NoSQL Data Model Evaluation on App Engine Datastore*

Felipe Ickert[1], Marcos Didonet Del Fabro[1], Eduardo Cunha de Almeida[1], Stefanie Scherzinger[2]

[1]Federal University of Paraná, Brazil (UFPR)
Curitiba – PR – Brazil

[2]Regensburg University of Applied Sciences
Germany

{fvickert,eduardo,didonet}@inf.ufpr.br

stefanie.scherzinger@hs-regensburg.de

Abstract. *Developing web applications with NoSQL databases as storage back-end can be rapidly done with existing application development frameworks. However, designing an unadapted NoSQL schema may cause serious impact on the application scalability. The impacts are often discovered only when the application is running and when the workload increases. In this article, we present a web-based tool with two different database schemas to show how the application scalability is impacted. The tool supports two different settings. First, concurrence-free write tests. Second, a simple benchmark to reproduce the behavior of concurrent writes in a cloud-based architecture. Our approach is presented in a demo app deployed in the cloud with two different NoSQL schemas revealing scalability bottlenecks.*

1. Introduction

With Platform-as-a-Service (PaaS) offering readily available infrasctructure, hosting an application in the cloud has became a commodity. A PaaS framework manages the complex deployment of the application across a large-scale hardware infrastructure, and comes with its own hosted databases, either relational or NoSQL.

NoSQL databases have as key aspect the simplicity of design, with less strict restrictions as we found in traditional DBMS (such as the ACID properties), with the main goal of achieving better scalability and availability. However, writing web applications using these databases is not always synonymous of scalability. Schema design made without care might cause severe performance bottlenecks when the workload increases. The bottlenecks may also affect availability of the application if the problem is discovered when it is running.

In this paper, we present a demo application in the cloud using a NoSQL database in order to emphasize that models done with poor design choices have impact in the performance of an application. We show how a naive schema choice can impact on the application.

This demo shows the difference and importance of the two schema designs of our case study revealing bottlenecks after some concurrency testing. Our demo is based on the

*A screencast of the tool is available at http://sbbd2013.cin.ufpe.br/screencasts.

work presented at [Scherzinger et al. 2013], where more details related to the approach can be found.

The remainder of this paper is structured as follows. In Section 2, we discuss a blogging application with alternative schema designs, as well as their impact on scalability. Our demo overview and implementation is presented in Section 3. Section 4 concludes with a summary and an outlook on future work.

2. Case Study

The core of our demo application consists of a blog application. While it may not be common, some very popular blogs may receive more than 4,500 comments over a single post. We implement two different NoSQL schemas as backend for this blog application. The first version is an article-oriented schema. The second version is na user-oriented schema. These designs use Datastore for persistence, a NoSQL backend provided with Google AppEngine [Google AppEngine 2013].

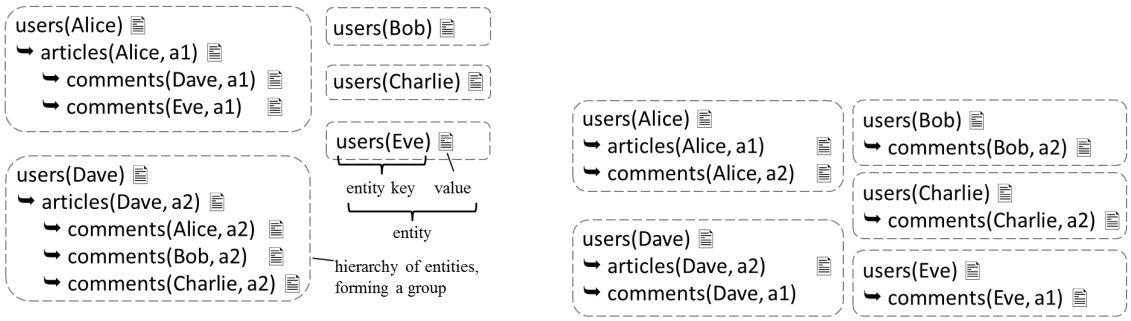
Datastore can be categorized as a key-value store[1] with entities as the basic unit of storage. Entities have a unique *key* and a *value*. Entities are made persistent by calling a simple *put*-operation, and retrieved by key using a *get*-operation. Different from a relational database, the schema is not fixed in advance. Entities need not to be consistent with any pre-defined structure. This makes Datastore a *schema-less* backend. It is nevertheless plausible to assume that developers maintain some loose notion of a schema, otherwise it becomes too complex to maintain an application. When the application logic requires strong consistency in updating several entities, the schemas and the corresponding data may be stored as an *entity group*. The system can then physically co-locate these entities, thus allowing for stricter consistency guarantees. Several systems built upon Megastore-like architectures provide this feature [Agrawal et al. 2012]. We describe the two different design below.

Naive design: The article-oriented schema. Figure 1(a) shows how entities could be grouped for a blogging application. We store an entity for each of the registered users Alice, Bob, Charlie, Dave, and Eve. Then "users(Alice)" is an entity key. The predicate "users" would then be the *entity kind*, a means of categorizing entities for the purpose of querying. As a key-value store, Datastore maps the key "users(Alice)" to the entity value containing data on Alice's user account, such as her password and preferences. In the figure, the value is depicted by a document icon, since we do not care for the exact contents.

Users may publish articles. "articles(Alice, a1)" is the key for an article written by Alice. The article entities may be stored with each user, forming a hierarchy of entities, and thus an entity group. Any comments on this article made by other users, such as a comment by user Dave with key "comments(Dave, a1)", are also stored within this group.

This may seem a natural model, since the conceptual relationships between entities are reflected in the hierarchy. However, by modeling the relationship between articles and comments in this manner, we force Datastore to co-locate all comments on a particular article in a single group. The write throughput for groups is throttled, causing writes

[1]Datastore is actually a *document store*, with structured values. Yet this distinction is of no consequence to our work.



(a) Grouping comments with articles. (b) Grouping postings with their authors.

Figure 1. Different blog schemas.

to fail if too many users are writing against a group simultaneously. With Google Datastore, at least one write can be handled against a group per second, while on average, 5–10 concurrent writes can be managed [Fuller and Wilder 2011]. Due to this limit and an optimistic concurrency control, concurrent writes against the same entity group are likely to fail.

A safer way: The user-oriented schema. The safest bet to avoid write contention is to never group any entities together. Yet storing each entity within its very own entity group comes at the cost of losing transactional safety and restricted query capabilities.

An approach that still allows for strong consistency on updates within a group and additional query options is to group all articles and comments with their authors, as depicted in Figure 1(b). Realistically, a user won't post several articles or comments per second. Hence, we consider it guaranteed that there will not be any concurrent writes against the same entity group. Thus, the alternative schema is safe from scalability bottlenecks.

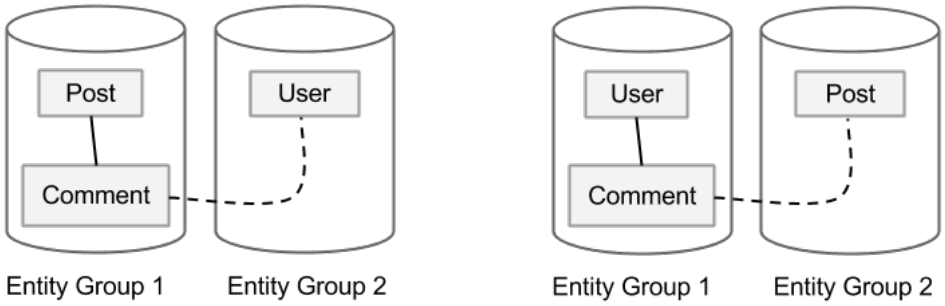
We present in the follwing sections the demo application that implements both design choices. More detailed results about the application performance can be found at [Scherzinger et al. 2013].

3. Blog Demo Overview

In order to build an application to map the two schemas presented in the case study and to show the bottleneck in an unsafe schema, we use three entities for the construction of two application blog schemas (*blog_app1 and blog_app2*). One entity represents the article (Entity Post), other represents the blog user (Entity User), and finally an entity represents a comment (Entity Comment) made by a user on an article.

We define as well the relationships between the entities in both schemas. We created two transaction classes representing the relationships between these three entities. First, one transaction class represents the post-oriented way, which is applied to *blog_app1* schema. Second, the user one is applied to *blog_app2*, considering that each transaction is guaranteed to be atomic and operates on entities in the same entity group or on entities in a maximum of five entity groups (cross-group transaction).

The first transaction class has the Entity Post as the parent of the Entity Comment,



(a) *blog_app1* (b) *blog_app2*

Figure 2. The alternative schema designs implemented.

which contains a reference to the User Entity. In this way we have a Entity Group with Entity Post plus Entity Comment, and another one with only the Entity User. In the second class, the Entity User is the parent of the Entity Comment with a reference to the Entity Post. In the same way, we have a Entity Group containing the Entity User plus Entity Comment and another Entity Group containing only the Entity Post. Each of these transactions classes represents the act of commenting an article. This characterizes the model as being scalability-safe or unsafe.

We see in Figure 2 that the entities are placed in different Entity Groups. This modeling difference is the main aspect that may impact in the application performance, since each entity group can be deployed in a different machine.

3.1. Implementation and Features

Our blog application is implemented in Java based on Google's AppEngine toolkit in the server side. It is publicly accessible at http://vanquish-ufpr-bottleneck.appspot.com and may be executed from any browser. All the requests in the client side are implemented using JQuery (Javascript framework) as well the benchmark part which uses Ajax calls to test the scalability of our different database schemas. Every time a client fires a blog writing request this task is processed by Java Servlets on the server side that communicate with the datastore, which tries to persist the data. If the datastore can not persist the data it raises a exception shown on client side.

The application consists of two main features: Single post or live test, where one can make a single comment on an article simulating a common blog. This feature is presented in two tabs, user-oriented and article-oriented, one per kind of schema, as shown in Figure 3. The other is the multiple post simulation (see Figure 4) where we can simulate a maximum of 999 comments in an article (user-oriented and article-oriented). The results of each transation is presented in real time.

We must first set up a number of users, which are created in the datastore, simulating the user subscriptions. After that the app provides two other inputs, one to include a number of comments on our first model (article-oriented) and another to include a number of comments on our second model (user-oriented). The application will perform

Figure 3. Blog live tests

asynchronous calls, respecting the users' number, to the Web Server, which will try to persist the comment. Every transaction is logged at the top of the page, Figure 5, as well a summary of these logs below the buttons where it is possible to check the bottleneck in the implementation of the unsafe data model.

In the article-oriented scenario, all comments on an article are stored in the same entity group as the article itself. If we run a series of 20, 100, 500, and 1000 write requests against one article, we get between 20% and 35% of write failures [Scherzinger et al. 2013].

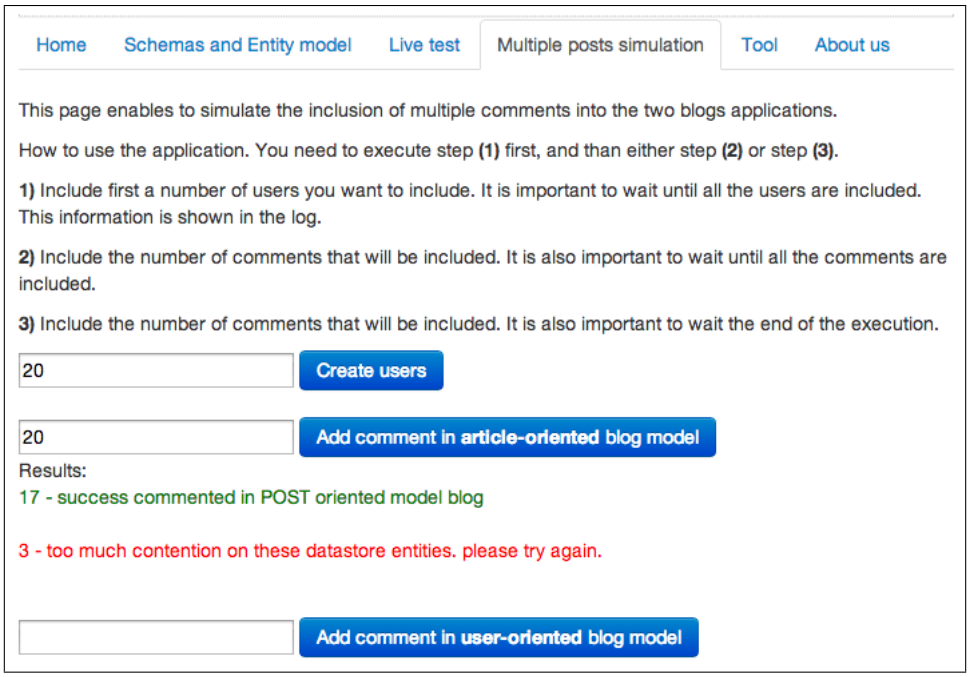


Figure 4. Multiple post simulation

4. Sumary and Outlook

In this paper, we have presented a demo application which implements two distinct blog schemas. We have shown how crucial is the design decision when developing NoSQL schemas for software-as-a-service applications. We have shown that the logical NoSQL schema has a vital impact on the overall scalability of an application. The schema is effectively responsible for physically balancing concurrent write requests. This relationship between schema and application performance is easy to miss for programmers who are new to this domain.

As future work, we plan to build a intelligent way that takes a NoSQL schema as input and that suggest others schemas which are more scalable in the cloud, or a way to help the developer to choose a safe schema before the code is deployed.

Acknowledgments. This work was partially funded by a CNPq grant.

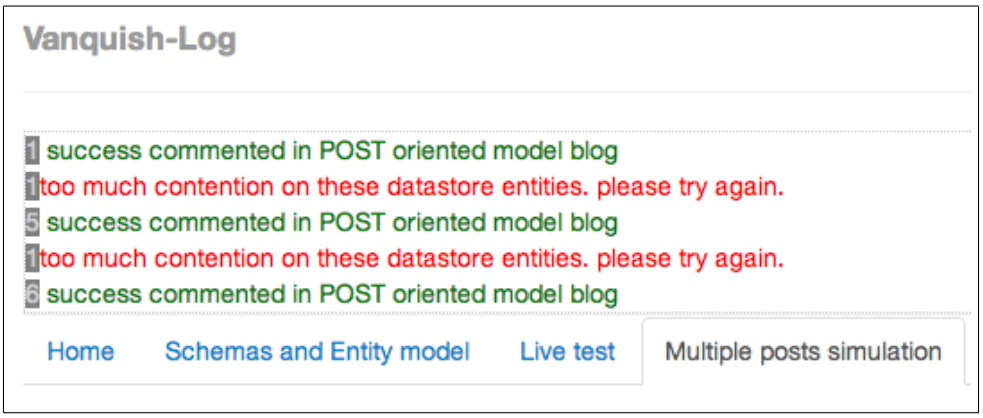


Figure 5. Multiple post simulation LOG

References

Agrawal, D., Das, S., and Abbadi, A. E. (2012). Data management in the cloud: Challenges and opportunities. *Synthesis Lectures on Data Management*, 4(6):1–138.

Fuller, A. and Wilder, M. (2011). "More 9s Please: Under The Covers of the High Replication Datastore". Google I/O Conference, presentation available on: `http://www.google.com/events/io/2011/`.

Google AppEngine (2013). Available on: `https://developers.google.com/appengine/`.

Scherzinger, S., De Almeida, E. C., Ickert, F., and Del Fabro, M. D. (2013). On the necessity of model checking nosql database schemas when building saas applications. In *Proceedings of the 2013 International Workshop on Testing the Cloud*, TTC 2013, pages 1–6, New York, NY, USA. ACM.