AutoShard – A Java Object Mapper (not only) for Hot Spot Data Objects in NoSQL Data Stores

Stefanie Scherzinger
OTH Regensburg
stefanie.scherzinger@oth-regensburg.de

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

Abstract: We demonstrate AutoShard, a ready-to-use object mapper for Java applications running against NoSQL data stores. AutoShard's unique feature is its capability to gracefully shard hot spot data objects that are suffering under concurrent writes. By sharding data on the level of the logical schema, scalability bottlenecks due to write contention can be effectively avoided. Using AutoShard, developers can easily employ sharding in their web application by adding minimally intrusive annotations to their code. Our live experiments show the significant impact of sharding on both the write throughput and the execution time.

1 Introduction

NoSQL data stores are highly appealing in web development, where applications require high scalability and high availability. Their flexible data model suits an agile software development style since the database schema does not have to be designed up front. Moreover, the sheer scalability of these systems is impressive: Due to a highly distributed architecture, they gracefully handle large amounts of users and data. To optimize user response time, several NoSQL data stores implement optimistic concurrency control [ADE12]. Transactions execute immediately, and then check at commit time if some other transaction has already committed a conflicting update. If so, the transaction is aborted and it is up to the application to handle the failure. This approach is appropriate for web applications where users spend most of their time browsing data, rather than generating content.

Since web applications are commonly read-intensive, users are less likely to concurrently update the same data. However, there are certain features in web applications that are inherently prone to write contention: This could be something as simple as a counter registering the number of visitors to a website, or the number of votes (or "likes") cast interactively against a blogpost. Write contention on *hot spot data objects* has a long legacy in database research, and at the same time is a timeless topic, e.g., recently motivating sophisticated concurrency control protocols that avoid shared-memory writes [TZK+13].

The established approach in the developer community for managing hot spot data objects in NoSQL data stores is *property* or *counter sharding* [ST14, San12] on the level of the database schema. Sharding physically balances write requests by dividing logical units of data into multiple shards. For example, instead of storing a *single* counter we maintain *multiple* shard counters. Updates are performed on one shard (chosen at random) and the sum over all shards is the overall counter value. While intuitive to grasp, a robust implementation of sharding is nevertheless a nontrivial task: On top of aggregated reads and distributed writes, sharding has to be safe under transactions. Custom-coding sharding requires a deep understanding of the underlying concurrency control mechanisms, is system-dependent, and introduces additional complexity to the application code.

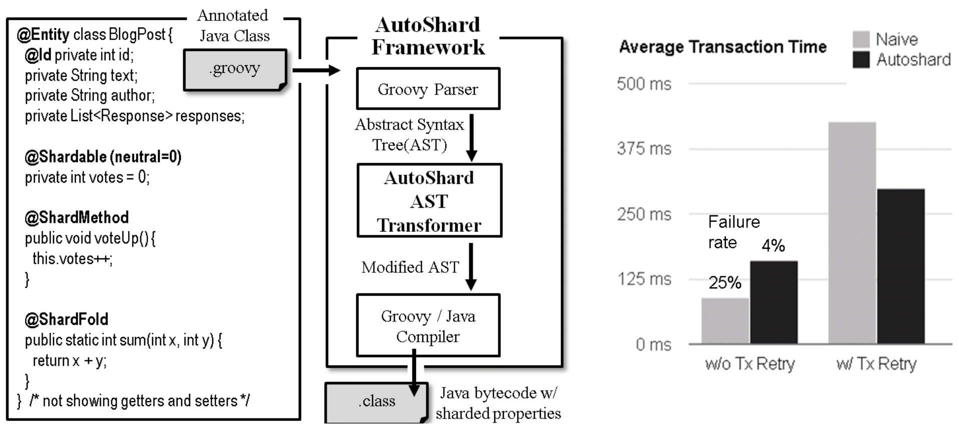
In this demonstration, we present *AutoShard*, a Java object mapper designed for NoSQL document stores of the Google Cloud Datastore [San12] family. AutoShard relieves the application developer from handling low-level sharding, and thus restores a clearer separation between logical and physical database design. We have shown in [ST14] that AutoShard significantly improves throughput and execution times of writes against hot spot data objects. AutoShard is easy-to-use, since it merely requires declarative annotations in Java classes and relies on self-modifying code to handle sharding transparently.

2 The AutoShard Object Mapper

The AutoShard object mapper is, to our knowledge, the first Java object mapper capable of sharding data automatically, based on simple annotations. Like other NoSQL object mappers, AutoShard takes care of the mundane marshalling of persisted entities into Java objects and back, thus greatly simplifying application development. In the following, we show how AutoShard addresses write contention. The Java class from Figure 1 (left) represents a blog post that can be voted up. As customary with object mappers, annotation `@Entity` specifies that an instance of class `BlogPost` is persisted as an entity. Annotation `@Id` marks the unique key.

The `votes` counter is a performance bottleneck when several users concurrently vote on the same blogpost. To solve this problem, we merely add the annotation `@Shardable` to declare that counter `votes` is to be sharded. 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 to decrement votes). Since in this example the shard method is incrementation, we specify zero as the neutral element (see `neutral=0`). This is exploited in initializing new shards. With annotation `@ShardFold`, we declare the static function `sum` 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.

The AutoShard approach relies on self-modifying code, by blending Java code with Groovy technology (*www.groovy.codehaus.org*). 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 1 (middle) shows the architecture of the AutoShard framework. A Java



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

  @Shardable (neutral=0)
  private int votes = 0;

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

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

Figure 1: Compilation of an annotated Java class (left) with the AutoShard framework (middle). Evaluation of property sharding with 2,0000 users incrementing 16 counters (right).

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 refer to [ST14] for the details of this compilation, and merely present the basic idea here.

For the sharded property `votes`, AutoShard's 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. This new implementation calls the original function both for the shard 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.

When a persisted `BlogPost` 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 (`sum` for class `BlogPost`) to aggregate over all shards. Second, the (internal) member `shard_votes` is initialized to the neutral element zero. 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.

When entity `BlogPost` is persisted after changes have been made, AutoShard first updates the main entity (if necessary). For the sharded class member, a random shard is loaded from storage. Its value is updated by invoking the `@ShardFold` method (`sum` for class `BlogPost`) on the loaded shard value and on `shard_votes`. The shard is persisted within a nested transaction, so we do not interfere with 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. The regular property `votes` still holds the current value.

3 Demonstration Description

In our demonstration we illustrate how AutoShard can effectively be employed in building scalable web applications that persist data in NoSQL data stores. Our scenario deals with a Java implementation of a voting tool. The application runs on Google App Engine and is backed by Google Cloud Datastore. We start with a naive implementation that does not take any precautions w.r.t. massively concurrent writes. We simulate an increasing number of users voting on blogposts. This causes write contention, and ultimately, failed web requests. We dynamically visualize the success rate as well as the average transaction time using Google Charts. For instance, Figure 1 (right) shows the effects of sharding counters in a voting app where 2,000 users concurrently update 16 counters.

We then add a few lines of annotations to the Java class declarations within Eclipse. Upon the push of a button, the Java code is re-compiled and deployed to Google App Engine. To the interested audience we can also go into the implementation details of the AutoShard AST Transformer. We repeat the experiment with the AutoShard-powered application. Now, the application handles web requests much more successfully: In the example from Figure 1, with the naive implementation, 25% of all transactions fail. Generating 16 shards with AutoShard, the failure rate can be reduced down to 4%.

In a second scenario, we add a simple retry mechanism and repeat failing transactions until they succeed. This simple approach obviously increases the chances of requests succeeding at the expense of higher execution times. We re-run our experiment with the naive and the sharded approach. Now, all requests succeed eventually, yet the sharded version shows superior transaction times (see Figure 1 (right)).

The audience may further investigate the details and differences between the naive and the sharded version by manipulating certain parameters such as the number of shards, or by employing alternative sharding strategies [ST14]. Again, AutoShard's easy-to-use annotation approach makes it convenient to modify the Java source code that is then compiled and deployed to Google App Engine. The audience will be able to assess the trade-offs between sharding strategy, success rate, and execution time under different workloads.

Acknowledgement: The authors would like to thank Werner Schier for contributing to the AutoShard code base as part of his Master thesis project at OTH Regensburg.

References

[ADE12] Divyakant Agrawal, Sudipto Das, and Amr El Abbadi. *Data Management in the Cloud – Challenges an Opportunities*. Synthesis Lectures on Data Management. 2012.

[San12] Dan Sanderson. *Programming Google App Engine*. Google Press, 2012.

[ST14] Stefanie Scherzinger and Andreas Thor. AutoShard - Declaratively Managing Hot Spot Data Objects in NoSQL Data Stores. In *Proc. WebDB'14*, 2014.

[TZK$^+$13] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, et al. Speedy Transactions in Multicore In-memory Databases. In *Proc. SOSP'13*, 2013.