AN EFFICIENT GOOGLE DATASTORE ADAPTER FOR RICH INTERNET APPLICATIONS

Johan Selänniemi and Ivan Porres
Department of Information Technologies, Åbo Akademi University, Joukahaisenkatu 3 A, Turku, Finland

Keywords: Platform as a service, Google App Engine, Databases, NoSQL, Rich internet applications.

Abstract: In this article we present the design of a database adapter for the Google Datastore and the Vaadin Rich Internet Application Framework. The adapter allows us to develop Vaadin applications that can use different database systems and can be deployed in a private infrastructure as well as in the Google App Engine platform. The adapter uses a two-level cache schema 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.

1 INTRODUCTION

Cloud computing has emerged as a flexible and scalable approach to deploy internet applications (Dikaiakos et al., 2009). Deploying applications to a cloud may bring important benefits such as overall lower operation costs and increased scalability on-demand. As a consequence, there is a rise in the number of cloud providers offering a wide range of virtualized hardware infrastructure (Infrastructure as a Service) and software platform services (Platform as a Service or PaaS).

Platform as a service providers introduce an attractive solution for the rapid development of web applications since they offer the main middleware and database components necessary to develop scalable applications readily available. A popular PaaS is the Google App Engine (GAE), Google's platform to deploy web applications to the cloud. There exists many other popular offerings such as Microsoft Azure and Heroku.

Usually, a PaaS offering supports only one or just a few selected programming languages and requires the use of a number of platform APIs that are specific for that PaaS offering. Currently, GAE requires that applications are written in the Java or Python programming languages. As persistent storage, GAE uses the Google Datastore, a non-relational database, which is built on top of BigTable (Chang et al., 2008). Access to the Datastore is provided through a low-level API that is specific for the Google platform (Sanderson, 2009). Other platforms require the use other languages and APIs.

From the point of view of the developer, this means that an application needs to use the specific features and APIs of the target deployment platform. As a consequence, porting an existing application to a platform may require a considerable effort. Also, it can be difficult to switch provider or even to delay the choice of provider to the later stages of development. The problem is accentuated if we take into account that the numerous offerings can make it hard for the developers to evaluate and choose the right platform (Bunch et al., 2010).

A solution to this problem follows a well-known pattern in software engineering: to introduce a new layer of abstraction. Instead of developing an application using the features of a specific cloud platform we can use a framework for web applications that has support for different cloud providers. Applications are then built on the abstractions and features used by the framework and they can be deployed in all the platforms supported by the framework with little or no changes. The challenge is to develop such a framework in a way that still provides the expected benefits of a cloud computing platform: on-demand scalability, simplified and flexible deployment and lower operation costs.

Concretely, in this article we present a database abstraction layer for the Vaadin (Vaadin, 2010b) web application framework. Vaadin is a java-based server-side rich internet application (RIA) framework that

Selänniemi J. and Porres I..
AN EFFICIENT GOOGLE DATASTORE ADAPTER FOR RICH INTERNET APPLICATIONS.
DOI: 10.5220/0003389104100417
In *Proceedings of the 1st International Conference on Cloud Computing and Services Science* (CLOSER-2011), pages 410-417
ISBN: 978-989-8425-52-2

provides many advanced 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. In this article, we present the main design decisions behind the development of a Vaadin Container for the Google Datastore and how performance and costs issues are taken into account using a two-level cache schema.

We proceed as follows. In Section 2 we discuss the main issues related to the database abstraction layer in rich internet applications. Section 3 presents the functional properties of the Datastore adapter for Vaadin while Section 4 presents a two-level cache proxy. The performance and scalability of the solution is demonstrated in the experiments described in Section 5. Section 6 contains our conclusions and final remarks.

2 DATABASE-DRIVEN RICH INTERNET APPLICATIONS

In this section we introduce two of the most important challenges in developing rich internet applications in a cloud environment such as GAE: database abstraction, performance and costs issues.

Although the work presented here is specific to the Vaadin framework, we believe that a similar approach can be also applied to other RIA frameworks. RIA frameworks often have well-defined interfaces to the underlying data sources. As an example, Figure 1 shows the main methods in the Vaadin Container interface with the corresponding interfaces of the ZK (Zk, 2010) and the Echo (Echo, 2010) frameworks.

Figure 1: Comparison of important methods in the component interfaces of Vaadin, ZK and Echo3.

2.1 Database Abstraction

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.

The structure of the Vaadin data model is similar to a table in a relational database; items being rows and properties being columns. 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 textitkind of entity, which in relational terms could be thought of as the table.

The main concepts of the container interface such as adding, removing, ordering, filtering and counting, are supported by the Datastore low level API. There are, however, incompatibilities between the Vaadin Container interface and the Datastore. The data structure imposed by the container interface, i.e. a table model, cannot be enforced 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 indexes for items, while the Datastore operates on a key-value basis.

2.2 Performance and Cost

Users can take advantage of Googles own high performing 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 (Chang et al., 2008). Some examples of advantages include automatic fault tolerance, data replication and scaling (Sanderson, 2009; Google, 2010). On top of all this users only have to pay for the data they actually store and query.

Despite all these advantages the Datastore poses some challenges to RIAs. A RIA tries to simulate a desktop like experience by extensive use of UI components that may produce more database requests than traditional web applications. A recent study has shown that the Datastore has a considerable variance in read performance (Iosup et al., 2010). As the UI components are highly dependent on low and even latency to deliver their user experience, the direct use of the Datastore could lead to a slow or even an unusable application.

Although offering a global view of data, Memcache offers latencies lower and with less variance than the Datastore (Iosup et al., 2010). 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 indexes and items by keys in separate methods. This is apart from data abstraction a challenge in 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 whole pieces of hardware. As accessing data from Memcache is cheaper than from the Datastore and accesses to local memory being virtually free, another factor, i.e. cost is added to performance.

3 DATABASE ADAPTER

In the following section we further explain our design, its challenges and solutions. We have designed and implemented an adapter between the Vaadin Container interface and the GAE low level API.

To a large extent, the challenges of designing the adapter derive from incompatibilities between the Datastore's non-relational model and Vaadin data model and interface. Below we explain our approach to solving these problems.

3.1 Enforcing a Schema in Google 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. 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 the 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.

3.2 Example: Address Book

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 (Vaadin, 2010a) lets users view addresses in a table. 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 1 shows the change in code needed to use our Adapter instead of the in-memory container.

Listing 1: Creating the Adapter.

```
private GAEContainer addressBookData
    = new GAEContainer
("People", true, false, CacheFactory
    .getCache(localCacheConfig),
    CacheFactory.getCache(
    memCacheConfig));
```

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 adapter as a data source as shown in Listing 2.

Listing 2: Binding the data source.

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

Listing 3 shows how to define the data model of our address book. All properties are of type string and have an empty string as default value. We should note that this example only uses the standard Vaadin interface. The responsibility of enforcing the data model in the Google Datastore is implemented in our adapter.

Listing 3: Defining the data model.

```
private static String[] fields = {
"firstname", "lastname", "Company",
    "Mobile Phone"};
for (String p : fields) {
  addressBookData.
      addContainerProperty(p, String
      .class, "");
}
```

3.3 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 relies on the order of items are affected. The Datastore only supports sorting on the property types listed in Section 3.1. Thus on sorting the Adapter validates that each supplied property can be sorted.

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

Therefore we created a customized interface for filtering as shown in Listing 4. 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, for example, inequality filters 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 the type corresponding to that property in the data model.

Listing 4: Adapter filtering interface.

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

void removeFilters();

void removeFilters(Object propertyId
    );
```

Listing 5 shows a code snippet from using filters in the Address Book example. The code allows the user to add filters to entries in the address book through a text field. As the customized interface is used, minor changes to the code were needed.

Listing 5: Add possibility to filter through text field.

```
final TextField sf= new TextField();
sf.addListener(new Property.
    ValueChangeListener() {
 public void valueChange(
     ValueChangeEvent event) {
    ...
    addressBookData.addFilter(
        propertyName, FilterOperator
        .EQUAL, sf.toString());

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

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.

3.4 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.

Given that one entry in the address book example has the first name 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 indexes, 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 indexes. 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 the 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 trough 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.

3.5 Counting

Vaadin Containers can be queried for the number of contained items. The `size` method returns the number of items. If no filters are applied, the `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 in extent to 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.

4 CACHING PROXY

Our attempted solution to improving latency is a proxy which a cache consisting of two levels: local memory and Memcache, with a modular design allowing for more levels to be added.

4.1 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.

Furthermore, 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 (Sanderson, 2009), 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 indexes should 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 in 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 indexes, 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.

4.2 Caching Indexes

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

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 depend-

ing 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.

4.3 Consistency

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

Due to the high demands on availability, a common approach to data consistency in web applications is optimistic locking (Herlihy, 1990). 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 write-backs. If the versions correspond, the item is updated, if not, an exception is thrown. The exception allows the user to take appropriate means.

4.4 Buffering

The Vaadin interface enables individual properties to be brought out 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.

5 EXPERIMENTAL RESULTS

To get an indication of the scalability of the adapter in the GAE we carried out a series of performance experiments using the different cache levels.

5.1 How the Cache Affects Response Time

Figure 2 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 latency if there is 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. No filters or sort orders were applied. The time to retrive data from the local cache is below 1ms and therefore is not shown.

The tests display approximately the same latencies regardless of requests per second, indicating that the functionality used in these tests scales at least to the tested load.

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

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

5.2 How the Cache Affects Operation Cost

To adress 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

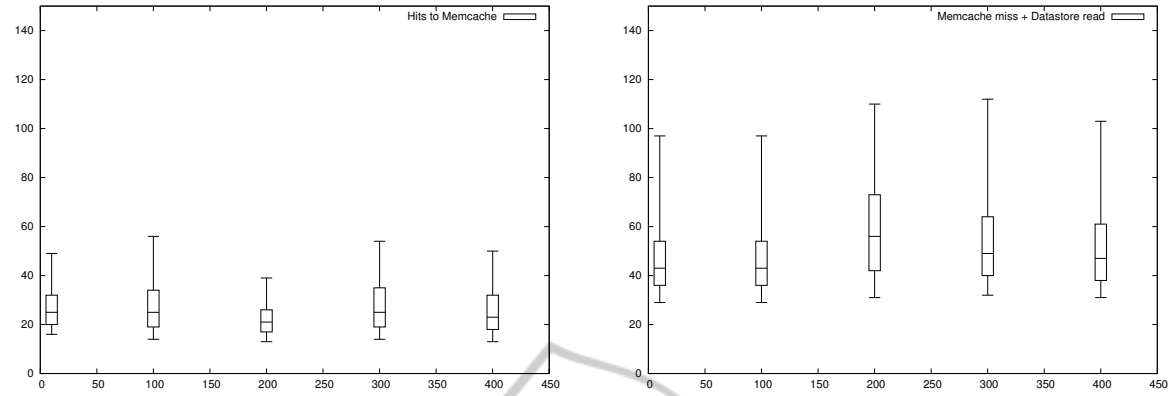

Figure 2: Read latencies for fetching a key and subsequently the item using the key; (Left) Latency when data is fetched from Memcache. (Right) Latency when data is read in Datastore after a miss in Memcache.

prices at different hit ratios. Figure 4 illustrates the cost for fetching 100,000 items, each at a time, with the same parameters as in Figure 2. The solid line indicates fetching without the adapter as a reference value using the low level API. The dashed lines show the cost results using the container for different hit ratios for misses going to Memcache and Datastore respectively. A hit comes from the local memory cache in both scenarios.

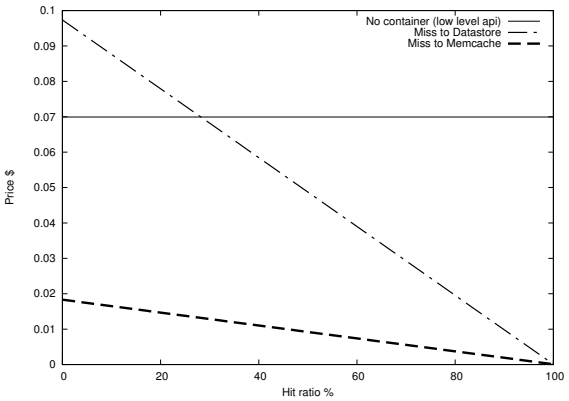

Figure 3: Comparison over cost of different level of the cache.

The values were calculated using the formulae shown in in Fig. 4, where p_l is the probability of a hit in the local cache, while p_m is 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 the order of 200 times cheaper than to a hit to Memcache, which is still 5 times cheaper than 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 to this 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.

6 CONCLUSIONS

In this article we have presented the main design decisions behind the development of a database abstraction layer of the Google Datastore for the Vaadin rich internet application and experimental performance and costs results. Our implementation is released as open source and is available to download (Selänniemi, 2010).

The solution presented here follows a well-known approach in software engineering: to add a new layer of abstraction over an existing component. Such kind of solution 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 a two-level cache. We believe that this cache approach may improve the efficiency and reduce the deployment costs when compared to direct access to the Datastore.

The challenge in our work resides in offering a solution that can fully utilize the scalability of a non-relational database, while at the same time deliver-

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

$$cost(p_l, p_m) = T_c(p_l, p_m) * cost_{cpus} \quad (2)$$

Figure 4: Calculation of computation costs.

ing the low latency needed by a RIA. The latency requirements can be achieved by introducing a multi-level cache that offers both benefits in performance and cost. There is a definite weighing between read speed, write speed and data consistency. Applications can gain a 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 to really 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.

ACKNOWLEDGEMENTS

We want to thank Joonas Lehtinen and Arthur Signell for their help and support during the whole development of this work. This work has been supported by the Cloud Software Project by Tivit, funded by Tekes, the Finnish Funding Agency for Technology and Innovation.

REFERENCES

Bunch, C., Kupferman, J., and Krintz, C. (2010). Active cloud db: A database-agnostic http api to key-value datastores. Technical report, Computer Science Department University of California, Santa Barbara.

Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. (2008). Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):1–26.

Dikaiakos, M. D., Katsaros, D., Mehra, P., Pallis, G., and Vakali, A. (2009). Cloud computing: Distributed internet computing for it and scientific research. *IEEE Internet Computing*, 13:10–13.

Echo (2010). Echo framework homepage. http://echo.nextapp.com/site/.

Google (2010). Why app engine. http://code.google.com/appengine/whyappengine.html.

Herlihy, M. (1990). Apologizing versus asking permission: optimistic concurrency control for abstract data types. *ACM Trans. Database Syst.*, 15:96–124. http://doi.acm.org/10.1145/77643.77647.

Iosup, A., Yigitbasi, N., and Epema, D. (2010). On the performance variability of production cloud services. Technical report, Faculty of Information Technology and Systems Department of Technical Mathematics and Informatics Delft University of Technology.

Sanderson, D. (2009). *Programming Google App Engine*. O'Reilly Media.

Selänniemi, J. (2010). Gaecontainer download page. http://vaadin.com/directory#addon/gaecontainer.

Vaadin (2010a). Tutorial. http://vaadin.com/tutorial.

Vaadin (2010b). Vaadin framework homepage. http://vaadin.com.

Zk (2010). Zk framework homepage. http://www.zkoss.org/.