On the Necessity of Model Checking NoSQL Database Schemas when building SaaS Applications

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

[1]stefanie.scherzinger@hs-regensburg.de – Regensburg University of Applied Sciences, Germany
[2]{eduardo | fvickert | didonet}@inf.ufpr.br – Federal University of Paraná, Brazil

ABSTRACT

The design of the NoSQL schema has a direct impact on the scalability of web applications. Especially for developers with little experience in NoSQL stores, the risks inherent in poor schema design can be incalculable. Worse yet, the issues will only manifest once the application has been deployed, and the growing user base causes concurrent write requests to increase. In this paper, we present a model checking approach to reveal scalability bottlenecks in NoSQL schemas. Our approach draws on formal methods from tree automata theory to perform a conservative static analysis on both the schema and the expected write-behavior of users. We demonstrate the impact of schema-inherent bottlenecks for a popular NoSQL store, and show how concurrent writes can ultimately lead to a considerable share of failed transactions.

General Terms

Validation

Keywords

Stress Testing, Database, Testing Methodology

1. INTRODUCTION

With Platform-as-a-Service (PaaS) offerings readily available, hosting your application in the cloud has turned into a commodity. The 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. For developers expecting big data, a NoSQL store is the backend technology of choice.

The learning curves in working with these high-end services seem steep, yet this impression can be treacherous. While easy to get started with, NoSQL systems bear their own pitfalls when it comes to writing scalable applications. Unfortunate design choices, made early on, can later manifest in severe performance bottlenecks when the workload increases, and ultimately even prevent scalability. Even worse, these problems arise once the application is up and running and the number of users is growing rapidly.

Such traps are known amongst experienced software engineers. Yet the research community seems to have mainly focused on matters of *physical schema* design, such as investigating data partitionings under transactions [7,8,15]. The impact that the *logical* schema design can have on the performance of an application has so far been neglected. The large body of work on relational schema design [1] can only be marginally transferred to NoSQL databases.

In this paper, we introduce a typical problem encountered by novices to NoSQL technologies. We raise the question how the testing community can protect developers from common pitfalls. In particular, we stress the need for model checking on the level of the database schema. The goal is to spot severe design flaws early on in the development process. Our paper makes the following contributions.

- We introduce a real-life design challenge to the testing research community. In particular, we draw attention to scalability bottlenecks lurking in the schema design of NoSQL backends.
- We argue that NoSQL stores are still in the process of being better understood by many developers who have grown up with relational databases and ACID transactions. This makes poor design choices likely when working with new technologies.
- Our work targets logical schema design, not the physical optimization of NoSQL databases. We are thus catering to the needs of the application developers rather than those of cloud service providers.
- We provide a method for static, conservative schema analysis that checks for schema-inherent bottlenecks. Our test assumes that the schema design and the expected user write behavior are known in advance.
- Our method for model checking a NoSQL schema draws on formal methods from tree automata theory.

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 model checking approach is presented in Section 3. We experimentally demonstrate the impact of schema design on application performance in Section 4. Section 5 is dedicated to related work. We then conclude with a summary and an outlook on future work.

2. CASE STUDY: MANAGING BLOGS

We model the schema for a blogging site to illustrate the impact of early design decisions on the scalability of a NoSQL-based web application. Our project follows the lead of beacon apps such as the Huffington Post website, which handles 70+ million comments on articles each year. Some ballpark numbers from late 2012 declare that users collectively write up to 25k comments per hour. While an article receives about 40 comments on average, comments are not evenly distributed: The most popular articles count over 4,500 comments, a single video even more than 170k comments [17,18].

We choose Datastore for persistence, a NoSQL backend provided with by Google App Engine [11,16] (or an alternative such as Hypertable [12] or Cassandra [2]). While our contributions are not limited to this particular platform, we stick to Datastore terminology throughout this paper.

Datastore is a key-value store with entities as the basic unit of storage.[1] 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. Unlike with a relational database, the schema is not fixed in advance. Entities need not cohere to 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 build an application on top of it. For instance, web applications typically use object mappers [16], thus enforcing a consistent mapping between the objects in the application code and the persisted entities.

Datastore supports queries in a restricted query language. Due to a weak consistency model, queries may temporarily return stale results. When the application logic requires strong consistency in updating several entities, the entities 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 BigTable-like architectures [5] provide this feature (c.f. Section 5).

Naive design: The article-oriented schema. Figure 1 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.

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.

While this schema may seem natural from a modeling point-of-view, it contains a dangerous design issue that function tests will not reveal: Systems like Datastore are tuned towards supporting massive parallel reads, at the cost of limiting parallel writes. By modeling the relationship between articles and comments in this manner, we force Datastore to treat all comments on a particular article as a single group. However, the write throughput for groups is throttled, causing writes 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 [10]. Due to this limit and an optimistic concurrency control, concurrent writes against the same entity group are likely to fail. During a heated debate over an article, this will inevitably lead to a bad user experience.

[1]Datastore is actually a document store, since the values are further structured. Yet this additional expressiveness is of no consequence to our work.

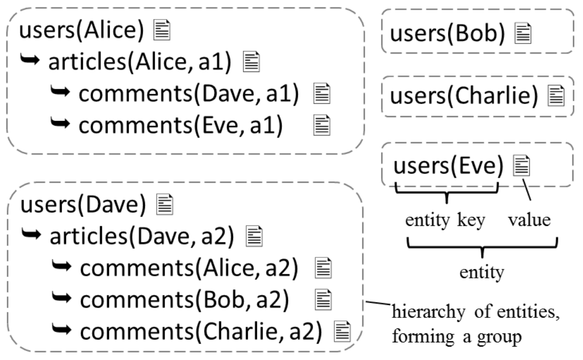

Figure 1: Grouping comments with articles.

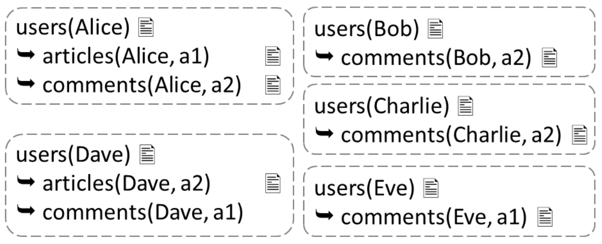

Figure 2: Grouping comments with their authors.

We take the vivid discussions on this dilemma in developer blogs and forums as a strong indication that the implications of the schema design for the ultimate scalability of an application are not easy to grasp for NoSQL novices.

A safer way: The user-oriented schema. The safest bet to avoid write contention is to never group any entities together. 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 postings with their author, whether articles or comments, as depicted in Figure 2. Realistically, a user won't post several articles or comments per second. Hence, we consider it guaranteed that there will never be any concurrent writes against the same entity group. Thus, the alternative schema is safe from such scalability bottlenecks.

3. MODEL CHECKING NOSQL SCHEMAS

Our contribution is an analysis that checks whether a schema is guaranteed to be free of concurrent write requests on the granularity of single entity groups.

We make these assumptions: (1) The schema is known, at least on the level of how entity keys are constructed and how entities are assembled inside entity groups. NoSQL systems are per se schema-less. However, any maintainable web application will require some consistent notion of the structure of entity groups. To account for large degrees of freedom, we describe the entity group structure by regular tree grammars, rather than by a relational schema. (2) Since the application code is available, we know the expected write behavior of users. (3) We assume that a single user won't execute any parallel writes against the same entity group. Of course, several users may write concurrently. This is reasonable for interactive web applications, where the application handles user web requests.

Safe schemas. Under these prerequisites, we can statically check whether a given schema is free of concurrent write requests against single entity groups. We then say the schema is *safe*. Otherwise, we conservatively assume that concurrent writes might occur, making the schema *unsafe*.

EXAMPLE 1. The schema from Figure 1 is unsafe when users write comments. During a heated debate, comments against the same article manifest in writes against the same entity group. In contrast, the schema from Figure 2 is safe since each user writes into a separate entity group. □

Outline of our approach. We encode the schema as a regular tree language. Our grammar formalism closely resembles DTDs, which define the valid structure of XML documents (essentially trees as well) [6]. We then derive a tree automaton $\mathcal{A}_s$ (the "schema" automaton) which accepts only trees that would be valid in an entity group. We capture what is known about user write behavior in the form of *update expressions*. They are the basis for a further tree automaton $\mathcal{A}_w$ (the "writes" automaton) which accepts a tree only if all updates operations occur on behalf of the same user. However, each tree in this language may be updated by a different user.

Leveraging automata theory, we check whether whether the language accepted by $\mathcal{A}_s$ is contained in the language accepted by $\mathcal{A}_w$. If this holds, then for any entity group in this schema, we can guarantee that all writes against this group are done by the same user.

There are two cases in which we classify a schema as unsafe. Either the containment check fails, or the automaton $\mathcal{A}_w$ is not deterministic to begin with. In either case, it is possible for two users to write against the same entity group.

3.1 Encoding Schemas

An entity group contains a tree of entities, possibly consisting of a single node. We next formalize the schema as a language of unranked trees. We abstract even further and focus on the entity keys, ignoring the values.

Formalizing the schema. We encode the tree structures of entity groups by an *extended regular tree grammar* [6]. These are regular grammars where the right-hand-side of a production contains a regular expression, allowing for the specification of languages over unranked, labeled trees.

Let the finite symbol set Σ contain all valid entity keys. This is the grammar alphabet.[2] For each symbol σ in Σ denoting a node label, we define a production of the form $\sigma ::= \rho$ where ρ is a regular expression over symbols from Σ. The right-hand-side of the production defines a pattern over the children of this node. We further identify a designated start production for an imaginary root node, defining which entities may be at the top-level in an entity group (a common trick in the literature on XML).

The following example defines the article-oriented schema from Figure 1. Out of convenience, we introduce this short hand for regular expressions: $|_{a\in\{a_1,\ldots,a_n\}} := a_1 \mid \cdots \mid a_n$.

EXAMPLE 2. We start with constructing the alphabet Σ. Let U be a finite set of user identifiers, and let A be a finite set of article identifiers. Then the alphabet is defined as

$$\begin{aligned}\Sigma = \{\mathit{users}(u) \mid u \in U\} \;&\cup\; \{\mathit{articles}(u,a) \mid u \in U, a \in A\}\\ &\cup\; \{\mathit{comments}(u,a) \mid u \in U, a \in A\}.\end{aligned}$$

Next are the grammar productions, defined for each user u in U and article identifier a in A:

$$\begin{aligned}
\mathit{root} &::= \;|_{u'\in U}\ \mathit{users(u')} && (1)\\
\mathit{users}(u) &::= \;\big(\,|_{a'\in A}\ \mathit{articles}(u,a')\big)^* && (2)\\
\mathit{articles}(u,a) &::= \;\big(\,|_{u'\in U}\ \mathit{comments}(u',a)\big)^* && (3)\\
\mathit{comments}(u,a) &::= \;\epsilon && (4)
\end{aligned}$$

Production (1) specifies that a user entity makes the topmost node in an entity group. (2) states that the user node has article nodes as children. (While a user won't write the same article twice, our grammar would allow this. Yet as long as the grammar is more general than the actual constraints, it is safe to simplify this way.) Production (3) specifies that an article node may have comments as children. The last production declares comments as leaf nodes. All entity groups in Figure 1 correspond to trees defined by this grammar. □

We point out that with extended regular tree grammars, we are not limited to trees of fixed depth. The grammar in the example above allows only trees that are at most three levels deep. Yet we could have just as well allowed recursive grammar productions, so that comments can be posted against comments. In the mere interest of an easily comprehensible example, we avoid any additional complexity.

Automata construction. Now that we have a way to encode the schema, we need a means for checking whether a given tree is part of this language. This is done by hedge automata [6]. We construct a hedge automaton which processes trees bottom-up, from the leaves towards the root. We

[2]In favor of a compact representation, we do not explicitly distinguish between grammar terminals and nonterminals. This follows the example of DTDs, which are also instances of extended regular tree grammars [6].

actually allow this automaton to be nondeterministic, since it can be transformed into a deterministic automaton [6].

A hedge automaton $\mathcal{A} = (\Sigma, Q, Q_f, \Delta)$ has an alphabet Σ, a set of states Q, final states Q_f, and state transition rules Δ. In our construction, we use the same alphabet as the grammar. For each symbol σ in the alphabet Σ, we define a corresponding state q_σ. For each symbol ϕ occurring in the right-hand-side of the start production $root ::= \rho$, we further declare the corresponding state q_ϕ to be final.

For each grammar production $\sigma ::= \rho$ with $\sigma \in \Sigma$, we define a state transition rule $q_\sigma \leftarrow \sigma(\rho')$, where ρ' is the regular expression obtained from ρ by replacing each symbol $\tau \in \Sigma$ by state q_τ. (Note that we do not define a transition for the grammar start production.) Transition rules assign states to nodes in bottom-up processing, and read as follows. When the children of a node labeled σ have all been assigned a state, and the sequence of their assigned states read from left to right matches the regular expression ρ', then this node is assigned state q_σ.

Example 3. Continuing our example, we define state transition rules for each user $u \in U$ and article identifier $a \in A$:

$$\begin{aligned} q_{users(u)} \leftarrow &\quad users(u)\big(\ (|_{a'\in A}\ q_{articles(u,a')})^*\ \big) \\ q_{articles(u,a)} \leftarrow &\quad articles(u,a)\big(\ (|_{u'\in U}\ q_{comments(u',a)})^*\ \big) \\ q_{comments(u,a)} \leftarrow &\quad comments(u,a)(\ \epsilon\ ) \end{aligned}$$

The final states are defined as $Q_f = \{q_{users(u)} \mid u \in U\}$. Figure 3 shows a run of this automaton on a tree from Figure 1. States are assigned to nodes bottom-up. The nodes to be processed next are highlighted in bold. As the root is assigned a final state, the automaton accepts. □

3.2 Encoding User Write Behavior

As the application code is known, we also know the users' update behavior. Key-value stores don't have powerful update languages. Typically, entities are persisted as key-value pairs and can be again retrieved from storage given their key.

Formalizing updates. We formalize a write as $put^u(k, v)$, persisting the key-value pair (k, v) on behalf of user u. This will overwrite any existing entity with the same key. We formalize a delete as $del^u(k)$, deleting the entity with key k from the store on behalf of user u. Our syntax is adapted from insert-delete-modify transactions [1].

Example 4. Users register and create accounts, publish articles, and may comment on articles. Let u denote a particular user, then we formalize his or her updates as

$$\begin{aligned} &put^u(\ users(u), v\ ) \\ &put^u(\ articles(u,a), v\ ) \\ &put^u(\ comments(u,a), v\ ) \end{aligned}$$

where v denotes some value to be stored, and a denotes some article identifier. v and a are treated as unbound variables. If users also delete all data they create, we further define

$$\begin{aligned} &del^u(\ users(u)\ ) \\ &del^u(\ articles(u,a)\ ) \\ &del^u(\ comments(u,a)\ ) \end{aligned}$$

where a denotes some article identifier. □

Automata construction. From the update expressions, we derive an automaton that accepts a tree only if all its nodes are guaranteed to be updated by a single user. These are the trees that denote safe entity groups, since there are no concurrent updates against the same entity group.

To specify this automaton, we again employ alphabet Σ. We define a state q_u for each user u, and make all states final. For each update expression of the form $put^u(k, e)$ or $del^u(k)$ we then define a state transition rule $q_u \leftarrow k(\ q_u^*\ )$.

When processing a particular entity tree bottom-up, the automaton will assign a state q_u to a leaf if this node could have been written by user u. It will assign this state to an inner node only if all of its children have been assigned state q_u, and the entity represented by the node itself could have been written by user u.

Example 5. Continuing Example 4, translating the write-operations yields these transition rules for each user u in U and each article identifier a in A:

$$\begin{aligned} &q_u \leftarrow users(u)(\ q_u^*\ ) \\ &q_u \leftarrow articles(u,a)(\ q_u^*\ ) \\ &q_u \leftarrow comments(u,a)(\ \epsilon\ ) \end{aligned}$$

The deletes do not contribute any additional rules. □

If the automaton such constructed is deterministic (i.e. it has no ambiguous transitions), it will only accept trees of entities that have been written by the a single user. In case the automaton should be nondeterministic, model checking classifies the schema as unsafe.

Example 6. The automaton just constructed is deterministic. It does not accept all trees from Figure 1, meaning this schema is unsafe: Let us pick out the tree from Figure 3(a). The leaf nodes are assigned the states q_{Dave} and q_{Eve}. There is no matching state transition rule for assigning their parent node a state. Since the root node cannot be assigned a final state, the tree is not accepted. Conceptually, when the leaf nodes carry different states, they were written by different users – possibly concurrently. The same automaton does accept all trees from Figure 2. □

3.3 Verifying Schema Safety

With hedge automata, we can check whether a tree is part of a given language. Even better, we can also make general statements without having to handle the individual trees of a language (of which there may be infinitely many): Let $L(\mathcal{A}_s)$ be the language defining the schema, and let $L(\mathcal{A}_w)$ be the language defining the safe writes (as constructed in

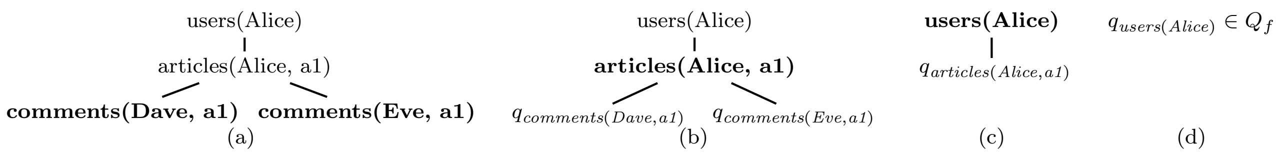

Figure 3: Bottom-up run of the automaton from Example 3.

the previous section). If $L(\mathcal{A}_s) \subseteq L(\mathcal{A}_w)$, then all entries in an entity group are guaranteed to be updated by a single user. Thus, the schema is safe under these update transitions. This check can be done on the level of the hedge automata and is a standard procedure described in [6].

Example 7. We model check the schema from Figure 1 with the automata $\mathcal{A}_s$ and $\mathcal{A}_w$ from Examples 3 and 5 respectively. Then $L(\mathcal{A}_s) \not\subseteq L(\mathcal{A}_w)$. For instance, the tree from Figure 3(a) is accepted by $\mathcal{A}_s$ but not by $\mathcal{A}_w$. Hence this schema is unsafe. □

Example 8. Let us describe the schema from Figure 2 by a grammar, again using alphabet Σ. We define productions for each user u in U and article identifier a in A:

$$\begin{aligned}
\mathit{root} &::= \quad |_{u' \in U}\ \mathit{users(u')} \\
\mathit{users}(u) &::= \quad \big(\, (|_{a' \in A}\ \mathit{articles}(u, a')) \\
&\qquad \mid \ (|_{u' \in U}\ \mathit{comments}(u', a)) \, \big)^* \\
\mathit{articles}(u, a) &::= \quad \epsilon \\
\mathit{comments}(u, a) &::= \quad \epsilon
\end{aligned}$$

We then derive the transition rules for automaton $\mathcal{A}'_s$:

$$\begin{aligned}
q_{users(u)} &\leftarrow \quad \mathit{users}(u)\big(\, ((|_{a' \in A}\ q_{articles(u,a')}) \\
&\qquad \mid \ (|_{u' \in U}\ q_{comments(u',a)}))^* \, \big) \\
q_{articles(u,a)} &\leftarrow \quad \mathit{articles}(u, a)\big(\, \epsilon \, \big) \\
q_{comments(u,a)} &\leftarrow \quad \mathit{comments}(u, a)\big(\, \epsilon \, \big)
\end{aligned}$$

As $L(\mathcal{A}'_s) \subseteq L(\mathcal{A}_w)$, this schema is safe: For each tree $t \in L(\mathcal{A}'_s)$ constructed according to the grammar, since t is also in $L(\mathcal{A}_w)$, the corresponding entity group is updated by a single user only. □

Discussion. To the best of our knowledge, our approach is the first to apply tree automata theory to model checking NoSQL schemas. Our approach is conservative in the sense that for all schemas verified as safe, it is guaranteed that no two users will ever write concurrently against the same entity group. However, it may produce false positives. There are schemas where concurrent writes occur without risk, since few users ever write concurrently. For example, sharding global counters effectively distributes the writes over several entity groups, thus increasing write throughput without avoiding concurrent writes [16]. While our decision procedure is conservative, it does detect practical instances of schema-inherent performance bottlenecks.

A known limitation of our method is that the finite set of entity keys must be enumerated in advance. We are currently investigating how we can generalize our approach to work with countably infinite alphabets.

4. EXPERIMENTAL EVALUATION

We experimentally tested the scalability of a schema in which comments are stored with articles ("article-oriented") against a schema where comments are stored with users who wrote them ("user-oriented"). We achieve concurrent writes by running a servlet on Google App Engine, triggering parallel Ajax calls against Datastore. Our testing application is publicly accessible and may be executed from any browser[3].

In the article-oriented scenario, all comments on an article are stored in the same entity group as the article itself. We created a single article, and ran a series of 20, 100, 500, and 1000 write requests against this single entity group. As Figure 4 shows, this causes between 20% and 35% of the writes to fail. In the user-oriented scenario, all comments are stored in the same entity group as the entity representing their author. Since all writes go against separate entity groups, we achieve a success rate of 100% (see Figure 4).

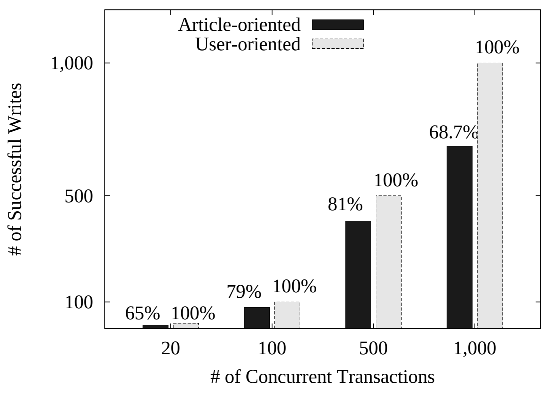

Figure 4: Successful writes per schema.

These observations confirm our claim that the logical schema design has a direct impact on the scalability of a software-as-a-service application. Interestingly, the effect can be confirmed for as few as 20 concurrent writes.

5. RELATED WORK

Schema-inherent bottlenecks are not an artifact of Google Datastore, but may be found in several systems that share a similar software stack (Megastore [4] and BigTable [5]). In the family of BigTable systems, including Hypertable [12], HBase [3], and Cassandra [2], transactions are executed to update only a subset of columns at row level. Groups of columns may be accessed together, but through a locality constraint indicated by the developers, which is the very principle underlying entity groups.

[3]http://vanquish-ufpr-bottleneck.appspot.com

In MongoDB [19], data is persisted as JSON documents to allow for dynamic schemas and to facilitate housing different applications. Transactions are performed at document level to guarantee atomicity. However, concurrent writes to the same JSON document are resource intensive and time consuming. Although auto-sharding may be used for horizontal scalability, concurrency and atomicity may influence the database schema here as well.

On the necessity of testing cloud database schemas, the investigation naturally heads towards the model transformation area where test oracles may explore variations of the database schema for validating scalability. Different transformations may be explored, including database fragmentation in order to scale out the database design [14]. The decomposition of a relation into fragments allows transactions and queries to execute concurrently, and it is used by most of the high-throughput main memory databases [13].

A possible fragmentation approach based on XML Schema is done by breaking ER-graphs into alternative hierarchies (trees) [9]. Due to the XML nature, this approach accepts multi-valued attributes that may be used to mimic *schemaless* databases. Unlike our approach, the fragmentation customizes the schema towards specific applications without taking performance into account.

In Schism [7], a workload-aware fragmentation approach improves the scalability of distributed databases. The approach creates a graph, where nodes are tuples and edges connecting the nodes are transactions that update them. The graph is fragmented to balance the transactional workload and to minimize costly cross-fragment transactions. While Schism delivers scalability to different types of applications that require distributed databases, its focus is on relational rather than NoSQL databases.

Two other approaches try to build fragments for transactional multi key accesses, which may be used in cloud data bases. G-Store [8] builds fragments for applications that require multi access to a group of keys. A fragment is built for each non-disjoint group of keys on-demand, where each fragment is kept in a single node (i.e., co-located data collection). Similarly, the One Hop Replication [15] builds fragments for social network databases to let transactions close to each other in a cluster of nodes (one-hop away in the best case). While their aim is to build fragments to let nodes with transactional multi key access close to each other, they may suffer from concurrency contention when a group of keys becomes popular, such as in our case study.

6. SUMMARY AND OUTLOOK

In this paper, we have discussed crucial design decisions in crafting NoSQL schemas for cloud-hosted web 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.

To address this knowledge gap, we perform model-checking so that application testers may discover schema-inherent bottlenecks. Our approach requires extracting an abstraction of the schema from a cloud database or the application code, as well as the write requests executed on behalf of users. We draw on formal methods from tree automata theory to perform a conservative static schema analysis. This reveals whether entity groups are guaranteed to be free of concurrent write requests. We show that our approach can detect performance contention that manifests in high rates of failed transactions. Based on this feedback, we were able to manually devise an alternative schema that scales up to highly concurrent write requests on Google Datastore.

Next, we will study the automatic recommendation of alternative schemas. Ideally, such a *schema advisor* for NoSQL databases would suggest alternative designs which show better performance under workloads with concurrent writes.

7. REFERENCES

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] Apache Cassandra. Available on: `http://cassandra.apache.org/`.

[3] Apache HBase. Available on: `http://hbase.apache.org/`.

[4] J. Baker, C. Bond, J. C. Corbett, J. Furman, et al. "Megastore: Providing Scalable, Highly Available Storage for Interactive Services". In *Proc. CIDR*, pages 223–234, 2011.

[5] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, et al. "Bigtable: A distributed storage system for structured data". In *Proc. OSDI*, pages 15–15, 2006.

[6] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. *Tree Automata Techniques and Applications*. 2007.

[7] C. Curino, Y. Zhang, E. P. C. Jones, and S. Madden. "Schism: a Workload-Driven Approach to Database Replication and Partitioning". *PVLDB*, 3(1):48–57, 2010.

[8] S. Das, D. Agrawal, and A. El Abbadi. "G-Store: A scalable data store for transactional multi key access in the cloud". In *Proc. SoCC*, pages 163–174, 2010.

[9] R. Elmasri, Y.-C. Wu, B. Hojabri, C. Li, and J. Fu. "Conceptual Modeling for Customized XML Schemas". In *Proc. ER*, pages 429–443, 2002.

[10] A. Fuller and M. Wilder. "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/`, May 2011.

[11] Google AppEngine. Available on: `https://developers.google.com/appengine/`, May 2013.

[12] Hypertable Inc. Available on: `http://hypertable.org/`.

[13] E. P. C. Jones, D. J. Abadi, and S. Madden. "Low overhead concurrency control for partitioned main memory databases". In *Proc. SIGMOD*, pages 603–614, 2010.

[14] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems, Third Edition*. Springer, 2011.

[15] J. M. Pujol, G. Siganos, V. Erramilli, and P. Rodriguez. "Scaling online social networks without pains". In *Proc. NetDB*, 2009.

[16] D. Sanderson. *Programming Google App Engine*. O'Reilly, 2013.

[17] J. Sonderman. "How the Huffington Post handles 70+ million comments a year". Article available on: `http://www.poynter.org`, October 2012.

[18] The Huffington Post. Available on: `http://www.huffingtonpost.com/theblog/index/`, May 2013.

[19] The MongoDB Company. "MongoDB Operations Best Practices. MongoDB v2.2.". Technical report, 10Gen, February 2013.