserting an event

```
HTTP/1.1 201 Created
Date: Tue, 01 Jul 2014 11:50:13 GMT
Access-Control-Allow-Origin: *
Location: http://localhost/event/example/53b2a07530045d07c2d68e05
Content-Type: application/json
Content-Length: 0
```

4.3 Implementation

Dolphin makes extensively use of MongoDB's Aggregation Pipeline [27]. The Aggregation Pipeline accepts a series of operations that are executed sequentially. Only the first stage can use indices when filtering or sorting data.

Metrics and time series are implemented using this Aggregation Pipeline. The specified filter is directly mapped to the `$match` operator of the pipeline. Additionally, if the request contains a `from` or `to` field, these get appended to the filter. After filtering, the `$group` operator is used to group by a field and compute an aggregation value (e.g. sum, min, max). For time series, the data is grouped by timestamp using the granularity specified in the request. This is implemented using the Date Aggregation Operators of MongoDB [29].

Funnels are implemented differently. In each funnel step a distinct search is done on the actor field using the provided filter. All resulting actor values are then passed to the next step that uses these values to filter only the actors that were found in the previous step.

The distinct search can be implemented using the `distinct` operation of MongoDB [30]. This command returns a document that contains all resulting distinct values. However, MongoDB has a maximum document size of 16MB. This limit was reached when calculating a funnel of about 7 million events. This also required a huge amount of memory to process all the data. A workaround for this problem is to use the Aggregation Pipeline in combination with a cursor that returns the results in batches.

The `distinct` operation makes better use of indices than the Aggregation Pipeline. That is why the funnel operation first tries to use `distinct`. This command fails quickly when the expected result size is bigger than 16MB. When it fails, Dolphin falls back to the Aggregation Pipeline. The results are then processed in batches by using a cursor, while partitioning the results in subsets of less than 16MB. All subsets are processed in parallel through the remaining funnel steps. Each step will only filter the actors it got as input, so the result of each step will always be less than 16MB. Therefore each remaining funnel step uses the more efficient `distinct` operation.

4.4 Quality Assurance

To ensure the quality of our product, we used several measures during the development. We have implemented a comprehensive automated test suite and our code has been validated twice by the Software Improvement Group (SIG)[1].

[1] `http://www.sig.eu/nl/`

4.4.1 Automated Software Tests

The whole test suit is designed to run automated in order to meet requirement 5. This way, there can be no hesitation to run the tests during development. The test suite actually consists of three types of tests: unit tests, integration tests and performance regression tests. Unit tests only test small units of code, in essence only one method at a time. Integration tests are used to actually use MongoDB during tests. Performance regression tests show whether an implemented code change has (significant) impact on the performance of the application. We used IntelliJ's coverage plugin in combination with the unit tests and integration tests to inspect whether all (important) code is being used by the tests. We achieved a total line coverage of approximately 86%. In this section we will discuss how we use the various types of testing.

Unit Tests

The first step in the testing process is running unit tests. In these tests we mocked all external dependencies of the class under test using Mockito. We also used Magnet.me's build server running Jenkins 1.567[2] for continuous unit testing. Our set of unit tests aims to test:

- Event models
  - Do the constructors create the correct object?
  - Do they get the right timestamp?
  - DAOs
    - Do they ensure indexes on the required fields of the events?
    - Do they use the correct MongoDB collection?
    - Is the timestamp formatted correctly?
  - Resources
    - Do they use the right DAO?
    - When inserting an event, do they return response code 201?
- Metrics
  - DAOs
    - Do they use the correct MongoDB collection?
    - Do they invoke the correct aggregation pipeline on MongoDB?
  - Resources
    - Do they invoke the right method with the right parameter on the DAO?

Integration Tests

As unit tests cannot determine whether MongoDB actually returns the expected results, we have also implemented some integration tests. These tests spawn an embedded MongoDB server using Flapdoodle Embedded MongoDB. They also launch the application, so that actual REST calls can be made.

[2] `http://jenkins-ci.org/`

Performance Regression Test

In order to determine whether a code change has (significant) impact on the performance of the application, we created a performance regression test. This test runs for x minutes with y threads on server z and tries to put as many events as possible. The results are collected in a directory and can be plotted.

We use Docker for running this test. This has several advantages. First, Docker ensures identical environments (in terms of installed software) for each test. This improves reliability and makes it easier to compare the results of multiple tests. Second, Docker makes it possible to setup and teardown a new environment within seconds. A disadvantage of running through Docker is that it may impact the performance. However, since this is a regression test, we are mainly interested in relative changes.

Our setup basically boils down to a simple Docker container (see section 6.1.2) with Apache JMeter 2.11[3] and a test runner (`bash` script). The container also contains a template for JMeter with some parameters that are being filled in at runtime using UNIX `sed`. JMeter is designed to load test functional behavior and measure performance. It was originally designed for testing Web Applications [1].

4.4.2 Software Improvement Group Recommendations

The SIG reviewed our code on maintainability. In total two reviews were performed, once in week 8 and once in week 10. The recommendations from both evaluations are included below.

> De code van het systeem scoort ruim 4 sterren op ons onderhoudbaarheidsmodel, wat betekent dat de code bovengemiddeld onderhoudbaar is. De hoogste score is net niet behaald door een lagere score voor Unit Interfacing.
>
> Voor Unit Interfacing wordt er gekeken naar het percentage code in units met een bovengemiddeld aantal parameters. Doorgaans duidt een bovengemiddeld aantal parameters op een gebrek aan abstractie. Daarnaast leidt een groot aantal parameters nogal eens tot verwarring in het aanroepen van de methode en in de meeste gevallen ook tot langere en complexere methoden. Wat hier opvalt is dat de methods met de meeste parameters zich bevinden in de 'MetricsDao'-class, veel van de methodes in deze class krijgen twee 'Optional' objecten als parameters. Omdat het in beide gevallen gaat om een 'String' zal een ontwikkelaar goed moeten onthouden dat de eerste parameter de 'matcher' en de tweede parameter een 'groupField' is. Het is onduidelijk waarom hier geen gebruik wordt gemaakt van een 'MetricsInput' (of soortgelijk) object. Het is aan te raden dit te documenteren of een type zoals 'MetricsInput' te introduceren.
>
> Over het algemeen scoort de code bovengemiddeld, hopelijk lukt het om dit niveau te behouden tijdens de rest van de ontwikkelfase. De aanwezigheid van test-code is in ieder geval veelbelovend, het is opvallend dat er meer test-code dan productie-code aanwezig is in de codebase. Erg goed! Hopelijk zal het volume van de test-code ook groeien op het moment dat er nieuwe functionaliteit toegevoegd wordt.

The main issue was the number of parameters of certain methods, especially in the `MetricsDao` class. This class took several parameters of which the last two were of type `Optional<String>`. Because this may lead to confusion we replaced these last two parameters by a parameter of type `MetricsInput`.

Two weeks later SIG performed another review. Besides addressing the feedback from the first evaluation we also worked on several new features. We were glad to here that the total score remained equal. However, Unit Interfacing was still mentioned for improvement. Apparently we failed to address the issue for every method, so some methods remain to be improved.

[3] `http://jmeter.apache.org/`

In de tweede upload zien we dat het codevolume is gegroeid terwijl de score voor onderhoudbaarheid ongeveer gelijk is gebleven.

Bij Unit Interfacing zien we net als bij de eerste upload dat er lijsten parameters zijn die meerdere keren voorkomen. Het eerder genoemde voorbeeld van de lijst “collection, field, matcher, groupField” is daar n van. Het introduceren van een object voor bij elkaar horende velden voorkomt dat je jezelf steeds herhaalt, en verlaagt op die manier de kans op fouten.

Uit deze observaties kunnen we concluderen dat een deel van de aanbevelingen van de vorige evaluatie zijn meegenomen in het ontwikkeltraject. Het is goed om te zien dat naast een verbetering in de onderhoudbaarheid er ook nog steeds een stijging in het volume van de test-code te zien is.

4.5 MongoDB Configuration

Our system is built on top of MongoDB. To meet the requirements, a good configuration is necessary. In this section we provide a recommendation on the production setup of MongoDB. We first indicate how to configure replica sets. Afterwards, we discuss the shard configuration and the use of config servers. Finally, we discuss the importance of having the right indices.

4.5.1 Replica Sets

MongoDB uses replica sets to provide redundancy and high availability. A replica set consists of a primary and zero or more secondaries. Writes are always done to the primary. Updates are asynchronously propagated to the secondaries through the oplog. If the current primary node becomes inaccessible, the other members can elect a new primary.

We propose using a three-node replica set. At least two nodes are needed to provide redundancy. However, if one of the two nodes becomes unavailable, both nodes will get demoted to secondaries and the database will no longer accept writes. Using three nodes instead of two keeps the cluster running in case any node fails. In general, using an odd number of members ensures that the replica set is always able to elect a primary [35]. Thus, a three-node replica set is the smallest production-safe setup.

4.5.2 Shards

At the scale at which Magnet.me currently operates, there will be no need for sharding. However, if the data size grows beyond what a single machine can handle, or if the query performance drops beyond what is acceptable, MongoDB can be put in a sharded configuration. An essential part of this configuration is choosing the correct shard key.

We propose using a hash of the `_id` field as the shard key. When using the automatically generated `_id`, the responsibility of generating unique ID’s is put at the database layer. Additionally, it distributes the data evenly across the shards. This allows queries to be distributed across the shards and can greatly increase query performance.

4.5.3 Config Servers

In a sharded setup, config servers are used to store metadata. This data is used by the `mongos` instances, which are used to route reads and writes. If the cluster has a single config server, then that config server is a single point of failure [28]. It is strongly adivised to run three config servers for production deployments. If one server goes down, the data is still writable. If two servers go down, the data is no longer writable but remains readable.

4.5.4 Indices

Our system heavily relies on indices for achieving good performance. Because the queries are not known beforehand, the system only ensures that indices exist on certain pre-defined fields. For example, in the predefined events (see requirements in section 2.3.2), an actor object is required. This object is validated at insertion and an index is created if one does not yet exist.

If in the future multiple queries are executed on other fields than the one already indexed, the user should create an index. These indices can be created on the background. This takes longer to complete, but has less impact on performance of the system. As the data size grows, indices become more important.

5 Performance Evaluation

To see how well the system actually performs we executed several performance tests. The goal of these tests was to get an impression of what the system is capable of. Additionally, we wanted to expose possible bottlenecks and performance hiccups.

Consequently, we have not tried to mirror a real-world setup. For example, to prevent MongoDB and Dolphin from competing for resources, both applications should be run on separate machines. These tests are merely meant to gives us some insight into the performance that *can* be achieved. We are therefore mainly investigated the effect of the change of a parameter.

We have used two different test setups which we will refer to as *test setup A* and *test setup B*. We start by describing the insertion performance on test setup A. We then continue with test setup B, where we tested the insertion performance, the impact of indices on insert performance, and the insert performance when indexing.

We did not reboot the test system between tests, nor did we restart the applications (Dolphin, Mongo). We did clean the database after each test. We also ran some tests before we started reporting (to warm up the application). Unless stated otherwise, Dolphin was executed with the default JVM heap parameters. The default initial heap size is 128MB and the default maximum heap size is 2048MB. JMeter was configured with a heap size of 512MB (both initial and maximum).

5.1 Test Setup A

In this setup, an HP Pavilion dv7-6c40ed with SSD (Samsung 840 EVO 250GB) running Windows 8 was used. This machine acted as server and ran both the database (MongoDB) and the REST API (Dolphin). To reduce interference, the tests were controlled by a separate machine. This machine, a MacBook Pro Retina, ran Mac OS X 10.9.3. The machines were connected through a Category 5e Ethernet cable. Additionally, a Thunderbolt to Ethernet converter was used. Both machines were configured with Java SE Runtime Environment (buid 1.8.0_05-b13) and Java HotSpot VM (build 25.5-b02).

5.1.1 Insertion Performance

We have inserted events through 200 threads over the course of 30 minutes. The average throughput was 5306 Requests Per Second (RPS). The average response time was 30ms with a standard deviation of 47ms. These figures are summarized in table 5.1.

Metric	**Measurement**
Average Load Time	30ms
Max Reponse Time	5046ms
Load Time St.Dev	47ms
Throughput	5306 RPS
Bandwidth	1155.6 kB/s

Table 5.1: Summary Report

Interestingly, the maximum response time exceeded 5000ms. Figure 5.1 dispays a histogram of the resposne times. From this figure it is clear that most requests return within a second.

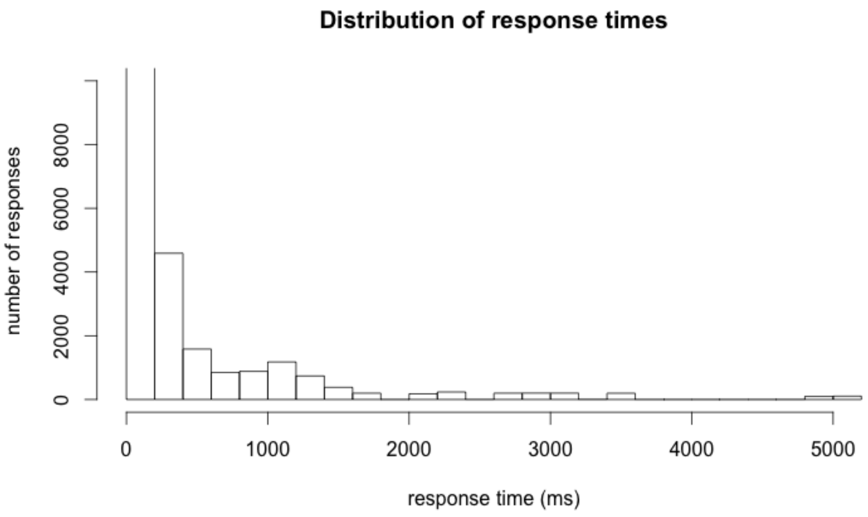


Figure 5.1: Histogram of response times

A plot of the response times over time reveals the real issue. Approximately every minute a spike is observed. We expect that these spikes are caused by the garbage collector. However, we could not confirm this relation, because we were unable to acurately track the garbage collector. Furthermore, as can be seen in the next section, these spikes were only present in test setup A.

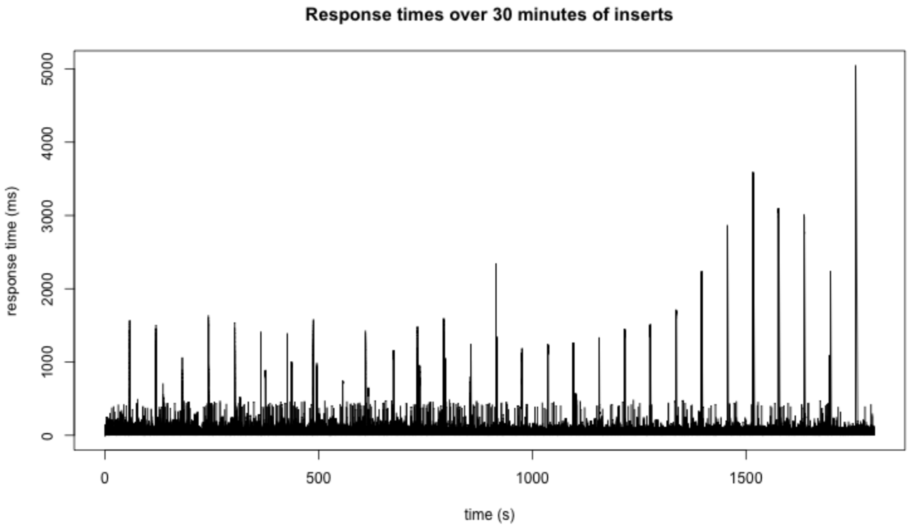


Figure 5.2: Response time over time

5.2 Test Setup B

In this setup, a MacBook Air (mid 2012) running Mac OS X 10.9.4 was used. This machine acted as server and ran both the database (MongoDB) and the REST API (Dolphin). The test was controlled the same machine as in the previous setup. The machines were connected through a Category 6 Ethernet cable. Additionally, both machines used a Thunderbolt to Ethernet converter. Both machines were configured with Java SE Runtime Environment (buid 1.8.0_05-b13) and Java HotSpot VM (build 25.5-b02).

5.2.1 Insertion Performance

We started off with a basic test setup. We configured JMeter to run for 10 minutes with 200 threads. It puts only bare events to the registration collection. These events contain just the required actor objects with a randomized `id` and a predefined `groupId`. The average response time during this test was 149ms, with an average of 1343 RPS.

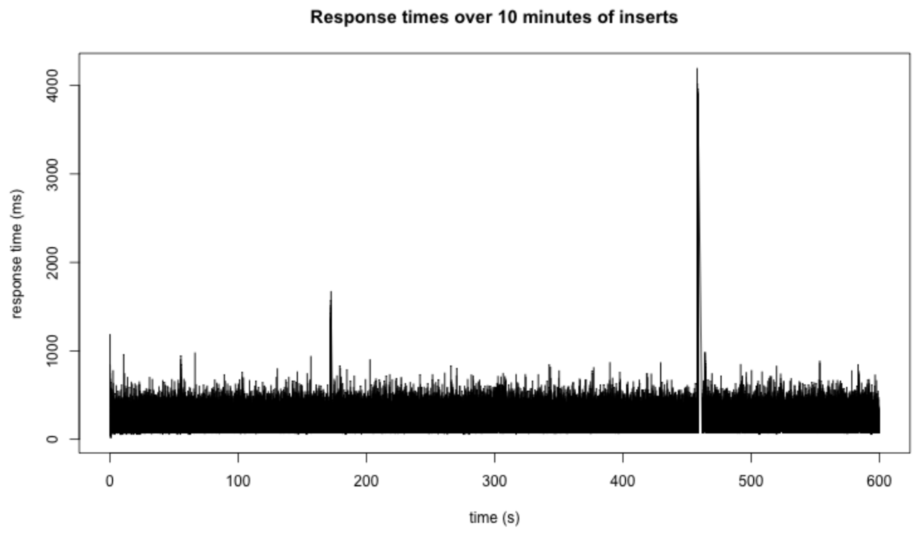


Figure 5.3: Response time over time

We also performed tests with different heap sizes and number of threads. We used a heap size of 1GB and 4GB for Dolphin. We also tested with JMeter running 100 threads and 400 threads. None of these parameters had significant influence on the response time. For completeness, plots of the response time for these tests have been included in Appendix F.

5.2.2 Impact Of Indices On Insert Performance

To determine the influence of indices on the insert performance, we first ran a test with inserts to the `visited` collection. Dolphin automatically ensures indices on the default fields (i.e. `timestamp` and `actor`), `entityId` and `entityType`. The insert performance dropped significantly, compared to the base test. This can also be seen in figure 5.4. The average response time was 187ms, with an average RPS of 1069.

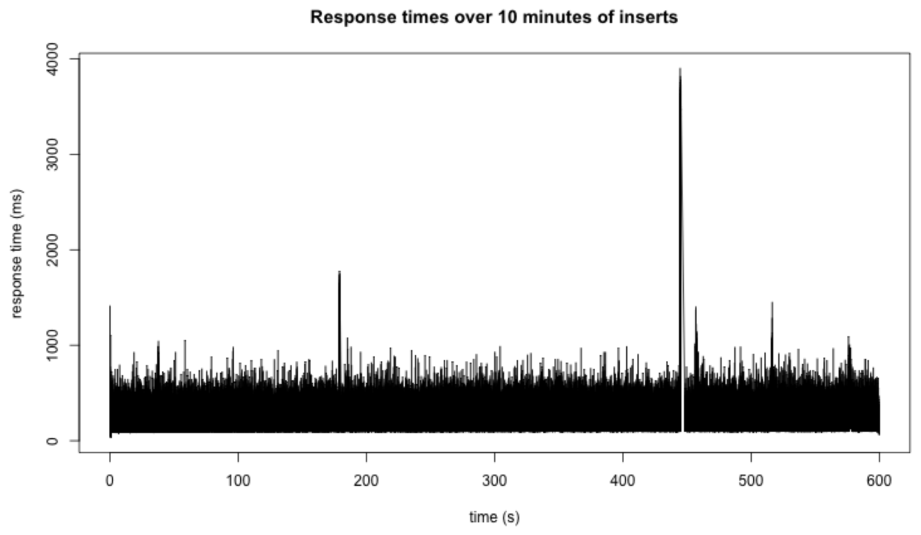


Figure 5.4: Insert performance when inserting visited events (response time over time)

In order to draw a fair comparison, we now insert the same type of events into the `detisiv` (`visited` spelled backwards) collection. Dolphin does not ensure the `entityId` and `entityType` indices on this collection. This directly influences the insert performance, as the average response time is now 134ms and the average RPS is 1489. Figure 5.5 also illustrates the increased performance, that is now comparable with the base test.

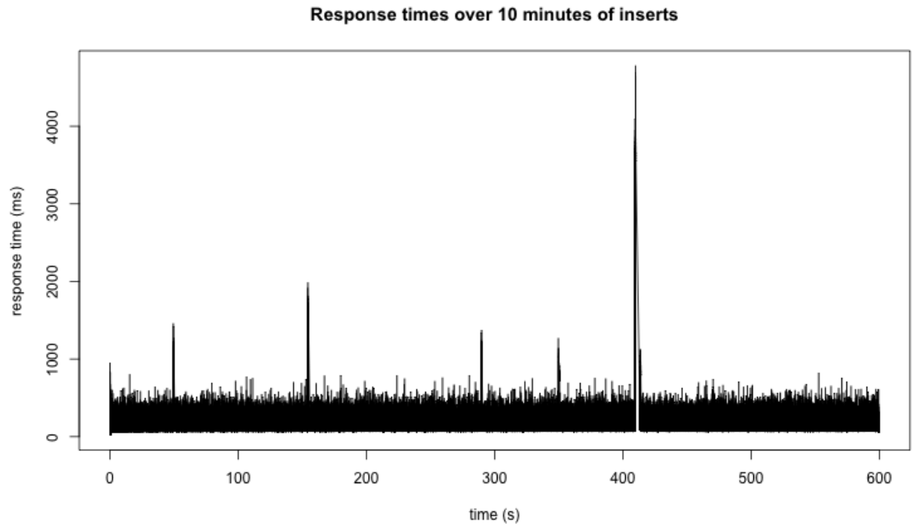


Figure 5.5: Insert performance when inserting detisiv events (response time over time)

To determine the location of this performance bottleneck, we stripped the part that ensures indices from Dolphin. We ran our test again, using the `registration` collection. The results are astonishing, as can be seen in figure 5.6. The average response time is much higher: 48ms. As a consequence, the average RPS is also much higher: 4146.

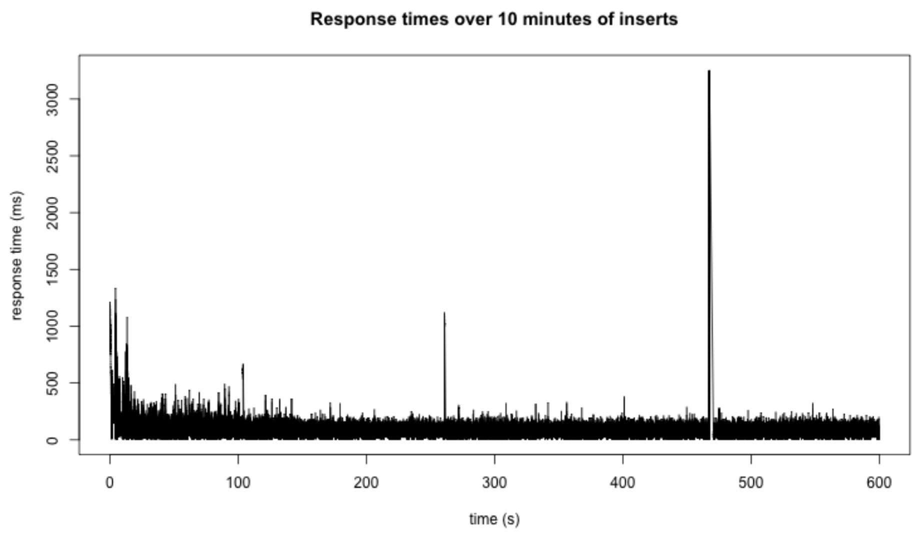


Figure 5.6: Insert performance when inserting events without indices (response time over time)

As a final test with regard to indices, we investigated whether the issue lies with MongoDB or with Dolphin. We ran the same test again, but now we defined the indices upfront using the MongoDB shell. Figure 5.7 reveals that the results are comparable to those without indices, with an average response time of 52ms. The RPS also remains quite high: 3871. It seems like the overhead of calling `ensureIndex` on every insert request is quite significant.

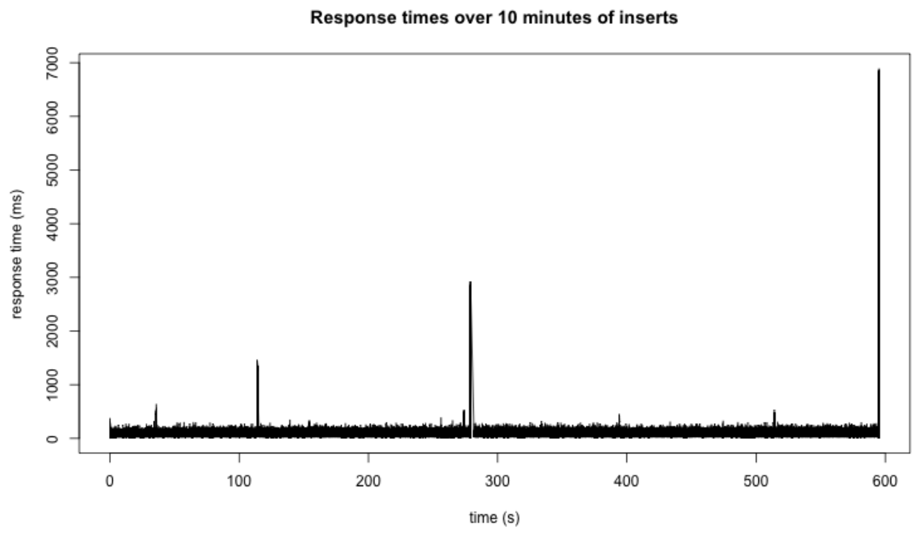


Figure 5.7: Insert performance when inserting events with pre-defined indices (response time over time)

5.2.3 Insert Performance When Indexing

For this test we created two collections: `registration` without events and `noitartsiger` with 10.8M events. All other parameters are the same as in the base test. We inserted events during 10 minutes into the `registration` collection. After two minutes, we started indexing the `timestamp` field in `noitartsiger`. When the inserts were completed, 1.3M events were indexed. It took about 1 minute and 12 seconds to index the remaining events.

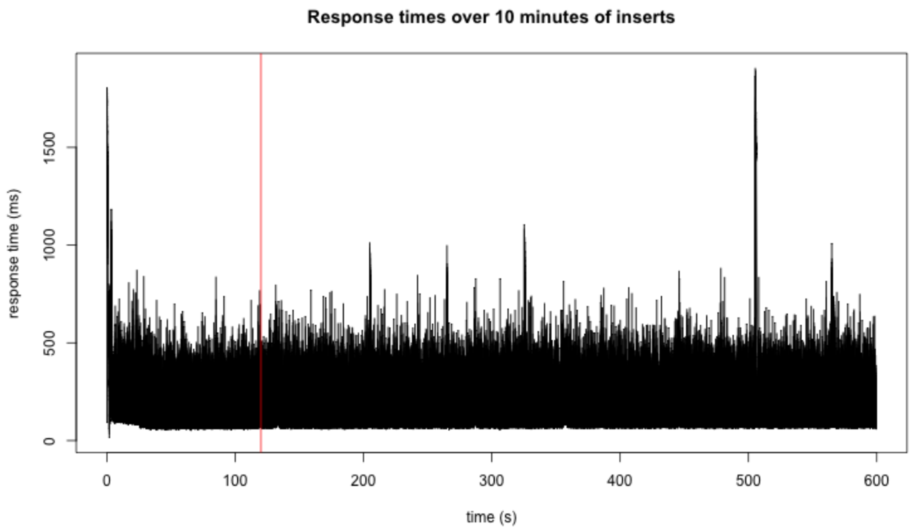


Figure 5.8: Response time while indexing

Figure 5.8 shows the response time of the requests. A vertical line has been added at $t = 120$ to indicate when the indexing started. As can be seen, the indexing has no significant impact on the insert performance.

6 Deployment

Dolphin is deployable as a Docker 1.0 container. In this chapter we first discuss the concepts of Docker and the recently released Docker Hub. We then continue by explaining how our application uses Docker and finally we discuss our experiences.

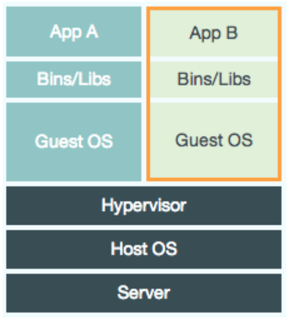
6.1 Docker

Docker is a platform to build, ship, and run (distributed) applications [15]. The applications are shipped in so called *containers*. To get an idea of what Docker is, we will first compare Docker to Virtual Machines (VMs). Afterwards, we will discuss containers and the Docker workflow.

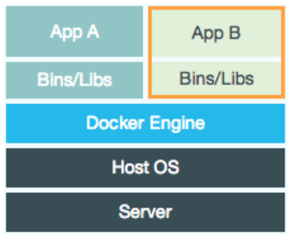
6.1.1 Docker Versus Virtual Machines (VMs)

The traditional way of deploying applications in an isolated environment is by the means of VMs. However, VMs have quite some overhead. For example, if you have an image of 1GB, and you spawn 1,000 of them, you will need 1,000GB of space. Not to mention the required CPU and memory resources. Because of this huge overhead, you cannot run many VMs on the same server.

From a users point of view, Docker containers are comparable to VMs. Docker gets rid of the space overhead by sharing common libraries. Additionally, Docker does not incur the overhead of running through a hypervisor. Figure 6.1 illustrates this difference.



(a) Virtual Machines



(b) Docker

Figure 6.1: Comparison between Virtual Machines (VMs) and Docker [15]

Docker containers run just like any other process and start within seconds. The processes run as isolated processes in userspace on the host operating system, allowing them to share the kernel with other containers. They use Linux Containers[1] to accomplish this.

6.1.2 Containers

Docker uses the concepts of *images* and *containers*. Images can be seen as the blueprints and containers as the instantiated images. Accordingly, you can spawn multiple containers from the same image.

[1]`https://linuxcontainers.org/`

Images are layerized, which means that every change applied to the image is stored in its own layer. As a result, fetching a newer version of the image will be much faster than the initial fetch, as the (untouched) layers can be reused from the cache. You can create an image in two ways: by committing to a *repository* and by writing a *Dockerfile*.

Using Docker repositories feels a bit like using (for example) `git` repositories. You can create a repository, pull a base image, make changes, commit these changes and push them to a remote repository. It is also possible to build an image using a Dockerfile and push the result to a repository. Even better, you can make changes on that built image and commit these changes to the repository. It is also possible to tag certain commits on the repository. That way, you can maintain multiple versions of your Docker images, for example a stable version and a beta version.

In a Dockerfile, you describe every step that has to be performed in order to build the image. When building an image from a Dockerfile, Docker implicitly commits each step of the build process. Using Dockerfiles is therefore basically like automating the process of making changes and committing those changes. See Appendix E for an example of a Dockerfile.

Another advantage of using containers is that Docker can link them together internally. For example, if you have an application A that needs to communicate with a database B, but A is not allowed to talk with another database C, you can link the container of A to the container of B. That way, A will not be able to communicate with (or even know about) C. By default all exposed ports of the containers are not exposed on the host OS, but you can expose them by adding a flag when running the container. When linking containers, you can give every container its own alias, which allows you to link multiple containers together without having to figure out their virtual IP addresses, even though they might expose the same ports.

6.2 Docker Hub

The building process can be performed on a private (build) server, or on the recently released Docker Hub[2]. Docker Hub is to Docker what GitHub[3] is to `git`: Docker Hub is a platform to collaborate and comment on Docker repositories. Docker Hub currently supports two types of repositories: the default repositories and automated build repositories.

Docker Hub can automatically build Docker images from Dockerfiles hosted on GitHub or Bitbucket[4] by the means of automated build repositories. This will make your life a lot easier, as they also support hooks. For example: you can add a post-hook for GitHub, so that when a developer pushes code to a specific branch, Docker Hub automatically builds the new Docker image. Additionally, you can add a post-hook to that process, so that your servers get notified whenever a new build has successfully been completed. That way, your servers can directly pull (and run) the new version.

6.3 Our Application And Docker

We use Docker for deploying our application and for executing performance regression tests (see section 4.4). As per requirement 7, the Dolphin Docker image is based on Ubuntu 12.04. Within Ubuntu 12.04, Java 8 and Maven are installed. The Dolphin source code then gets imported into the image, and Maven is being used to package the Java Archive (JAR). The Dockerfile in Appendix E is the Dockerfile that does all this.

[2]`https://hub.docker.com`
[3]`https://github.com/`
[4]`https://bitbucket.org/`

By building Dolphin in the Dockerfile, you can build the image once and then pull it on multiple servers. When you run the container, the Dolphin JAR gets started directly and automatically. This way, Dolphin can be distributed fairly easily, allowing you to quickly start multiple Dolphin instances within seconds.

6.4 Our Experiences With Docker

Because Docker is a relatively new technology, there are no strict guidelines on how to use it. In this section we will briefly discuss some of the things we experienced during the use of Docker.

Build Process At first, we built the Dolphin image on the host OS. This seemed to be nice, as your image remains very small. You do not need dependencies such as Maven, and the source code is also excluded from the image. However, this decision made the automated build process more complicated, because you need a build server running the right Java version and Maven. We now build Dolphin during the build process of the image, so that the container is directly runnable and you do not need the dependencies on the build server.

Private Repositories When pulling from a private repository, you have to somehow authenticate first. It took some time to discover that you need to use `docker login` with your credentials for this. At the moment, there is no way of using SSH keys for this.

Data Persistency When running a container with data that should be persisted, you should make sure the data directory is shared with the host OS or copied onto the host OS before you destroy the container (for example when you deployed a newer version). This seems trivial, but it is very easy to forget.

Commit Workflow Committing to a Docker repository requires a different workflow than the one you are probably used to. You run a command on a container, and when that command has finished, you commit the resulting container. When you want to perform multiple operations at once, you probably need to run `bash` on the container. However, the preferred way to perform multiple operations at once is through a Dockerfile.

Development Docker has more advantages on the operations side than it has on the development side. Docker does not run natively on Mac OS X or Windows, which means you have to run some kind of virtualization. We used Vagrant 1.6.2[5] with VirtualBox 4.3.12[6] for this. We spawned a Core OS 353.0.0[7] machine that natively supports Docker.

Amazon EC2 Micro Instance We also tried running the Dolphin Docker container on an Amazon EC2 Micro Instance[8] with Core OS 353.0.0. It took less than 10 minutes from a push to GitHub to running the latest build on that instance. Docker Hub immediately started building after the push and directly triggered the post-hook after it had finished. Pulling a newer version of the container onto the same instance took even less time (because of the cache reuse): about 5 minutes.

[5]`http://www.vagrantup.com/`
[6]`https://www.virtualbox.org/`
[7]`https://coreos.com/`
[8]`http://aws.amazon.com/ec2/instance-types/`

7 Process

This chapter describes the process of developing the application. We start by describing the software development methodology. We then continue with an outline of the main challenges that we were confronted with. Finally, we reflect on the process and indicate how we can improve ourselves.

7.1 Software Development Methodoloy

During the project, we were required to use the Magnet.me workflow. Magnet.me therefore supplied access to their repositories on GitHub[1] and to HipChat[2]. We will discuss these tools first. Afterwards, we briefly describe the roles of the team members and our planning.

7.1.1 GitHub Development Workflow

GitHub adds a social layer on top of `git` in order to allow developers to collaborate, review code changes and keep track of issues. Every new piece of functionality is developed in a separate branch. When the developer is satisfied with the code or when he needs help, he opens a so called Pull Request. A Pull Request allows other developers to review the set of changes. When all team members are satisfied, the feature branch gets merged into the master branch, taking the code into production.

GitHub also allows you to easily track issues. Additionally, these issues can be linked to milestones. Every issue was fixed in a separate branch, matching the method described above.

7.1.2 Communication Over HipChat

Magnet.me promotes transparancy. Therefore, all data has to be available to all employees and all works have to be available (and promotable) via a URL. Accordingly, all internal communication is also transparant. Magnet.me uses HipChat for this, a chat system with persistent chat rooms. Each morning we had to communicate over HipChat what we were planning to do that day, and at the end of each day we communicated what has actually been accomplished.

Magnet.me promotes flexible working places and times. Therefore it is to the team to decide where and when to work. The team was thus not bound to the Magnet.me office, except for the first two weeks. This way, the team was able to meet the employees and to learn about the workflow of the company. While working remote, HipChat allowed easy communication amongst the team members.

7.1.3 Team Roles

During the project, all project team members were equal. This implied that we all contributed to the project in the same way with the same amount of input/control. The workload was also evenly distributed amongst us.

Alex Nederlof is CTO at Magnet.me and he both guided the project and acted as the company contact. Additionally, he is the customer. Georgios Gousios coordinated the project as supervisor from the Delft University of Technology.

[1]`https://github.com/`
[2]`https://www.hipchat.com/`

During the project, we had intensive contact with Alex Nederlof. Especially during the first weeks, when we had to get the requirements clear. We also had regular (weekly) meetings with Georgios Gousios to discuss the progress of the project.

7.1.4 Planning

For the project we used approximately 10 weeks, with 42 working hours per week. The total time was split in roughly four phases: problem analysis, research, development, and evaluation. The first week was spent on investigating the actual problem. The following the two weeks were used to research a suitable database system, as evaluated in chapter 3. After these three weeks, we began the development of Dolphin. This continued for approximately five weeks. The last weeks were spent on evaluating the developed system (performance tests), on writing this report and on preparing the presentation.

7.2 Main Challenges

The main challange was dealing with the lack of functional requirements. During the first few weeks, we kept interviewing the client in order to find more functional requirements. However, this did not result in significantly more information. After a few weeks we realized that we were approaching the problem from the wrong side. Instead of returning to the client, we started figuring out what interesting features our system *could* support. This eventually resulted in the functionality listed in section 4.1.

While there was a lack of functional requirements, there was an abundance of non-functional requirements. For example, we were instructed to write the application in Java, communicate using a REST API and ship the product in a Docker container. Some of these non-functional requirements are legitimate. Others, we think, could have better been formulated as suggestions. We feel that these non-functional requirements pushed us into a particular direction. If we had followed a more problem-centric approach, we might have taken different decisions.

Finally, some non-functional requirements were controversial. Initially we were instructed to use an *event sourcing* architecture [23]. After a few weeks we dropped the concept of event sourcing, because it solves a different problem. Other requirements changed along the way. Initially we were instructed to use Puppet for deployment, but after a few weeks we were told to ship our product in a Docker container.

A totally different challenge was choosing the right data store. Originally, we decided to use Elasticsearch. This decision was based on several arguments, such as its powerful searching capabilities, the very easy setup of scalability, and the availability of the ELK stack (Elasticsearch, Logstash[3] and Kibana[4]).

However, after using Elasticsearch for a week, we experienced many issues. Its query possibilities are very limited and the Domain-specific Language (DSL) to describe these queries is really complex. Certain queries (e.g. aggregating nested objects, nested queries) cannot even be performed in Elasticsearch. Additionally, certain results are inaccurate. For example, computing aggregates over shared data gives an approximation. Finally, storing nested objects (such as our Actor) also had many negative implications.

Because of these issues, we looked further into MongoDB. MongoDB overcame all these Elasticsearch hassles, and it feels more mature (i.e. better documentation, integration with other tools). Therefore, we decided to switch from Elasticsearch to MongoDB.

[3]`http://logstash.net/`

[4]`http://www.elasticsearch.org/overview/kibana/`

7.3 Reflection

In retrospect, we can draw some conclusions from our process. In this section we will shortly reflect on the used software development methodology and discuss the learned lessons.

The Github workflow, as described in section 7.1.1, worked really well. For example, GitHub issues really helped us to keep track of the work that had to be done. We organized issues in weekly milestones. However, estimating which issues could be completed within one week was hard. The main reason for this was our lack of experience. This got better during the course of the project.

Though we intended to not use a waterfall model, we waited until the last few weeks before executing performance tests. These tests revealed critical performance issues for which we had to find a solution in the last week. If we had performed these tests ealier, we could have incorporated this in our initial design.

As discussed in section 7.2, there were little functional requirements. The lack of focus of the project description made it hard for us to get started. However, this also gave us the opportunity to get to know many new technologies. We have now learned how to approach a project in which you are expected to think outside the box.

We believe that we succeeded in delivering a system that is capable of storing and retrieving events in a way that satisfies Magnet.me's needs. We feel confident that other developers can use our solution and continue working on the project.

8 Conclusion And Future Work

In this chapter we first present a conclusion of the research. In doing so, we provide an answer to the questions that were posed in the introduction. We finish the chapter with recommendations for future work.

8.1 Conclusion

We used *events* to capture important interactions of a user with the Magnet.me website. MongoDB is used for storing theses events and for executing basic query functionalities on them. On top of MongoDB we built Dolphin. Dolphin exposes a developer-friendly REST API for inserting and retrieving events.

Dolphin supports the following hardcoded event types: JavaScript error events, modified events and visited events. Additionally, Dolphin was built with extensibility in mind, making it easy to add new types of events. All events have at least one required field: the actor, i.e. the user that triggered the event. The hardcoded event types have certain additional required fields, such as the entity (identifier and type) on which the event was triggered. Dolphin ensures that MongoDB has indices on all these required fields, so they can be queried as fast as possible.

Additionally, Dolphin is able to compute analytical features: metrics (count, sum, average, min, and max), time series and funnels. The metrics give answers to several simple questions, such as *"How many users registered in the last ten minutes?"* or *"What was the maximum loadtime for page x?"*. Time series are basically series of metrics, that answer questions such as *"How many registrations where there per day during the last week?"*. Funnels help to determine how many users performed steps in a certain order, such as *"How many users registered and how many of them actually logged in?"*.

All components of the system can be shipped in Docker containers, making it easy to deploy on various servers. The Docker container with Dolphin in it can be automatically built. The container contains all dependencies and compiled source code. Whenever the container is launched, Dolphin starts immediately.

Because of its modular setup, the system is easily horzontally scalable. Dolphin connects to a MongoDB URI, which means that from the Dolphin's point of view it does not matter whether it connects to a cluster or a single instance. MongoDB provides the ability to partition the data horizontally (i.e. shard the data) in order to scale the database. Finally, because of the stateless nature of our REST API, Dolphin can easily be scaled as well.

We also implemented a large automated test suite, consisting of unit tests and integration tests. These tests have a total line coverage of approximately 86%. Additionally, we implemented a (rather simple) performance regression test to determine whether a code change has impact on the performance of the system. The Software Improvement Group (SIG) evaluated the code of Dolphin twice, giving a score of over four stars (out of five).

8.2 Future Work

We were expected to deliver a working product within ten weeks. We knew up front that this was too little time for implementing all possible features. Our main goal has been to create a well thought out system to store and retrieve events that can easily be extended. This section lists possible improvements to the current system as well as new features.

8.2.1 Paging

Paging, sometimes also known as *pagination*, is the process of dividing the results into pages. Returning only a single page with results is less CPU intensive and uses less bandwidth then returning all results. Currently, our application does not support any form of paging. With the few events that are currently processed, this is not really an issue. However, if the data grows as expected, it will become practically impossible to extract events from the application. Also, depending on the use case, a user may only be interested in certain parts of the data. For example, if a user is interested in the last 10 events in a certain collection, there is no need to return all events in that collection.

8.2.2 All Events Endpoint

Currently, the application does not allow a user to query among multiple collections. More precisely, the metrics and timeseries can only be computed on events within a single collection. Also, it is only possible to extract events per collection. It would be very useful to run queries across collections. This will, for example, allow the user to get a list of the last 10 events that occured throughout the system.

8.2.3 Dynamic Event Type Specification

The application makes a distinction between general events and specific events. Currently, the following specific event types are implemented: JavaScript error events, modified events and visited events. The difference between general events and specific events is that specific events require additional validation at insertion. To create a new event type, a developer should create a new Java class, implement the necessary functionality, and recompile the application. This tedious process can be improved by dynamically loading event definitions. A developer should be able to define a new event type in, for example, a YAML file and put this in a predefined folder so that the application automatically loads the definition.

8.2.4 Machine Learning

In our project we have focused on using these events to provide analytics. However, the events contain tremendous business value and can be used for many other purposes. For example, the events can be used as input to a machine learning system. This would allow Magnet.me to predict user behavior, offer personalized content, help users to discover interesting data and many more. Examples of machine learning systems that can be integrated are Apache Mahout[1] and Prediction.io[2].

8.2.5 Precomputed Metrics

One of the requirements was to persist raw data (see section 2.3). This enables our system to cope with any event that can be represented as a JSON string. However, performing computations on the full data set can take up to a few minutes. Furthermore, if the user is interested in several metrics, the process of extracting information out of the system can get very cumbersome. It would be extremely

[1]`https://mahout.apache.org`
[2]`http://prediction.io/`

useful for the company to generate reports on the fly. This can be achieved by pre-computing metrics in the background and presenting the user the aggregated data. Metrics of interest are not known beforehand, therefore, a general framework should be created to process data in the background and output aggregated results.

A Project Proposal

Project Description

Magnet.me is visited by hundreds of people per day. To analyze their behavior on our site, we want to track the actions they perform on the site, and how often they do. Tracking such information generates tons of data, requiring a big data solution to persist and analyze the information. During this project, you will be responsible for setting up such a system, and generating the first reports. The project will consist of the following phases:

1. Research which NoSql/BigData solution suits our needs best.
2. Set-up the proposed solution, write an API for Magnet.me to connect to.
3. Run performance tests, see how the system scales.
4. Generate a report that tells us how often users log in, and which parts of the site they visit when they do.

You will learn: The big data solution, puppet, auto-deployment, performance testing, API design and working remote with HipChat and GitHub.

Company Description

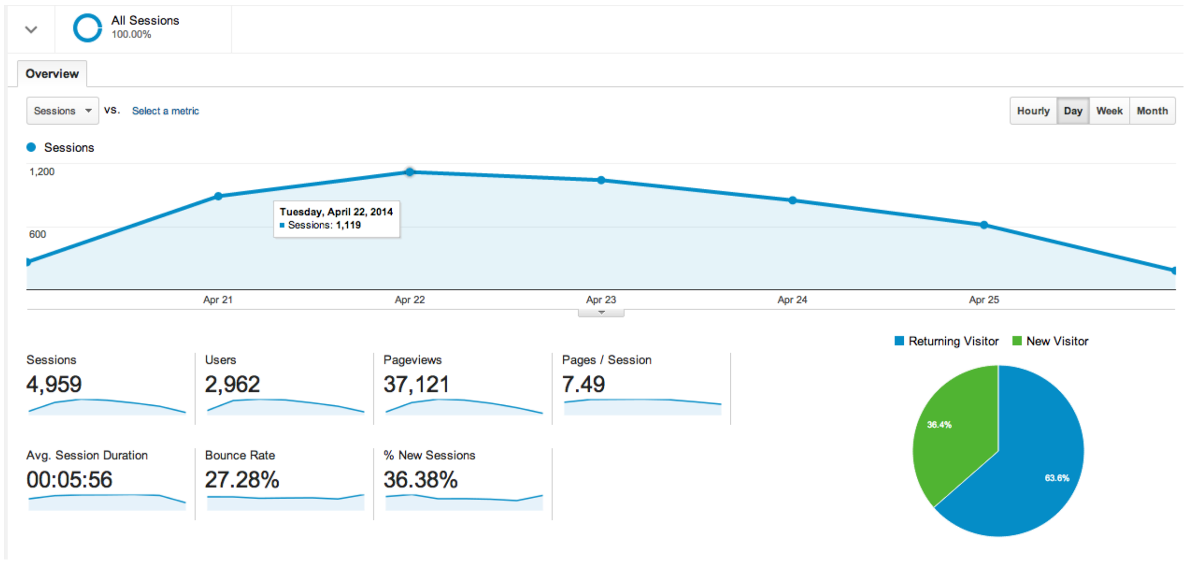
Magnet.me is a young, Rotterdam-based start-up that allows students to get into contact with the companies they are interested in. On contrary to LinkedIn, on Magnet.me you only connect with the companies that are relevant to you and that are interested in you. This gives you a much better insight into your career opportunities and hiring potential. Magnet.me grew from 3 to 18 employees in 2013 and is still going strong with plans to go abroad in 2014. If you want to work with young, talented people, on something that will have significant impact, Magnet.me is the perfect place for you.

Auxiliary Information

Project is only allowed to be implemented in Java, or JavaScript.

B Google Analytics Report

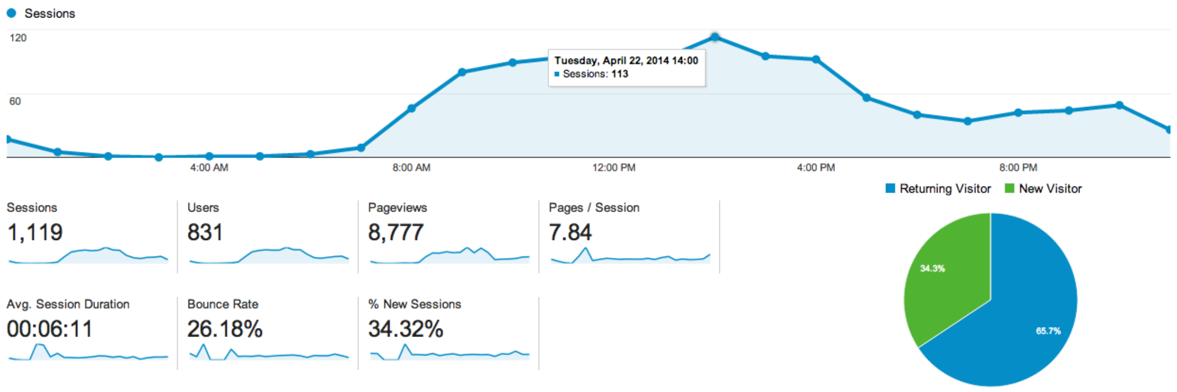
Figures B.1 and B.2 show the usage of the Magnet.me website on Tuesday April 22, 2014. The first shows an overview of the day, the second shows an overview per hour.



Sessions	Users	Pageviews	Pages / Session
4,959	2,962	37,121	7.49

Avg. Session Duration	Bounce Rate	% New Sessions
00:05:56	27.28%	36.38%

Figure B.1: Website usage (day overview)



Sessions	Users	Pageviews	Pages / Session
1,119	831	8,777	7.84

Avg. Session Duration	Bounce Rate	% New Sessions
00:06:11	26.18%	34.32%

Figure B.2: Website usage (hour overview)

C Dependencies

To get up and running quickly, we made use of numerous existing software solutions. This page lists the most important software dependencies and a short description.

Dropwizard 0.7.0 (`http://dropwizard.github.io/dropwizard/`)
Dropwizard is a Java framework for developing ops-friendly, high-performance, RESTful web services.

Flapdoodle 1.46.0 (`https://github.com/flapdoodle-oss/de.flapdoodle.embed.mongo`)
Flapdoodle is an embedded MongoDB process for running integration tests.

Hamcrest 1.3 (`http://hamcrest.org/JavaHamcrest/`)
Hamcrest is a library of matchers for usage in JUnit tests.

Guice 3.0 (`https://code.google.com/p/google-guice/`)
Guice is a dependency injection framework.

Jackson 2.4.0 (`http://wiki.fasterxml.com/JacksonHome`)
Jackson is a multi-purpose Java library for processing JSON data format. Jackson aims to be the best possible combination of fast, correct, lightweight, and ergonomic for developers.

JUnit 4.11 (`http://junit.org/`)
JUnit is a unit testing framework to write repeatable tests.

Log4j 1.2.17 (`http://logging.apache.org/log4j/1.2/`)
Log4j is a logging package that serves a simple API for easily inserting log statements into the code.

Mockito 1.9.5 (`https://code.google.com/p/mockito/`)
Mockito is a test framework that allows you to mock dependencies of a class and verify interactions on these mocks.

MongoDB 2.6 (`http://www.mongodb.org/`)
MongoDB is a document-oriented database. It stores JSON-like documents with dynamic schemas. We used the Java MongoDB driver to communicate with MongoDB.

Project Lombok 1.14.0 (`http://projectlombok.org/`)
Project Lombok replaces the need of writing boilerplate code (such as getters and setters) with annotations.

Reflections 0.9.9-RC2 (`https://github.com/ronmamo/reflections`)
Reflections allow you to query metadata of the source code (such as all classes with some annotation) on runtime.

D API Documentation

Introduction

This document gets updated during development. As a consequence, there may be some inconsistencies between this document and the most recent implementation.

This section continues with several general API remarks. The next section explains event insertion, retrieval, and querying.

Headers

All resources accept and produce JSON. You must specify the following header in every request:

`Content-Type: application/json`

Additionally, you should use the following header to specify the API version:

`Accept: application/me.magnet.dolphin-v1+json`

Actor

Some requests to the API require an `actor` object. This object has the following fields:

Field	Description
id (mandatory)	The identifier that is used by Magnet.me internally
groupId (mandatory)	The group the actor is in

Storing Events

The system supports *general* events and *specific* events. A specific event is a special kind of event that requires additional fields and for which additional constraints apply. The specific events are:

- *JavaScriptError* events
- *Modified* events
- *Visited* events

We first discuss general events and then proceed with the specific events.

General events

Resource

`PUT /event/:collection`

Body

Field	Description
timestamp	An ISO 8601 timestamp. Example: 2014-05-19T22:34:17Z
actor	An object that is used to specify the actor

It is possible to specify additional key-value pairs to provide more event information.

Output

The HTTP status code is 201 CREATED if the insertion of the event was successful. Additionally, the `Location` header specifies the location of the entity.

JavaScriptError events

As a special case of storing events, you can store *JavaScriptError* events. Additional validation is performed on this type of events.

Resource

`PUT /event/javascripterror`

Body

Field	Description
timestamp	An ISO 8601 timestamp. Example: 2014-05-19T22:34:17Z
actor	An object that is used to specify the actor
entityId	The identifier that is used by Magnet.me internally
entityType	The type of the entity
error	The javascript error

It is possible to specify additional key-value pairs to provide more event information.

Output

The HTTP status code is 201 CREATED if the insertion of the event was successful. Additionally, the `Location` header specifies the location of the entity.

Modified events

As a special case of storing events, you can store *modified* events. Additional validation is performed on this type of events.

Resource

`PUT /event/modified`

Body

Field	Description
timestamp	An ISO 8601 timestamp. Example: 2014-05-19T22:34:17Z
actor	An object that is used to specify the actor
entityId	The identifier that is used by Magnet.me internally
entityType	The type of the entity

It is possible to specify additional key-value pairs to provide more event information.

Output

The HTTP status code is 201 CREATED if the insertion of the event was successful. Additionally, the `Location` header specifies the location of the entity.

Visited events

As a special case of storing events, you can store *visited* events. Additional validation is performed on this type of events.

Resource

`PUT /event/visited`

Body

Field	Description
timestamp	An ISO 8601 timestamp. Example: 2014-05-19T22:34:17Z
actor	An object that is used to specify the actor
entityId	The identifier that is used by Magnet.me internally
entityType	The type of the entity

It is possible to specify additional key-value pairs to provide more event information.

Output

The HTTP status code is 201 CREATED if the insertion of the event was successful. Additionally, the `Location` header specifies the location of the entity.

Retrieving Events

Resource

`POST /event/:collection`

Body

Field	Description
from (optional)	A lower ISO 8601 date string to filter the results on, e.g. ‘2014-05-25T00:00:00.000Z’
to (optional)	An upper ISO 8601 date string to filter the results on, e.g. ‘2014-05-25T00:00:00.000Z’
filter (optional)	A raw MongoDB filter object for `$match`

Output

Returns a filtered list of events, with status code 200 OK and the following body:

```
[
    { .. },
    { .. },
    ..
]
```

Metric: Average of a common field in events

Resource

`POST /average/:collection/:field`

Body

Field	Description
from (optional)	A lower ISO 8601 date string to filter the results on, e.g. ‘2014-05-25T00:00:00.000Z’
to (optional)	An upper ISO 8601 date string to filter the results on, e.g. ‘2014-05-25T00:00:00.000Z’

Field	Description
filter (optional)	A raw MongoDB filter object for `$match`
groupField (optional)	The field to group the data on

Output

Returns a list of the average(s), with status code 200 OK:

```
[
    {
        "group": (Object) valueOfGroupField,
        "value": (int) averageOfField
    },
    ...
]
```

When the groupField is omitted, the value of `_id` is 0. The list then consists of one item.

Metric: Count number of events

Resource

```
POST /count/:collection
```

Body

Field	Description
from (optional)	A lower ISO 8601 date string to filter the results on, e.g. '2014-05-25T00:00:00.000Z'
to (optional)	An upper ISO 8601 date string to filter the results on, e.g. '2014-05-25T00:00:00.000Z'
filter (optional)	A raw MongoDB filter object for `$match`
groupField (optional)	The field to group the data on

Output

Returns a list of the count(s), with status code 200 OK:

```
[
    {
        "group": (Object) valueOfGroupField,
        "value": (int) numberOfItems
    },
    ...
]
```

When the groupField is omitted, the value of `_id` is 0. The list then consists of one item.

Metric: Maximum value of a common field in events

Resource

`POST /max/:collection/:field`

Body

Field	Description
from (optional)	A lower ISO 8601 date string to filter the results on, e.g. '2014-05-25T00:00:00.000Z'
to (optional)	An upper ISO 8601 date string to filter the results on, e.g. '2014-05-25T00:00:00.000Z'
filter (optional)	A raw MongoDB filter object for `$match`
groupField (optional)	The field to group the data on

Output

Returns a list of the maximum(s), with status code 200 OK:

```
[
    {
        "group": (Object) valueOfGroupField,
        "value": (int) maximumOfField
    },
    ...
]
```

When the groupField is omitted, the value of `_id` is 0. The list then consists of one item.

Metric: Minimum value of a common field in events

Resource

`POST /min/:collection/:field`

Body

Field	Description
from (optional)	A lower ISO 8601 date string to filter the results on, e.g. '2014-05-25T00:00:00.000Z'
to (optional)	An upper ISO 8601 date string to filter the results on, e.g. '2014-05-25T00:00:00.000Z'
filter (optional)	A raw MongoDB filter object for `$match`
groupField (optional)	The field to group the data on

Output

Returns a list of the minimum(s), with status code 200 OK:

```
[
    {
        "group": (Object) valueOfGroupField,
        "value": (int) minimumOfField
    },
    ...
]
```

When the groupField is omitted, the value of `_id` is 0. The list then consists of one item.

Metric: Sum of a common field in events

Resource

`POST /sum/:collection/:field`

Body

Field	Description
from (optional)	A lower ISO 8601 date string to filter the results on, e.g. ‘2014-05-25T00:00:00.000Z’
to (optional)	An upper ISO 8601 date string to filter the results on, e.g. ‘2014-05-25T00:00:00.000Z’
filter (optional)	A raw MongoDB filter object for `$match`
groupField (optional)	The field to group the data on

Output

Returns a list of the sum(s), with status code 200 OK:

```
[
    {
        "group": (Object) valueOfGroupField,
        "value": (int) sumOfField
    },
    ...
]
```

When the groupField is omitted, the value of `_id` is 0. The list then consists of one item.

Funnel of events

Resource

`POST /funnel`

Body

An array of step objects. The contents of the step object are explained in the table below. The number of steps is not limited (two steps are shown below).

Field	Description
from (optional)	A lower ISO 8601 date string to filter the results on, e.g. '2014-05-25T00:00:00.000Z'
to (optional)	An upper ISO 8601 date string to filter the results on, e.g. '2014-05-25T00:00:00.000Z'
collection	The collection to apply the funnel to
filter (optional)	The filter to apply

Below is an example of a funnel request:

```
[
  {
    "collection": "registration",
    "filter": {
      "referer": "Google"
    }
  },
  {
    "collection": "login"
  }
]
```

Output

For each step, the number of distinct actors that were also available in the previous step is returned.

```
[
    42,
    24
]
```

Timeseries

A timeseries is a series of a metric. For example, it can be used to count the number or registrations per day or to calculate the average response time per actor ID. Possible values for the `metric` parameter in the URI are min, max, avg, sum, and count. The field parameter in the URI is required for all metrics except `count`.

Resource

```
POST /timeseries/:metric/:collection/:field
```

Body

An object with the following fields:

Field	Description
from (optional)	A lower ISO 8601 date string to filter the results on, e.g. '2014-05-25T00:00:00.000Z'
to (optional)	An upper ISO 8601 date string to filter the results on, e.g. '2014-05-25T00:00:00.000Z'
filter (optional)	A filter object (see top of the page)
granularity	One of the following strings: second, minute, hour, day, month, year

For example:

```
{
    "from": "2014-05-25T00:00:00.000Z",
    "to": "2014-05-31T00:00:00.000Z",
    "granularity": "day"
}
```

Output

An array that contains an object for each second/minute/hour/etc. For example, if the granularity was `day`, then an array with objects for each day is returned. The object specifies a `from`, `to` and `count`. If the count is zero, it is omitted from the results.

```
[
    {
        "from": "2014-06-01T14:28:01Z",
        "to": "2014-06-01T14:28:02Z",
        "count": 1
    },
    {
        "from": "2014-06-01T14:27:01Z",
        "to": "2014-06-01T14:27:02Z",
        "count": 1
    }
]
```

E Example Dockerfile

Listing E.1: Dockerfile used by Dolphin

```
# We base our image on Ubuntu 12.04 (Precise Pangolin) with Java 8
    and Maven installed
FROM rickw/ubuntu12-java8-maven

# Expose the ports of Dolphin; note that EXPOSE only exposes ports
    within Docker.
EXPOSE 8080
EXPOSE 8081

# Adds source code to the Docker image
ADD . /dolphin
WORKDIR /dolphin

# Builds and packages Dolphin
RUN mvn clean package

# Default runs with server and config.yml arguments
CMD ["server", "config-docker.yml"]
ENTRYPOINT ["java", "-jar", "target/dolphin-jar-with-dependencies.
    jar"]
```

F Extra Performance Test Results

This appendix contains several additional test results for the performance tests discussed in chapter 5. The first two plots show the irrelevance of changes in the maximum heap space, the second two plots show the irrelevance of changes in the number of JMeter threads.

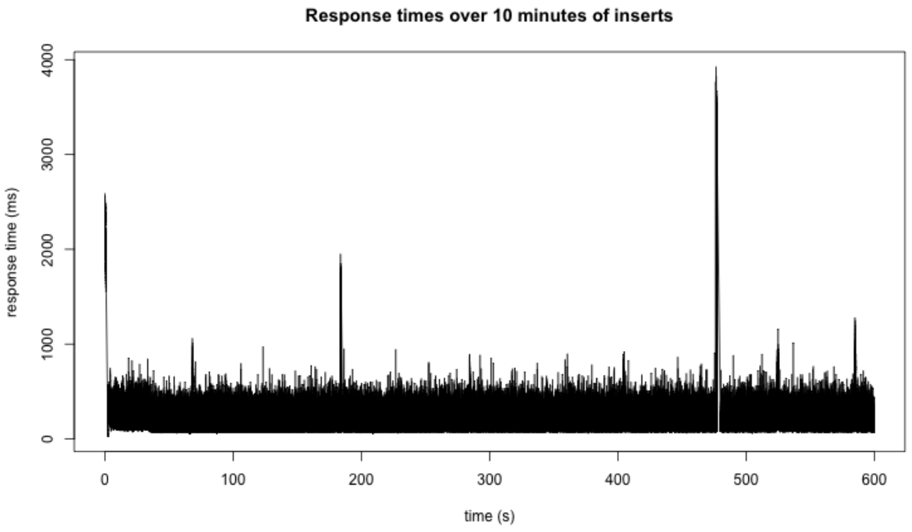


Figure F.1: Insert performance with maximum heap space of 4GB (response time over time)

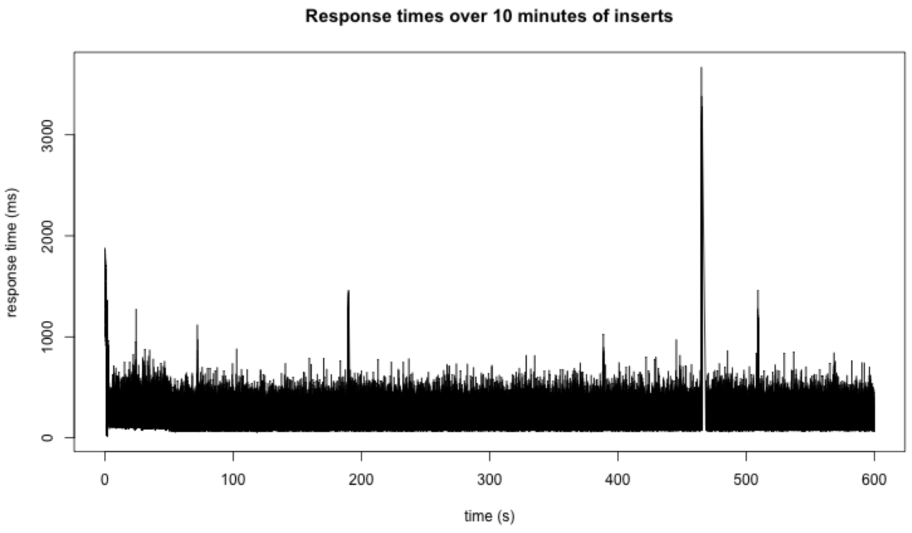


Figure F.2: Insert performance with maximum heap space of 1GB (response time over time)

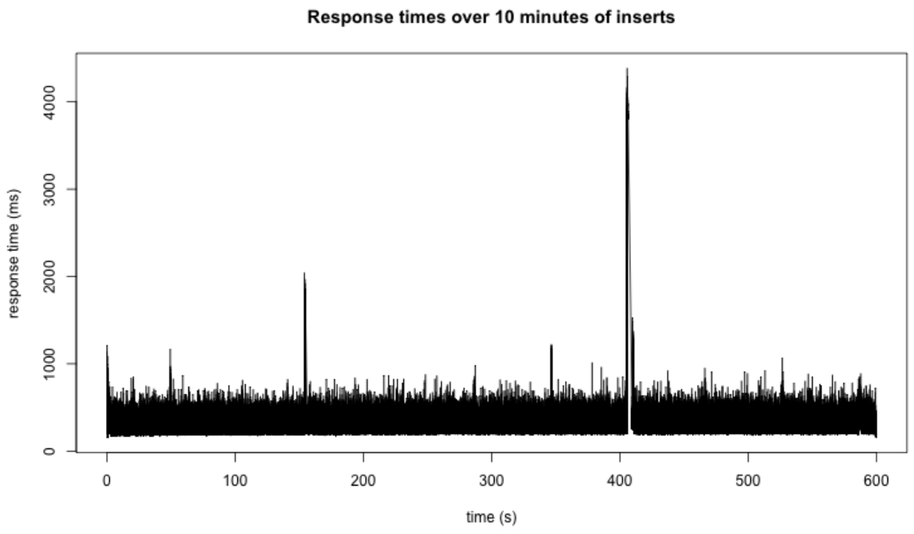


Figure F.3: Insert performance with 400 threads of JMeter (response time over time)

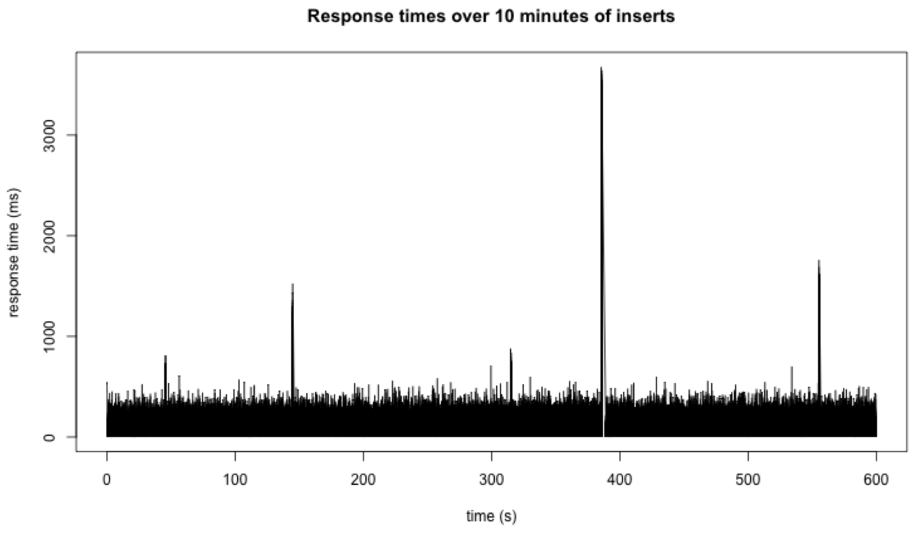


Figure F.4: Insert performance with 100 threads of JMeter (response time over time)

Bibliography

[1] Apache JMeter. http://jmeter.apache.org/. [Accessed: June 19, 2014].

[2] Elastic Search as a primary data store. http://elasticsearch-users.115913.n3.nabble.com/Elastic-Search-as-a-primary-data-store-td4036336.html. [Accessed: May 8, 2014].

[3] ES failed to recover after crash. https://groups.google.com/forum/#!topic/elasticsearch/HtgNeUJ5uao/. [Accessed: May 8, 2014].

[4] Map-Reduce performance in MongoDb 2.2 and 2.4. http://stackoverflow.com/a/12680165. [Accessed: May 9, 2014].

[5] Strong Consistency in MongoDB. https://groups.google.com/forum/#!topic/mongodb-user/8ILNHEOUIY4. [Accessed: May 6, 2014].

[6] Does ElasticSearch's Java Client use REST API? http://stackoverflow.com/questions/12156662/does-elasticsearchs-java-client-use-rest-api, 2012. [Accessed: May 6, 2014].

[7] Jae Hyeon Bae, Danny Yuan, and Sudhir Tonse. The Netflix Tech Blog: Announcing Suro: Backbone of Netflix's Data Pipeline. http://techblog.netflix.com/2013/12/announcing-suro-backbone-of-netflixs.html, 2013. [Accessed: May 9, 2014].

[8] Cassandra. Architecture in brief. http://www.datastax.com/documentation/cassandra/2.0/cassandra/architecture/architectureIntro_c.html. [Accessed: May 9, 2014].

[9] Cassandra. Cassandra Query Language (CQL) v3.1.5. http://cassandra.apache.org/doc/cql3/CQL.html. [Accessed: May 8, 2014].

[10] Cassandra. Cassandra Wiki. http://wiki.apache.org/cassandra/FrontPage. [Accessed: May 1, 2014].

[11] Cassandra. Data Replication. http://www.datastax.com/documentation/cassandra/2.0/cassandra/architecture/architectureDataDistributeReplication_c.html. [Accessed: May 1, 2014].

[12] Cassandra. Snitches. http://www.datastax.com/documentation/cassandra/2.0/cassandra/architecture/architectureSnitchesAbout_c.html#concept_ds_c34_fqf_fk. [Accessed: May 9, 2014].

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

[14] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 205–220. ACM, 2007.

[15] Docker. What is Docker? https://www.docker.com/whatisdocker/. [Accessed: June 12, 2014].

[16] Druid. Design. http://druid.io/docs/0.6.105/Design.html. [Accessed: May 6, 2014].

[17] Druid. Querying. http://druid.io/docs/0.6.105/Querying.html. [Accessed: May 6, 2014].

[18] Elasticsearch. Getting started. http://www.elasticsearch.org/guide/en/elasticsearch/guide/current/getting-started.html. [Accessed: May 8, 2014].

[19] Elasticsearch. Mapping Relations Inside Elasticsearch. http://www.elasticsearch.org/blog/managing-relations-inside-elasticsearch/. [Accessed: May 8, 2014].

[20] Elasticsearch. Optimistic Concurrency Control. http://www.elasticsearch.org/guide/en/elasticsearch/guide/current/optimistic-concurrency-control.html. [Accessed: May 8, 2014].

[21] Fangjin Yang et al. Druid: A Real-time Analytical Data Store. 2013.

[22] Jian Fang. MongoDB features and comparisons with Cassandra and HBase. http://johnjianfang.blogspot.nl/2012/04/mongodb-features-and-comparisons-with.html, 2012. [Accessed: May 1, 2014].

[23] Martin Fowler. Event sourcing. http://martinfowler.com/eaaDev/EventSourcing.html, 2005. [Accessed: April 22, 2014].

[24] Clinton Gormley and Zachary Tong. *Elasticsearch: The Definitive Guide*. O'Reilly Media, 2014.

[25] Guy Harrison. Cassandra and Hadoop - Strange Bedfellows or a Match Made in Heaven? http://www.dbta.com/Columns/Notes-on-NoSQL/Cassandra-and-Hadoop---Strange-Bedfellows-or-a-Match-Made-in-Heaven-75890.aspx, 2011. [Accessed: May 1, 2014].

[26] MongoDB. Aggregation Introduction. http://docs.mongodb.org/manual/core/aggregation-introduction/. [Accessed: May 9, 2014].

[27] MongoDB. Aggregation Pipeline. http://docs.mongodb.org/manual/core/aggregation-pipeline/. [Accessed: May 9, 2014].

[28] MongoDB. Config Servers. http://docs.mongodb.org/manual/core/sharded-cluster-config-servers/. [Accessed: June 25, 2014].

[29] MongoDB. Date Aggregation Operators. http://docs.mongodb.org/manual/reference/operator/aggregation-date/. [Accessed: May 12, 2014].

[30] MongoDB. distinct. http://docs.mongodb.org/manual/reference/command/distinct/#dbcmd.distinct. [Accessed: June 26, 2014].

[31] MongoDB. FAQ: Concurrency. http://docs.mongodb.org/manual/faq/concurrency/. [Accessed: May 6, 2014].

[32] MongoDB. $match (aggregation). http://docs.mongodb.org/manual/reference/operator/aggregation/match/#pipe._S_match. [Accessed: May 9, 2014].

[33] MongoDB. MongoDB Drivers and Client Libraries. http://docs.mongodb.org/ecosystem/drivers/. [Accessed: May 9, 2014].

[34] MongoDB. MongoDB Wire Protocol. http://docs.mongodb.org/meta-driver/latest/legacy/mongodb-wire-protocol/. [Accessed: May 9, 2014].

[35] MongoDB. Replica Set Deployment Architectures. http://docs.mongodb.org/manual/core/replica-set-architectures/. [Accessed: June 25, 2014].

[36] MongoDB. Replication Introduction. http://docs.mongodb.org/manual/core/replication-introduction/. [Accessed: May 1, 2014].

[37] MongoDB. Sharding Introduction. http://docs.mongodb.org/manual/core/sharding-introduction/. [Accessed: May 1, 2014].