AN EFFICIENT NoSQL DATABASE ADAPTER FOR A RICH INTERNET APPLICATION FRAMEWORK

Johan Selänniemi

Master of Science Thesis
Supervisor: Ivan Porres
Software Engineering Laboratory
Department of Information Technologies
Faculty of Technology
Åbo Akademi University
ABSTRACT

This thesis presents the design and implementation of an adapter between the Google App Engine Datastore and the Vaadin Rich Internet Application framework. The adapter layer is based on the Vaadin data-model interfaces and enables applications to be easily run with different data backends. The challenges to running Vaadin Rich Internet Applications with a non-relational database such as GAE Datastore are centered around data abstraction and performance. The adapter manages data abstraction and uses a proxy to improve performance and reduce operation costs. Experimental results show that the use of the adapter does not hinder the ability of the Google App Engine platform to scale web applications on-demand to high loads.
ACKNOWLEDGEMENTS

I owe my gratitude to the Software Engineering laboratory at Åbo Akademi and Professor Ivan Porres for his great ideas and feedback and for giving me the opportunity to do research in such an interesting topic. I am grateful for the collaboration with Vaadin and Arthur Signell.

Most of all, thanks to my friends and colleagues, especially Niclas Snellman and Jens Smeds for many constructive discussions and encouraging remarks.

Johan Selänniemi
Turku, Februari 2011
# CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>i</td>
</tr>
<tr>
<td>Acknowledgements</td>
<td>ii</td>
</tr>
<tr>
<td>Contents</td>
<td>iii</td>
</tr>
<tr>
<td>List of Figures</td>
<td>iv</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2 Background</td>
<td>3</td>
</tr>
<tr>
<td>2.1 The Cloud Software Program</td>
<td>3</td>
</tr>
<tr>
<td>2.2 NoSQL databases</td>
<td>4</td>
</tr>
<tr>
<td>2.2.1 Google BigTable</td>
<td>5</td>
</tr>
<tr>
<td>2.3 Google App Engine</td>
<td>6</td>
</tr>
<tr>
<td>2.3.1 Global memory</td>
<td>8</td>
</tr>
<tr>
<td>2.3.2 Local memory</td>
<td>9</td>
</tr>
<tr>
<td>2.3.3 Datastore</td>
<td>10</td>
</tr>
<tr>
<td>2.4 Rich Internet Applications</td>
<td>11</td>
</tr>
<tr>
<td>2.4.1 Vaadin</td>
<td>12</td>
</tr>
<tr>
<td>2.4.2 Vaadins data model and interface</td>
<td>13</td>
</tr>
<tr>
<td>2.5 Vaadin and the Datastore</td>
<td>15</td>
</tr>
<tr>
<td>2.5.1 Database abstraction</td>
<td>15</td>
</tr>
<tr>
<td>2.5.2 Performance and cost</td>
<td>16</td>
</tr>
<tr>
<td>3 Design</td>
<td>18</td>
</tr>
<tr>
<td>3.1 Adapter design</td>
<td>18</td>
</tr>
<tr>
<td>3.1.1 Enforcing a schema in GAE Datastore</td>
<td>19</td>
</tr>
<tr>
<td>3.1.2 Sorting and filtering</td>
<td>19</td>
</tr>
<tr>
<td>3.1.3 Positional indexes</td>
<td>21</td>
</tr>
<tr>
<td>3.1.4 Counting</td>
<td>22</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

2.1 Google App Engine platform overview. ........................................ 7
2.2 Example of a RIA: GrooveShark. ............................................. 12
2.3 Vaadin data model. ............................................................... 13
2.4 Comparison of important methods in the component interfaces of Vaadin, ZK and Echo3. ......................................................... 16

4.1 GAECContainer UML diagram. .................................................. 27
4.2 Address book application. ......................................................... 40

5.1 Read latencies for fetching a key and subsequently the item using the key; (Left) Latency when data is fetched from Memcache. (Right) Latency when data is read in Datastore after a miss in Memcache ......................................................... 46
5.2 Comparison over cost of different level of the cache. ................. 47
CHAPTER
ONE

INTRODUCTION

The use of cloud computing platforms to deploy web applications and web services has seen a significant increase over the past few years. This has lead to the rise of a sizable assortment of cloud providers. Deploying web applications to a cloud may bring important benefits such as lower operation costs and increased scalability. Cloud providers offer a wide range of services with varying features which can make it hard for the developers to evaluate the platforms [2]. Significant changes to an existing application may also be required since different cloud providers offer different Application Programming Interfaces (API).

An increasingly popular way of offering cloud services is Platform-as-a-Service (PaaS). With PaaS the provider takes care of the underlying hardware and virtualization mechanisms so that the programmer does not need to know the underlying details of the system but rather works with the system through APIs. The PaaS usually supplies the programmer with development tools for debugging, deploying and managing the application [17].

A popular Platform-as-a-Service is Google App Engine (GAE), Google’s platform for deploying web applications to the cloud. Currently, App Engine requires that applications are written in the Java or Python programming
languages. For persistent storage GAE uses the Datastore, a non-relational database, which is built on top of BigTable [4]. Access to the Datastore is provided through a low-level API [21].

GAE is free to try out and therefore has a low threshold for initial use. Still, porting or developing an application for a specific cloud platform such as GAE may require a considerable effort. As a consequence, it can be difficult to switch provider or even to delay the choice of provider to the later stages of development.

The solution proposed in this thesis is to use a framework for web applications that has support for different cloud providers. If applications are built using the abstractions and features of the framework then we can deploy applications in all the platforms supported by the framework with little or no changes.

Concretely, in this thesis we present a database adapter for the Vaadin [24] web application framework. Vaadin is a java-based server-side Rich Internet Application (RIA) framework that provides many advanced graphical user interface (GUI) components as well as support for different data backend solutions. Each data backend is supported as a different implementation of the same interface: the data container.

The thesis begins with presenting the background by discussing NoSQL databases, Google App Engine, the Datastore, Rich Internet Applications, Vaadin and the challenges to using the Vaadin data-model with Google’s datastore. In Chapter 3 we describe the challenges in more depth and state our design decisions behind the solutions. In Chapter 4 we present the implementation of the abstraction layer and its most important classes and interfaces. We also provide an example of a practical application modified to operate with our component in Google App Engine. We round off by presenting a performance evaluation of the abstraction layer in Chapter 5 and conclusions and further work in Chapter 6.
In this chapter we present the background needed to understand the concepts presented in this thesis. We start off by presenting the program in which our work was performed, the Cloud Software Program [20]. The chapter continues with the possibilities and challenges with NoSQL databases and Google BigTable, the underlying technology for Google’s datastore.

We describe the Google App Engine cloud for web applications and its components with focus on memory and the Datastore. The chapter continues with Rich Internet Applications with emphasis on the Vaadin framework and the Vaadin data-model. We end the chapter by introducing the challenges to running Vaadin with the Google datastore.

2.1 The Cloud Software Program

The cloud software program is part of the Finnish Strategic Centres for Science, Technology and Innovation (Tivit). The goal of the program is to improve the position of the finnish software industry in global markets.
The main goals of the program are new cloud business models, lean software enterprise model and open cloud software infrastructure [20]. The program consists of partners from the industry and the academic world.

The Software Engineering laboratory at Åbo Akademi participates in the program with research on testing of Rich Internet Applications and abstraction of non relational databases.

2.2 NoSQL databases

NoSQL, or non-relational, does as the name suggests refer to a database that in some ways operate differently than a relational one [6]. Relational databases have been the industry standard for storing persistent data, with big vendors such as IBM, Oracle and Microsoft dominating the market for decades [23]. However as cloud computing grows more and more popular, the problems of SQL databases as a one-type-fits-all solution become more apparent. Relational databases work well with structured data, such as sales figures, but have problems with unstructured data, for instance images and documents, which are popular in web applications. There are databases that utilize the NoSQL approach such as Apache CouchDB, Google BigTable, Cassandra and Amazon SimpleDB which are already in use by big companies.

As the non-relational view on data differs from the relational one there are several important differences. Data in NoSQL databases is not stored in tables with a static structure but rather as unstructured data by which items of the same type may have different properties. The NoSQL databases does not generally support the concept of primary and foreign keys. Storing data in a distributed way makes joining tables as in SQL-join operations highly inefficient and are usually not supported. Non relational databases does not natively support the traditional concepts of atomicity, consistency, isolation and durability (ACID) for enforcing constraints on data within the database [1].

NoSQL databases offer several advantages over relational ones. First off
utilizing a non-relational view on data means that the database is better equipped to respond to the scalability needs of cloud computing applications. Trying to squeeze data that is not relational into a table structure will result in a complex data model. Partly by providing a less complex set of operations and reducing the precision of transactions, NoSQL databases offer higher performance [18]. The distributed take on storing data enables better data availability and fail safety.

However, NoSQL also introduces limitations that are yet to overcome. As NoSQL databases does not commonly offer a sophisticated querying language such as SQL, manual query programming is necessary, which can be complex and time consuming. NoSQL databases does not offer substantial measures to enforce data-model consistency within the database. The programmer must enforce the data-model on the application level, which is error prone. Maintaining data consistency can be a challenge since NoSQL databases offers less support for transactions. Next we will have a look into Google’s NoSQL implementation called BigTable.

2.2.1 Google BigTable

In 2003 Google created a distributed storage system called BigTable. The storage system was designed to accommodate the needs for a set of Google’s own cloud services. BigTable was constructed with wide applicability, scalability, high performance and high availability in mind. BigTable’s data model differs significantly from that of a traditional relational database. Contrary to a full relational model and advanced SQL-constructs, BigTable offers a simple data model that can be dynamically controlled.

In BigTable terms a table is a ”sparse, distributed, persistent, multidimensional sorted map” [4]. The table is made up out of three dimensions: rows, columns and time. Both row and column keys consists of strings and data is sorted lexicographically according to the row key. Column keys are grouped into column families which may contain an arbitrary amount of columns of different datatypes. Thus, consecutive rows in BigTable do not necessarily have the same amount of columns or the same data types. The time di-
mension consists of timestamps for the data which is used for maintaining distinct versions on a per field level. The user can specify the amount and age of the versions so that old data can be handled by BigTable’s garbage collector.

BigTable supports transactions for performing atomic operations on a row-level basis but has no transaction support across rows [4]. The storage system runs on Google's infrastructure, sharing its resources with other applications for storing data GFS, Google's distributed filesystem, is used. For various synchronization tasks BigTable uses Chubby, a distributed lock service for course-grained control [3].

Implementation wise there are three major components: a master server, a client library and several tablet servers. The master handles load balancing, data assignment, tablet locations and garbage collection. The tablet servers serve read and write requests and split tablets to fit with the GFS file system. Due to extensive caching of tablet locations on the client side the master server will not sustain an acute load. Until recently BigTable was only used internally but in 2008 Google opened access to the proprietary database through App Engine and its Datastore API.

### 2.3 Google App Engine

Google App Engine (GAE) is a service provided by Google for hosting web applications. App Engine is designed to scale to large amounts of concurrent users while the details of the scaling are not visible to the programmer. App Engine provides support for both Python and Java. The Java Virtual Machine in GAE provides much of the same functionality as that of a traditional one with some exceptions to thread handling. Apart from the JVM the run-time environment also contains libraries and the application code.

Figure 2.1 shows an overview of the GAE Platform. Incoming requests arrive at a load balancer that distributes the request to an appropriate server depending on the load. Serving of static files are performed by sep-
Figure 2.1: Google App Engine platform overview.
arate servers, and are routed accordingly by the load balancer. Dynamic requests are routed to a server which starts an instance to serve the request. Several instances serving different applications may be running on one server simultaneously which create security concerns. To address this, each instance runs in an isolated sandbox. As starting new instances is quite costly, servers try to keep instances in memory between requests for better performance. Section 2.3.2 discusses the effects and possible applications of this.

Since the request-response model is stateless there is no guaranteed way of saving data on servers instances between requests. App Engine provides persistent data storage through the Datastore. The Datastore is discussed in detail in section 2.3.3. To speed up the access of data, GAE provides users with an implementation of Memcache with a global view of data between servers. Memcache is further discussed in section 2.3.1. Additionally, GAE provides a range of APIs for various tasks such as working with images and fetching URLs.

In Google App Engine, users are only billed for the resources their applications actually use, for example CPU time and stored data. CPU time can be further divided into time spent in calls to the, APIs e.g. to the Datastore, and time spent in the runtime environment. Google allows users to use a fixed amount of resources for free everyday, which makes the platform an attractive alternative to new users.

### 2.3.1 Global memory

For the purposes of persisting data, web applications use databases with an underlying non-volatile storage device such as a hard-drive. GAE applications use the Datastore for persistent data. Accesses to hard-drives are comparatively slow to that of ram-memory. Most web applications use memory to cache data for faster read access. In a distributed environment the memory used for the cache is located on multiple servers. A distributed memory cache provides a global view of cached data on all server instances. Since data is stored in memory there is no guarantee of the durability of
data. Thus, all data should also be stored in the non-volatile storage.

GAE provides a distributed cache service called Memcache [5]. The GAE Memcache is based on the Memcached system [8]. The Memcache service is a key-value store for arbitrary data types. Keys are limited to 250 bytes and values to one megabyte. The GAE Memcache has no support for transactions but set-operations are atomic, meaning that either the value gets completely set or the old value is retained. Different strategies for setting data exists, for example add-only-if-not-present or add-always.

The most obvious way to use memcache is to cache key-entity pairs and check the cache before the Datastore. On updates Memcache must be updated to reflect the data in the datastore. The datastore and Memcache cannot be transactionally updated which could leave incorrect data in Memcache. To minimize the effects of erroneous data, the life time of cached data can be specified. By using the memcache service the amount of queries to the datastore can be reduced, thus, lowering costs.

### 2.3.2 Local memory

Applications run on GAE in an isolated sandbox environment. Once an application is loaded into the sandbox its application code and resource files are stored in memory to speed up subsequent requests [21]. The local memory is shared between all application instances in the same sandbox. As usual with global variables and threads, enforcing thread safety is necessary.

Arbitrary data can be stored in the local memory by declaring it with the Java static attribute. The static data has the same life time as the application instance. The data is lost once the instance is destroyed. We have used the possibility of storing static data between requests to build a cache level above Memcache. There is no guarantee on the life time of an application instance nor to which instance a request will get routed. The fact that data could be present but is not guaranteed does not matter in terms of a cache as it would just mean a cache miss in the case of a new instance.
Since the static variables are stored in local memory on each server instance, there is no global consistency of data between servers. The application instance has a limited amount of memory to its disposal so limiting the amount of data stored is essential.

2.3.3 Datastore

For persistent storage GAE uses the Datastore, a non relational object database built on BigTable. The Datastore does not support all of the constructs found in SQL-databases, such as joining tables. This has a considerable effect on how data is modeled. The data model builds on the terms of entities, properties and kinds. Data is stored as entities which can have several properties with names and values with one of the supported data types. Each entity is of a certain kind. The kind, is among other things, used when querying for data.

At first glance the Datastore might seem like an SQL database, entities corresponding to rows, properties to columns and kinds to table names. There are however several important differences. An entity need not have the same amount of properties as another entity of the same kind. An entity may even declare a property with the same name as another entity, but with a different data type. Properties in the datastore can store multiple values, referred to as multi-valued properties. An entity has an unique key that can be used for fetching or querying the entity but is not part of the entity itself as opposed to SQL keys. Due to the loose model of the Datastore much of the data-model must be maintained in the application.

Querying is supported by a set of, compared to SQL, primitive filters such as equal or greater than. Querying supports ordering of properties which affects the arrangement of the returned entities. The Datastore maintains an index for each defined query to increase the access speed. A big difference to that of querying in SQL databases is that in the Datastore all types of queries must be specified in advance to run-time and replicated through the infrastructure. Since the values in the indexes must be maintained on run-time, writes to the Datastore are several magnitudes slower than
reads. Having close to linear querying speed by sacrificing write speed is acceptable in most web applications.

In the Datastore all updates on single entities are atomic which means the entity is either updated completely or not at all. Transactions across multiple entities are also supported allowing entities to be updated on the same server. Transactions across multiple entities are however only possible if the entities belong to the same entity group. The entity group must be selected when creating the entity and cannot be changed after that. This due to the fact that entity groups help the Datastore to store entities that are subject to transactions together.

The Datastore can be accessed either through middleware frameworks such as JPA or JDO or through a low level API. The API provides abstractions for all of the common operations on entities, properties, and and tasks such as querying and fetching by keys.

### 2.4 Rich Internet Applications

This section introduces Rich Internet Applications from a general point of view and more closely, our subject of study, the Vaadin RIA framework. Traditionally, web pages have had a document-like way of presenting data with modest possibilities for user interaction. The upside of this simple architecture is universality. As our society is becoming increasingly media centric, the capabilities of traditional web applications are coming up short. The current hot topic is Rich Internet Applications which allows for more powerful presentation and interaction capabilities on top of the existing HTTP protocol.

The term RIA is quite broad but a RIA tries to extend the original architecture of the web with the benefits of desktop applications by adding technologies such as AJAX \[9\] to the HTTP protocol. For example traditional web applications consists of multiple pages which are reloaded according to the request-response model of HTTP. On the contrary Rich Internet Applications run in solely one page and allows page elements to be loaded
individually. Figure 2.2 provides an example of a Rich Internet Application called Grooveshark. Grooveshark is a sophisticated music player that runs solely in the browser.

![Figure 2.2: Example of a RIA: GrooveShark.](image)

The popularity of Rich Internet Applications has resulted in the development of a substantial amount of frameworks aiming to ease and speed up the development process. A relevant part of this effort is the effective modeling of data.

### 2.4.1 Vaadin

Vaadin is a server-side framework for developing Rich Internet Applications in Java. The open source framework is developed by its community and a finnish company called Vaadin Ltd. The development of the framework started already in 2000 [13]. The Vaadin framework offers a programming model that allows users to write their RIAs solely in Java. The framework consists of a server-side part and a client side part. The client side part runs in the browser and uses GWT [12]. The actual control and business logic is located on the server.
2.4.2 Vaadin data model and interface

This section describes the interface of the Vaadin data-model which provide the foundation of the abstraction used in our abstraction layer. The data-model is defined by by a set of Container interfaces. The Containers provides a flexible way of managing items with common properties and can be connected to the UI-components in the framework. The structure of the Vaadin data model shown in figure 2.3 bears a close resemblance to a table in a relational database, items corresponding to rows and properties to cells. In GAE terms, a Vaadin item is analogue to an entity and a Vaadin property to a GAE property. Each instance of a Container is bound to a certain type of item, which could in SQL terms be thought of as the table name.

We have focused on five major interfaces in the Vaadin Container API: the basic Container interface, Container.Ordered, Container.Indexed, Container.Sortable and Container.Filterable.

![Figure 2.3: Vaadin data model.](image)

The Container interface adds basic methods for manipulating data such as adding and removing items, getting the number of items in the container and methods for defining the data model. Each item has a unique id which can be used to retrieve the item. Counting the amount of entities of a certain kind in the datastore is possible but has linear complexity.
Methods such `removeAllItems` were neglected since they were deemed not to be scalable. Users define the data model by repeatedly calling the `addContainerProperty` to add properties. The method takes the following parameters: `propertyId`, `type` and `defaultValue`. A newly created item has the specified default values for all properties. Additionally, the interface imposes requirements on the structure of the data in the container:

- All Items in the Container must include the same number of Properties.
- All Items in the Container must include the same Property ID sets.
- All Properties in the Items corresponding to the same Property ID must have the same data type.
- All Items within a container are uniquely identified by their non-null IDs.

As these constraints cannot be modeled in the Datastore, they must be maintained within the Container implementation.

The `Container.Ordered` interface adds ordering to the Container. The concept of ordering falls naturally with the Datastore, since entities are in fact ordered by their keys in the order they were added. However adding entities in other places than at the end is not possible.

The `Container.Indexed` extends ordering with positional indexes. Each item in the container has a positional index starting from zero. The `getIdByIndex` method takes a positional index as input and returns an item ID which can be used to retrieve the actual item. The inverse, i.e. determining the index given a item ID, is problematic as it would require traversing the Datastore until the item ID is found.

The `Container.Sortable` and `Container.Filterable` interfaces enables users to customize the order and visibility of items in the container by adding sort orders and filters to properties. Depending on active sort orders and filters, the items will have different indices. Noteworthy is that
the methods in the previously mentioned interfaces are also dependent on the filters and sort orders.

**Vaadin in Google App Engine**

The GAEApplicationServlet [26] servlet container developed by Vaadin allows Vaadin applications not using the Datastore to be deployed in Google App Engine. Since the GAE execution environment is distributed, the servlet serializes the application at the end of each request and deserializes it in the beginning of the next one. The serialized application is stored in Memcache and the datastore. Consequently, all classes that are not declared as static or transient must implement the Java serializable interface.

**2.5 Vaadin and the Datastore**

In this section we summarize two of the most important challenges in developing Rich Internet Applications with frameworks such as Vaadin with GAE Datastore; database abstraction, performance and costs issues.

**2.5.1 Database abstraction**

The main concepts of the Container interface such as adding, removing, ordering, filtering and counting, are supported by the Datastore low level API. However, there are incompatibilities between the Vaadin Container interface and the Datastore.

The data structure imposed by the Container interface, e.g. a table model, cannot be enforced directly within the Datastore. The Vaadin interface enables filtering on partial strings, comparable to the SQL LIKE command, whilst the Datastore does not. The Vaadin interface introduces the notion of Java-like positional indices for items, while the Datastore operates on a key-value basis.
Although the work presented here is specific to the Vaadin framework, we believe that a similar approach can also be applied to other RIA frameworks. These frameworks often have well-defined interfaces to the underlying data sources. As an example, Figure 2.4 compares the main methods in the Vaadin Container interface with the corresponding interfaces of the ZK [27] and the Echo [7] frameworks.

![Figure 2.4: Comparison of important methods in the component interfaces of Vaadin, ZK and Echo3.](image)

The Vaadin framework has a data model with a container interface that already has implementations for other back-ends such as in-memory, Lucence, JDBC and JPA. These different implementations indicates widespread support for the container interface within the Vaadin framework. Our addition of the Datastore to the already implemented technologies demonstrates the transparency of switching between a non relational and relational database.

### 2.5.2 Performance and cost

Users can take advantage of Googles own high performance infrastructure by running their applications on App Engine and its Datastore. The Datastore utilizes the same technology as Googles in-house applications, for example GFS and BigTable [4]. Some examples of advantages include automatic fault tolerance, data replication and scaling [21, 11]. On top of this, users only have to pay for the data they actually store and query.
Despite all these advantages, the Datastore introduces some challenges to RIAs. A RIA tries to simulate a desktop like experience by extensive use of UI components. These may produce more database requests than traditional web applications. A study has shown that the Datastore has a considerable variance in read performance [15]. As the UI components are highly dependent on low and uniform latency to deliver a good user experience, the direct use of the Datastore could lead to a slow or even unusable application.

Although offering a global view of data, Memcache offers latencies lower and with less variance than the Datastore [15]. Additionally, data can be stored in local memory on server instances between requests. Combining the local memory and Memcache to a multi-level cache could give significant performance benefits.

The Vaadin Container interface splits fetching of keys by positional indices and items by keys in separate methods. This is, apart from data abstraction, an issue for performance since it would require two round-trips to the Datastore. Additionally, counting the amount of entities in the Container is time consuming.

Moreover, due to the billing model of GAE, an application is billed on a per query basis. This raises questions about cost efficiency and optimization to larger extent than that of traditional setups where the client already owns or rents full installations of hardware. As accessing data from Memcache is cheaper than from the Datastore and accesses to local memory being virtually free, another factor, cost has to be considered in addition to performance.
CHAPTER THREE

DESIGN

In this section we further explain the challenges of running Vaadin applications with the GAE Datastore. To each challenge we present our solution and design decisions. To respond to the challenge of the data abstraction we have designed an adapter between the Vaadin Container interface and the GAE Datastore low level API. To improve the performance and cost benefits the adapter also works as a proxy with a multi-level cache.

3.1 Adapter design

To a large extent, the challenges of designing the adapter derive from incompatibilities between the Datastore’s non-relational model and Vaadin’s data model and interface. The main challenges are to enforce the data schema, efficiently support sorting, filtering and positional indexes. In the following section we explain our approach to solving these problems.
3.1.1 Enforcing a schema in GAE Datastore

The Vaadin data model assumes that each item in a container has a fixed number of properties and each property has a fixed type. On the contrary, Google datastore does not impose such restrictions.

As a consequence our adapter maintains the data schema and implements the necessary checks to ensure that data, which is written and read from the database, conforms to the schema. In order to define the data model, the programmer calls the method `addContainerProperty`, specifying the type and default value of each property.

The adapter allows any serializable class to be used as a property type. The Java Date and String are stored as such in the Datastore while the Long, Short, Integer, Float, Double types are stored as Long. This enables us to sort and filter using these properties. All other types are serialized and stored as blobs. Blobs or binary large object are arbitrary collections of data. When reading an entity, the adapter automatically converts the data to the right type using the information about the property type in the data model.

When adding a new entity, our adapter does not explicitly store properties containing a default value. Instead a null value is used. This may reduce the amount of data stored per entity, while still allowing us to filter and sort by these properties.

When an entity is read, the null values are replaced by the default property values described in the data model. Since we do not allow null values for properties, it is not possible to mistake a null value for a default value.

3.1.2 Sorting and filtering

Users can customize the adapter by adding sort orders and filters to properties. Both the Vaadin interface and the Datastore API enables sorting in both ascending and descending order for individual properties.
Sorting is done in the Container interface by calling the `sort` method with an arbitrary amount of property names and desired sort orders as arrays. Once sorted, all methods that rely on the order of the items are affected. The Datastore only supports sorting on the property types listed in Section 3.1.1. Thus on calling the `sort` method, the Adapter validates that each supplied property can be sorted.

The Vaadin Container interface supports filtering on substrings with the options of partial matching and ignoring case. The Datastore only supports matching on exact values, hence, supporting this functionality is not possible.

Therefore we created a customized interface for filtering as shown in Listing 3.1. Filters are added to properties using the `addFilter` method. The Adapter supports the following filters: `less than`, `less than or equal to`, `equal to`, `greater than`, `greater than or equal to` and `not equal`. The Datastore has constraints on possible filter combinations. Inequality filters for example, are only allowed on one property. The Adapter validates each new filter to the existing ones, throwing an exception if the filter combination is not possible. Additionally the Adapter checks if the property is supported for filtering and if the supplied value is of a type corresponding to that property in the data model.

Listing 3.1: Adapter filtering interface

```java
void addFilter(Object propertyId, FilterOperator filter, Object value)
  throws IncompatibleFilterException;

void removeFilters();

void removeFilters(Object propertyId);
```

The methods in the Adapter are affected by the applied sort orders and filters, e.g. the `size` method should return different results depending on which filters are applied. Hence, the adapter stores the sort orders and filters, and applies them to every relevant query to the Datastore.
3.1.3 Positional indexes

Given a set of sort orders and filters, each matched item in the container will have its own positional index. The `getIdByIndex` method returns the ID of an item, given a positional index. The ID can be used by the `getItemById` method to fetch the actual item.

As an example, imagine that an entry of type person has a property first name with the value Aaron and there are no other persons with the first name starting with "A". If the `firstname` property is sorted in an ascending order, the item with Aaron would be first in the Adapter. Thus calling `getIdByIndex` with "0" would return the ID for that item. Sorting the same property in a descending order would place the item last in the Adapter. Furthermore, adding filters to properties reduces the visibility of items. For example adding an equal filter with the value "Aaron" to the `firstname` property would mean that only items having the first name "Aaron" are visible to the `getIdByIndex` method.

To support positional indices, the Adapter must take each filter and sort order into consideration. The Datastore supports offsets on queries with a maximum size of 1000. The offset indicates how many entities the Datastore will skip in the matched query prior to returning results.

The adapter uses offsets to enable positional indices. To find the ID corresponding to a positional index, the adapter performs a query with all filters, sort orders and an offset equal to that of the positional index. Due to limitations on the maximum size of the offset, the Adapter cannot determine the key corresponding to a positional index in one query for indexes larger than 1000. In these cases, the Adapter uses cursors to step through each chiliad until a cursor is obtained for the chiliad in which the index resides. For example the positional index 1200 would require the Adapter to first perform a query to find the cursor for the 1000th entity, and subsequently perform a query with an offset of 200 using that cursor.
3.1.4 Counting

Vaadin Containers can be queried for the number of contained items. The \texttt{size} method returns the number of items. If no filters are applied, the \texttt{size} method returns the full amount. If filters are applied, the amount of items matched by the combination of filters is returned.

Counting the amount of entities matched by a query is supported by the Datastore, but grows linearly in time and is therefore not scalable to large quantities.

To avoid counting items using queries to the Datastore when no filters are applied, we store the amount of items as metadata in the Datastore. This enables us to determine the size merely by fetching the entity containing the metadata. The metadata is transactionally updated for each add and remove of an item. This introduces additional overhead to adds and removes but was deemed justifiable extending on GAE’s methodology of prioritizing fast reads over writes.

As each combination of filters has its own amount of items, we do not store the size as metadata when filters are applied.

3.2 Proxy design

Our attempted solution to improving latency is a proxy with a cache consisting of two levels: local memory and Memcache. Our design is modular allowing for more levels to be added. The biggest challenges in designing the cache are caching positional indices efficiently, while at the same time ensuring cache consistency.

3.2.1 Memcache and local memory

The GAE platform provides an implementation of Memcache for caching data. The implementation uses the standard Memcache API and stored
data is global to the entire application. We use Memcache as the middle level in our cache hierarchy.

Static variables are stored in memory on server instances between sessions in GAE. The top level uses this to cache data with a very fast access speed. The stored data is local to each server. There is no guarantee to which server instance a particular request will go [21], however in a cache application context, a request arriving at a new instance can simply be thought of as a cache miss.

The adapter fetches data in chunks from the Datastore, enabling preloading. There are performance benefits from preloading data to UI components, such as a table, since data is accessed spatially. The chunk size and life time of data can be specified for each of the cache levels.

The slower but global Memcache can be configured to bring in larger chunks of data with longer life time while the local memory level works on smaller pieces that are refreshed at a higher pace. Since there could be an arbitrary amount of combinations of applied filters, we allow the users to decide if positional indices should be cached when filters are applied.

The implementation for the local memory level essentially consists of customized linked hash maps. The size method is extensively used by Vaadin components such as Table. To further reduce the latency of this method, we allow sizes to be cached at the local memory level. As the conceptual size varies depending on which filters are applied, the user can choose whether to cache sizes when filters are applied. If specified, one size will be cached for each filter combination. To ensure thread safety, each operation that changes data in the hash maps uses a Reentrant lock for read and write operations.

To customize the memory usage of the local memory level, maximum capacity for positional indices, items and sizes can be specified separately. An update strategy can be chosen for discarding data once the maximum capacity is reached. We support Least Recently Used and First In First Out.
3.2.2 Caching indexes

One of the biggest challenges for designing the cache is that the Vaadin interface does not simply work on a key-value basis but relies on positional indices to retrieve the keys. Only caching keys-items would give small performance benefits as querying the Datastore would still be necessary to determine the positional indices.

In our implementation we cache both key-item pairs and positional index-key pairs. Each set of positional indexes is uniquely identified by its sort orders and filters. Since the positional indexes vary depending on sort orders and filters, there exists many sets of positional indexes-keys per set of keys-items. Hence, these are stored separately. This allows the user to set different life time for positional indexes and key-item data.

In order to obtain the item that corresponds to a given positional index, the Vaadin container interface requires us to invoke two different methods: `getIdByIndex` and `getItemById`. This may require two different queries to the Datastore API, consuming more time and money. Since we have observed that these two methods often are called in sequence, we instead perform a single query in the method `getIdByIndex` that returns all the necessary information and store it in the cache. When the method `getItemById` is invoked, the necessary information is often already in the cache so we save one call to the Datastore.

3.2.3 Consistency

Our take on data consistency is that key-item pairs are updated and discarded on updates and removes, while positional indices are not updated in the cache. Ultimately this means that positional indices temporarily can point to wrong items. To optimize the effects of invalid positional indices, users can adjust life time and caching options of positional indices with filters applied.

Due to the high demands on availability, a common approach to data con-
sistency in web applications is optimistic locking [14]. We enable optimistic locking on a per item basis. To allow this functionality, each entity in the Datastore is provided with a version number. When retrieving an item, the Adapter equips the item with the version number. A comparison of the item’s version and the entity version is performed in a transaction on writebacks. If the versions correspond, the item is updated, if not, an exception is thrown. The exception allows the user to take appropriate means.

3.2.4 Buffering

The Vaadin interface enables the possibility of bringing out individual properties from the container. Whole entities are the smallest units that can be fetched from the Datastore. This means that it will be as time consuming to fetch a property as a whole entity.

It is likely that two properties that belong to the same item will be accessed together. By caching the item to which the property belongs, performance benefits can be gained from subsequent requests for other properties from the same item.

Likewise, the Vaadin Container interface expects that properties are updated each time their values are changed, which would mean a write operation to the Datastore each time a property is changed. Although we support write-trough, the user can toggle the functionality and buffer property changes and then commit them in groups per item.
CHAPTER
FOUR

IMPLEMENTATION

This chapter describes the implementation of the adapter and proxy. Figure 4.1 shows an overview of the class structure of the implementation, the GAEContainer. The GAEContainer class provides the methods available to the application programmer. The task of the provider is to provide the GAEContainer class with data. The provider uses the Datastore wrapper to communicate with the Datastore and the Cache implementations in the cache hierarchy. We end the chapter with an example of moving a practical Vaadin web application to the GAE cloud.

The Vaadin GAE servlet serializes the objects when storing them in the Datastore between request. Thus, all classes that are to be serialized must implement the serializable interface. All configuration regarding the caches and container e.g. data-model are serialized with the application context so that the next request will have the same settings.
Figure 4.1: GAEContainer UML diagram.
4.1 GAEContainer

The GAEContainer class provides the methods for interacting with the abstraction layer implementing the Container interfaces. Since the Container.Filterable interface was not feasible, the custom QueryableContainer interface described in section 3.1.2 is used instead. The GAEContainer class is responsible for maintaining the data-model i.e. mappings for default property values, property types and which properties are available for sorting. Input values are validated so that the other classes in the system can assume that they are valid. The class creates Vaadin properties and items from GAE properties and items. Creation of properties entails type conversion for natively supported values and serialization of general objects as discussed in section 3.1.1.

Listing 4.1 shows the constructor that is provided to the programmer for creating the container. Each instance of a GAEContainer is tied to a specific kind of entity in the Datastore e.g. person or address. If property write trough is set to false the GAEContainer class will buffer changes to properties until a manual commit is performed. The versioned flag specifies if writes to the Datastore should be done with optimistic locking. To enable manual commits and optimistic locking we created an extention to the Vaadin Item interface; the VersionedGAEItem interface. The constructor accepts an arbitrary amount of cache levels, i.e. classes that implement the Cache interface. The cache levels are passed to the CachingProvider.

Listing 4.1: GAEContainer constructor.

```java
public GAEContainer(String kind, boolean propertyWriteTrough,
                     boolean versioned, Cache ... Cache)
```

The class allows other classes to register themselves as listeners. The listeners are notified of changes made to properties and items that were brought out from the container since sort orders and filters that are applied to properties should be taken into consideration for queries to the datastore. The system stores all applied sorts and filters in a QueryRepresentation object.
4.1.1 QueryRepresentation

The QueryRepresentation class is used to internally represent a given combination of sort orders and filters. The class contains methods for creating Datastore queries from the internal representation and methods for generating unique identifiers. The identifiers are for example used when saving data in the cache. The QueryRepresentation also ensures that filter combinations are possible when added. Invalid filter combinations generate a IncompatibleFilterException.

4.1.2 VersionedGAEItem

VersionedGAEItem is an extension of the Vaadin item interface. The new interface adds the capability of manual commits and optimistic locking. When the GAEItem is created in the GAEContainer class, it is supplied with a version number. The version number is, like other properties, taken from the Datastore entity. The core optimistic locking functionality is implemented in the Datastore wrapper. Listing 4.2 shows the method for performing manual commits. If optimistic locking is enabled the method may throw one of the exceptions. If write-through was set to true the commit method will have no effects since the changes were instantly committed when the property was changed.

Listing 4.2: VersionedGAEItem commit method

```java
public void commit() throws ConcurrentModificationException, NoSuchElementException
```

4.2 Cache

The Cache interface is implemented by all cache implementations. The interface defines methods for putting entities, getting entities, removing entities and checking the line size of the cache. The most important methods of the interface are listed in listing 4.3. Entities can be added or updated
with the `put` method that accepts a map of keys/entities. Either a single entity or a collection of entities can be queried from the cache using the `get` method. The `getLineSize` method returns the size of one line in the cache, i.e. how many entities the cache should be updated with at a time.

Listing 4.3: Basic Cache methods.

```java
/**
 * Adds a batch of entities to the cache.
 *
 * @param entities map of key and entity pairs to be added
 */
void put(Map<Object, Object> entities);

/**
 * Gets the entities corresponding to the given keys.
 * Might return a partial map in case not all keys existed.
 *
 * @param keys keys of the entities
 * @return map of key and entity pairs of the keys that existed
 */
Map<Object, Object> get(Collection<Object> keys);

/**
 * Gets an entity.
 *
 * @param key key of the entity
 * @return the entity corresponding to the key or null
 */
Object get(Object key);

/**
 * Line size says how many objects this cache wants to be updated with at a time.
 *
 * @return the line size of this cache
 */
int getLineSize();
```

The `Cache` interface only stores key-item pairs. Cache implementations can extend the type of data they store by implementing the `IndexCache` and `SizeCache` interfaces.
The `IndexCache` interface adds the capability of storing positional indices-keys as described in section 3.1.3. The `putIndexes` adds or updates a set of positional indices-keys starting from the given `index_start`. The positional indices are represented as an array of keys. Each offset in the array represents an index relative to the `index_start`. The `queryIdentifier` is a unique identifier for the query that the positional indices correspond to. Analogously, to fetch an array of positional indexes, the `queryIdentifier`, start index and amount are supplied. For example, to retrieve positional indices 100-200 the `getIndexes` method is given a start index of 100 and amount 100. The 50th element in the returned array then corresponds to the 150th positional index.

Listing 4.4: IndexCache methods

```java
/**
 * Gets a chunk of keys mapped to indexes.
 * @param queryIdentifier a unique identifier for the given kind
 * and sort combination
 * @param index_start the first index of the chunk
 * @param amount amount to fetch
 * @return an array with each element pointing to a key starting
 * from index_start
 */
Object[] getIndexes(String queryIdentifier, int index_start, int amount);

/**
 * Puts a chunk of keys mapped to indexes.
 * @param queryIdentifier a unique identifier for the given kind
 * and sort combination
 * @param index_start the first index of the chunk
 * @param indexes an array with each element pointing to a key
 * starting from index_start
 */
void putIndexes(String queryIdentifier, int index_start, Object[] indexes);
```

The `SizeCache` adds the capability of caching sizes for a given kind and query as described in section 3.1.4. The `size` and `updateSize` methods
takes an identifier which identifies to which query the size belongs.

Listing 4.5: SizeCache methods

```java
/**
 * Gets this caches view of the amount of entities in the datastore.
 * @param identifier unique identifier of kind and applied filters
 * @return amount of entities
 */
int size(String identifier);

/**
 * Update this caches view of amount of stored entities.
 * @param identifier unique identifier of kind and applied filters
 * @param new_size the updated size
 */
void updateSize(String identifier, int new_size);
```

Two cache levels were implemented: MemCache and Local Memory. The MemCache class acts as a wrapper over the GAE MemCache implementation. The Local Memory class works as the top level in the cache and is completely implemented in memory.

### 4.2.1 MemCache

The MemCacheImpl class mediates access to GAE’s native MemCache. The class implements the Cache and IndexCache interfaces. Listing 4.6 shows the constructor of the MemCacheImpl class. The life time of data in MemCache is specified with Expiration objects. The implementation stores positional indexes and entities separately, which makes it possible to have different life times for the two. The cacheFilteredIndexes flag indicates whether the cache should store positional indexes for queries that has filters applied.

Listing 4.6: MemCacheImpl constructor
/**
 * @param lineSize Length of one cache line
 * @param indexLifeTime Life time of indexes
 * @param itemLifeTime Life time of raw data
 * @param cacheFilteredIndexes Should indexes with filters be cached
 */

public MemCacheImpl(int lineSize, int indexLifeTime, int itemLifeTime, boolean cacheFilteredIndexes)

4.2.2 Local memory

The LocalMemoryImpl class uses the local memory of server instance to cache data. The class implements the Cache, IndexCache and SizeCache interfaces. Listing 4.7 shows the constructor of the LocalMemoryImpl class.

Entities, positional indices and sizes are stored separately in hash maps. An extension to the native Java hash map was created in order to enforce a maximum size, and a remove strategy once the map is full. The remove strategy and maximum size of each map is specified through the constructor. The hash maps are stored as static variables which means that data will exist in a server instance until it is destroyed. This means that subsequent requests to the same server instance will benefit from the cache.

For it to be possible to set a life time for the entities stored in the hash maps, a MemoryCacheItem class was created. The MemoryCacheItem objects are used are used internally in the LocalMemoryCache and contain the entity and a time stamp from when it created. The getValue method is used to extract the entity from the MemoryCacheItem. The method compares the time that the object was created against the maximum life time of a cache item. If the time is less than the maximum life time, the method returns the entity, otherwise null is returned and the MemoryCacheItem is removed from the hash map. The life time of data is set through the constructor.

33
To ensure thread safety, the implementation utilizes read-write locks using the Java `ReentrantReadWriteLock` implementation. Similarly, each time a hash map is read it must wait to acquire a read-lock. Each time a hash map is updated a write-lock must be acquired. Many read-locks can be acquired at the same time, meaning that reading can be done in parallel. However, if a write-lock was obtained all methods requiring a read-lock must wait until the write-lock is released. The locks are applied on a per hash map basis for improved performance. For instance, if the hash map containing positional indices is locked for updating, the hash map containing entities can still be read.

Some of the classes such as the locks are not serializable. Also, the actual cache data, i.e. the hash maps should not be serialized. When a new session is started the unserialized object’s hash maps and locks will be null. Hence the local memory cache defines a post-deserialization method that populates the hash maps and locks.

Listing 4.7: LocalMemoryCacheImpl constructor

```java
/**
 * @param lineSize Length of one cache line
 * @param indexLifeTime Life time of indexes
 * @param itemLifeTime Life time of raw data
 * @param cacheFilteredIndexes Should indexes with filters applied be cached
 * @param sizeLifeTime Life time of cached sizes
 * @param indexCapacity Number of lines with indexes to store
 * @param itemCapacity Number of entities to store
 * @param sizeCapacity Number of sizes to store
 * @param cacheFilteredSizes Should sizes with filters applied be cached
 * @param LRU If true use LRU else FIFO
 */
public LocalMemoryCacheImpl(int lineSize, int indexLifeTime, int itemLifeTime, boolean cacheFilteredIndexes, int sizeLifeTime, int indexCapacity, int itemCapacity, int sizeCapacity, boolean cacheFilteredSizes, boolean LRU)
```
4.2.3 Cache creation

The Cache implementations’ constructors accepts various configuration parameters. The CacheFactory class was implemented to simplify the creation of Cache instances to the user. The class has methods for creating instances of both the local memory and the MemCache implementations. Listing 4.8 shows the methods available for cache creation.

Listing 4.8: CacheFactory methods

```java
/**
 * Creates a @link LocalMemoryCacheImpl
 */
public static Cache getCache(LocalMemoryCacheConfig c)

/**
 * Creates a @link MemCacheImpl
 */
public static Cache getCache(MemCacheConfig c)
```

To create a Cache the user must supply the factory with a CacheConfig object that specifies the configuration properties of the cache to be created. The abstract class CacheConfig has two sub classes: LocalMemoryCacheConfig and MemCacheConfig. The config classes have static Builder classes which have methods for setting each property in the config. The methods return the Builder so that method calls can be chained. To allow new users to use caching in their applications without the details of setting the parameters, we created default configs for both implementations. The default configs are created with the factory.

4.3 CachingProvider

The DataProvider acts as a proxy between the GAECContainer class and the Datastore wrapper. The DataProvider interface defines various methods for adding, removing and fetching data. The CachingProvider interface extends the basic interface with caching support by using the Cache imple-
All caches must at least implement the functionality described in the Cache interface. Caches are added with the addCache method.

The CachingProviderImpl class implements the CachingProvider interface. The implementation supports an arbitrary amount of cache levels. When the GAEC container class requests data from the provider, it first checks the caches prior to querying the datastore. For each method that supports caching; the caches are traversed for data starting from the first cache added. The type of data that is cached is: items, positional indexes and sizes. On a cache miss, the next level in the hierarchy is checked and ultimately if no cache contained the data, the data is requested from the datastore. All cache levels that did not contain the data are updated. If no caches were added the provider, no caching is used and the Datastore wrapper is queried directly.

The CachingProviderImpl class supports any of the Cache, SizeCache and IndexCache interfaces. The provider's size method shown in listing 4.10 checks each of the added caches. If the given cache implements the SizeCache interface it is queried for the size otherwise it is skipped. The check if performed as in listing 4.9.

```
Listing 4.9: Code for finding SizeCaches.
for(Cache cache : caches){
    if(cache instanceof SizeCache){
        //query the cache
    }
}
```

```
Listing 4.10: The provider's size method.
/**
 * Gets the amount of entities in the datastore for a given kind and filter order.
 *
 * @param query desired kind and filter combination
 * @return amount of entities
 */
int size(QueryRepresentation query);
```
The procedure for checking the caches for positional indexes when calling the `getKeyByIndexFromStart` works in a similar manner. The following list outlines the main steps taken when the method is called:

- Loop through all caches
- For each cache loop until the index is found or until all caches has been checked.
  - Check that the cache implements the `IndexCache` interface.
  - Determine the amount of indices to be fetched.
  - Fetch the indices from the cache.
  - If the indices did not exist in the cache, add it to the list of caches that should be updated.
- If the indices were found in the cache, update all caches.
- If the indices were not found in the cache, query the datastore using the datastore wrapper.
  - Update the caches.

The amount of indices to be fetched depends on the line size of each cache. If no caches exist in the list of caches to be updated, only one index is fetched since the method should only return one index. Otherwise the amount will be set according to the caches that are going to be updated. Since each cache can have a different line size the amount of indexes fetched is always determined by the cache with the largest line size.

Listing 4.11: Provider method for fetching a key given a positional index.

```java
/**
 * Given a query representation fetch the key of index index starting from beginning.
 *
 * @param query the query representing the current sort order
 * @param index offset from the first entity
 * @return the key corresponding to the index or null if it did not exist
 */
```
public Key getKeyByIndexFromStart(QueryRepresentation query, int index);

We use arrays as the data structure for storing positional indexes where the length of the array is equal to the line size of the given cache. The array contains the keys pointing to the actual items. The array is stored in a cache as an identifier-array pair where the identifier describes the query and relative offset of that chunk of positional indices. When updating a cache with a smaller line size the array is sliced to be of appropriate length.

If a datastore query is needed to find the keys to the positional indexes, we update the items in the caches at the same time for the reasons described in section 3.2.2.

### 4.4 Datastore

The Datastore interface defines a wrapper with methods needed by the DataProvider to interact with the Datastore. The DatastoreImpl class implements the Datastore interface. To interact with the GAE Datastore, the DatastoreImpl class uses the DatstoreService interface supplied by Google. The wrapper class implements the functionality for querying the datastore with the use of positional indexes as described in section 3.1.3.

Entities are updated in the Datastore with the put method shown in listing 4.12. The versioned flag indicates if versioning i.e. optimistic locking should be used. First, a check is performed to determine if the entity’s key is complete. Since an entity will only have a complete key if it was previously stored in the datastore, we can know whether it is a new entity or old entity. If the entity is new, a version field with value 0 is added to the entity prior it being written to the Datastore. By using the key to check if the entity is newly created we save one call to the datastore. If the entity isn’t new i.e. it has a complete key, the corresponding entity is fetched from the datastore. Furthermore, the version fields are compared. If they match, the version number is increased by one and the entity is
updated. Otherwise a ConcurrentModifcationException is thrown. The exception propagates down to the commit call of the VersionedGAEItem which was performed by the user. If the corresponding entity could not be found in the Datastore a NoSuchElementException is thrown instead. All steps are performed within a transaction to ensure consistency.

Listing 4.12: Datastore wrapper interface method for adding and updating entities.

```java
public Entity put(Entity entity, boolean versioned) throws ConcurrentModificationException, NoSuchElementException

public int size(Query query)
```

If optimistic locking was turned on and the user tries to save a item that does not have a version number in the datastore, a new version number is created. This makes it possible to start using optimistic locking for entities that were not created with the GAEContainer.

The size method shown in listing 4.12 is used to count the amount of entities matching a certain query as described in section 3.1.4. We store the amount of entities for each kind of entity as metadata in the datastore. The size metadata is identified by the entity’s kind as the key. Each time an entity is added or removed, the corresponding _size entity is updated. The result that the size method returns is dependent on the applied filters. Since the amount of possible filter combinations is arbitrary, we only store the amount of entities as metadata for queries with no filters applied.

If the query given to the size method has filters or the _size entity corresponding to the query’s kind does not exist, the datastore is queried for the entities and the method returns the size of the list. It should be noted that this approach is slow for a query matching a large amount of entities.
4.5 Example usage

We illustrate the functionality of our solution and the simplicity of moving a basic Vaadin Application to GAE with a Vaadin tutorial application originally designed to use an in-memory container implementation. The Address Book application[25] lets users view addresses in a table. Figure 4.2 shows a live screen shot of the application being run with the GAECContainer in Google App Engine. Readers are encouraged to view the original example before continuing.

![Address Book Application](image)

Figure 4.2: Address book application.

The browsing can be customized by sorting on different fields such as "first name" and further by applying filters to the fields. Entries can be added, removed or changed. Listing 4.13 shows the change in code needed to create and use our container instead of the in-memory container. The container is configured to use no optimistic locking and property write-through, meaning that properties are written to the Datastore when they are updated.
The fields in the address book will therefore be updated instantaneously. Since the local memory cache is passed to the constructor before the memcache configuration, the local memory cache will be higher in the cache hierarchy.

Listing 4.13: Creating the Adapter.
```java
//code for creating the in-memory container
//private IndexedContainer addressBookData = new IndexedContainer()
/*
create the GAEContainer
with the caches, write through and no optimistic locking
*/
private GAEContainer addressBookData = new GAEContainer("People",
true, false, CacheFactory.getCache(localMemoryCacheConfig),
CacheFactory.getCache(memCacheConfig));
```

Listing 4.14 shows the code for creating the cache configurations that are passed to the GAEContainer constructor. Both ways of creating configurations are displayed. The LocalMemoryCacheConfig is created with manual methods by chaining calls on each setter of the builder class. The MemCacheConfig is created with the getDefaultMemCache method which returns a cache with default parameters.

Listing 4.14: Creating the cache configurations.
```java
static final LocalMemoryCacheConfig localMemoryCacheConfig = new
    LocalMemoryCacheConfig.Builder()
    .withCacheFilteredIndexes(true)
    .withCacheFilteredSizes(true)
    .withIndexCapacity(50)
    .withIndexLifeTime(60)
    .withItemCapacity(500)
    .withItemLifeTime(60)
    .withLineSize(20)
    .withRemoveStrategy(RemoveStrategy.LRU)
    .withSizeCapacity(10)
    .withSizeLifeTime(60).Build()

static final MemCacheConfig memCacheConfig = CacheFactory.
    getDefaultMemCacheConfig();
```
Listing 4.15 shows how to define the data model of our address book. All properties are of type string and have an empty string as the default value. We should note that this example only uses the standard Vaadin interface. The functionality of enforcing the data model in the Datastore is implemented in our container.

Listing 4.15: Defining the data model.

```java

for (String p : fields) {
    addressBookData.addContainerProperty(p, String.class, "");
}
```

The native table component used to display the entries can, as with the in-memory container, conveniently interact with the data automatically by defining the container as a data source as shown in Listing 4.16.

Listing 4.16: Binding the data source.

```java
contactList.setContainerDataSource(addressBookData);
```

Listing 4.17 shows a code snippet with filters in the Address Book example. The code allows the end-user to add filters to entries in the address book through a text field. Since the customized interface is used, minor changes to the code were needed. Noteworthy is that since customized filtering is used, the filtering is case-sensitive as opposed to the original example which was case-insensitive.

Listing 4.17: Filter through text field.

```java
private void initFilteringControls() {
    for (final String propertyName : fields) {
        final TextField sf = new TextField();
        sf.addListener(new Property.ValueChangeListener() {
            public void valueChange(ValueChangeEvent event) {

```
addressBookData.removeContainerFilters(pn);

/* old code for adding filter
addressBookData.addContainerFilter(propertyName,
     sf.toString(), true, false);
*/
addressBookData.addFilter(propertyName,
     FilterOperator.EQUAL, sf.toString());

getMainWindow().showNotification(
     "" + addressBookData.size() + " matches
     found");

...
In this chapter we present the performance tests executed on the container. We start of by describing the test suite and continue by describing each test in separate sections. To get an indication of the scalability of the proxy we performed a latency test on the different cache levels. The results from the latency test is presented in section 5.1. To address the question of possible costs benefits that could be gained from the proxy, in addition to performance benefits, we created a model to compare the prices at different hit ratios. The model and the results are presented in section 5.2

To perform the tests in GAE we created a custom servlet container. The servlet creates and instance of the container and invokes the methods specific to the test. By running a light-weight servlet instead of the Vaadin servlet we escape the extra overhead from Vaadin framework tasks, such as serialization, which are not relevant to the performance of the container.

Prior to invoking the methods a timer is started and when the methods are completed, the duration is measured. The parameters and the test duration are then logged using the GAE logging facility. Some of the tests was performed at such high loads that the server logs were congested. Therefore we created a BASH script that, once started, continuously retrieves
the logs server from the server. After the test has completed we use Python scripts to parse the data and calculate the results. Finally the graphs are plotted with Gnuplot [10].

The load was generated with Autobench [19], a tool that uses httperf [16] and adds the capability of coordinating a cluster of computers. Our cluster consisted of eight university computers. Each test has its own Autobench test script containing the test parameters. The test parameters needed by the servlet are sent by using cookies.

5.1 Cache latency comparison

Figure 5.1 shows a graph depicting the 5th percentile, quartiles, and 95th percentile latencies for one of the most common use-cases: fetching a key by positional index and subsequently the corresponding item. The figure in the right side represents the latency if there was a miss in local memory, a miss in Memcache and a read in Datastore. The figure in the left side represents latency if there is a miss in local memory and a hit in Memcache. The time unit is milliseconds.

The tests where performed at a varying load from 10 to 400 requests per second. The load was initially set to 2 requests per second and then increased by steps of 2 requests per second until the desired load was reached. Each step was performed for two minutes to allow GAE’s load balancer to react to the increasing load. Finally once the desired load was reached the test was executed for another 10 minutes in order to get the test data.

No filters or sort orders were applied. The time to retrieve data from the local cache is below 1 ms and therefore not shown.

The tests display approximately the same latencies regardless the amount of requests per second. This indicates that the functionality that was tested scale up until the levels of load used the tests.

No significant reduction in average latency is obtained by using Memcache. This is in partly due to the fact that we need to perform two queries to Mem-
cache to retrieve one item: one query for the positional index and one query for the actual item. The latency from the Datastore increases with higher positional indexes and therefore also the benefits from using Memcache.

We also observed that there is more fluctuation on latency when accessing the Datastore. Therefore we consider the performance of Memcache being more predictable.

![Figure 5.1: Read latencies for fetching a key and subsequently the item using the key; (Left) Latency when data is fetched from Memcache. (Right) Latency when data is read in Datastore after a miss in Memcache](image)

### 5.2 Cache optimization

Figure 5.2 illustrates the cost of fetching 100,000 items, one item per fetch, with the same parameters as in Figure 5.1. The solid line indicates fetching without the adapter, as a reference value using the low level API. The dashed lines show the cost using the container for different hit ratios. In the first scenario a miss is represented by a miss in local memory and a read from Memcache. In the second scenario a miss is represented by a miss in local memory and Memcache followed by a read from Datastore. A hit comes from the local memory cache in both scenarios. The cost parameter was calculated from a mean of 100 values for each of the three
cases. The cost was measured and logged as shown in listing 5.1.

![Graph showing cost comparison](image)

Figure 5.2: Comparison over cost of different level of the cache.

Listing 5.1: Code for measuring cpu and api quota time used.

```java
QuotaService qs = QuotaServiceFactory.getQuotaService();
long start_cpu = qs.getCpuTimeInMegaCycles();
long start_api = qs.getApiTimeInMegaCycles();
//run the test
long cpuSeconds = end-start_cpu;
long apiSeconds = end_api-start_api;
log.info("Total megacycles "+(cpuSeconds+apiSeconds) + ">
API_megacycles "+apiSeconds+ "CPU_megacycles "+cpuSeconds);
```

The values were calculated using the formula

\[
T_c(p_l, p_m) = \frac{p_l T_{hl} + (1 - p_l)(T_{ml} + p_m T_{hm} + (1 - p_m)(T_{mm} + T_d))}{1200}
\]  

(5.1)

\[
cost(p_l, p_m) = T_c(p_l, p_m) * cost_{cpus}
\]  

(5.2)
where $p_l$ is the probability of a hit in the local cache and $p_m$ the probability of a hit in the memcache cache, assuming a miss in the local cache. $T_{hl}$ and $T_{ml}$ are the average CPU times in megacycles for a hit and miss in the local cache. Similarly $T_{hm}$ and $T_{mm}$ are the average API and CPU times in megacycles for a hit and miss in the memcache cache. Finally $T_d$ is the average time for a datastore access. The times in megacycles are divided by 1200 to convert them to CPU seconds and multiplied by the cost of a CPU second to obtain the cost of a read operation.

The graph shows the cost benefits of the different levels of the container. In our adapter, a hit to the local memory level is an order of 200 times cheaper than to a hit to Memcache, which is still 5 times cheaper than a fetch to the Datastore. As the graph indicates and which could be expected, the use of the adapter will be more expensive than the direct use of the low level API for low hit ratios. The explanation being the additional cost of keeping the cache up to date. The actual cost benefit to be gained from real usage depends on the hit ratio of the data in the application.
CONCLUSIONS AND FURTHER WORK

In this thesis we have presented the main design decisions behind the development of a database adapter between the Google Datastore and the Vaadin Rich Internet Application framework and experimental performance and costs results. Our implementation is released as open source and is available for download [22].

The solution presented here follows a well-known approach in software engineering, which is to add a new layer of abstraction over an existing component. A solution of that kind is not successful if the new layer is considered more complex or less efficient than the existing component. We tackle the first challenge by ensuring that the new container exhibits the same interface as the existing Vaadin containers. The second challenge is addressed by employing a two-level cache. We believe that this cache approach may improve the efficiency and reduce the deployment costs compared to direct Datastore access.

The challenge in our research resides in offering a solution that can fully utilize the scalability of a non-relational database, while at the same time delivering the low latency needed by a RIA. The latency requirements can be achieved by introducing a multi-level cache that offers benefits in both
performance and cost. There is a trade-off between read speed, write speed and data consistency. Applications can gain an increase in performance by reducing global data consistency to a per server instance consistency. Also, by momentarily reducing the consistency of positional indexes in the cache, the adapter can better deliver a latency acceptable to that of UI components in a RIA.

Experimental results show that the use of the adapter does not hinder the ability of the Google App Engine platform to scale web applications on-demand for high loads. To fully evaluate the performance of the solution in a production environment more extensive testing should be performed, especially testing with practical RIAs. It would also be interesting to compare the performance of our implementation to that of native middleware in GAE such as JPA or JDO.
BIBLIOGRAPHY


6.1 Introduktion

Användningen av datormoln för att distribuera webbapplikationer och tjänster har ökat signifikant under de senaste åren. Trenden har gett upphov till en stor mängd leverantörer av datormoln. Att använda datormoln för att distribuera webbapplikationer kan ge fördelar såsom lägre driftskostnader och ökad skalbarhet. Leverantörerna av datormolnen erbjuder en stor mängd egenskaper vilket kan göra det svårt för utvecklare att välja bland datormolnen [2]. En existerande applikation kan behöva signifikanta ändringar eftersom olika leverantörer erbjuder olika gränssnitt för tillämpningsprogram.

Ett allt mer populärt sätt att erbjuda tjänster för datormoln är *plattform-som-tjänst*. Denna typ av tjänst innebär att leverantören tar hand om den underliggande hårdvara och mekanismer för virtualisering så att programmeraren inte behöver känna till detaljerna för systemet utan endast gränssnitten. Tjänsten innehåller ofta verktyg för avsökning av
fel i programkoden, uppladdning av applikationen till datormolnet och administration av applikationen.


6.2 Bakgrund


Google App Engine erbjuder en distribuerad cache kallad Memcache [5].


6.3 Design

Som svar på utmaningen med dataabstraktion har vi designat en adapter mellan Vaadins containergränssnitt och Google Datastore. För att förbättra prestandan har vi designat en cache med flera nivåer.
Utmaningarna i att designa en adapter härstammar från inkompatibiliteter mellan Vaadins datamodell och Datastores icke-relationella datamodell. Huvudutmaningarna berör upprätthållning av dataschemat samt effektivt stöd för sortering, filtrering och positionsindex.


6.4 Prestanda

För att evaluera skalbarheten av cachen utfördes test för svarstider för de olika nivåerna i cachen. För att evaluera möjliga kostnadsfördelar av cachen, skapade vi en modell för att jämföra kostnader vid olika proportioner för träffar och missar i cachen.

6.5 Slutsats

I detta slutarbete presenterades de huvudsakliga designbesluten för utvecklingen av en databasadapter mellan Google Datastore och Vaadin, samt experimentella prestandatest och kostnadsfördelar för vår implementering. Implementeringen är utgiven som öppen källkod och finns tillgänglig för nedladdning [22].


Experimentella resultat visar att användningen av adaptorn inte hindrar Google App Engine att hantera webbapplikationer med hög belastning. För att till fullo bedöma prestandan i en produktionsmiljö behövs mer tester, speciellt med praktiska rika internetapplikationer. Det vore också intressant att jämföra prestandan av vår implementation med Googles egna komponenter såsom JPA eller JDO.