CHALMERS
UNIVERSITY OF TECHNOLOGY

Management of historical data in continuous integration systems

Master's thesis in Software Engineering

ANDERS GUSTAFSSON
JONAS LERGELL

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2015

Master's thesis 2015:NN

Management of historical data in continuous integration systems

ANDERS GUSTAFSSON
JONAS LERGELL

Department of Computer Science and Engineering
Software Engineering Division
Chalmers University of Technology
Gothenburg, Sweden 2015

Management of historical data in continuous integration systems
ANDERS GUSTAFSSON
JONAS LERGELL

Supervisor: Graham Kemp, Department of Computer Science and Engineering
Examiner: Miroslaw Staron, Department of Computer Science and Engineering

Master's Thesis 2015:June
Department of Computer Science and Engineering
Software Engineering Division
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Abstract

Continuous integration is an agile software practice where code is checked in frequently and subsequently built and tested automatically. Due to the maturity of this practice at the company Ericsson, the frequency of these automated builds is increasing at the company and as such, much more data about the development process is generated. However, the software systems that gather and present this data was not designed to scale with the current data growth. This has led to several problems surrounding the continuous integration process and the evolution of the software systems that support this process.

This master's thesis reports on the design and evaluation of two different prototypes for management of historical data in continuous integration systems. One of these prototypes uses a NoSQL database for storing historical data and one uses a relational database. The constructed prototypes where designed specifically to address problems related to scalability and performance of the current continuous integration systems in use at Ericsson. Evaluation shows that the prototypes solve scalability problems and increase performance of current systems by separating live and historical data. The value of the prototypes is further motivated by investigating how historical data in a continuous integration system can be utilized for the benefit of the company that has this data.

Keywords: data management, continuous integration, relational databases, nosql, polyglot persistence, big data

Acknowledgements

We would like to thank Ericsson for giving us the opportunity to conduct this thesis. Special thanks to John Haagen, our supervisor at Ericsson and everyone in the Framework team. We would also like to thank our supervisor Graham Kemp for all his help and feedback and our examiner Miroslaw Staron.

Contents

1

Introduction

In recent years, software companies have started to generate much more data about their development processes as a result of the wide adoption of continuous integration. Fowler [1] defines continuous integration (CI) as a software practice where developers in a team integrate work frequently, typically once a day or more [1]. Each integration is then verified by an automated build that runs tests, checks for code standard violations etc. [1]. According to Duval et. al [2], the proposed benefits of using CI include an increased project visibility where problems like failing builds can be noticed instantly. Therefore, data generated through the use of CI is crucial to the success of the practice, since CI stresses the need for transparency, information sharing and information visualization [1, 2]. Collecting this data also opens up companies to analyze trends about their development processes in the long term [2, 3]. However, Schram and Anderson [4] state that having software systems that generate large amounts of data on a daily basis is not without consequence and likely introduces challenges. These include having the appropriate technical infrastructure and software in place to deal with data growth and also supporting different use cases for operating on collected data.

The company Ericsson recently began adoption of continuous integration. At the company, there are two high level use cases for data collected from the continuous integration process. The first is to visualize newly created data, in real time, to developers. The second is to perform analytics, making use of historical data. This use case is of interest for the management staff as a means to quantify the effectiveness of the development process, to see if for example the proportion of successful builds increase over time.

Since the adoption of continuous integration at Ericsson, several software systems in support of this process have been put into use, including a web application developed internally called CIMS (Continuous Integration Management System). It has several instances, one for each of the larger software products being developed, and its main purpose is to address the first high level use case, namely to visualize data for developers. However, it is currently also used for the second use case.

Recently it has been observed that the amount of builds registered into CIMS is increasing. At the same time the amount of test cases for each build is also increasing. The technology stack of CIMS was originally not made to scale with this rate of

monotonic growth which in turn has introduced problems that impact performance, maintainability, changeability and other factors.

The influx of data for one of the larger developed software products are shown in figure 1.1. The trend is increasing, with different factors involved in describing why it looks like it does. As the situation is today, the data growth this trend describes has caused a variety of problems with the current infrastructure supporting the continuous integration process.

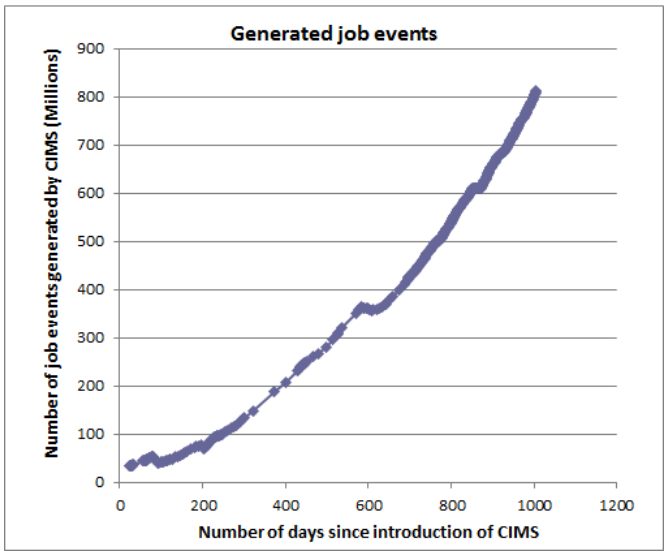

Figure 1.1: Diagram showing the data growth for one instance of CIMS.

One of these problems with the current implementation of CIMS stems from the need to evolve the system, specifically its data model and schemas. Currently, all system data is stored in a single relational database. For evolution to take place, all data must be migrated to match new versions of the system. At current rates of stored data, these migrations can take as much as a week's time to complete. Due to the long running migrations, the evolution of CIMS is impaired. This has halted efforts to improve the continuous integration process, thus reducing Ericsson's ability to respond to market changes.

1.1 Problem description

The context for this thesis is the intersection of continuous integration systems and data management. The focus is on the management of historical data and problems related to monotonic data growth. As such, we have broken down the research problem into two parts; the first part is to investigate different solutions to manage historical data that grows monotonically. The second part is to justify the value of having a solution by understanding how historical data in continuous integration systems can be used and what value it brings to the organization that possess the data.

Both these problems were addressed using a design research approach. This approach is described in detail in the sections 1.2, 1.3 and 1.4 where we also present the research purpose and research questions.

1.2 Research Purpose

The purpose of this study is to identify and evaluate solutions for management of historical data in a continuous integration system, more specifically, to evaluate the suitability of polyglot persistence (see section 2.4) in that context. Based on the problem description and the research purpose, we ask the following research questions:

RQ1: How can historical data in continuous integration systems be managed in the long term?
RQ2: What value can a solution to management of historical data in continuous integration systems bring to an organization?

1.3 Research Method

This thesis will be conducted using design research methods based on knowledge from [5, 6, 7]. Design research is performed by building and evaluating artifacts that are designed with the purpose of addressing a specified business need or problem [5].

Peffers et al. [6] describe an overall process for conducting design research which for the purposes of this thesis has been implemented as in figure 1.2. As can be noted in this figure, one important aspect of design research is its focus on iterative processes, where evaluation should inform reformulation of solution objectives and also give feedback to upcoming iterations of design and development. This part of the process will be implemented by performing several iterations of design/development and demonstration/evaluation. If necessary, solution objectives can be changed as an increased understanding of the problem is gained.

According to [6], there are several possible entry points for a design research study. In our case the entry point is an objective-centered solution, since the need for the study originated from an industrial problem that could be addressed by constructing an artifact.

In this thesis, our implementation of the process described by Peffers et al. [6] laid out the overall framework used to design the study. First, we studied and defined the problem in more detail in collaboration with the CIMS development team. After this, we proceeded with the formulation of solution objectives which set the scope

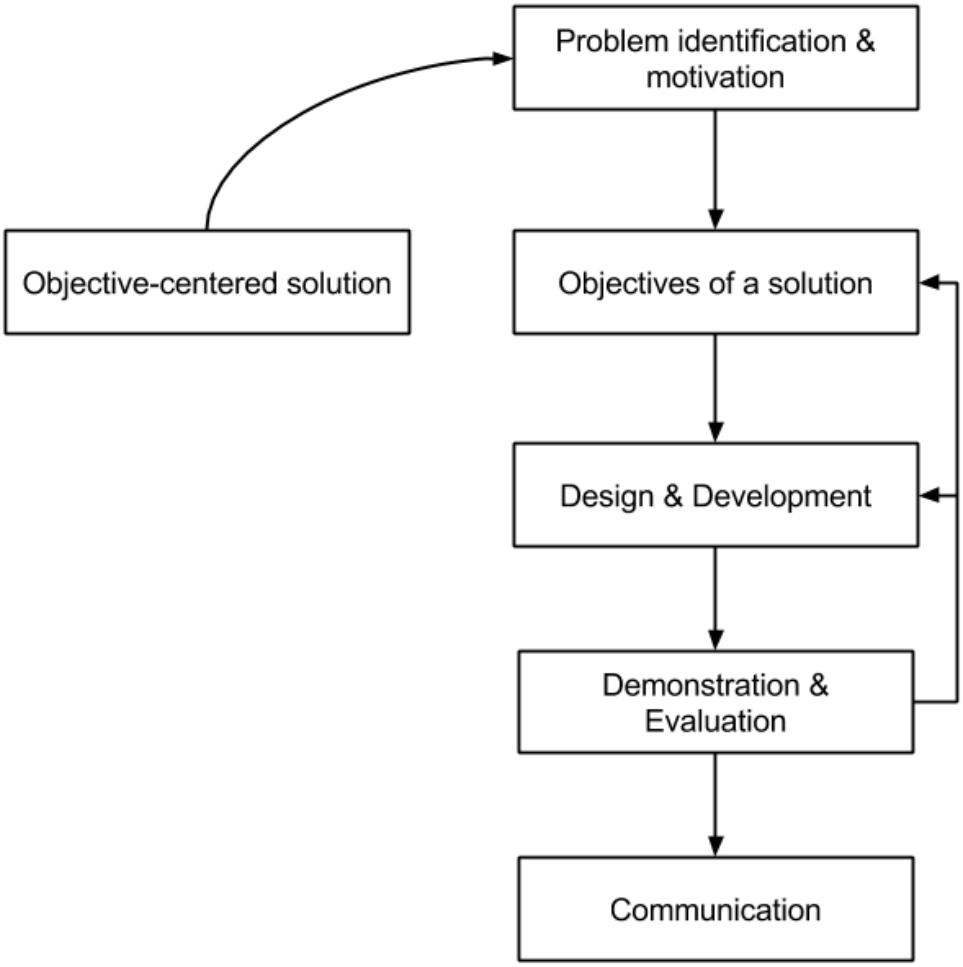

Figure 1.2: The design research process as described in [6].

for this thesis. During this process, an overview of related work was written.

With solution objectives in place, work began on the two bigger phases, namely design/development and evaluation. In a given iteration of design/development, an artifact in the form of a prototype was either built from scratch or further developed.

For the evaluation process, options given by [7] were used to define properties of the artifact, its purpose and actual approaches used for evaluation. The choices made are shown in figure 1.3.

Further explanations of some of these choices are warranted; the goal of the evaluation is to gather knowledge about the problem and its domain so as to put upcoming decisions made at Ericsson on a rational basis. As such, evaluation goes in line with the knowledge function described in [7].

A prototype solution was requested by stakeholders at Ericsson which is why the choice was made to evaluate using this method. Since we evaluate a prototype it is the artifact itself that is evaluated as opposed to the construction of the artifact.

Given that the thesis was conducted in collaboration with Ericsson, the artifact was evaluated in an organizational context and as such against the real world. Finally, we chose to do evaluation after the construction (ex post) of the artifact.

Variable	Value				
Approach	**Qualitative**	**Quantitative**			
Artifact Focus	**Technical**	Organizational	Strategic		
Artifact Type	Construct	Model	Method	**Instantiation**	Theory
Epistemology	**Positivism**	Interpretivism			
Function	**Knowledge**	Control	Development	Legitimization	
Method	Action research	Case study	Field experiment	Formal proofs	
	Controlled experiment	**Prototype**	Survey		
Object	**Artifact**	Artifact construction			
Ontology	**Realism**	Nominalism			
Perspective	Economic	Deployment	**Engineering**	Epistemological	
Position	Externally	**Internally**			
Reference Point	Artifact against research gap	**Artifact against real world**	Research gap against real world		
Time	Ex ante	**Ex post**			

Figure 1.3: Evaluation matrix. Choices made are marked in bold and colored.

1.4 Solution objectives

In adherence with the design research process, a set of objectives that define a successful solution has been defined in collaboration with the CIMS development team. The main objective is to develop an archive system that physically separates stale data from live data in order to further support the evolution of CIMS.

While stale data is not accessed often, it is still useful for analytics purposes and to respond to trouble reports from customers running older versions of Ericsson products. Therefore a secondary objective is to retain as much functionality as possible in terms of queries on the archived portion of the data. The specific queries that the archive should support have been defined through discussion with the CIMS development team and the metrics team.

For the combination of CIMS and the archive system to be future proof it must also be able to scale better with overall data size than CIMS currently does. This ensures that the problem is simply not pushed further into the future but rather solved in its entirety.

In summary, 3 main objectives have been identified. They are as follows;

1. Develop an archive solution that physically separates live from stale data.
2. Retain a set of important queries on archived data.

3. Introduce long term scalability.

1.5 Report structure

This report is structured in the following way; In chapter 2 the background and related work is described. Chapter 3 describes the domain of CIMS and the business context in more detail. In chapter 4 a description is given of how the artifact addressing the identified problem was designed and implemented. This is followed in chapter 5 by a description of how the artifact was evaluated. Results and discussion of this evaluation are given in chapter 6. The report is then wrapped up in chapter 7 with conclusions.

2

Background

In this chapter, a review of related work is presented. First, the context for the identified research problem is covered, namely that of software development using continuous integration. This context is connected to the field of big data which in broad terms is about management of large quantities of data. Finally, sections about NoSQL and polyglot persistence are more technical in nature and knowledge from these fields is used to aid the design process of the archive prototypes.

2.1 Continuous Integration

As was stated in the introduction, CI is a software practice where work is integrated frequently, preferable more than once a day [1]. A very central concept to CI is the automated build, which should be run in many stages of the development cycle. Duval et al. [2] state that the purpose of the build is to "put source code together and verify it is working as a cohesive unit". Builds typically compile code, test, inspect, deploy and so on [2]. CI stresses the need for this process to be automated as no part other than perhaps starting the build should be performed manually. CI also implies use of some version control system (VCS) to integrate work and a CI server that waits for changes in the VCS and automatically runs integration builds when they are detected [1].

It is useful to make a distinction between private and integration builds. The difference can be illustrated with an example workflow; first, when a developer is done with a task, a private build should be run. Only if it is successful the work should be pushed to the used VCS which will trigger an integration build. While private builds can be seen as trial and error, where they can fail often and be fixed, the goal for the integration build is that it should never fail. If it does, then it becomes a top priority to get it fixed [1].

This kind of workflow is useful to understand and relate to the context at Ericsson and CIMS. However, CIMS is not a CI server. Instead it aggregates, stores and presents data from different systems, including the CI server, VCS and code review systems.

2.2 Big Data

Since the internet was invented, big data is considered to be the phenomenon that has received the most attention in the computing industry [8]. Today's popularity for big data is mainly derived from the current maturity in technologies combined with the ongoing decline in hardware costs [8, 9]. This has led to the development of systems which enable the processing and handling of big data [8].

Big data can be described by three characteristics; volume, velocity and variety. Volume is the quantity of data generated. As an example, Walmart gathers more than 2.5 Petabytes of customer transaction data per hour [3]. The velocity of data gathering should be nearly real-time, which enables the organization to be more agile. Data gathered can vary in structure and origin, which gives the ability to correlate data from different sources. If any of these three characteristics are so extreme that they can't be processed using traditional technologies, it is considered to be big data [8]. The boundary of when data becomes big data is influenced by which sector a system is in use. It is always in movement along with advancements in technology for that sector [9]. In the context of CIMS, the relational database has reached such boundaries. This is due to the volume of data in CIMS and the forthcoming increase in influx of data.

When traditional relational databases are used, due to its limitations, analytics can only use relative small data sets to create models. The wanted level of certainty in the models is thereby harder to achieve [9]. With big data and its technologies, analytics now has the possibility to use far more data when creating predictive models. As described by Minelli et al. in [9], big data enables analytics to see the whole iceberg, not just what is visible above the surface. With the capability to gather massive amounts of data, more intelligence can be mined from the data which can, if done right, be translated into business value [3].

2.3 NoSQL databases

Cattell [10] provides an overview of the current dialogue in the database world, primarily regarding big data trends and how to address the problem of scalability depending on technology and applications. The paper focuses on four different categories of databases; key value stores, document stores, extensible document stores and relational databases. Out of these, the first three fall under the category of NoSQL databases. Differences with traditional SQL databases typically boils down to trade-offs made about consistency in favor of scalability and flexible schemas.

NoSQL databases have been designed primarily to meet modern demands on online applications that need to handle a large amount of concurrent users and store large amounts of possibly unstructured data [10]. As described above, a common way of comparing NoSQL and SQL databases is in terms of consistency and demands

on it. SQL databases typically have ACID [11] transactions, meaning 'Atomicity, Concistency, Isolation and Durability'. ACID transactions are strict in the sense that they ensure that the every user always has exactly the same view of the database. Such constraints are very important for some use cases like the handling of banking transactions but less important for others, such as status updates in a social network.

When consistency can be relaxed, weaker constraints can be employed. This has given rise to the BASE [10] acronym, meaning 'Basically Available, Soft State, Eventual Consistency'. Databases using this approach are eventually consistent, meaning updates will eventually propagate throughout the system. The idea is that when relaxing consistency, much greater performance and scalability is possible [10].

The NoSQL databases mentioned in [10] also provide scalability by horizontal scaling in a distributed manner, meaning that the data in the database can be spread across multiple servers. This distribution of data among multiple servers enables the database to meet the modern demands in both throughput and scalability [12] mentioned earlier. A distributed service is often designed with the desire to achieve three key characteristics; consistency, availability, and partition-tolerance [13]. However, these are impossible to achieve all at once in a distributed environment [13]. Therefore a prioritization must be made as a distributed database system can only possess two out of these three characteristics, in [13] called the CAP theorem. Due to the prioritization made by the NoSQL variants mentioned in [10], they achieve scalability and availability in favor for consistency.

Four different categories of NoSQL data stores mentioned in [10] and [14] are described in the sections below.

2.3.1 Key value stores

Key value stores uses a key-value index as structure, each value corresponds to a key which is used for lookups and deletes for that value [10]. The data model used is a map or dictionary, which is simpler, compared to a relational table and gives a faster query speed compared to a relational database [15].

2.3.2 Document stores

Compared with the key value store where the key is used for look-ups, a document store uses a query based on the structure of the document combined with certain attributes like a primary key in order to retrieve data. The structure of the documents in a document store does not need to be defined upfront and the structure of the documents does not need to be homogeneous. As such, the document store is said to be schema-less [14]. The data structure of a document is a hierarchical tree which can contain lists, nested objects and maps [10, 14]. Due to the freedom in a document's structure and depending on which document store used, there exists

different techniques to model relationships between objects in a document store. These offer different benefits in terms of performance, size on disk etc.

2.3.3 Graph Databases

Many NoSQL databases mentioned in [10] consists of blobs of data with low coupling. However, in the case of graph databases the opposite is true. Graph databases consist of small records with high and complex coupling. A graph is a data structure containing nodes that are connected with arcs [14]. This highly interconnected structure gives the ability to query information based on the relationships between nodes, for instance "find the parts included in product X that were manufactured by Y and shipped by Z". The relational databases would model these relationships with foreign keys, but when querying complex relationships, the joins required becomes heavily performance demanding. The querying and traversal between the relationships becomes efficient and cheap in graph databases compared to the relational databases [14].

2.3.4 Column-Family Stores

These databases are categorized as extensible document stores by [10]. The data model is similar to relational databases in that data consists of rows and columns [10]. However, these systems are designed to be distributed by sharding records either horizontally or vertically across nodes [16]. Similar to documents in a document-store, rows in a column-family store need not adhere to a predefined schema and can contain different attributes.

2.4 Polyglot persistence

There is a consensus in contemporary database literature that relational databases are not going to go away anytime soon [10, 14, 15]. They are, however, no longer the one and only option for every problem. Instead, a contemporary viewpoint is that no data model can and should have the purpose of describing all kinds of domains [14, 15]. In practice this means that trying to model all data to fit into one kind of database, relational or otherwise, is not a prudent task. Instead, other strategies [17] exist for addressing the task where a more recent one is the concept of polyglot persistence [14].

The word polyglot means something that is made up of several languages. In the field of software this is often meant to mean a system that is built with several different programming languages. Similarly, polyglot persistence means that a system leverages several different databases. Rarely does all the data of a software system

match nicely with one kind of data model. So instead of using the same data model for everything, databases with different data models and characteristics are used for specific kinds of data within a system. That is in essence, the idea behind polyglot persistence. To illustrate, figure 2.1 shows a design example of an e-commerce platform where different types of data exist along with database choices that are well suited to store this data.

Polyglot persistence is closely related to the rise in popularity of NoSQL databases. These databases should not be considered general purpose [10, 18] but should instead be used for solving specific problems involving specific kinds of data. For the purposes of this study, polyglot persistence is an interesting approach that warrants further investigation. Since use cases for live and historical data are different and given the fact that the amount of collected historical data will grow forever, we believe that using different kinds of databases for live and historical data might be beneficial.

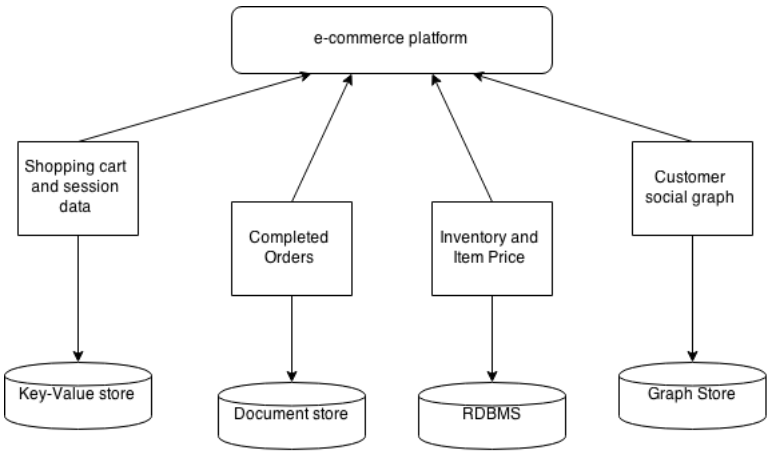

Figure 2.1: Example implementation of polyglot persistence, adapted from [14].

3

Domain

This chapter begins with a high level description of the domain. This is followed by a description of the schema in the relational database of CIMS and a depiction of the data influx rate in CIMS. The chapter ends with potential uses cases for an archive solution and the database queries needed for those use cases.

3.1 High level schema

For the feasibility of this study, the domain of CIMS has been narrowed to some key entities, which are the following; products, revisions, builds, test suites, test cases and trouble reports. Excluding trouble reports, these entities form a hierarchical structure as seen in figure 3.1. Products are at the top of this hierarchy. They represent the different software systems that are being developed at Ericsson. Products are developed iteratively, where a released iteration is called a revision. At the revision level, developers make changes to source code, configuration, tests etc. which are then checked into a revision control system and subsequently built and tested. This is referred to in domain terms as a build. Each build runs a number of test suites, which in turn consist of a number of test cases.

As has been noted, a trouble report sits outside of the hierarchy. This entity represents a bug or defect which arrives from external (i.e. a customer) or internal sources. It can be mapped to different entities from products down to specific test cases. Trouble reports are handled by implementing a fix in a specific build.

3.2 Existing relational implementation

The relational database schema in use by CIMS represents the above mentioned entities in a different abstraction. Representation of these entities is described in depth in this section. The schema of the relational database in CIMS is presented in figure 3.2.

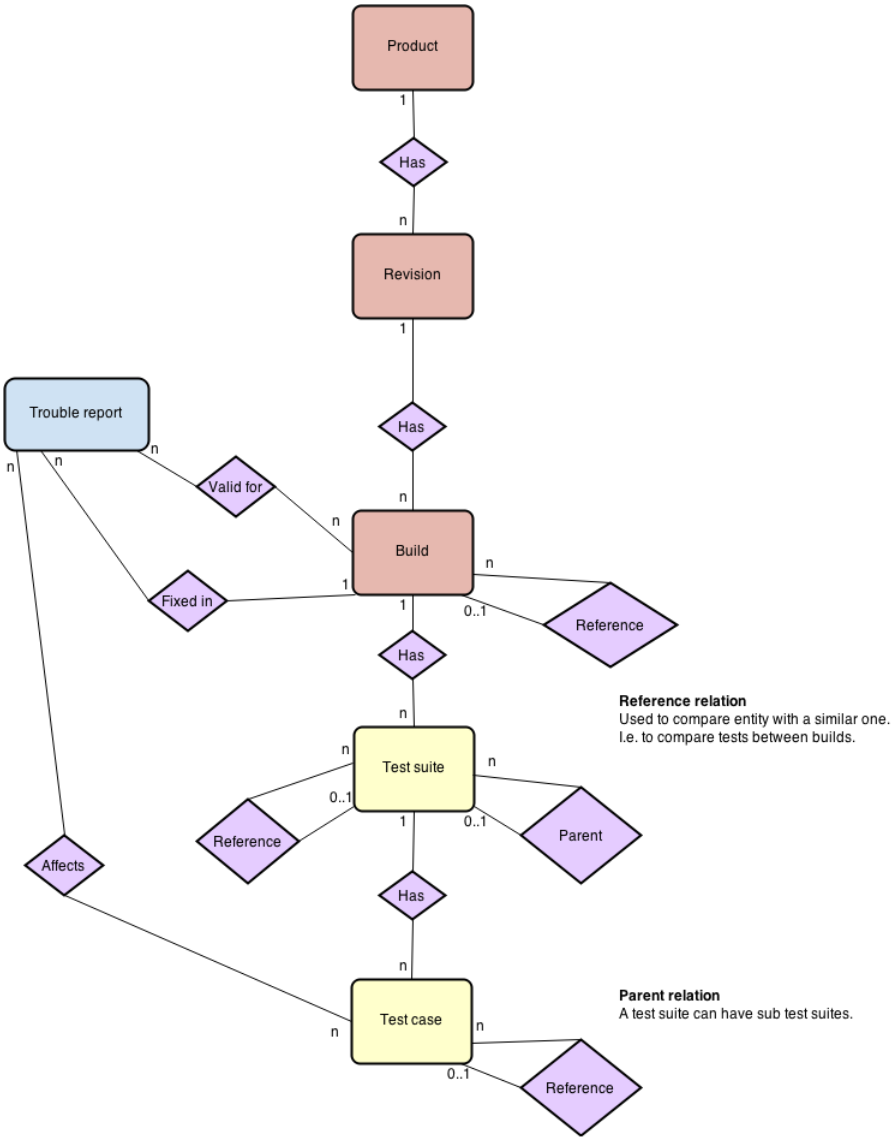

Figure 3.1: High level diagram of the domain.

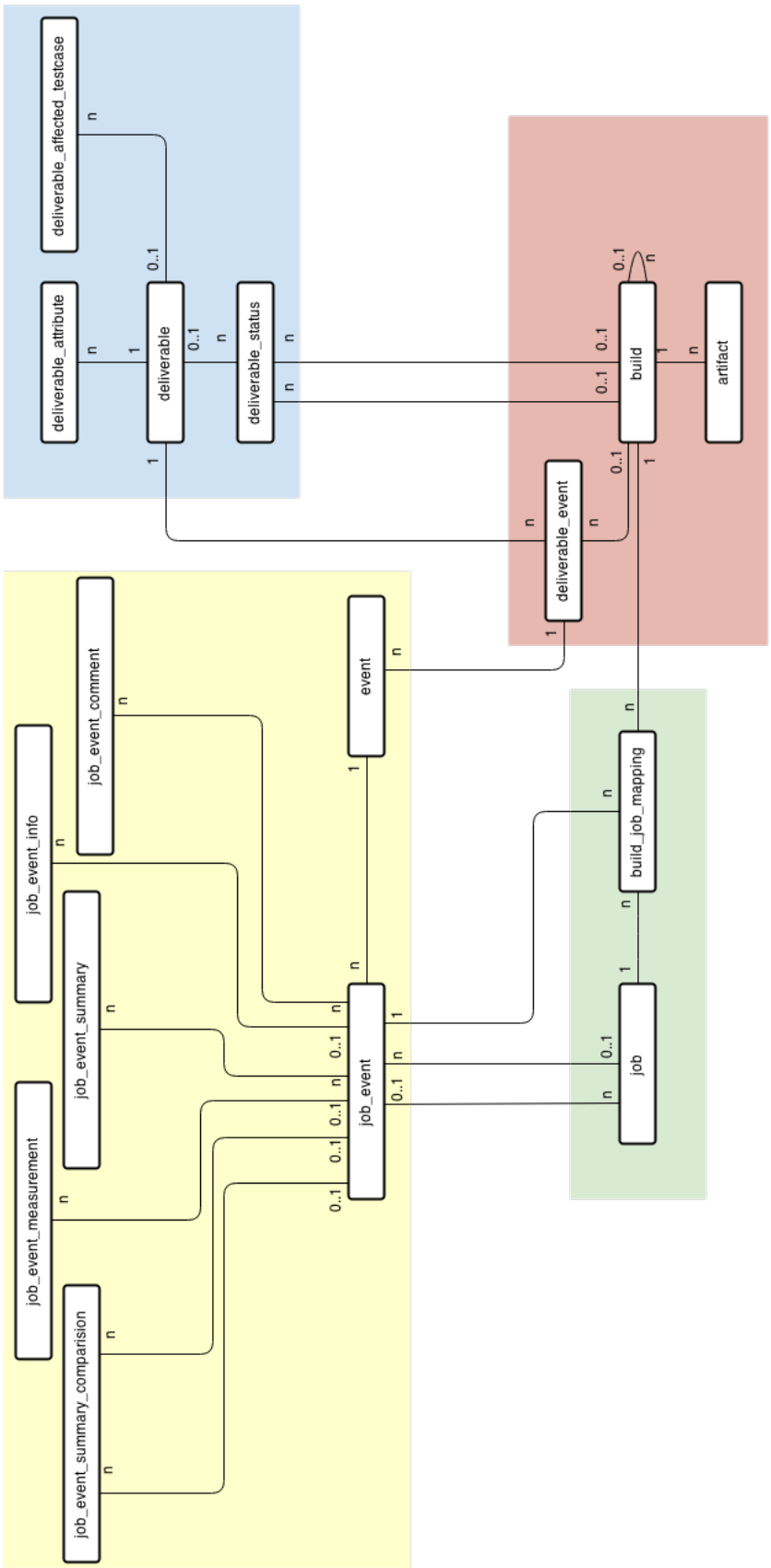

Figure 3.2: Diagram of the relational database in CIMS.

3.2.1 Build

In the relational database, a build is represented by the build table. Each build can possess a number of artifacts and deliverable events, as seen in figure 3.3. An artifact can for instance be the path to a binary file that is included in the build. A build also contains attributes about which product and revision it belongs to. The deliverable_event table is for defining which deliverables (i.e. trouble reports) that belong to the build.

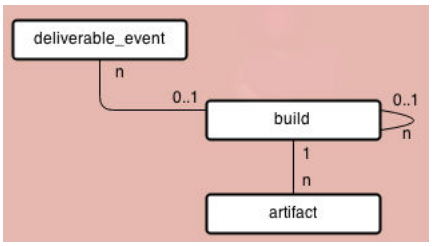

Figure 3.3: Diagram of tables related to builds.

3.2.2 Deliverable

A deliverable in the relational database represents a trouble report described in figure 3.1. As seen in figure 3.4 a deliverable is defined by the following tables; deliverable, deliverable_event, deliverable_status and deliverable_affected_testcase. These tables contain normalized data about a deliverable.

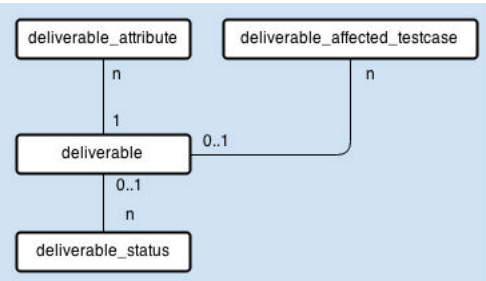

Figure 3.4: Diagram of tables related to trouble reports.

3.2.3 Job event

The relational representation of a test case and test suite described in figure 3.1 is the job event entity. Each test case and test suite registered in CIMS becomes a job event and the job events that are related forms a tree structure describing the execution of a test suite. As seen in figure 3.5 a job event is represented by the

job_event table which is linked to multiple tables containing normalized data about the job event.

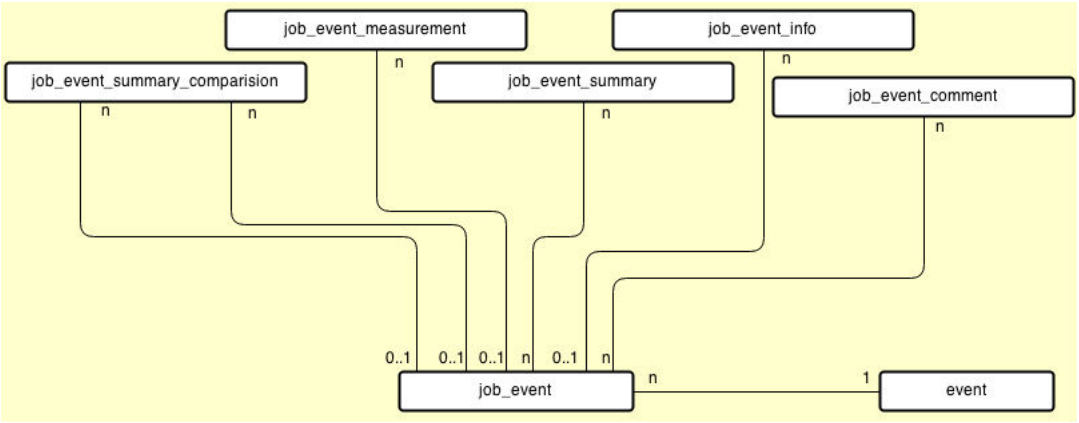

Figure 3.5: Diagram of tables related to test cases.

3.2.4 Job

A job in the relational implementation does not have a directly corresponding entity in the high level diagram visualized in figure 3.1. The job is a technical entity that groups related test cases and test suites. As seen in figure 3.6 a job is represented by the job table which is linked to a build via the build_job_mapping table.

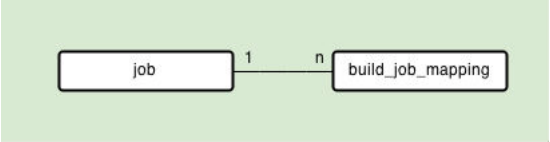

Figure 3.6: Diagram of tables related to jobs.

3.3 Use cases

Several use cases have been identified for the archive that can provide business value to Ericsson and thereby motivates why saving data in an archive is viable. These include the ability to have information about each build, even builds that were executed years ago and the ability to perform analysis on larger data sets.

The first use case is an important part in Ericsson's process of managing bugs that is found by the customer running live versions of a product. Even if CI leads to shorter release cycles, the lead time of deploying new software at a customer is still relatively long. It can also be the case that the customer doesn't have the need or resources to upgrade to a newer software version. This leads to that customers can use software

that was released years ago. When a customer finds a bug or their system crashes, in order to quickly fix that bug, it is crucial to have all information about that particular build in use by the customer. Information such as the outcome of all tests performed, warnings in the build procedure are examples of such necessary information.

The second use case can give Ericsson further insight in the quality and trends in their development processes. For Ericsson to be able to improve in their processes of building and designing software, meta data of the software's developed needs to be analyzed. The data in CIMS needs to be correlated with other sources of information, such as the trouble report system. This use case will provide value to roles such as managers, they can in nearly real time see if the quality of the produced software is declining or increasing compared to earlier measurements and can be more certain that it is not just a temporary event because they will have the ability of doing analysis on years of data.

3.3.1 Queries

In order to manage the use cases mentioned in the section above, the archive needs to support a large set of queries. Below we describe one of the more interesting queries, namely looking up a test case history. It is described in domain terms and in terms of the relational implementation of CIMS. More queries can be seen in appendix A.

3.3.1.1 Get test case history

This query should return the history of a test case, identified by its name. It should group the results by product and for each of these it should show the builds containing the specific test case and the verdict of the test case in that build. This query cannot be expressed as a single SQL statement and is implemented on the application level in CIMS by performing several SQL queries and building a dictionary with the results. Pseudo code for this query is show in figure 3.7

3.4 Data generation rate

We end this chapter by presenting the current data generation rates for builds and events (i.e. test cases and trouble reports). This is useful so as to put the growth trends in a context. One part in designing a solution is to be aware of these trends. In the introduction the accumulated number of job events where shown in figure 1.1. The figure was chosen to illustrate the overall current data growth trend for one specific product at Ericsson. The cause for this trend is twofold. First the number

```
job_events =
    SELECT * FROM cims_job_event
    WHERE name = 'my_test_case'

// List comprehension
job_names = job_event.job_name FOR job_event IN job_events

jobs = SELECT * FROM cims_job WHERE job_name IN job_names

// List comprehension
root_job_event_ids = job.root_jobevent_id FOR job IN jobs

root_job_events =
    SELECT * FROM job_event WHERE id IN root_job_event_ids

builds =
    SELECT cims_build.* FROM cims_build
    INNER JOIN cims_build_job_mapping
    ON cims_build_job_mapping.build_id = cims_build.id
    WHERE cims_build_job_mapping.job_name IN job_names

(Use list of job_events, jobs,
root job events and builds to create result dictionary)
```

Figure 3.7: Pseudo code describing the test case history query.

of builds registered is increasing, as can be seen in figure 3.9. Second, the number of events (which are primarily made up of job events) for a build is increasing as well, albeit not that aggressively. This ratio is shown in figure 3.10. As such, every month, the number of generated events increases, as seen in figure 3.8.

While the purpose of this study is not investigate why more builds are registered and why builds increase in size, as already stated we believe these figures are important to put the problem in a context. They clearly show the current situation and in part motivate why this has recently become a problem at Ericsson.

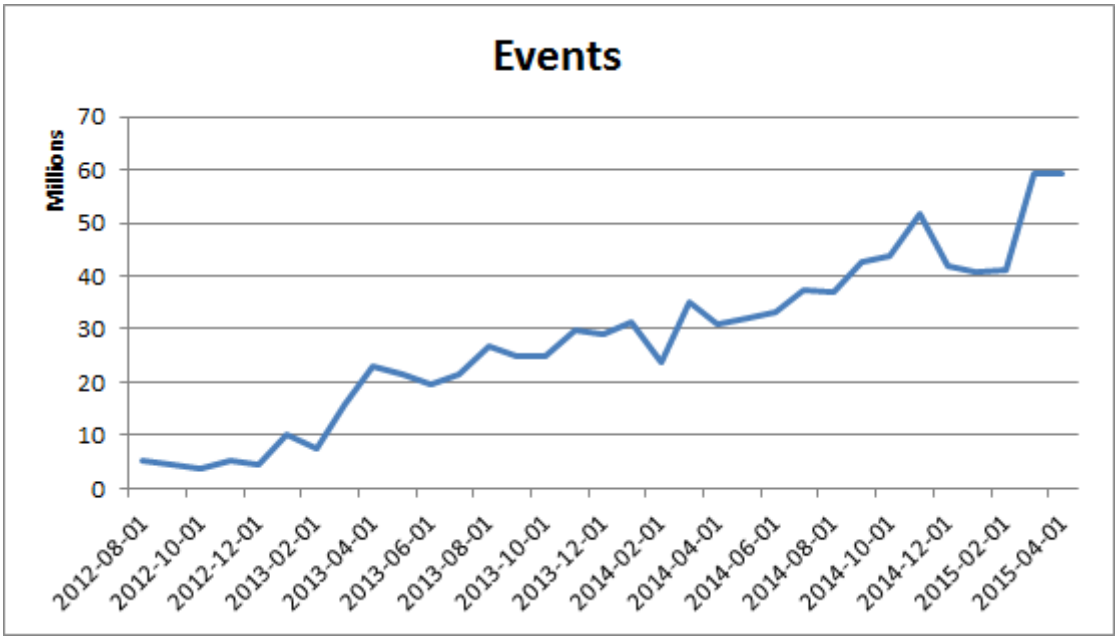

Figure 3.8: Generated events per month. Not accumulated.

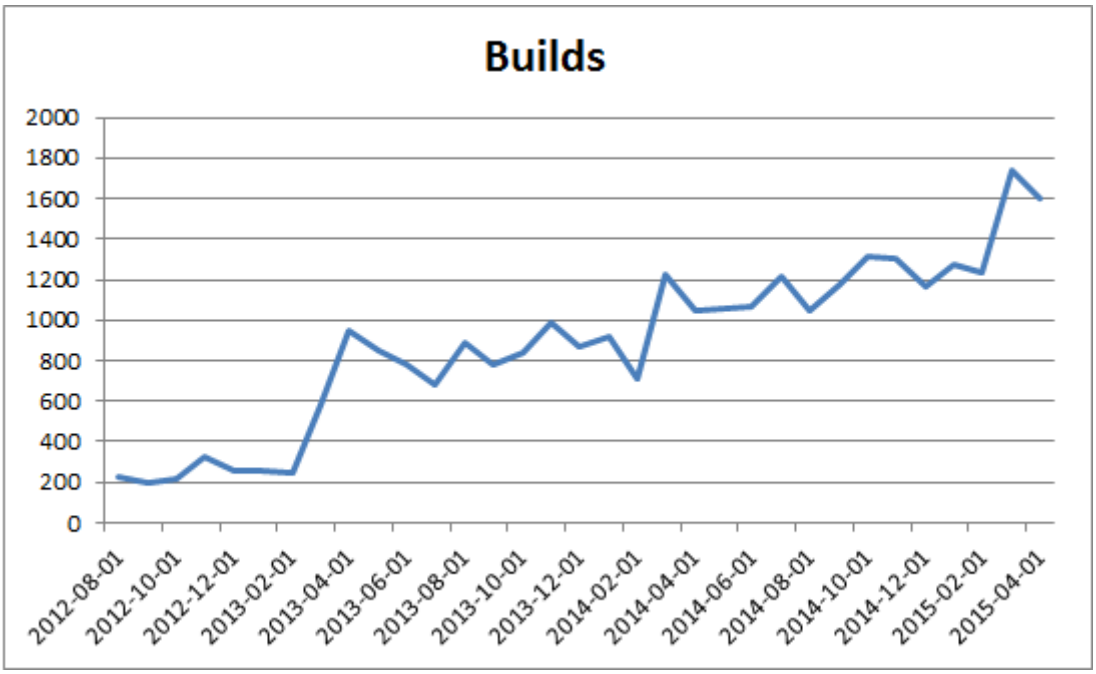

Figure 3.9: Generated builds per month. Not accumulated.

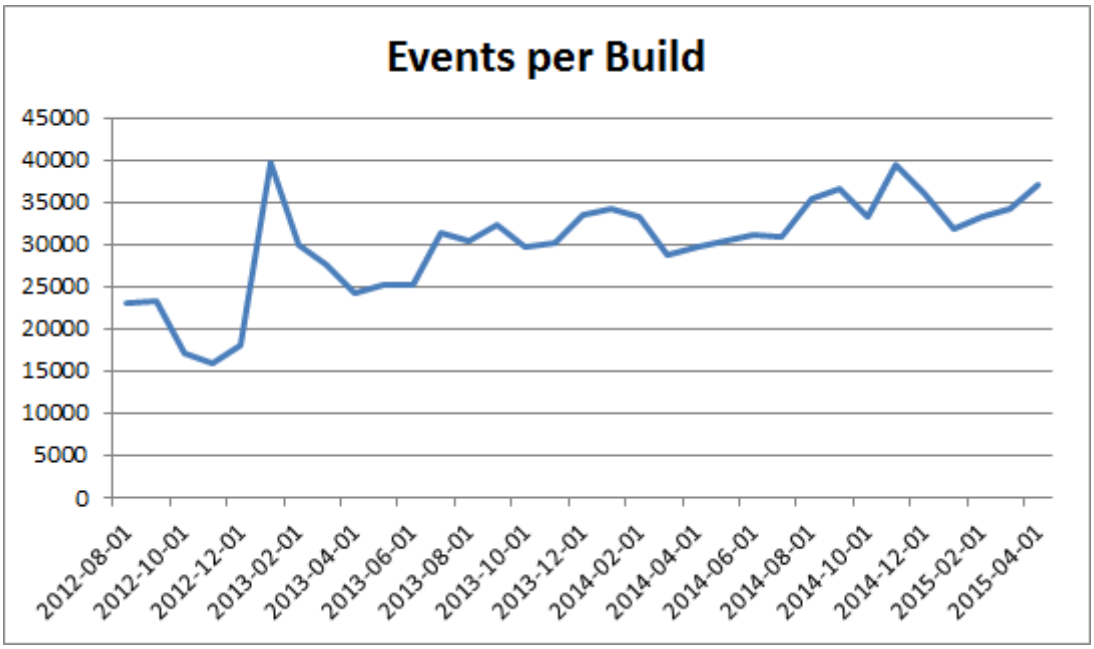

Figure 3.10: Events per build. Not accumulated.

4

Design

This chapter describes the design of the artifacts that are to be implemented in the form of prototypes. Some choices related to the design, like database schemas and choice of databases for the prototypes are motivated in more detail. Initially, several candidate databases were considered but these were narrowed down to two choices that were found to be interesting enough to move forward with. Before describing the different design choices, we recall the previously defined solution objectives:

1. Develop an archive solution that physically separates live from stale data.
2. Retain a set of important queries on archived data.
3. Introduce long term scalability.

Through the review of related work, it has been established that non-relational databases might be appropriate as a part of a solution to the identified problem of this thesis. However, it has also been suggested [15] that the introduction of non-relational storage should not replace current relational databases, but should instead work as a complement to them. We argue that a polyglot persistence solution is a natural approach to the identified problem.

Such a solution would then address objective 1, namely to physically separate live from stale data. Before implementation, a decision has to be made on what types of databases should be used as the underlying storage for the archive. If the only objective was data separation then the simplest choice would be to completely duplicate system data in its current form to a similar database. However, this does not take into account other objectives and would likely not solve the underlying problem.

When selecting what databases to use we need to address the issue of long running schema migrations. One way this could be addressed is by using a database with a flexible schema, capable of handling heterogeneous data. This database should also have query capabilities that allow us to effectively address objective 2; the support of important queries.

In order for the archive system to scale naturally with the current level of data growth (objective 3), it is prudent to select a database with horizontal scaling. In

addition to this, it would be highly beneficial with more efficient data compression than what is currently in use by the relational database of CIMS.

While we began looking at two different NoSQL databases and one relational database to use for implementing archive prototypes, we ended up moving forward with one NoSQL database (Elasticsearch) and one relational database (TokuDB). They are described in more detail in sections 4.1 and 4.2. The reason for selecting these two was due to an increased knowledge over time about the domain, archive use cases and the technologies themselves. Following the initial iterations, we came to the conclusion that Elasticsearch was the more interesting of the two NoSQL alternatives to pursue further and that doing so would yield more interesting results.

4.1 Elasticsearch

Elasticsearch [19] is a NoSQL database classified as a search engine, with many similarities to a document store, specifically, its data model and the ability to scale horizontally. The difference is mainly in query capabilities, which are optimized for full text search. However, supporting full text search comes at a price of higher demands on hardware. We believe this to be an interesting trade-off to consider, and a comparison between the other candidate database and Elasticsearch should yield useful practical information on how to proceed with the development of an archive solution after the study has been conducted.

4.1.1 Data model for non-relational archive

With the domain description as a basis, the schema of the relational database in CIMS should be transformed to suit the strengths of the chosen NoSQL database. A level of denormalization is prudent since any relationship between collections in a document-store has to be implemented in application logic. However, no relationships at all would not be suitable for the types of queries that need to be supported. As such, the data model of the archive needs to be well designed. Given the use of document-stores and their data model, one natural change is to convert one to many relationships into lists of embedded documents. Another change is to denormalize all one to one relationships.

To illustrate the transformation process, we present a diagram in figure 4.1 describing a proposed non-relational schema for the archive. This can be compared to the original relational schema in figure 3.2.

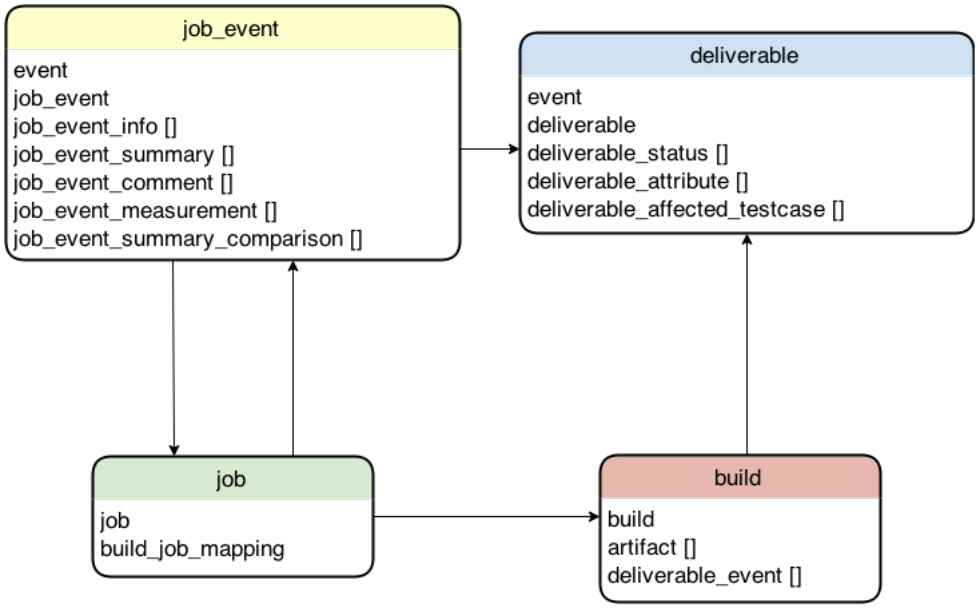

Figure 4.1: Representation of the non-relational model. Bracket pairs represent a list of embedded documents.

4.2 TokuDB

The second candidate database is MySQL using TokuDB [20] as the storage engine. Storage engines are software components that handle SQL operations without changing the overall architecture and interfaces of the database [21], in this case MySQL. TokuDB aims to bring MySQL up to par with NoSQL databases in terms of scalability and flexibility, while still retaining the ACID transactions that are common in relational databases. It does this by addressing many of the weaknesses of MySQL and its most widely used storage engine, InnoDB, which is also the storage engine for CIMS instances at Ericsson. Some examples of improvements TokuDB has over InnoDB are; much higher compression rates, a new type of index (Fractal Tree [22]), and hot schema migrations and index creations. Hot in this context means to perform these operations without at the same time locking database access.

An archive prototype will be developed using MySQL with TokuDB as the storage engine. This will provide a contrast to Elasticsearch and has potential benefits that the NoSQL approach does not have. For example, the same schemas can be used for CIMS and the archive, making data transfer seamless between the two. However, the effectiveness of this solution is highly dependent on whether schema migrations are manageable with large data sets.

4.3 Software components

A set of software components should be developed in order to effectively support the databases used for the archive prototypes. These components serve as glue code between the relational database of CIMS and the database of the archive. An overview of the proposed software architecture containing these components is presented in figure 4.2.

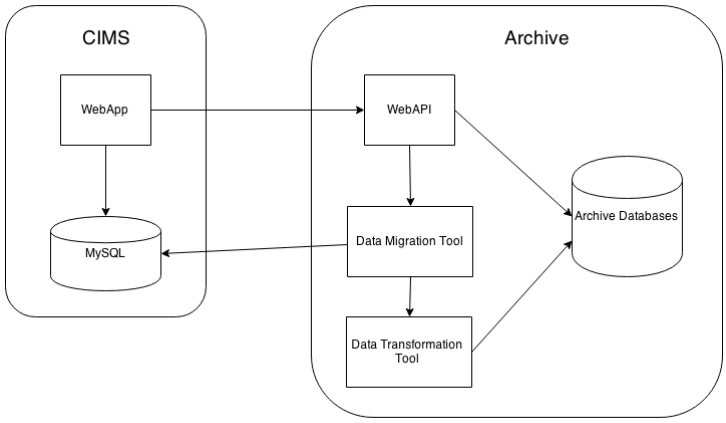

Figure 4.2: High level representation of the prototype architecture.

4.3.1 Data migration tool

Data needs to be extracted from the live relational database before it can be transformed and migrated elsewhere. This component groups data into chunks that makes sense to migrate as a unit.

4.3.2 Data transformation tool

After data has been extracted, it is transformed to fit the new schema in figure 4.1. In order to transform the relational data to a viable non-relational structure, the data needs to be denormalized. This is made to avoid expensive application-side joins that incorporate multiple round trips to the database. As seen in figure 4.1, the tables corresponding to a 'job_event' and an 'event' has been grouped into a single document. Once data has been transformed it is ready for insertion into the archive.

4.3.3 Archive API

To demonstrate the capability of the archive to respond to queries, an API should be developed that capture these queries.

For Elasticsearch and most NoSQL databases, operations on data can be made via an HTTP REST API, where operations are defined by JSON objects.

The set of important queries defined in section 3.3 can be translated as follows for the different archive databases. For brevity we only show an example of looking up a test case history, seen in figure 4.3. The rest of the translated queries are included in appendix A.

```
job_events = POST http://localhost:18843/job_event/_search
{
    "query" : {
        "filtered" : {
            "filter" : {
                "term" : {
                            "name" : "my_test_case"
                }
            }
        }
    }
}
job_names = job_event.job_name FOR job_event IN job_events
jobs = POST http://localhost:18843/job/_search
{
    "query" : {
        "filtered" : {
            "filter" : {
                "terms" : {
                            "job_name" : "job_names"
                }
            }
        }
    }
}
root_job_event_ids = job.root_job_event_id FOR job in jobs
root_job_events = POST http://localhost:18843/job_event/_search
{
    "query" : {
        "ids" : {
            "values" : ["root_job_event_ids"]
        }
    }
}
build_ids = job.build_job_mapping_build_id FOR job in jobs
http://localhost:18843/build/_search
{
    "query" : {
        "ids" : {
            "values" : [build_ids]
        }
    }
}
(Construct result object from list of builds, jobs and job events)
```

Figure 4.3: Implementation of the test case history query using Elasticsearch.

5

Evaluation

In this chapter the different evaluation criteria of the artifact are described and motivated. The artifact is evaluated by metrics that capture the solution objectives and provide relevant data used to address research questions. The results of the evaluation are presented in the corresponding sections within chapter 6.

5.1 Data separation

How well the separation of data works is evaluated qualitatively by reasoning about the general process of migrating data from the live database in CIMS to the database of the archive. This process is dependent on the effectiveness of the software components that are constructed to support the database of the archive. A crucial part of this process is the transformation and denormalization of relational data. The process of separation is also dependent on how the archive database is configured and how suitable it is for the task.

5.2 Query support

The usefulness of the archive depends on what queries it supports and how these perform at scale. As a basis for evaluation, the set of important queries defined previously in section 3.3 is used. Evaluation is done by comparing this set to the set of queries implemented in the archive prototypes which are previously described in section 4.3.3.

5.3 Scalability

The following points in this section focus on the ability of an archive prototype to scale well with current growth trends. This includes disk usage, memory utilization, insertion time and horizontal scalability.

5.3.1 Disk usage

For the archive to scale, disk usage must be manageable, even with billions of documents in the archive. The metric used is derived from 'Memory utilization' defined in [23], memory is in this case the disk space used by the database of the archive. In order to assess the archive's capacity for long term scalability compared to the live database, measurements are done on both migrated data in the archive and the equivalent data in a CIMS instance.

5.3.2 Memory utilization

The metric used originates from the internal metric defined as 'Maximum memory utilization' in ISO/IEC standard TR 9126-2 [24]. Measurement is made by using the archive to perform tasks like insertion and querying and estimating the amount of memory that is needed to perform the task adequately.

5.3.3 Insertion time

For the archive to be viable in practice, insertion of new documents into the archive must be efficient. Even with a large number of documents already in the archive, the insertions must be made in adequate time, which in this case means faster insertions than the data generation rate for a CIMS instance. Currently, the CIMS instance for the largest product being developed produces about 2 million job events per day, and as already stated the trend is increasing (see figure 3.8). Measurements are done on insertions of data originated from the live database and generated dummy data similar in size and structure to real data.

5.3.4 Query response time

To assess how performance of a prototype scales with the size of the data set, query response times will be measured on some important queries that an archive should support.

5.3.5 Schema changes

Since one of the archive prototypes (TokuDB) uses a relational database, its database schema must match the schema of the CIMS instance that feeds the archive with data. Therefore whenever the schema is changed in the CIMS instance, the schema in the archive must be changed as well. To see how well this operation performs with a large data set, schema changes will be done on the archive prototype and

the running time of the operation will be measured. This evaluation point does not apply to Elasticsearch, since it is essentially schema-less.

5.3.6 Horizontal scalability

The ability of the archive to scale horizontally is evaluated by qualitatively investigating each constructed prototype's capability to do so. In this case, capability is defined as the database's characteristics in the following areas: the amount of upfront configuration needed to start a cluster and the amount of configuration needed to add new nodes to a running cluster.

5.4 Integration with CIMS

In order to evaluate the suitability of the polyglot persistence approach, some new functionality that leverages the archive database will be built into CIMS. More specifically, we have chosen to implement the viewing of builds. This new functionality should work as follows:

- User tries requests the web page for a specific build.
- System checks to see if the build and its related data exist in the live relational database.
- If a build is found in the live database then it is presented to the user as it normally would.
- If a build is not found then the system checks if the build exists in the archive database.
- If a build is found in the archive database then it is presented using the archive API.

5.5 Organizational suitability

The final point of evaluation is to assess how suitable a solution is from an organizational perspective. There are multiple factors to consider when evaluating the archives' suitability in the organization. For example, who the target users of the archive are, willingness to learn and adopt new technology, willingness to make use of archived data, increased maintenance effort and system complexity, and so on. These points would be discussed with relevant people in the organization and would also be extended by observations made throughout the course of the study.

6

Results and Discussion

In this chapter we present results from the evaluation phase described in chapter 5. Given the iterative nature of design research, some adjustments were made between iterations. As such, the results included in this chapter focus on the last iteration of development.

Access was given to a dedicated server[1] in order to achieve more control of the environment so as to get more dependable measurements. However, it should be emphasized that the purpose of doing the measurements was not to perform an experiment and we do not claim to have done so. Rather, the goal was to assess initial feasibility of the solutions and see if they would perform in an adequate manner. More specifically, we ask if the measurements of disk usage, memory utilization, insertion time, query response time etc. are good enough to warrant going forward beyond prototype construction.

6.1 Data separation

For evaluation and testing purposes, access was given to a CIMS instance with a database containing about 2 years' worth of data (see figure 1.1), where the job event table contains about 480 million rows. This is the largest database with real data that could be accessed for evaluation, without disturbing production environments. Using the developed software components for transforming and migrating data, all relevant data from the CIMS instance was migrated to the databases of the archive prototypes. Overall the migration process worked well in terms of transforming data to fit the non-relational model and using the developed software components to perform their respective tasks.

[1]Specs: Redhat Linux 6.6, 256GB RAM, 2TB storage

6.2 Query support

The set of important queries defined in section 3.3 has been implemented for the archive prototype using Elasticsearch. This section describes the outcome of this implementation along with reasoning about the complexity of the queries. Query support for TokuDB is not addressed since in this case, queries and schema are identical to CIMS so no translation is needed.

The set of important queries translated based on the non-relational schema (section 4.1.1) was defined previously in section 4.3.3. These queries were implemented as part of an API component than can be used for testing purposes and by other systems. Implementation of all queries was successful in the sense that the queries can certainly be expressed using the query language of Elasticsearch, if application side joins are allowed. However, the effectiveness of the queries is highly dependent on the design of the non-relational schema. Several differences in query properties of the archive prototype compared to the CIMS database has been noted, as is to be expected. Due to the denormalization of data, fewer joins are required for the non-relational databases and the joins that are performed are done on the application level. Joins on this level leads to more round trips to the database, but lightens the computational resources used by the database. As long as application side joins are simple in nature, i.e. fetch one record, get its id and fetch related records, we believe using application side joins are acceptable. For anything else, data should be denormalized so that joins are not needed.

6.3 Scalability

In this section, results are presented from the evaluation of scalability for the archive prototypes. It includes disk usage, memory utilization and insertion time, for both Elasticsearch and TokuDB. Since MySQL/TokuDB does not scale horizontally, only Elasticsearch is covered by this evaluation point.

6.3.1 Disk usage

The first migration performed was that of the previously mentioned CIMS instance with 480 million rows in the job event table. Disk usage is presented in figure 6.1. There is a noticeable difference in disk usage by the different archive prototypes and the live database. Comparing to InnoDB which is the MySQL storage engine in use by CIMS, the two prototypes using Elasticsearch and TokuDB both provides better utilization of disk space.

In order to perform larger migrations, dummy job event data was generated up to a total of 3.3 billion records and inserted into the archive databases (see figure 6.2).

The dummy data was similar in both structure and record size compared to the real data. The reason for not going beyond 3.3 billion was simply that the testing environment had 2TB of local storage. However, with current growth trends, 3.3 billion records would still represent several years' worth of historical data.

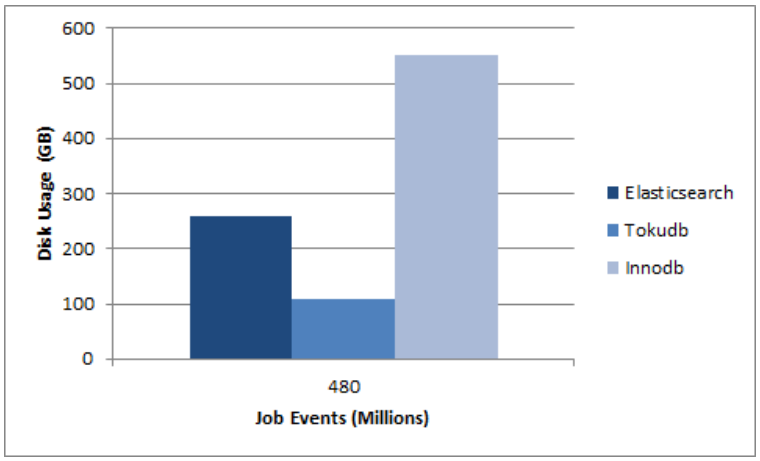

Figure 6.1: Disk usage after migrating data from CIMS instance with real data.

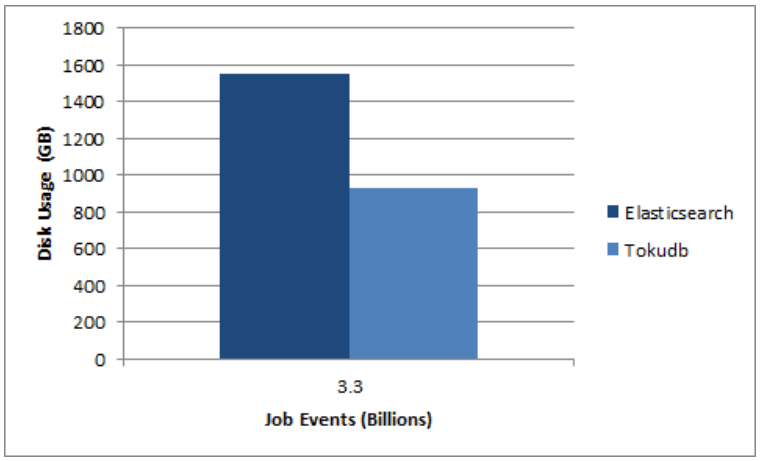

Figure 6.2: Disk usage after having a combination of real data and generated data.

6.3.2 Memory utilization

6.3.2.1 Elasticsearch

In Elasticsearch some configuration related to memory usage is needed to achieve stable performance. It is recommended to set the JVM heap size that Elasticsearch uses to no more than half of total RAM available but never more than 32GB [25, 26]. Elasticsearch will typically use more memory than just the heap size, but this usage will be delegated to the operating system. Tests were done with 8GB, 16GB and 32GB heap size for both querying and insertion. Insertion using an 8GB heap would make Elasticsearch crash since garbage collection could not keep up. But

16 and 32GB gave stable insertion performance. However, there was no observable difference in insertion rates between 16 and 32GB heap sizes. Notably, when doing bulk insertion Elasticsearch would quickly reserve half of available system memory but no more (according to the default configuration). So insertion rates seem to scale with available RAM.

On the other hand querying was different. Performing many subsequent queries would not pool RAM like insertion did. Also no positive effect on response time could be observed by increasing the heap size. Elasticsearch was stable with 8, 16 and 32 heap sizes and response times were the same.

6.3.2.2 TokuDB

The default configuration for TokuDB regarding memory utilization is the same as Elasticsearch, to use no more than half of total RAM available. During bulk insertion, with up to 3.3 billion rows in the database, TokuDB quickly reserved half of available memory. No tweaking of the configuration was needed to get a stable system when doing bulk insertions. As such, the result is largely the same as for Elasticsearch; insertion rate again seem to scale with available RAM.

6.3.3 Insertion time

6.3.3.1 Elasticsearch

Elasticsearch was configured to use a single node setup. The CIMS instance with 480 million job events was then migrated. With this configuration the process took 21 hours. This figure is not very reliable since it is also dependent on the database server of the CIMS instance. To get more dependable results we measured insertion rates for dummy data up to a total of 3.3 billion documents. The generation of data was performed locally. During the insertion of dummy data we could achieve a mean insertion rate of about 33 million documents per hour. This number, compared to the data generation rate in CIMS described in section 3.4 would mean that one month of data generated by a CIMS instance could be archived in less than a day. This is a positive result since it means that the archive system would spend little time doing insertions and most of its time responding the queries.

What is also notable is that even using a single node setup we observed little to no performance degradation for insert rates when the total data set grew. When generating documents from 0 to 3.3 billion the insert rate was as far as we could observe constant. In figure 6.3, we present a graph of insertion rates in relation to data set size.

When switching to a cluster with 4 nodes (4 Elasticsearch instances running on the same machine), much greater insert rates could be achieved and we observed an

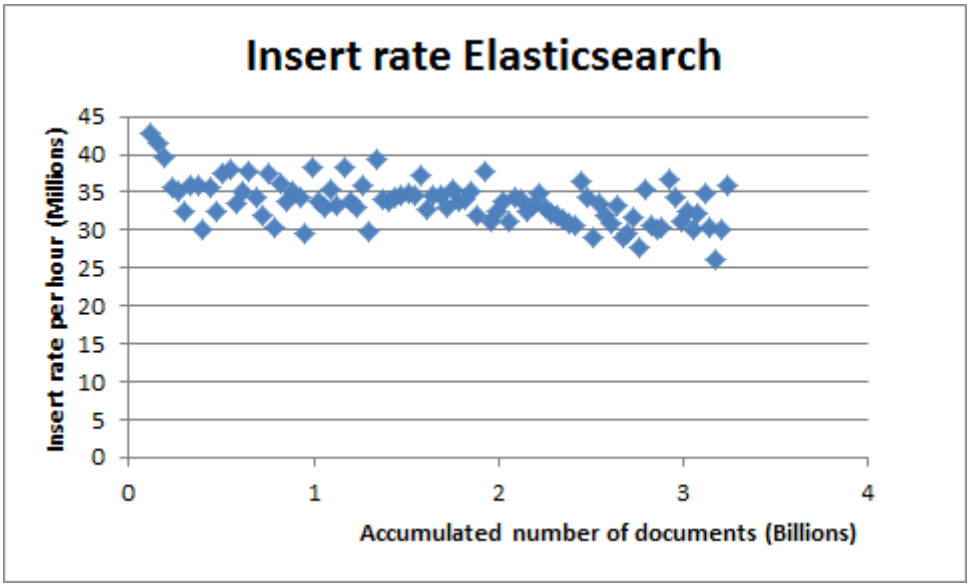

Figure 6.3: Insertion rate of dummy job event data into Elasticsearch.

initial insert rate of 60 million documents per hour in this case.

These reported insertion rates were achieved with little configuration of Elasticsearch. The only important configuration variable to set is the JVM heap size, which as previously mentioned was set to a minimum of 16GB to get stable performance.

The insertion rate measurements show that Elasticsearch is more than capable of handling the current growth trends.

6.3.3.2 TokuDB

A MySQL instance using TokuDB was setup using mostly standard default options. Insertion of dummy data was then performed in the same way as with Elasticsearch. In figure 6.4 a graph with insertion rates is shown. In this case some degradation seems to exist over time. There is also a noticeable dip in the insertion rate at about 2.4 billion rows but the rate quickly recovers. We have no clear explanation for this and can only speculate as to why it happened. It could for example be due to disk allocation by the database or operating system.

Considering the current growth trends, these insertion rates are still much higher than the amount of data that gets generated by a CIMS instance. Therefore the insertion rates should still be considered acceptable.

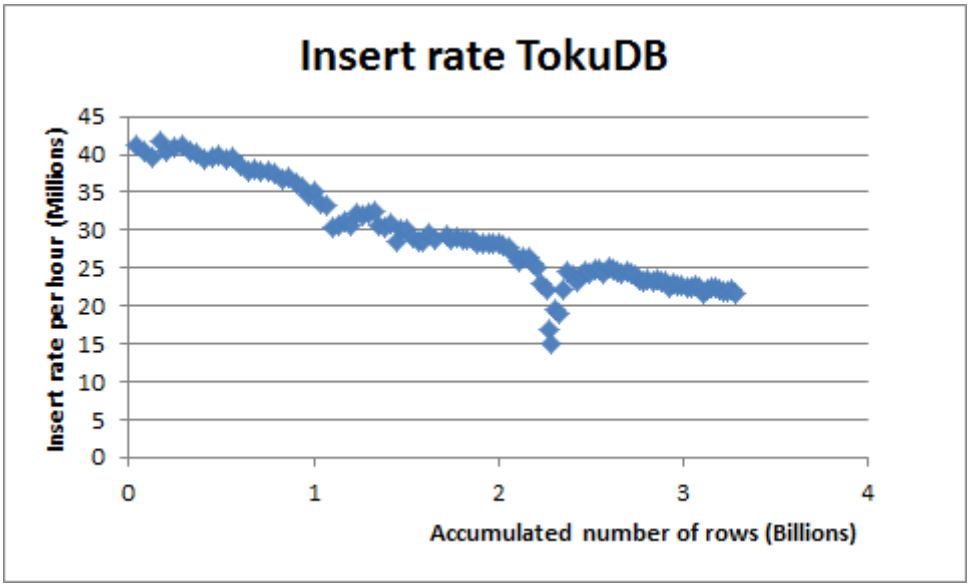

Figure 6.4: Insertion rate of dummy job event data into MySQL/TokuDB.

6.3.4 Query response time

Measurements were made in regards to query response time with the large data set (3.3 billion documents) containing dummy job events. We looked specifically at two queries in this case, namely to look up a job event by its name (used for creating test case histories) and to look up all children for a job event (used to create a test tree). For the former query, the result set will grow as the overall data set grows, since more and more test cases with a specific name is inserted. Another interesting thing to note about this query is that data retrieval is random across the data set since test cases with the same name exist in many builds spanning years' of time.

For the latter, the result set size will likely grow much more slowly. It is not dependent on the overall data set size but instead depends on if the size of test suites grow. This is in the hands of the developers who construct the test suites. In comparison to looking up a test case by name, looking up test suite children is not random across the entire data set. This because test cases from the same build and test suite are inserted together. As such, querying for test suite children should be faster than querying by test case name. But the response time is also highly dependent on the size of the result set.

As stated previously, the purpose of these tests is not to prove or make claims about which solution is better but rather to assess if they perform adequately. It is likely that there is room for improvement in our configuration and code that would give even better results.

6.3.4.1 Elasticsearch

In figure 6.5 the response times when querying by name is shown. The figure has 400 data points in total, where for each point a name is selected randomly from a list of all names of test cases from one CIMS instance. There are about 16000 unique names in total. The outliers in the figure are due to selecting a name more than once and results where cached for the second time query was executed with the same name. Notably, the response time is long for this query because of the large result sets. If the whole result set is not needed but only the first few hundred hits, a more realistic scenario in every day usage, then the response time would reduce to a few seconds.

The second chosen query, to look up all the children of a test suite was tested on the large data set and provided promising results. In the large data set, the dummy data that was generated have test suites all containing 1000 children test cases. In the test that was performed, the average response time when querying for all children of a test case was 0.134 seconds.

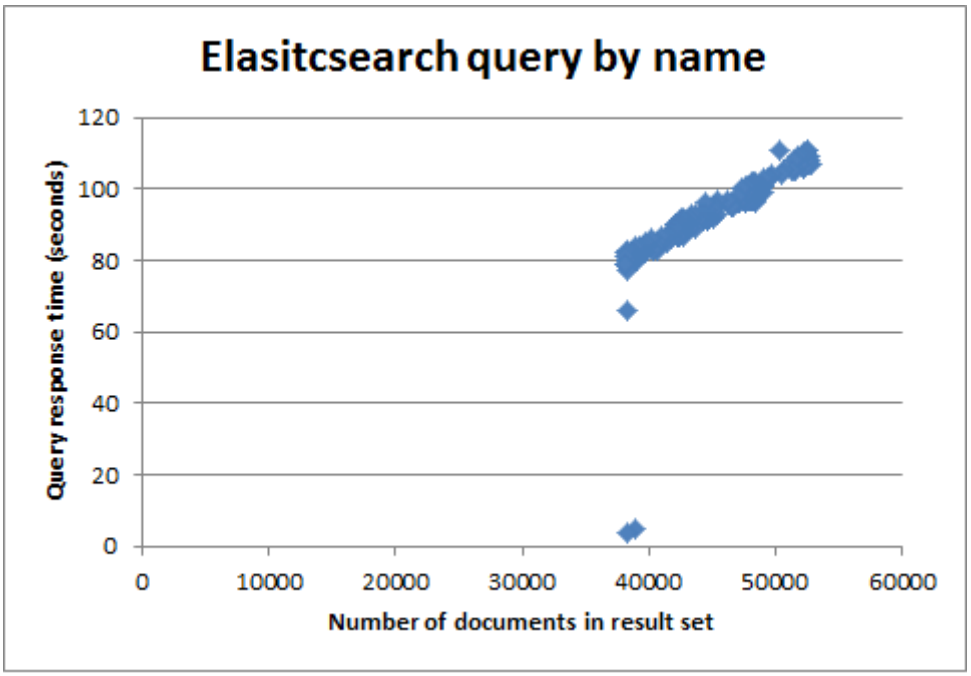

Figure 6.5: Query response time in Elasticsearch when looking up job events by name.

6.3.4.2 TokuDB

Figure 6.6 shows response time for TokuDB when querying by name. 400 test case names were selected randomly and subsequently queried on. We note that, when querying by name, TokuDB is slower overall than Elasticsearch but that the response times are still very reasonable, given the size of the result sets. As expected,

subsequent queries on the same name means results are cached and thus finish in just a few seconds. Querying for all children of a test suite showed good results for TokuDB where the average response time was 0.102 seconds for test suites with 1000 children test cases.

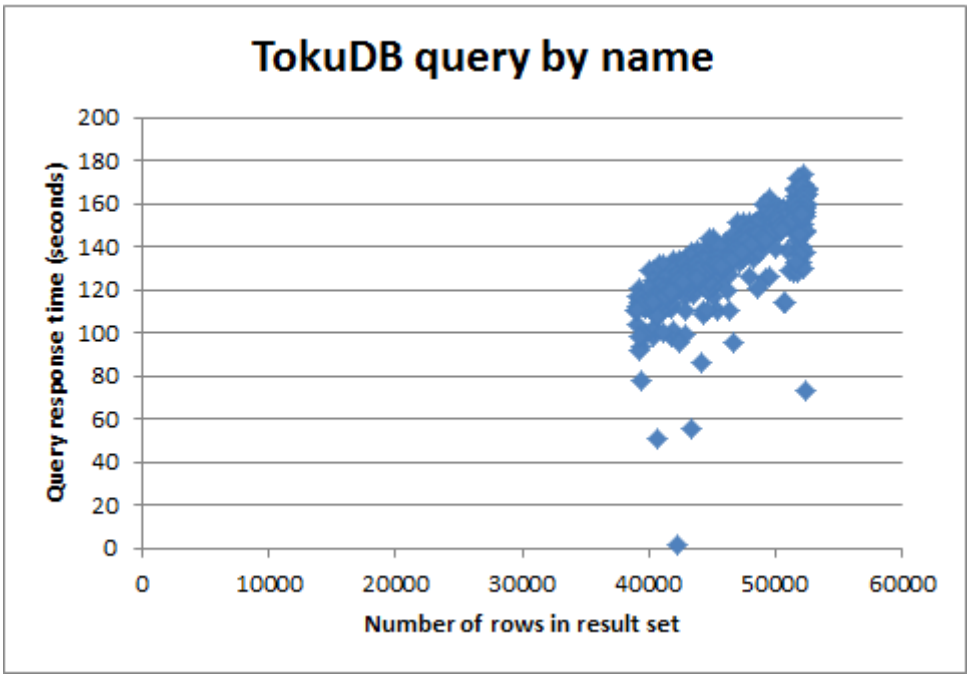

Figure 6.6: Query response time in MySQL/TokuDB when looking up job events by name.

6.3.5 Schema changes

Given that one of the main problems with the current implementation of CIMS is long running schema migrations in MySQL using InnoDB, an archive prototype must effectively address this issue and minimize or remove the need to perform these schema migrations.

6.3.5.1 Elasticsearch

Elasticsearch has the data model of a document-store in that it stores data in the JSON format and does not enforce a predefined schema on inserted documents. This means that the need to perform schema migrations is basically removed and that data with varying structure can be inserted. As already mentioned, this was one of the main reasons for choosing a document-store as a candidate for the archive database.

6.3.5.2 TokuDB

When using a relational database as an archive database, the schema between the live database and the archive needs to be kept in sync. TokuDB has basically solved the issue of long running schema migrations by introducing concepts like hot schema changes and lazy migration. When performing an alter table operation on a table, the change is not immediately propagated to disc for all rows, instead it is done the next time the row is retrieved by a query like a select statement. As such, typical alter tables operations using TokuDB reduces to a few seconds, even on huge tables.

Performing schema changes was tested on the large data set with 3.3 billion rows with good results. For example, adding a new integer column with a default value to the job event table took just a few seconds.

6.3.6 Horizontal Scalability

While Elasticsearch performed well in tests using a single node setup with large data sets, having the possibility to go smoothly from single node to cluster is important to consider. Before adding more nodes to an Elasticsearch cluster, the amount of shards per index must be defined as it sets the upper bound for the total amount possible nodes in the cluster.

In Elasticsearch, all data in an index is split up into shards. These shards can themselves not be split and as such, one shard is always stored in its entirety on a single node. However, nodes can store multiple shards from the same index, as is the case in the single node setup that was tested on. In this setup, four total shards were predefined per index. In production, the total number of shards is typically set to a much higher amount.

Given the setup that was predefined, the maximum number of deployable nodes was four. The single node setup was then converted to a four node cluster where all nodes resided on the testing machine. Doing the conversion was as trivial as starting new Elasticsearch processes configured with the same cluster name an already running Elasticsearch process. The cluster then automatically handles shard allocation to available nodes.

6.4 Integration with CIMS

Proof of concept functionality was requested by stakeholders at Ericsson to demonstrate the possibility of integrating the archive with CIMS and the suitability of polyglot persistence. As mentioned previously, the viewing of builds was selected as a suitable piece of functionality to evolve to use the archive.

To demonstrate the new archive build viewer we used the following somewhat contrived use case as a basis:

- Initially the archive is empty.
- User views list of all builds.
- User identifies build that has a reference build.
- User deletes the reference build.
- System moves build to archive before it is deleted.
- User requests to view reference build.
- System presents data about reference build using the archive API.

This use case is mostly used for demonstration purposes and we believe a more realistic way of moving data from the live database to the archive would be to use scheduled jobs that do this on for example a weekly basis.

It was possible with reasonable effort to implement the new build viewer in CIMS and use the archive API to present build data. However, the view was somewhat simplified given time constraints and because of the way CIMS is designed, where it is not trivial to change the underlying data source for web page components. Overall, through this implementation it is shown that to be possible without too much effort to integrate the archive with CIMS.

6.5 Organizational suitability

This chapter ends with discussion about what was found out about the organizational suitability of the two constructed prototypes.

6.5.1 Elasticsearch

There are a few reasons as to why we believe using Elasticsearch would work well for the organization. First of all, CIMS already employs a number of different[2] data stores other than MySQL, so in some sense it is already a polyglot persistence solution. Therefore, development of CIMS requires familiarity with multiple databases and their respective ways of representing data. Going from this situation to also using Elasticsearch should not be considered too problematic. Through discussion with the CIMS development team this opinion is somewhat cemented as they are

very open to discussing and trying out new technologies and have been doing so before this thesis was conducted.

Furthermore, in an article by Stonebraker and Cattell [16], several rules are given to aid the selection of what they classify as simple operation data stores, namely databases that scale horizontally and avoid complex operations like cross node joins. One notable rule is to consider out-of-the-box behavior of a candidate database. The better this behavior is the less of a pain it will be to initially configure the candidate database. Our experience of Elasticsearch and its out-of-the-box behavior is certainly positive in many respects. The amount of configuration to get stable performance was minimal. As mentioned earlier, the only variable we had to consider in order to get stable performance was the JVM heap size. On top of this, it was easy to go from a single node setup to a cluster.

However, while the initial configuration was pain free, there are many things to consider when it comes to schema and query design in Elasticsearch. For example, it is important to know upfront what kinds of queries that the field of a document or the document as a whole should support. Notably, if a field requires to be queried on by an exact value comparison, this must be configured upfront (typically by declaring the field as 'not analyzed'). As such, most of the time spent on Elasticsearch was not done configuring the database but rather optimizing schemas and exploring the rich query possibilities.

Overall, we think Elasticsearch is an interesting candidate to consider moving forward and see no real problems as to why it would not be possible to use, taking the organizational context into consideration.

6.5.2 TokuDB

Since TokuDB is a storage engine for MySQL, for this solution the live database and the archive would be identical for everything but the storage engine. As such, this solution has very little friction to use as everyone working with CIMS is already very knowledgeable about relational databases and MySQL in particular. The solution is simple and would solve many of identified problems related to monotonic data growth. Through the other points of evaluation for TokuDB we have shown that by using TokuDB as a storage engine, it is very possible to scale to billions of rows on a single machine when it comes to both insertion rate and query performance.

[2]CIMS uses Redis [27] and Sphinx [28] for caching.

7

Conclusion

This project was conducted with two overall goals; first, to design and develop solutions for management of historical data in a continuous integration system at Ericsson. Second, to investigate what value the proper management of historical data can bring to the organization. Using a design research approach, two prototypes were designed and evaluated against a set of predefined solution objectives.

7.1 Contributions

Many companies are adopting continuous integration and could face problems similar to the one at Ericsson. For a practitioner the process in which we designed our solution could be used to identify some steps to take toward a solution in the practitioners own context. For example, to create a high level schema that can be used as a basis for transforming existing data models, or to identify what queries need to be supported for an archive and how well these translate over to the data model of an archive database.

As for academic contributions, this thesis adds to knowledge about software engineering and continuous integration in particular. This was achieved by taking a design research approach to address practical issues of CI in the context of data management. Adopting CI naturally means generating large amounts of data about the development process and as such, one can ask what value can be extracted from this data. In this thesis we have identified some ways this data can be utilized.

7.2 Revisiting research questions

In this section we relate findings from the results of the design research process and the review of related work to the context at Ericsson and use this knowledge to respond to the research questions.

7.2.1 RQ1 - How can historical data be managed?

At the end of the design research process in this thesis, two different archive prototypes had been implemented, one using a relational database (MySQL/TokuDB) and one using a NoSQL database categorized as a search engine (Elasticsearch). Through the evaluation of these prototypes, we show that they effectively address the solution objectives that were defined in adherence with the design research process. The two solutions are different to each other in many respects as they provide different potential in benefits and drawbacks, which we elaborate further below.

Using a relational database as an archive is the simplest solution and would solve many of the data growth problems that motivated the work conducted in this thesis. The main benefits are that developers at Ericsson are already experienced with the technology and should have few problems implementing a solution. In support of this, findings from our study indicate that existing relational technology can be augmented to scale to well enough for the data growth problem to be solved for a long time. However, some functionality is harder to implement, such as a real time free text search and large scale analytics.

A search engine solution is also a viable option. Technology like Elasticsearch is quite mature and naturally solves scalability since the technology was designed with this in mind. In addition, the possibility for full text search and analytics are greater using a search engine as opposed to a relational database. One important factor is also the ability of horizontal scaling, where nodes can run on commodity hardware compared to a relational database that has much higher hardware demands [29]. This gives Ericsson the ability of saving the infrequently accessed data on cheaper disks. There are drawbacks, however. Developers are likely not as experienced with the technology and people who wish to use the search engine must learn new ways to query for data. Introducing a non-relational archive also brings a new data model into the code base. This means that developers need to manage two different data models for the same data. In any case, we believe this is the future in many ways. We also believe organizations should learn to use technologies other than relational databases when the problem and use case calls for them, thereby moving towards polyglot persistence solutions.

7.2.2 RQ2 - Value of a solution

Our opinion is certainly that it is valuable to have an archiving solution for historical data in a continuous integration system. Given the maturity of big data technologies, the possibility to store and analyze large data sets is becoming less and less expensive. In the context of software development, this lets the company (Ericsson) learn more about and quantify the efficiency of its development process. Managers are also given the ability to take data-driven decisions with years of data as a basis. As stated by McAfee and Brynjolfsson [3], these advantages are of paramount importance in order to be more productive and profitable than competitors.

In addition, having a continuous integration and deployment process in place leads to a higher frequency of builds and releases. However, it does not necessarily follow that the upgrade cycle of customers match with the release cycle of the software development process. As such, different customers potentially use a variety of old and new releases. If problems arise with an older release, it is very important that all data related to this release is stored and can be accessed quickly.

Furthermore, at companies such as Ericsson, the continuous integration tools in use are often business critical entities. Providing high up time and reliability for these entities should be considered of high importance. By physically separating live and stale data, the live data set which is accessed much more frequently is kept at a constant and manageable size which should greatly increase the performance of applications that access the live data. From a development perspective, keeping the live data set small is beneficial since database schema changes will take much less time compared to if all data is stored in the live database. This enables instances of CIMS to keep up with the evolution rate of the system. Finally, when live and stale data is separated, the stale data which is accessed much less frequently and by fewer people, can be stored on cheaper machines, thereby reducing overall cost.

7.3 Future work

We believe that we have identified some interesting points for future work when it comes to analyzing the continuous integration process using accumulated data. Since adoption of CI at Ericsson, more builds are registered every month and at the same time, builds contain more and more test cases. It would be interesting to see what effects this has on the developed software. Does quality increase? Are more defects detected?

Finally, this study addressed a problem in an industrial context, at one company. It would certainly be interesting to investigate what the situation is at other companies regarding data generated through the continuous integration process. Are the uses cases we have identified relevant at other companies as well? As far as we know, little research has been done on the value of historical data in continuous integration systems.

Bibliography

[1] M. Fowler, M. Foemmel. Continuous Integration (2006, May 01). [Online] Available: http://www.martinfowler.com/articles/continuousIntegration.html.

[2] P. Duvall, S. Matyas and A. Glover, Continuous integration. Upper Saddle River, NJ: Addison-Wesley, 2007.

[3] McAfee, A., and Brynjolfsson, E. 'Big Data: The Management Revolution.' Harvard Business Review 90:60–68, 2012.

[4] A. Schram and K. Anderson, 'MySQL to NoSQL', Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity - SPLASH '12, 2012.

[5] A. R. Hevner, S. T. March and J. Park, "Design Science in Information Systems Research," MIS Quarterly, vol. 28, no. 1, pp. 75-105, 2004.

[6] K. Peffers, T. Tuunanen, M. Rothenberger and S. Chatterjee, 'A Design Science Research Methodology for Information Systems Research', Journal of Management Information Systems, vol. 24, no. 3, pp. 45-77, 2007.

[7] A. Cleven, P. Gubler and K. Hüner, 'Design alternatives for the evaluation of design science research artifacts', Proceedings of the 4th International Conference on Design Science Research in Information Systems and Technology - DESRIST '09, pp. 1-8, 2009.

[8] K. Krishnan, Data warehousing in the age of big data. Morgan Kaufmann, 2013.

[9] M. Minelli, M. Chambers and A. Dhiraj, Big data, big analytics. Wiley, 2013.

[10] R. Cattell, 'Scalable SQL and NoSQL data stores', ACM SIGMOD Record, vol. 39, no. 4, pp. 12-27, 2011.

[11] C. Mullins, Database administration. Upper Saddle River, NJ: Addison-Wesley, 2013.

[12] D. Abadi, 'Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story', Computer, vol. 45, no. 2, pp. 37-42, 2012.

[13] S. Gilbert and N. Lynch, 'Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services', SIGACT News, vol. 33, no. 2, pp. 51-59, 2002.

[14] M. Fowler and P. Sadalage, NoSQL distilled. Boston, Mass.: Addison-Wesley, 2012.

[15] Jing Han, Haihong E, Guan Le and Jian Du, 'Survey on NoSQL database', 2011 6th International Conference on Pervasive Computing and Applications, pp. 363-366 2011.

[16] M. Stonebraker and R. Cattell, '10 rules for scalable performance in 'simple operation' datastores', Commun. ACM, vol. 54, no. 6, p. 72-80, 2011.

[17] C. Ireland, D. Bowers, M. Newton and K. Waugh, 'A Classification of Object-Relational Impedance Mismatch', 2009 First International Conference on Advances in Databases, Knowledge, and Data Applications, pp. 36-43, 2009.

[18] J. Sharp, D. McMurtry, A. Oakley, M. Subramanian and H. Zhang, Data Access for Highly-Scalable Solutions: Using SQL, NoSQL, and Polyglot Persistence. Microsoft patterns & practices, 2013.

[19] Elastic.co, 'Elastic · Revealing Insights from Data (Formerly Elasticsearch)', 2015. [Online]. Available: https://www.elastic.co/. [Accessed: 24- Apr- 2015].

[20] Tokutek.com, 'TokuDB® for MySQL | MariaDB vs MySQL | MySQL Performance', 2015. [Online]. Available: http://www.tokutek.com/tokudb-for-mysql/. [Accessed: 24- Apr- 2015].

[21] Dev.mysql.com, 'MySQL :: MySQL 5.1 Reference Manual :: 14 Storage Engines', 2015. [Online]. Available: https://dev.mysql.com/doc/refman/5.1/en/storage-engines.html. [Accessed: 24- May- 2015].

[22] Tokutek.com, 'Learn About TokuMX™ and TokuDB® Distributions', 2015. [Online]. Available: http://www.tokutek.com/resources/technology/. [Accessed: 22- May- 2015].

[23] International Organization for Standardization, Software engineering - Product quality - Part 3: Internal metrics, Japan, 2002

[24] International Organization for Standardization, Software engineering - Product quality - Part 2: External metrics, Japan, 2002

[25] Docs.oracle.com, 'Java Virtual Machine Technology', 2015. [Online]. Available: http://docs.oracle.com/javase/8/docs/technotes/guides/vm/. [Accessed: 22- May- 2015].

[26] Elastic.co, 'Limiting Memory Usage', 2015. [Online]. Available: http://www.elastic.co/guide/en/elasticsearch/guide/master/_limiting_memory_usage.html. [Accessed: 07- May- 2015].

[27] Redis.io, 'Redis', 2015. [Online]. Available: http://redis.io. [Accessed: 22- May- 2015].

[28] Sphinxsearch.com, 'Sphinx | Open Source Search Engine', 2015. [Online]. Available: http://sphinxsearch.com/. [Accessed: 22- May- 2015].

[29] A. Moniruzzaman and S. Akhter Hossain, 'NoSQL Database: New Era of Databases for Big data Analytics - Classification, Characteristics and Comparison', International Journal of Database Theory and Application, vol. 6, no. 4, pp. 1-14, 2013.

Appendix A

A.1 Relational queries

Get build

```
SELECT * FROM cims_build WHERE id = 'some_id'
```

Get build information

```
SELECT product_name, revision_name, verdict, start, stop
FROM cims_build WHERE id = 'some_id'
```

Get build for root test suite

```
SELECT cims_build.* FROM cims_build
INNER JOIN cims_build_job_mapping
ON cims_build_job_mapping.build_id = cims_build.id
WHERE cims_build_job_mapping.job_id = 'some_simple_id'
```

Get root test suite for test case/test suite

```
SELECT cims_job_event.* FROM cims_job
INNER JOIN cims_job_event
ON cims_job_event.id = cims_job.root_jobevent_id
WHERE cims_job.job_name =
(SELECT job_name FROM cims_job_event WHERE id = 'some_id')
```

Get test case by name

```
SELECT * FROM cims_job_event WHERE name = 'some_name'
```

A.2 Elasticsearch queries

Get build

```
POST http://localhost:18843/build/_search
{
    "query" : {
        "ids" : {
             "values" : ["CXS101289_10_LLVCORA"]
        }
    }
}
```

Get build information

```
POST http://localhost:18843/build/_search
{
    "query" : {
        "ids" : {
            "values" : ["CXS101289_10_LLVCORA"]
        }
    },
    "fields" :  ["product_name","product_revision",
                    "verdict","start","end"]
}
```

Get build for root test suite

```
POST http://localhost:18843/job/_search
{
    "query" : {
        "ids" : {
            "values" : ["1234"]
        }
    },
    "fields" :  ["build_job_mapping_build_id"]
}
```

Get root test suite for test case/test suite

```
POST http://localhost:18843/job_event/_search
{
    "query" : {
        "ids" : {
            "values" : ["fff70f3a-c69e-44c1-b9f5-7e5ae5a59610"]
        }
    },
    "fields" :  ["job_name"]
}

POST http://localhost:18843/job/_search
{
    "query" : {
        "ids" : {
            "values" : ["result from previous query"]
        }
    }
}
```

Get test case by name

```
POST http://localhost:18843/job_event/_search
{
    "query" : {
        "filtered" : {
            "filter" : {
                "term" : {
                            "name" : "some_name"
                }
            }
        }
    }
}
```