Naïve, lock-free, client-side sharding with online addition of shards

Omkar J. Tilak *Alexandre Hardy*

Abstract

We describe a client-side sharding system which follows the naïve approach of hashing keys in documents to determine the shard that a document belongs to. Our database clients support a limited set of (NoSQL like) operations. We extend this simple approach by describing a scheme for rebalancing data between shards when new shards are added. The rebalancer is a client of each shard similar to the other clients in the system. We describe the algorithms required to present a consistent view to clients of the sharded database system during rebalancing. These algorithms permit a limited set of database operations to be executed in a lock-free fashion. Our clients do not rely on any notion of logical or global clocks to achieve a consistent partial event ordering.

1 Introduction

Sharding is a simple and naïve approach to scale out database capacity and performance. Although there are several databases that support sharding and hide the details from the client, it is useful to be able to implement a sharding system on the client side, for the following reasons (some of which are already provided by available databases):

- We reduce the complexity of the server.
- We can avoid a bottle neck of requiring all clients to communicate with a coordinating master.
- We have the potential to introduce heterogeneous sharded systems, which may be an attractive proposition for migrating data from one database system to another.

Sharding based on a hash of key fields of a document provides high performance and exceptional simplicity. We extend this approach to an environment where shards can be added dynamically with limited disruption, and where the clients continue to operate in spite of rebalancing of data on the shards.

2 Related Work

Since the advent of the distributed and cloud computing paradigms; scalable, distributed and fault tolerant data stores have been at the forefront of the database research community. A significant amount of research work has been done on designing scalable database systems for various types of workloads [2]. Various schemes of distributing and managing the data have been proposed for key-value stores, document stores and graph based data stores.

As a prominent feature of various distributed data stores, hash-based schemes have been widely used to distribute the data evenly among multiple nodes in the data cluster. [12] provides an efficient scheme for storing multi-dimensional data in distributed hash tables. Node join and leave operations are proposed to handle system churn and to maintain a certain load factor at all times. Such data distribution schemes can be thought of as complimentary to the data management algorithms proposed in this paper. Although we use a simple hashing scheme in our approach, various other data distribution schemes can be plugged into our algorithm and act as the data distribution layer. [17] uses a hierarchical distributed hash technique to store and efficiently query spatio-temporal data using a graph representation. However, such approaches are rather specialized in nature and focus on a particular data store platform.

Various database platforms store data in a key-value pair format. Such key-value based database platforms and data management tools have been used commercially by various organizations. Google, for example, uses Spanner [9] and MegaStore [3] as its data distribution and retrieval platform. Spanner supports distributed transactions and provides a globally consistent notion of time. However, to achieve this, it uses specialized hardware by equipping database servers with GPS and atomic clocks for time synchronization. On the other hand, our client-side data management platform can be deployed on commodity hardware thus eliminating the cost of expensive hardware. We simulate global time based on the ordering among various operations and thus maintain a consistent view that gets offered to multiple, distributed clients. Amazon uses Dynamo [11] as the platform to power its e-commerce applications. Dynamo makes use of well-known distributed database management constructs (such as data partitioning based on geographic locations, object versioning to attain consistency) to provide a consistent view to the application. PNUTS [8] is a geographically distributed and centrally managed database system that powers Yahoo's massive web application suite. Data is either hashed or ordered for quick retrieval. PNUTS makes use of a message bus to distribute operational data to the database servers. PNUTS offers consistency guarantees on a per-record basis that are transactional in nature but not completely serializable. In [10], authors propose an extension to the basic key-value paradigm by allowing applications to combine multiple keys to form key-groups. The transactional semantics are provided on a per-keygroup basis. This approach uses a traditional key-value store as the underlying platform. However, a single node in the cluster gains ownership of all the keys in a key group. This makes the scheme susceptible to failures.

There are various open source distributed database solutions available which offer varied paradigms (SQL, NoSQL, key-value, document store, object store) of data store [6]. MongoDB [7, 4] is an open-source document database which supports data replication for high-availability. Data distribution is supported in the form of shards. MongoDB shards (divides) data among multiple machines based on range or based on the hash value of the shard key. The data is automatically moved between shards based on load, access pattern and data churn caused by inserts, updates and deletes. However, MongoDB sharding is done entirely on the server side. Our approach offloads that responsibility to the client and allows servers to only worry about data replication. Cassandra [15] supports partitioning and replication. Failure detection and recovery are done automatically by the cluster. However, Cassandra has a weaker concurrency model wherein replicas are updated asynchronously and replication factor is maintained by a read-repair approach.

The commodity storage solutions offered by various vendors have been leveraged to implement databases. In [5], authors explain one such approach. Here, a database service is implemented on top of Amazon's S3 service. Read, write, and commit protocols are presented which together constitute the database service. Such approaches can be considered as complimentary to the approach presented in this paper. Our data management and query algorithms can run on top a myriad of database services as long as they offer some basic database functionalities.

Significant work is also being done to extend the functionality of different languages so that it becomes easier to write code that deals with queries on distributed databases. For example, Java can be extended with distributed transactional constructs [14] so that it is easier to write queries on distributed datastore and access them using JDBC and MPI. Although such approaches do make writing correct distributed queries possible, our approach addresses the issue of correctness by defining algorithms so that the ordering of the operations is preserved even in a concurrent context.

Our approach differs from these solutions in a fundamental manner. These industry techniques push the data management and consistency logic to the servers whereas we put that logic in the clients. This allows us to keep the servers "dumb" and make use of commodity hardware to run our platform. Our approach does not require any messaging platform, except to establish some global state. No communication is required between clients to perform the standard set of database operations that our client library supports. This also increases the scalability and operability of the system as new clients can be launched without any extra setup required. Our approach also aims to be generic in nature and can be applied to various types of databases. Finally, we rely on the underlying database to provide ordering of operations so that the clients do not require any notion of a shared clock [16].

3 Overview

We make a number of assumptions in our system, which is consistent with our particular use of the database system. The algorithm is designed around these assumptions, and is thus not a completely general algorithm, but rather a useful subset of the typical operations executed against a database.

We assume that the only atomic operations available to clients are:

- Add (insert) a document into a collection, with notification of conflicting primary keys.
- Remove a named (by primary key) document from a collection.
- Update the entire contents of a named document (partial updates, and increment operations are not supported).

- Get (retrieve) a named document from a collection.

By atomic, we mean that

- the state of a document is well defined during any of these operations, corresponding to the state either before or after the operation.

- on completion of the concurrent execution of these operations on a document, the final state of the document coincides with the state achieved by serially executing some permutation of the concurrent operations.

We do not require chronological ordering of operations, except from the perspective of the client. That is, if the database client executes operations in a serial fashion, then the chronological ordering of those operations must be preserved. However if a database client executes a number of operations concurrently, then we do not guarantee any chronological ordering of the operations.

We also support queries over document collections where we guarantee:

- If a document has not been deleted, it will be returned as part of the results of the query (assuming it matches the query criteria).

- A document will not occur twice in the results of a query, if that document has not been updated during the query.

- Only queries that do not mutate documents are permitted. For mutations, the individual atomic operations listed above should be used.

Our priority in this case is not to miss any data, but allow some data to be duplicated in a result set (although in most cases we avoid these duplicates).

Note that the algorithm is lock-free, but this does not mean that the shards do not use locks to implement the atomic operations. Instead, it means that the client does not have to perform any locking to guarantee consistency of the data.

The algorithm we present has more overhead than traditional sharding, and thus we want to limit the time during which this algorithm is applied. To do so, each client maintains state which determines when this algorithm is going to be executed.

In the next section we describe how the database clients agree on the state of the system, and how transitions from one state to another is effected, followed by a discussion of our algorithms for lock free database operations during the rebalancing phase.

4 State Transitions

When a new shard is added, the system enters the rebalancing phase. The naïve sharding algorithm does not guarantee consistent results when we enter the rebalancing phase. For example, a sharding system based on the hash of the primary key of a document, will compute a new shard for every document. Assume that the rebalancer has not moved these documents yet. If a database client tries to retrieve a known document using the new algorithm, it may end up fetching the document from the new shard rather than the shard where the document is actually located, and will thus incorrectly infer that the document no longer exists.

We thus need a special algorithm to execute during rebalancing to ensure consistency. However, it is imperative that all database clients execute the same algorithm at any point in time to prevent mutations that can result in an inconsistent state. It is not acceptable for one client to assume there are 2 shards, while another client works with 4 shards, where 2 shards are common between the database clients.

To ensure agreement between clients, we store the current state in an independent metadata store. This metadata store is extremely lightweight, and is consulted infrequently. The metadata stored for the current state is:

- The number of shards before rebalancing was started (if rebalancing).
- The number of shards currently available for use.
- The current version of the database system – a monotonically increasing number which records the number of times the number of shards has changed.
- The state of all database clients (there is only one such state since all clients must agree on that state).
- Related to the above field: whether rebalancing is in progress.

Our state transition diagram is illustrated in figure 1. The states are as follows:

- normal – The normal naïve sharding algorithm (high performance) is executed in this state.
- enter_rebalance – A barrier state: join the enter_rebalance barrier. Database clients are notified of this state change through a notification mechanism, after the new database shards are confirmed to be available. No new operations may be performed until all known database clients have entered the barrier. The last database client to enter this barrier will trigger the transition to the rebalancing state. All clients will use the new database version number at this point.

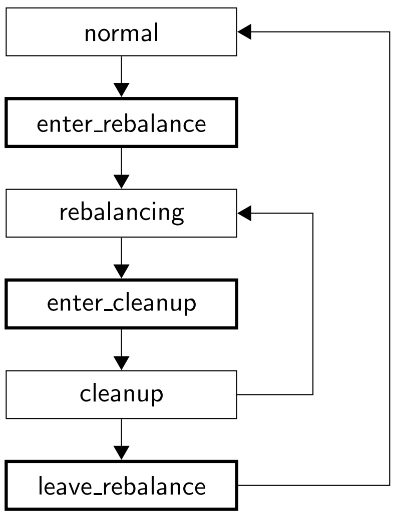

Figure 1: State transition diagram for database clients.

- rebalancing – The rebalancer process begins to execute in this state. When the rebalancer process has finished moving documents around, it will notify database clients of the transition to the enter_cleanup state. While in the rebalancing state, clients will execute a slower algorithm that ensures that the data in the database system remains consistent.

- enter_cleanup – A barrier state: join the enter_cleanup barrier. The last database client to join this barrier will trigger the transition to the cleanup state. No database operations are permitted in this state, the operations are queued until the database client enters a state where operations can be performed.

- cleanup – The rebalancer removes any remaining documents which are not necessary for the naïve sharding algorithm. There should be very few documents left which satisfy this criteria. Database clients are not permitted to perform operations and therefore queue any database requests. When the rebalancer has finished the cleanup, it notifies database clients of the change to the leave_rebalance state.

- leave_rebalance – A barrier state: join the leave_rebalance barrier. The last database client to join this barrier will trigger the transition to the normal state.

Barriers are used as the synchronization mechanism between database clients, to ensure that the same algorithm is executed by all clients. Before entering a barrier, the database client waits for all existing operations to complete, and subsequently joins the barrier. Once all known database clients have joined the barrier, each client can then proceed to process any queued database requests. The query operation is handled somewhat differently and is discussed in a later section. We have used ZooKeeper [13] to implement the barrier mechanism, where the number of participants in the barrier is determined by a strictly increasing counter stored in ZooKeeper.

State	**Algorithm**
normal	naïve
enter_rebalance	none
rebalancing	consistent
enter_cleanup	none
cleanup	none
leave_rebalance	none

Table 1: Algorithms executed in each of the states.

The algorithm executed in each one of these states is listed in table 1.

Next we introduce some notation for describing the consistent algorithm applied during rebalancing, and follow that with a collection of algorithms for common database operations which provide consistent results during rebalancing.

5 Notation

For simplicity in the following discussions, we ignore the possibility of multiple databases per shard, and multiple collections of documents per shard. Instead, we presume each shard is a single collection of documents which all share the same notion of a primary key. The algorithms are easily extended to collections and databases by considering each collection as a shard in the algorithms which follow. The notation we will use is:

- d – a document with a number of attributes.

- d_p – The primary key of the document d.

- S_i – The set of all documents on shard i. Each primary key may occur no more than once in each shard.

- $S_i(k_1 = v_1, k_2 = v_2, \ldots, k_n = v_n)$ – The set of documents in shard i with attributes $(k_1, k_2, \ldots, k_n) = (v_1, v_2, \ldots, v_n)$. Note that $d \in S_i(k_1 = v_1, k_2 = v_2, \ldots) \Rightarrow d \in S_i$. Although we focus on equality, this notation can be extended to include arbitrary criteria that can be evaluated on a single document.

- $d \in S_i$ – **True** if the primary key of d is already present in a document in S_i. **False** otherwise.

- $S_i \leftarrow S_i + \{d\}$ – Insert d into the shard S_i. The result of this action is DUPLICATE iff $d \in S_i$, or SUCCESS otherwise. This is an atomic operation.

- $S_i \leftarrow S_i - \{d\}$ – Delete the document with the same primary key as d from the shard S_i. The result of this action is NOTFOUND iff $d \notin S_i$, or SUCCESS otherwise. This is an atomic operation.

- $S_i(k_1 = v_1, k_2 = v_2, \ldots, k_n = v_n) \leftarrow d$ – Update the documents in shard i with attributes $(k_1, k_2, \ldots, k_n) = (v_1, v_2, \ldots, v_n)$ to have attributes as given by d. The attributes specified by $(k_1, k_2, \ldots, k_n)$ must include the primary key of the document, and the value of the primary key must be d_p. Each update is an atomic operation. The result of this action is NOTFOUND iff $d \notin S_i(k_1 = v_1, k_2 = v_2, \ldots, k_n = v_n)$, or SUCCESS otherwise.

- We define the upsert operation as: $S_i \leftarrow S_i \cup \{d\}$ where

$$S_i \cup d = \begin{cases} S_i + \{d\}, & d \notin S_i \\ (S_i - \{d\}) + \{d\}, & d \in S_i \end{cases}$$

  This is an atomic operation. The upsert operation inserts the document into the shard if it is not already present, or updates the document otherwise.

- We define the parameterized upsert operation as: $S_i \leftarrow S_i \underset{q}{\cup} \{d\}$ where

$$S_i \underset{q}{\cup} d = \begin{cases} S_i + \{d\}, & d \notin S_i(q) \\ (S_i - \{d\}) + \{d\}, & d \in S_i(q) \end{cases}$$

  and q is a tuple of the form $(k_1 = v_1, k_2 = v_2, \ldots, k_n = v_n)$. This is an atomic operation.

- result = action – Perform the specified action and store the result.

Every insertion or update sets the document version to the new version v_{new}.

6 Algorithms

In this section, we describe the individual operations supported by the database client, and how they are implemented to ensure consistency during the rebalancing phase.

The key issue to address is that documents may not map to the same shard after the number of shards have been increased. Thus any database operation may in fact extend across two shards: the shard the document originally resided on (the old shard) and the shard on which we expect the document to reside after rebalancing completes (the new shard). If the old shard is identical to the new shard for a given document, then no rebalancing is required (other than some minor updates mentioned later). In this case, the conventional naïve sharding algorithm can be executed. Thus we only consider the case where the old shard and the new shard differ.

It will be helpful to establish some invariants or rules to reduce the probability of mistakes in the algorithm. The invariants we have selected during rebalancing are as follows:

- Only the rebalancer may modify or remove documents from the old shard.

- Any database operations may only mutate documents on the new shard for that document.

- Any document migration must be performed in such a way that at any point in time the document:
  - Exists on the old shard, or
  - Exists on the new shard

Each document has two additional attributes:

- version: A number identifying the generation of the document, which is tied to the number of times the shard count has changed. All documents before rebalancing have a version associated with the previous number of shards, which we call the old version. Documents which have been moved to the new shard are tagged with a different version identifier, which we call the new version. Every mutation on the new shard must change the version attribute to the new version. In simple environments, the version number may simply be the number of shards, although this may cause problems if the number of shards are reduced.

- tombstone: As we will see later, deleting a document is a non-trivial matter, and thus we need a way to mark documents as being deleted. The tombstone attribute indicates whether the document has been deleted.

We now discuss each of the supported operations along with associated pitfalls.

6.1 Add

A document can only be added if no other document with the primary key already exists. Since an add operation is a mutation, we only apply mutations on the new shard for this document. To insert a new document we perform an atomic operation which:

- Inserts a new document on the new shard, unless a tombstone for this document exists.

Algorithm 1 Add

```
Require: A document $d$, old shard $S_{old}$ and new shard $S_{new}$, $S_{old} \neq S_{new}$
  procedure ADD($S_{old}$, $S_{new}$, $d$)
    if $d \in S_{new}$ then
      // Conditionally update the document
      $r = S_{new}(d_p, \text{🪦}(d) = \textbf{True}) \leftarrow d$
      return $r$
    else
      if $d \in S_{old}$ then
        return DUPLICATE
      else
        // Perform an upsert
        $r = S_{new} \leftarrow S_{new} \underset{\text{🪦}(d)=\textbf{True}}{\cup} \{d\}$
        return $r$
      end if
    end if
  end procedure
```

Algorithm 2 Update

```
Require: A document $d$, old shard $S_{old}$ and new shard $S_{new}$, $S_{old} \neq S_{new}$
  procedure UPDATE($S_{old}$, $S_{new}$, $d$)
    // Check if the document exists on the old shard
    if $d \in S_{old}$ then
      $r = S_{new} \leftarrow S_{new} \underset{\text{🪦}(d)=\textbf{False}}{\cup} \{d\}$
    else
      $r = S_{new}(d_p, \text{🪦}(d) = \textbf{False}) \leftarrow d$
    end if
    return $r$
  end procedure
```

- If a tombstone for this document exists, then update the tombstone to be this document (and no longer a tombstone).

We rely on the *upsert* operation on the shard to perform this action, and we also rely on a unique constraint on the primary key. If the unique constraint is violated, then the database client will be informed that the document already exists. In addition to this simplistic scenario, we need to take into account that the document may already exist on the old shard. So before proceeding with the above algorithm, we first determine if the document exists on the old shard. If so, then the add fails due to a duplicate key error. The complete algorithm is provided in algorithm 1.

Note that we only provide the algorithm for $S_{old} \neq S_{new}$. For $S_{old} = S_{new}$ we execute the standard naïve sharding algorithm.

Algorithm 3 Delete

```
Require: A document $d$, old shard $S_{old}$ and new shard $S_{new}$, $S_{old} \neq S_{new}$
  procedure DELETE($S_{old}$, $S_{new}$, $d$)
    // Check if the document exists on the old shard
    if $d \in S_{old}$ then
      $r = S_{new} \leftarrow S_{new} \underset{\text{🪦}(d)=\textbf{False}}{\cup} \{\text{🪦}(d)\}$
    else
      $r = S_{new}(d_p) \leftarrow \text{🪦}(d)$
    end if
    return $r$
  end procedure
```

6.2 Update

We utilize *upsert* and *update* operations on individual shards to perform a cross-shard update. A document can be updated only if it already exists on some shard. The update operation mutates the document. So to maintain the invariant, we only apply updates to the new shard for this document. The update operation is performed in the following manner:

- If the document is found on the old shard, then insert the updated document to the new shard (or update the document on the new shard if it is already present).
- Otherwise, if the document is found on the new shard, then update the document on the new shard. No insert is permitted if the document is not found on the old shard.

In both cases we add conditions that documents may only be updated if the document is not a tombstone, since deleted documents cannot be updated.

We rely on the fact that parameterized *upsert* to the new shard will fail if the document to be updated is a tombstone. Similarly, the update operation on the new shard will fail if the document does not already exist on the new shard. The complete algorithm is provided in algorithm 2.

6.3 Delete

A document can only be deleted if it exists on some shard. Since only the rebalancer can remove documents from a shard, we maintain this invariant by not physically deleting the documents during the delete operation. Instead, we only mark a document when it is deleted. The marking is done by inserting an additional field called *tombstone* into the document and setting it to **True** to indicate a valid tombstone. The rebalancer then takes care of physically deleting all the tombstones when it is done

Algorithm 4 Get

```
Require: A document primary key $d_p$, old shard $S_{old}$
    and new shard $S_{new}$, $S_{old} \neq S_{new}$
  procedure GETSHARD($S_i$, $d_p$)
      if $d_p \in S_i$ then
          if $S_i(d_p)$ is 🪦$(d)$ then
              return DELETED
          else
              return $S_i(d_p)$
          end if
      else
          return NOTFOUND
      end if
  end procedure
  procedure GET($S_{old}$, $S_{new}$, $d_p$)
      $r$ = GETSHARD($S_{new}$, $d_p$)
      if $r$ is DELETED then
          return NOTFOUND
      end if
      if $r$ is not NOTFOUND then
          return $r$
      end if
      $r$ = GETSHARD($S_{old}$, $d_p$)
      if $r$ is DELETED then
          return NOTFOUND
      end if
      if $r$ is not NOTFOUND then
          return $r$
      end if
      $r$ = GETSHARD($S_{new}$, $d_p$)
      if $r$ is DELETED then
          return NOTFOUND
      end if
      return $r$
  end procedure
```

Algorithm 5 Rebalance

```
  procedure COMPUTESHARD($d$, $n_{new}$)
      // This process will calculate the shard where the
      //  document should be stored
      // This can be done in a variety of ways
      // We chose to use a hash and mod approach
      // Here, h indicates the hash function
      $j = (h(d_p) \bmod n_{new}) + 1$
      return $S_j$
  end procedure
  procedure REBALANCE($v_{old}$, $v_{new}$, $n_{old}$, $n_{new}$)
      for all $S_i \in \mathbf{S}$ do
          for all $d \in S_i(version = v_{old})$ do
              $S_{new} \leftarrow$ COMPUTESHARD$(d, n_{new})$
              if $S_i = S_{new}$ then
                  $S_{new}(d_p)_v \leftarrow v_{new}$
              else
                  // Ignore any conflicts
                  // A conflict means the document has
                  //  already been rebalanced
                  $S_{new} \leftarrow S_{new} + \{d\}$
              end if
          end for
      end for
      $\mathbf{Q} \leftarrow$ all active queries
      $\mathbf{Q}' \leftarrow \mathbf{Q}$
      while $\mathbf{Q}' \cap \mathbf{Q} \neq \emptyset$ do
          $\mathbf{Q}' \leftarrow$ all active queries
      end while
      for all $S_i \in \mathbf{S}$ do
          $S_i \leftarrow S_i - S_i(version = v_{old})$
          $S_i \leftarrow S_i - S_i($🪦$ = \mathbf{True})$
      end for
  end procedure
```

moving the documents around. The delete operation is implemented as follows:

- If the document is found on the old shard, then insert a tombstone for this document on the new shard.
- Otherwise, if the document is found on the new shard, then update the document on the new shard and set *tombstone* to **True**.

The addition of a tombstone document is actually performed using an *upsert* operation. For the upsert we simply modify the existing document and insert a tombstone field in it and set it to value **True**. If the document already contains a tombstone field, then we simply set its value to **True**. The complete algorithm is provided in algorithm 3. Please note that 🪦 indicates the tombstoning operation where we insert a tombstone field into the document.

6.4 Get

A *get* operation does a lookup for the document based on a primary key. Since the rebalancer might be rebalancing the document that we are trying to lookup, the *get* operation has to do multiple lookups before declaring that a document does not exist. The lookup has to be done in following sequence to make sure if a document exists in the sharded database. First check if the document exists on new shard, then check if the document exists on old shard and then finally check if the document exists on the new shard again. If these three lookups fail to produce the required document, then we can declare that the document does not exist. The complete algorithm is provided in algorithm 4.

6.5 Rebalance

The *rebalance* operation is executed when a new shard is added to the system. When a new shard is added, the existing data that is residing on the old shards needs to be moved to appropriate shards based on the new number of shards in the system. For each shard in the system, the rebalancer basically reads all the documents from that shard that have the old version number. Then for each such document, it finds the new shard location for this document and then puts the document in its new location. After all the required documents from all the shards are moved to their appropriate locations, all the documents belonging to the older version are removed. Also, all the documents that have a valid tombstone flag are removed.

If the rebalancer finds any document that does not need to be removed, then the rebalancer simply changes the version attribute of that document to the new version.

The complete algorithm is provided in algorithm 5.

There is one peculiarity we need to mention. Since queries are often implemented with cursors, it is possible that cursors iterating through the shards may miss a rebalance operation and omit a document. The rebalancer must thus wait for all existing queries to terminate before removing the old version of documents.

6.6 Query

The *query* operation needs to execute the query on all the shards in the system and then combine (merge) the results obtained from individual shards to produce the final result. Since the *rebalance* operation might be moving documents around while query is performed, the merging of results needs to be performed by taking this factor into consideration.

The *merge* process works as follows. First, the query is executed on individual shards and the results of the operation are stored in individual queues. Then, while there are still documents left in some queue, we first read the top document from each shard queue. Then we find the minimum document(s) based on the value of the primary key. Then among this set of minimum documents we find the document that has the highest version number. If this document does not have a valid tombstone flag, then this document is added to the final result set. Then from all individual shard result queues, we delete the document(s) whose primary key matches with that of the document that was just added to the final result queue. The complete algorithm is provided in algorithm 6.

Algorithm 6 Query

```
procedure ADDSORTCRITERIA($q, m$)
    // Add the sort key m to the existing sort keys
    //  in the query q
    // Assume query has sort keys $\{k_1, k_2, \ldots, k_n\}$
    //  in that order
    // Then the new sort key order will be
    // $\{k_1, k_2, \ldots, k_n, m\}$
     $r = \{\text{EXISTINGSORTKEYS}(q) + m\}$
    return $r$
end procedure
procedure QUERY($q$)
    // SR stores the query results obtained from individual shards
    $SR \leftarrow \{\}$
    // R stores the final result set of the query
    $R \leftarrow \{\}$
    $q' \leftarrow \text{ADDSORTCRITERIA}(q, d_p)$
    for all $S_i \in \mathbf{S}$ do
        $SR \leftarrow SR + S_i(q')$
    end for
    while $\sum_i |SR_i| > 0$ do
        $d \leftarrow \min_{d_p}\{\top(SR_i)\}$
        $d \leftarrow \max_{d_v}\{\top(SR_i) | \top(SR_{i_p}) = d_p\}$
        if $d$ is not 👻$(d)$ then
            $R \leftarrow R + d$
        end if
        for all $SR_i \in SR$ do
            $SR_i \leftarrow SR_i - \{d\}$
        end for
    end while
    return $R$
end procedure
```

7 Validation

We first consider every sharding mutation operation. All our mutation operations consist of exactly two database calls:

- A query (fetch of a single document) from the old shard.
- An atomic single shard mutation operation on the new shard.

Since all single shard mutations on a shard are atomic, all combinations of mutations will be serialized in some way. So we are guaranteed that single shard mutation operations cannot affect the correctness of the algorithm in the sense that individual mutations cannot interleave in such a way that the outcome of any sharding mutation is unpredictable. All that remains is to validate that the algorithms do indeed perform the actions we require.

7.1 Fault Tolerance

The rebalancer algorithm ensures that even if the rebalancing process fails at any point in its execution, it leaves the database in a consistent state. The rebalancer can be resumed on any node at any time in the future with no special provisions. Since the rebalancer never violates the invariants listed in section 6, all the other algorithms will function consistently and correctly in the presence of a rebalancer failure.

7.2 Testing Framework

Reasoning about the algorithms helps to validate their correctness, but it is desirable to have some stronger guarantee such as a mathematical proof of correctness. We have favored a different approach. To validate the algorithms, we implemented a simulator which is able to determine the order in which atomic actions on a shard occur. We generate a number of test scenarios for every permutation of interactions between two concurrent requests. These tests are executed, and the final state after the operations have completed is evaluated to see that the outcome is consistent, and that some ordering of the operations has been applied. The same algorithm can be applied to more than two operations, but the time consumed by these tests begins to outweigh the gains at more than three simultaneous operations. Our expectation is that three operations will behave correctly if two operations behave correctly, since any two operations can be ordered, with the third applied after that (since we have verified that mutation operations serialize).

We have executed our test framework to convince ourselves that the implementation is correct. We perform each test for a number of scenarios:

- Document does not exist on the old shard.
- Document does exist on the old shard.
- Tombstone for the document exists on the new shard.
- Document has been rebalanced and exists on the old shard and new shard.
- Document has been rebalanced and exists on only the new shard.

The test framework confirms the correctness of our algorithms.

8 Results

We computed a number of tests to determine the overhead, in real terms, of the algorithms proposed in this paper. These tests were performed on a cluster with 3 shards, with three replicas in each shard. The 9 compute nodes hosting the database were all running on virtual machines on a cloud service, so a possibility exists that some virtual machines may have contended for disk access with other virtual machines. However, the dataset was small enough that the database server would be able to aggressively cache the dataset. MongoDB was used as the underlying database for each shard. The “normal” operations were executed with a conventional sharding algorithm with 3 shards, whereas the “rebalancing” operations apply the algorithms listed in this article (moving from 2 shards to 3 shards). No rebalancer was running during either of these tests. We have enumerated a number of different cases, including failure cases, to attempt to discern the impact of performing multiple database operations. Since shards provide the most benefit for concurrent operations, we also include a separate measure of 20 concurrent operations, whereas the default case performs all operations serially. Each test run consists of 100 operations of the specified type. We repeated the full battery of tests 100 times to reduce the effect of outliers and report the average time in each case. We selected python for the client side implementation of the algorithms specified in this paper, as well as the “normal” sharded client. We have tuned the python client to eliminate the effect of garbage collection in the results, to obtain a test measure free of effects unrelated to the test (or as near as we could get).

Table 2 presents the overhead involved in executing

Operation	Normal	Rebalancing	%
Add	3.88	3.93	1%
Add (duplicate)	3.66	3.90	6%
Delete	7.54	7.55	0%
Delete (missing)	7.76	8.49	9%
Get	6.93	6.92	0%
Get (missing)	7.83	7.75	0%
Query1	0.16	0.14	0%
Query2	0.07	0.07	0%
Update	7.60	7.78	2%
Update (missing)	7.96	8.22	3%

Table 2: Overhead of executing operations during rebalancing.

Operation	Normal	Rebalancing	%
Add	2.74	2.86	4%
Add (duplicate)	2.76	2.82	2%
Delete	5.43	5.49	1%
Delete (missing)	6.51	6.70	3%
Get	5.48	5.42	0%
Get (missing)	6.30	6.49	3%
Update	5.56	5.56	0%
Update (missing)	6.57	6.82	4%

Table 3: Overhead of executing operations during rebalancing (parallel).

the basic operations while rebalancing is in progress. Table 3 lists the same operations where 20 operations are executed concurrently. The first two columns report the time, in seconds for 100 operations to be performed. The last column indicates the overhead of executing the rebalancing algorithm, based on these measurements. The two queries listed in the table were only executed once per test run, and covered a subset of the data loaded during the test. In this test run, the overhead of applying the algorithms in this paper were minimal. Thus the other operations in the system dominated any overhead introduced by our sharding layer. In some cases the proposed rebalancing algorithms outperformed the naïve sharding approach, which we ascribe to some variance in the test environment.

9 Conclusion

We have described a client side implementation of sharding, with the ability to add shards (or remove shards) without taking the sharded database offline. The limited set of operations we support are typical of many NoSQL and document oriented databases, which means this approach is applicable to a wide variety of available database systems which have not necessarily been designed to scale out in this way. Database operations are ordered based on the destination shard for each operation, so ordering is achieved with no global clocks. Each shard is responsible for ordering the operations arriving at that shard, and the ordering of operations from a client perspective is easily described in terms of the ordering of operations at each shard.

10 Acknowledgments

The tombstone image was obtained from openclipart.org [1].

References

[1] `http://openclipart.org/detail/171351/rip-by-cyberscooty-171351`. Accessed: 2013-09-24.

[2] Divyakant Agrawal, Amr El Abbadi, Beng Chin Ooi, Sudipto Das, and Aaron J. Elmore. The evolving landscape of data management in the cloud. *Int. J. Comput. Sci. Eng.*, 7(1):2–16, March 2012.

[3] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of the Conference on Innovative Data system Research (CIDR)*, pages 223–234, 2011.

[4] Kyle Banker. *MongoDB in Action*. Manning Publications Co., Greenwich, CT, USA, 2011.

[5] Matthias Brantner, Daniela Florescu, David Graf, Donald Kossmann, and Tim Kraska. Building a database on S3. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 251–264, New York, NY, USA, 2008. ACM.

[6] Rick Cattell. Scalable SQL and NoSQL data stores. *SIGMOD Rec.*, 39(4):12–27, May 2011.

[7] Kristina Chodorow and Michael Dirolf. *MongoDB: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2010.

[8] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, August 2008.

[9] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst.*, 31(3):8:1–8:22, August 2013.

[10] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. G-Store: a scalable data store for transactional multi key access in the cloud. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 163–174, New York, NY, USA, 2010. ACM.

[11] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, October 2007.

[12] George Giakkoupis and Vassos Hadzilacos. A scheme for load balancing in heterogenous distributed hash tables. In *Proceedings of the twenty-fourth annual ACM symposium on Principles of*

distributed computing, PODC '05, pages 302–311, New York, NY, USA, 2005. ACM.

[13] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX annual technical conference*, pages 145–158, Berkeley, CA, USA, 2010. USENIX Association.

[14] Kazuaki Ishizaki, Ken Mizuno, Toshio Suganuma, Daniel Silva, Akira Koseki, Hideaki Komatsu, Yohei Ueda, and Toshio Nakatani. Parallel programming framework for large batch transaction processing on scale-out systems. In *Proceedings of the 3rd Annual Haifa Experimental Systems Conference*, SYSTOR '10, pages 15:1–15:14, New York, NY, USA, 2010. ACM.

[15] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[16] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[17] Matthew Malensek, Sangmi Pallickara, and Shrideep Pallickara. Autonomously improving query evaluations over multidimensional data in distributed hash tables. In *Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference*, CAC '13, pages 15:1–15:10, New York, NY, USA, 2013. ACM.