AutoShard – Declaratively Managing Hot Spot Data Objects in NoSQL Document Stores

Stefanie Scherzinger
Technical University
of Applied Sciences Regensburg
Regensburg, Germany
stefanie.scherzinger@oth-regensburg.de

Andreas Thor
Deutsche Telekom
University of Applied Sciences
Leipzig, Germany
thor@hft-leipzig.de

ABSTRACT

NoSQL document stores are becoming increasingly popular as backends in web development. Not only do they scale out to large volumes of data, many systems are even custom-tailored for this domain: NoSQL document stores like Google Cloud Datastore have been designed to support massively parallel reads, and even guarantee strong consistency in updating single data objects. However, strongly consistent updates cannot be implemented arbitrarily fast in large-scale distributed systems. Consequently, data objects that experience high-frequent writes can turn into severe performance bottlenecks. In this paper, we present AutoShard, a ready-to-use object mapper for Java applications running against NoSQL document stores. AutoShard's unique feature is its capability to gracefully shard hot spot data objects to avoid write contention. Using AutoShard, developers can easily handle hot spot data objects by adding minimally intrusive annotations to their application code. Our experiments show the significant impact of sharding on both the write throughput and the execution time.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*concurrency, distributed databases, transaction processing*

General Terms

Design, Performance, Algorithms

1. INTRODUCTION

NoSQL document stores are highly appealing in web development, especially for applications that require high scalability and high availability. Conveniently, NoSQL data stores are readily available with established web hosting platforms, such as Google App Engine [18]. The flexible data model that these systems commonly provide suits an agile software development style since the database schema does not have to be designed up front. New attributes can be easily added on-the-fly as required by new features. The simple data access methods (i.e., put() and get(key)) and the limited query capabilities are usually sufficient for web applications. Moreover, the sheer scalability of these systems is impressive: Due to a highly distributed architecture, these systems gracefully handle large amounts of users and data.

Seventeenth International Workshop on the Web and Databases (WebDB 2014), June 22, 2014 - Snowbird, UT, USA.

NoSQL document stores are capable of scaling out over tens of thousands of nodes. With such an architecture, arbitrary transactions with ACID semantics are usually not feasible, and effects of eventual consistency come into play. Yet the application logic often demands strong consistency. Therefore, systems such as Google Cloud Datastore that have been designed with web applications in mind, allow for strongly consistent updates in restricted cases, e.g. when changes affect a single data object only [5].

Yet strongly consistent updates come at the cost of slower writes. With Google Cloud Datastore, the supported limit for writing against a single data object is merely one update per second [4]. This can result in *write contention*, an effect not unique to this particular system, e.g. [2].

Commonly, web applications are read-intensive, so in the presence of few writes, this limit may not even be noticed. However, there are certain features in web applications that are inherently prone to write contention, such as a global counter recording page visits, or many users hitting a *like* button on a popular image.

Example 1. Consider crowd sourcing tools such as *Google Moderator* for ranking user-submitted questions (see Figure 1). In 2008, this tool has been successfully employed in the U.S. presidential debates, with one million votes from 20,000 people in just 48 hours[1]. Thanks to a sophisticated implementation, *Google Moderator* is highly scalable. However, a naive implementation on a backend such as Google Cloud Datastore may not scale: If the counter tracking positive votes on a question is implemented by a *single* document, already tens of users concurrently voting on the same popular question will cause write contention. Ultimately, this triggers runtime errors during peak times.

How to deal with so-called *hot spot data objects* has already been subject of study back in the late 70s and early 80s. Typical use cases in those days were the number of available seats on a plane, or the overall balance of bank accounts. The idea of exploiting the semantics of data items and their transactions has been termed *semantics-based transaction processing*, and has lead to sophisticated solutions, such as the IMS Fast Path system [16], or the Escrow method [13]. A comprehensive survey can be found in [15].

Yet the challenge of managing hot spot data objects in today's web applications comes with several novel aspects:

1. NoSQL data stores frequently implement *optimistic concurrency control*, whereas solutions designed for re-

[1]`http://en.wikipedia.org/wiki/Google_Moderator`

Figure 1: Users vote questions up or down in *Google Moderator* during the U.S. presidential debates, creating hot spot data objects.

lational database systems usually assume pessimistic locking mechanisms to be in place.

2. When NoSQL data stores are used as database-as-a-service, it is impossible for developers to extend or even customize the functionality of their backends. Thus, we need to be able to handle write contention on the level of the database application, rather than by physical database design.
3. Finally, NoSQL data stores commonly do not provide full ACID transaction behavior, so eventual consistency effects must be considered.

The established approach in the developer community for managing hot spot data objects in NoSQL data stores is application-level *sharding* of documents, e.g. as suggested in [9, 18] for Google Cloud Datastore, [8] for Amazon SimpleDB, and [2] for Couchbase. Like with traditional data fragmentation and allocation in distributed systems [14], sharding data objects effectively distributes write requests across physical nodes. However, sharding is now managed in the application code: For example, instead of storing a single counter for voting up a *Google Moderator* question, we maintain 10 shard counters. Updates are then performed on a single, randomly chosen shard, and the sum across all shard counters yields the overall vote count.

While the idea is intuitive, getting sharding right is not trivial. Writing custom sharding code (e.g. as exemplified in [9]) requires a deep understanding of the underlying technology and its transactional behavior. Additionally, this introduces a new level of complexity in the application code, which in return increases the development and testing effort. This is not to be underestimated when sharding is added in hindsight, to already existing code. Also, coupling the sharding code with the application logic enforces the technological lock-in with one particular database provider.

What is missing today is a well-principled machinery for sharding that does not amount to major code refactoring.

Contributions. Our main contributions are:

- We give a systematic overview for sharding hot spot data objects in NoSQL document stores. In particular, we introduce static and dynamic *property sharding*, as well as *entity group sharding*.
- We present AutoShard, a novel Java object mapper specifically designed for NoSQL document stores. AutoShard relieves developers from having to deal with low-level property sharding, and thus restores a clearer separation between logical and physical database design. AutoShard is designed for ease-of-use, merely requiring simple declarative annotations in Java classes. Its architecture relies on self-modifying code to transparently generate sharding code.
- We evaluate static property sharding with AutoShard and thereby demonstrate a significant improvement on both the write throughput and the average execution times of writes against hot spot data objects.

Organization. The remainder of this paper is organized as follows. Section 2 presents various sharding strategies. We discuss the tradeoffs in sharding, i.e. eliminating severe performance bottlenecks and thereby trading in eventual consistency effects that are still tolerable for a large class of practical applications. In Section 3, we introduce AutoShard, our Java object mapper that unburdens developers from writing custom sharding code. Instead, developers conveniently specify which data members are to be sharded by adding annotations to their Java code. We present details on the AutoShard architecture in Section 4. We experimentally evaluate our implementation of AutoShard (Section 5), and conclude with a summary of our work and the directions for future research.

2. SHARDING STRATEGIES

We next give an overview of common sharding strategies as established in the developer community. AutoShard implements two ways of sharding atomic document properties:

- In *static property sharding* (e.g. [2, 9]), the number of shards is fixed, whereas
- in *dynamic property sharding* (e.g. [8]), the number of shards grows on demand.

Property sharding is applicable to a large class of NoSQL document stores. We illustrate the idea behind these approaches, and further discuss *entity group sharding*, where sharding is applied to groups of documents. Entity groups are a feature specific to Google Cloud Datastore [18] and its underlying software layer, the Google-internal Megastore [1].

2.1 Static Property Sharding

Let us resume our discussion of building a scalable voting application. An example of a voting question is shown below in JSON format. We refer to persisted objects as *entities*.

```
{"kind" : "Question", "id" : 42,
 "question" :
      "How do you plan to improve public education?",
 "author" : "Phil R",
 "responses" : [
    {"response" :
     "i have earned $1048 dollars just by ad clicks",
     "author" : "twodollarclick"} ],
 "votes" : 76}
```

Each entity is assigned a *kind*, which is simply a classification of the entity as a question in this case. Each entity has a unique identifier and further *properties* (c.f. attributes). Properties may be atomic, multi-valued, structured, and even nested (e.g. like the list of responses above).

The rate at which users may vote on this question is physically limited. In systems such as Google Cloud Datastore, only a minimum write throughput of one write per second per entity is guaranteed (with 5 to 10 concurrent writes achievable on average [3]). Yet a controversial question is likely to receive concurrent votes. Write contention then causes runtime errors, and ultimately, results in data loss, since not all updates can be persisted.

The recommended approach is to *shard* property `votes`, creating $n + 1$ entities instead. One entity stores the question without the `votes`-property, we refer to it as the *main entity*. The value of the `votes`-property is distributed over n single entities, the *shards*. This is shown below. The first shard with identifier `42-1` stores the original value of the `votes`-property, whereas the `shard_votes`-property in all other shards has been set to zero. We can always obtain the total number of votes for a given question by computing the sum over the `shard_votes` across all n shards.

```
{"kind" : "Question", "id" : 42,
 "question" :
    "How do you plan to improve public education?",
 "author" : "Phil R",
 "responses" : [
    {"response" :
     "i have earned $1048 dollars just by ad clicks",
     "author" : "twodollarclick"} ]}

{"kind" : "Shard",  "id" : "42-1",
 "question" : "42", "shard_votes" : 76}

{"kind" : "Shard",  "id" : "42-2",
 "question" : "42", "shard_votes" : 0}      ...

{"kind" : "Shard",  "id" : "42-n",
 "question" : "42", "shard_votes" : 0}
```

Whenever the question is voted on, a single shard is picked at random, `shard_votes` is incremented, and the shard is persisted again. This can usually be executed as an atomic action, and effectively distributes concurrent writes across the n shards, rather than all concurrent updates affecting a single entity. Since addition of integers is commutative and associative, this is mathematically sound.

Tradeoffs. By sharding the hot spot counter, we have eliminated a crucial scalability bottleneck in our voting application. As we show in Section 5, sharding significantly improves the write throughput on single entities, while keeping the average transaction time within acceptable bounds. Yet sharding hast two inherent drawbacks, owing to the particularities common to many NoSQL document stores:

1. Range queries over shards may not be supported.
2. The computation of the total number of votes may show eventual consistency effects.

Let us elaborate on drawback (1): NoSQL document stores commonly provide very restricted query languages. For instance, Google Cloud Datastore would support the following query over the original, unsharded question entity:

```
select * from Question where votes > 50
```

Yet the query language is not expressive enough to compute the equivalent query in the presence of shards. Consequently, developers need to write custom code to retrieve these questions. On the good side, queries filtering over unsharded properties can still be expressed.

(2) We next consider compromises in consistency. In many NoSQL data stores, updating a single entity is a strongly consistent action (c.f. [17]). Let us assume that shards `42-2` and `42-3` from our example have been updated concurrently:

```
{"kind" : "Shard",  "id" : "42-2",
 "question" : "42", "shard_votes" : 1}

{"kind" : "Shard",  "id" : "42-3",
 "question" : "42", "shard_votes" : 1}
```

At this point, the total number of votes reaches 78. Let us try to retrieve this value. Since the query language is not expressive enough to aggregate across several entities, we first issue a query to fetch all shards for question `42`:

```
select * from Shard where question = 42
```

This query is evaluated across a large cluster of nodes, and thus may not return a strongly consistent result. Next, we programmatically aggregate over the `shard_votes`. Due to the effects of eventual consistency, repeatedly executing the query at time of the updates may return the stale results 76 or 77, and eventually will return the consistent value 78.

In an application such as the voting app, temporarily stale results are tolerable, as long as queries return the consistent state by the time that the result is to be utilized (e.g. when the presidential debate actually begins). Therefore, sharding trades strong consistency for scalability in terms of concurrent writes. This is a valid tradeoff for applications where we are mainly interested in a ballpark number (e.g. counting the number of visitors to a website), and where the order of updates does not matter (unlike an auctioning site, for example). After all, the alternative is an application that suffers from runtime errors and data loss at peak times.

2.2 Dynamic Property Sharding

We now introduce an alternative approach, which we refer to as dynamic property sharding. The previous discussion of tradeoffs applies here as well. In our running example, we start with a *single* shard where the property `shard_votes` is set to the original value of `votes`.

```
{"kind" : "Shard",  "id" : "473",
 "question" : "42", "shard_votes" : 76}
```

For each user who increments the counter, we add a new shard and let the NoSQL data store assign a unique key. Thus, when two users increment the counter concurrently, two new shards with `shard_votes`=1 are added:

```
{"kind" : "Shard",  "id" : "119",
 "question" : "42", "shard_votes" : 1}

{"kind" : "Shard",  "id" : "236",
 "question" : "42", "shard_votes" : 1}
```

Even under immense write load, increments can be executed without *any* concurrent writes against a single entity. This comes at the cost of higher storage requirements. Again, the total shard value is computed by aggregating over all shards. This may temporarily yield stale results, again due to eventual consistency effects. An independent batch process, e.g., run nightly or when the system is under less load, compacts the shards that have accumulated. This reduces the number of shards, as well as the storage costs. Since there is a single thread writing (or rather, deleting) the shards, this does not cause write contention. Dynamic property sharding scales more gracefully under peak loads, yet amounts to a considerable implementation effort, involving background batch processes or MapReduce jobs.

```
@Entity class Question {
  @Id private int id;
  private String question;
  private String author;
  private List<Response> responses;

  @Shardable (neutral=0, shards=10)
  private int votes = 0;

  @ShardMethod
  public void voteUp() {
    this.votes++;
  }

  @ShardFold
  public static int foldVotes(int x, int y) {
    return x + y;
  }
  /* ... not showing getters and setters ... */
}
```

Figure 2: Java class with AutoShard annotations.

2.3 Entity Group Sharding

Entity groups are a particular feature of Google Cloud Datastore [5] and Megastore [1]. Entities can be arranged in groups by defining a hierarchy between entities. By physically co-locating the entities inside a group, the system can guarantee ACID updates within the scope of the group.

Different from our original data design of a question with nested responses, we can store the responses in the same group as the question. Below, property `parent-id` references the question as the root of the hierarchy.

```
{"kind" : "Question", "id" : 42,
 "question" :
    "How do you plan to improve public education?",
 "author" : "Phil R",
 "votes" : 76}

{"kind" : "Response", "id" : 47, "parent-id" : 42,
 "response" :
     "i have earned $1048 dollars just by ad clicks",
 "author" : "twodollarclick"}

{"kind" : "Response", "id" : 67, "parent-id" : 42,
 "response" : "Crucial for our future",
 "author" : "Stan S"}
```

As with single entities, Datastore limits the number of concurrent writes against an entity group. Several users responding to a question in a heated debate thus turn the group into a hot spot data object. In entity group sharding, we consequently shard entity groups, now distributing writes over several groups. To restore all entities from the original group, we compute the union of entities from across several groups. This improves the rate of successful writes, at the cost of making certain atomic updates impossible.[2]

3. THE AUTOSHARD OBJECT MAPPER

The main focus of our work is a novel object mapper framework for automatically rewriting annotated Java classes with property sharding. Like other object mappers, (e.g. [6, 7,11,12]), AutoShard takes care of the mundane marshalling of persisted entities into Java objects and back, thus greatly simplifying application development. Just like established object mappers, AutoShard relies on Java language metadata annotations.

We show how AutoShard helps with our running example, the voting app. The Java class from Figure 2 represents a `Question` that can be voted up. In the following, we resolve write contention by property sharding.

As is customary with object mappers, the annotation `@Entity` specifies that an instance of class `Question` is to be persisted as an entity. The annotation `@Id` marks the unique key of the persisted entity.

The mapping of an instance of class `Question` onto a persisted entity is straightforward. As discussed previously, a *single* entity is a performance bottleneck with multiple users voting concurrently on the same question. To solve this problem by sharding, we merely add annotations. The annotation `@Shardable` specifies that the class member `votes` is to be sharded.[3] When processing shards, the method annotated with `@ShardMethod` will be applied to a single shard, rather than the global value of the votes counter. We could even declare several sharding functions (e.g., to increment and decrement votes). Since in this example the shard method is incrementation, we specify zero as the neutral element (see `neutral=0`). This information is exploited in initializing new shards. Further, we request static property sharding with ten shards (specifying `shards=10`). If no shard limit is specified, AutoShard shards dynamically.

With annotation `@ShardFold`, we declare the static function `foldVotes` as the folding function. This function is called for aggregating over all shards. We may specify even more complex folding operations, as long as they are commutative and associative. It is the responsibility of the developers to correctly annotate their Java classes.

4. THE AUTOSHARD ARCHITECTURE

The AutoShard object mapper is, to our knowledge, the first Java object mapper to shard properties based on simple annotations. Our approach relies on self-modifying code by blending Java code with Groovy technology [10]. Groovy is a dynamic language that runs in the JVM and smoothly inter-operates with Java code. Further, Groovy allows us to annotate code structures for transformations in the abstract syntax tree (AST) during compilation.

Figure 3 shows the architecture of the AutoShard framework. A Java class with AutoShard annotations serves as input. The Groovy parser produces an AST and our AutoShard AST transformer restructures this tree. Class members annotated as `@Shardable`, as well as the sharding and the folding method, are now transformed.

We consider the modifications required for the class from Figure 2. When compiling for Google Cloud Datastore, AutoShard's Groovy-based compiler generates the Java class shown in Figure 4. For the sharded property `votes`, compilation introduces a new (private) attribute `shard_votes` that stores a single shard value. The body of user-defined method `voteUp` is transferred to a private method, and the original method is replaced as shown in Figure 4. This new implementation calls the original function both for the shard

[2] Google Cloud Datastore only allows ACID transactions involving up to five entity groups [18].

[3] Figure 2 shows the simple case of a single sharded class member. Naturally, the syntax of AutoShard annotations also allows for several data members to be sharded.

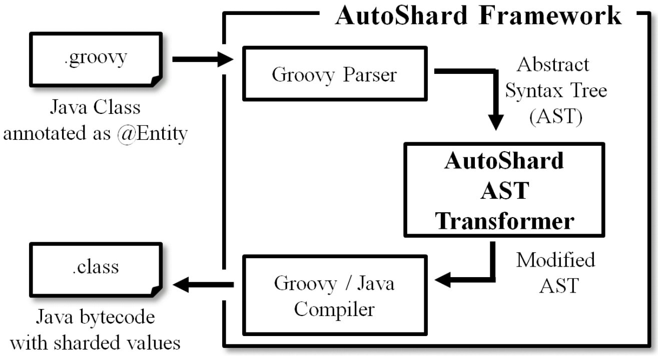

Figure 3: The AutoShard framework architecture for compiling annotated Java classes.

value (`shard_votes`) and for the actual value (`votes`). Since the signatures of the class methods do not change, the remaining application code need not be adapted.

At runtime, we use the modified `Question` class during loading, updating, and saving entities:

Loading: When a new instance of a `Question` is loaded, AutoShard retrieves the main entity to map all unsharded class members. For the sharded class member `votes` it reads all shards and generates *two* data members. First, the (regular) data member `votes` is initialized to the aggregated shard value. AutoShard uses the `@ShardFold` method (`foldVotes` for class `Question`) to aggregate over all shards. Second, the (internal) data member `shard_votes` is initialized to the neutral element zero.

Updates: When shard method `voteUp` is invoked for updating the counter value, the update is executed on both the (regular) data member `votes` as well as the (internal) data member `shard_votes`. This ensures that whenever the application code accesses `votes`, it sees the expected value.

Saving: When entity `Question` is persisted after changes have been made, AutoShard first updates the main entity. For the sharded class member a random shard is loaded from storage, as shown in Figure 5. Its value is updated by invoking the `@ShardFold` (`foldVotes` for class `Question`) method on the loaded shard value and on `shard_votes`. The shard is persisted within a nested transaction, so that we do not interfere with any transactions that may be running in the remaining code. Since the sharded value is re-set to the neutral element, it will capture future updates. Note that the regular property `votes` still holds the current value.

Persisting entities and retrieving them by key are the main building blocks for web applications when interacting with the NoSQL backend. NoSQL document stores also provide basic query languages. For example, Google Cloud Datastore allows queries on entities of the same kind (or type) using simple property filters. Queries on unsharded properties or entities are not affected by AutoShard compilation. Hence, they can be run without changes (c.f. Section 2.1).

5. EVALUATION

We investigate the runtime benefits of sharding with AutoShard. We have implemented AutoShard with property sharding for Google Cloud Datastore, a commercial NoSQL document store handling 6.3 trillion daily requests[4]. Our evaluation scenario deals with a Java implementation of a voting tool in the style of Google Moderator. The application is hosted on Google App Engine.

We start with a naive implementation that does not take precautions for handling concurrent writes. A shell script simulates an increasing number of users voting on popular questions, e.g., 75 voting requests per second are equally spread across 16 questions. This causes write contention on the level of persisted entities. As seen in Figure 6, for this naive implementation without any transaction retries (naive "w/o Tx retry"), 25% of the transactions fail due to write contention. This failure rate is obviously unacceptable for real world web applications.

We then repeat the experiment with a sharded version, where we have added the AutoShard annotations from Figure 2 to the code and have recompiled the application. This time, the application experiences only 4% of failed requests (see AutoShard "w/o TxRetry"). This improvement is due to the fact that write contention is reduced by distributing writes across multiple shards. However, a failure rate of 4% can still be considered alarming. Note that in this experiment, we can observe a slight increase in the average transaction time, due to the overhead imposed by sharding.

To ensure that all votes are indeed persisted, we add transaction retries ("w/TxRetry"), so transactions retry until they succeed. This obviously increases execution time. We repeat the experiment with the naive implementation, as well as with the code generated by AutoShard, and visualize the performance results in Figure 6. Both versions show a 100% success rate, yet the average transaction time for the sharded version is clearly superior (300ms vs. 430ms).

```
class Question {
  private int votes;     // the aggregated value
  private int shard_votes; // single shard value

  // internal method with body of
  // original voteUp method
  private void _voteUp() {
    this.votes++;
  }

  // new voteUp method
  public void voteUp() {
    int tmp = votes;
    votes = shard_votes;
    _voteUp(); // updating the shard value
    shard_votes = votes;

    votes = tmp;
    _voteUp(); // updating the aggregated value
  }

 // the @ShardFold method
  public static int foldVotes(int x, int y) {
    return x + y;
  }
  /* ... not showing unsharded class members,
     getters, and setters ... */
}
```

Figure 4: The modified Java class `Question` as generated by AutoShard during compilation.

[4]Quoting Urs Hölzle in his keynote at *Google Cloud Platform Live 2014*, available online at `https://cloud.google.com/events/google-cloud-platform-live/`.

```
public void save(Question q) {
 DataStore.put(q);     // save the main entity
 BEGIN TRANSACTION
  // read ONE shard value at random
  shard = DataStore.getRandomShard(q);

  // fold shard value with object shard property
  // using the @ShardFold method
  shard.shard_votes =
  Question.foldVotes(shard.shard_votes, q.shard_votes);

  // save the updated shard
  DataStore.put(shard);

  // re-initialize local shard_votes data member
  q.shard_votes = 0; // 0 = neutral element
 END TRANSACTION
}
```

Figure 5: Pseudo-code for saving sharded Question entities with AutoShard.

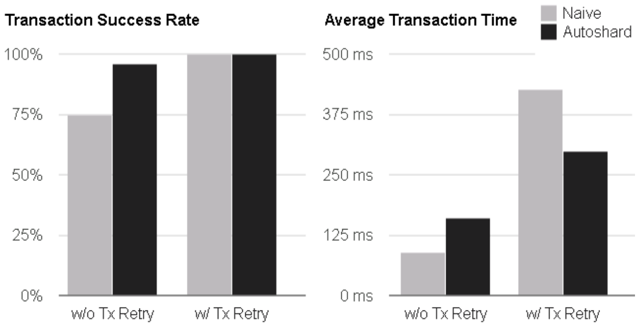

Figure 6: Evaluation of static property sharding where 2,000 users vote on 16 questions. A naive implementation shows an unacceptable failure rate of over 25%. AutoShard reduces the failure rate down to 4% (using 16 shards). When adding a transaction retry mechanism to ensure a 100% success rate, AutoShard significantly reduces the average transaction time compared to the naive implementation.

This brief evaluation scenario confirms the usefulness of AutoShard, i.e., Autoshard is capable of gracefully sharding hot spot data objects. An extended evaluation will compare static and dynamic sharding and will analyze the impact of the number of shards on the transaction time.

6. SUMMARY AND FUTURE WORK

In this paper we have presented AutoShard, a ready-to-use object mapper for Java applications running against NoSQL document stores. In addition to mapping Java objects to persisted entities, AutoShard is capable of sharding properties so that hot spot data objects can be managed gracefully. This form of application-managed sharding ties in with the long tradition of efforts to avoid write contention over hot spot data objects (c.f. [15]).

A main strength of AutoShard is the ease with which data objects may be sharded, namely by merely annotating the Java code. We have demonstrated the merits of AutoShard by contrasting the performance of a naive implementation of a realistic web application with a sharded version generated by AutoShard. Our experiments show the significant impact of property sharding on the throughput of write requests.

In our future work, we are investigating a generic approach to property and entity group sharding that is not specific to a certain data store. Programmers should be able to describe all relevant data store properties (e.g., consistency model, ACID guarantees) so that AutoShard can implement suitable sharding strategies. We will examine how to automatically identify properties that require sharding as well as to automatically determine a suitable number of shards. We are currently extending AutoShard so that queries involving sharded properties or entities are handled transparently (whenever possible). AutoShard is scheduled to be made available as open source software.

7. REFERENCES

[1] 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.
[2] Couchbase Inc. *Making the Shift from Relational to NoSQL.* Couchbase Whitepaper. 2013. http://info.couchbase.com/Relational-to-NoSQL.html.
[3] A. Fuller and M. Wilder. More 9s Please: Under The Covers of the High Replication Datastore. Google I/O Conference 2011. https://www.youtube.com/watch?v=xO015C3R6dw.
[4] Google Inc. Balancing Strong and Eventual Consistency with Google Cloud Datastore, 2014. https://cloud.google.com/developers/articles/balancing-strong-and-eventual-consistency-with-google-cloud-datastore.
[5] Google Inc. Google Cloud Datastore, 2014. https://developers.google.com/appengine/docs/java/datastore/.
[6] Google Inc. Using JDO with App Engine, 2014. https://developers.google.com/appengine/docs/java/datastore/jdo/.
[7] Google Inc. Using JPA with App Engine, 2014. https://developers.google.com/appengine/docs/java/datastore/jpa/overview.
[8] M. Habeeb. *A Developer's Guide to Amazon SimpleDB.* Developer's Library. Pearson Education, 2010.
[9] Joe Gregorio. Sharding Counters, January 2013. https://developers.google.com/appengine/articles/sharding_counters.
[10] D. Koenig, G. Laforge, and A. Glover. *Groovy in Action.* Manning Publications Co., 2006.
[11] Morphia. A type-safe Java library for MongoDB, 2014. http://code.google.com/p/morphia/.
[12] Objectify AppEngine. The simplest convenient interface to the Google App Engine Datastore, 2014. https://code.google.com/p/objectify-appengine/.
[13] P. E. O'Neil. The Escrow transactional method. *ACM Trans. Database Syst.*, 11(4):405–430, Dec. 1986.
[14] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems, Third Edition.* Springer, 2011.
[15] K. Ramamritham and P. K. Chrysanthis. *Executive Briefing: Advances in Concurrency Control and Transaction Processing.* IEEE Computer Society Press, Los Alamitos, CA, USA, 1996.
[16] A. Reuter. Concurrency on high-traffic data elements. In *Proc. PODS*, pages 83–92, 1982.
[17] P. J. Sadalage and M. Fowler. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence.* Addison-Wesley Professional, 1st edition, 2012.
[18] D. Sanderson. *Programming Google App Engine.* Google Press, 2012.