Efficient Node Bootstrapping for Decentralised Shared-Nothing Key-Value Stores

Han Li and Srikumar Venugopal

The University of New South Wales
Sydney, Australia
{hli,srikumarv}@cse.unsw.edu.au

Abstract. Distributed key-value stores (KVSs) have become an important component for data management in cloud applications. Since resources can be provisioned on demand in the cloud, there is a need for efficient node bootstrapping and decommissioning, i.e. to incorporate or eliminate the provisioned resources as a members of the KVS. It requires the data be handed over and the load be shifted across the nodes quickly. However, the data partitioning schemes in the current-state shared nothing KVSs are not efficient in quick bootstrapping. In this paper, we have designed a middleware layer that provides a decentralised scheme of auto-sharding with a two-phase bootstrapping. We experimentally demonstrate that our scheme reduces bootstrap time and improves load-balancing thereby increasing scalability of the KVS.

Keywords: Cloud computing, Key-value Stores, Elasticity, Performance.

1 Introduction

Distributed key-value stores (KVSs) [3,5,10] have become a standard component for many web services and applications due to their inherent scalability, reliability and data availability, even in the face of hardware failures. While KVSs have been mostly used in data centres, many enterprises are now adopting them for use on servers leased from Infrastructure-as-a-Service (IaaS) cloud.

IaaS providers offer compute resources in the form of virtual machines (VMs), which can be provisioned or de-provisioned anytime on-demand. To deal with increasing workload, new VMs are acquired to improve the system's capacity (i.e. scale up). Since IaaS providers normally follow the "pay-as-you-go" pricing model, redundant VMs can be shut down in the face of declining demand (i.e. scale down) to save on economic costs . In this paper, the process of incorporating a new empty VM as a member of KVS is termed as node *bootstrapping*. In contrast, the process of eliminating an existing member with redundant data off the KVS is called node *decommissioning*.

The storage model of a KVS determines its performance of data movement during node bootstrapping and decommissioning. In shared storage KVSs, the persistent data is stored in the underlying networked attached storage or distributed file system (DFS). The data can be migrated between nodes without

D. Eyers and K. Schwan (Eds.): Middleware 2013, LNCS 8275, pp. 348–367, 2013.
© IFIP International Federation for Information Processing 2013

actual data transfer, simply by exchanging the metadata (e.g., identifiers or ownership) of data blocks in the shared storage [9]. In contrast, shared-nothing KVSs consist of distributed nodes, each with their own separate storage, coordinated as a distributed hash table (DHT). When a new node joins the system, it has to obtain data from its peers. This process is slow in the case of KVS with large data volume. Thus, it is non-trivial to bootstrap or decommission a node *quickly* and *frictionlessly*, i.e. without affecting the online query processing.

The challenge of node bootstrapping in the shared-nothing KVSs lies in redistributing the data when a new node is added. Specifically, it requires a mechanism that *partitions* the key space of a database and then *re-allocates* the partition replicas during node bootstrapping. Moreover, most shared-nothing KVSs [10,15] are essentially DHTs, deployed in a completely decentralised architecture (i.e. peer-to-peer, or P2P). There is a need for decentralised coordination between the peers to execute data partitioning.

This paper aims at improving the efficiency of node bootstrapping for decentralised shared-nothing KVSs. The goal of efficiency is three-fold. First, the side-effect of data movement (against front-end query processing) should be minimised. Second, data consistency and availability should be maintained during bootstrapping. Third, the load in terms of both data volume and workload that each node undertakes, should be re-balanced after bootstrapping. Node decommissioning is also discussed, but it is applied with caution to avoid data loss.

In this paper, we describe the design of a middleware layer that provides a decentralised scheme of data partitioning and placement to improve the efficiency of node bootstrapping. The main contribution of this paper is a decentralised auto-sharding scheme, extending from the concept of "virtual node" [19], that consolidates each partition of data into single transferable replicas to eliminate the overhead of migrating individual key-value pairs. Through sharding, the data volume of each partition replica is confined into a bounded range.

We also discuss a related placement algorithm, that evenly re-allocates the partition replicas when a node is bootstrapped and decommissioned, with the objectives of: i) rebalancing the volume of data; ii) maintaining high data availability; and iii) minimising data movement at startup for quick bootstrapping. We have also implemented a token ownership mechanism to provide eventual consistency when a replica is migrated between nodes.

We have implemented these partitioning and placement mechanisms on top of Apache Cassandra, an open source KVS, to build *ElasCass*. We present experimental evaluations, carried out using public IaaS cloud, that demonstrate that our proposed scheme of data partitioning and placement reduces the time to bootstrap nodes, distributes data and workload more evenly among the nodes, and improves throughput of the KVS.

The rest of this paper is structured as follows. In the next section, we discuss the state-of-the-art in node bootstrapping and replica placement in distributed databases. The system design is presented in Section 3. The data consistency issue is discussed in Section 4. We present the experimental evaluations in Section 5. Finally, we conclude in Section 6.

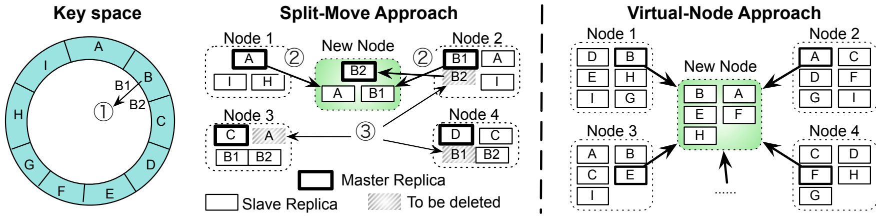


Fig. 1. A node joins the key-value store

2 Background and Related Work

The bootstrap process for a KVS executing on IaaS begins with provisioning a VM as a node and starting a KVS process. The next step is for the node to acquire a list of key ranges from existing nodes. Finally, the node acquires the data belonging to the key ranges. At this point, the node is ready to serve queries. We denote the time between the start of the KVS process and the point when the node is ready to serve queries as the *bootstrap time*. The efficiency of bootstrap is determined by the acquisition of the key ranges and the associated data. This is determined by the data management - partitioning and placement - strategies, the state-of-the-art in which is discussed in the following sections.

2.1 Partitioning in Key-Value Stores

Figure 1 illustrates several approaches for migrating the data during node bootstrapping, described as follows.

***Split-Move* Approach**. This approach is commonly used in distributed hash tables (DHTs), and was adopted by Cassandra [15]. Typically, consistent hashing [13] is used, as it introduces minimal disruption when a hash table (e.g. a key range or a partition) is resized during node bootstrapping. The key space is split into a list of consecutive key ranges, each assigned to one node. Thereby, each node maintains one master replica for its own range, and also stores the slave replicas of several other key ranges for high availability. When a new node joins the KVS, the key space of the database is re-partitioned. One or several existing partitions are *split* into two sets of data (e.g. $B1$ and $B2$ as in Figure 1). One is retained in the existing nodes. The other set of data is *moved* pair-by-pair and reassembled at the new node.

There are multiple drawbacks to this approach. One is the overhead of moving individual key-value pairs. When a partition is split, the node contributing the subset has to scan its entire dataset to prepare a list of key-value pairs for the new node, which, on receiving the data, has to reassemble the key-value pairs into files. Both scanning and reassembling are heavyweight operations.

The other drawback is that, consistent hashing aims at remapping a minimised number of keys when the number of nodes changes. As a result, only a limited number of nodes can participate in bootstrapping, each undertaking relatively heavy workload. According to Amazon [10], this bootstrapping approach

is highly resource intensive, and is only suitable to run at a lower priority. However, low priority results in significantly slow bootstrapping, which adapts less quickly to dynamic load.

***Virtual-Node* Approach**. A virtual node is a consolidated data partition that is transferable as a single unit. The idea of "virtual node" was introduced in Chord [19] and other consistent hashing systems, upon which KVSs such as Dynamo [10], Voldemort [1] and Cassandra [15] are based. Other KVSs, such as BigTable [3] and PNUTS [5], use the term "tablet" instead.

This approach avoids the overhead of scanning and reassembling as in the **split-move** approach. In practice, the key space is over-partitioned, such that the number of virtual nodes is made much greater than the data nodes. Each data node is assigned many virtual nodes. Hence, a new node can be bootstrapped by multiple existing nodes, each offering one or several virtual nodes. Thereby, each participating node shares a relatively small amount of workload in bootstrapping.

However, there is a lack of efficient data partitioning schemes for completely decentralised KVSs. The current-state research efforts [10,15] use a *simplified* partitioning strategy, wherein the key space is split into static key ranges of equal length, or hashed into buckets with equal capacity. Although this strategy avoids complex coordination amongst the peer nodes, it leads to data skew for biased key distributions. Data skew results in some "giant" partitions that are difficult to migrate because of the large volume of data [5].

One refinement is to re-hash the inserted keys using uniform hash functions, most of which, however, are not order-preserving, making the support of range queries more difficult. For those uniform order-preserving hash functions, there is a fundamental limitation: the key space is discrete and cannot adapt to any arbitrary application key distributions [1]. Alternatively, PNUTS [5] proposed to shard the tablets (i.e. partitions) into bounded sizes. However, it relies on a centralised component that limits the efficiency of partitioning in the KVS.

***Metadata-Only* Approach**. This approach is used by shared storage KVSs such as BigTable [3], Spanner [7] and HBase [2]. The persistent data is not stored in the nodes of KVSs, but in underlying distributed file systems such as GFS [11] or HDFS [18]. For this storage model, Das et al. [9] proposed that data can be migrated by exchanging only the **metadata** (i.e. *identifiers* or ownership) of data blocks between the nodes of database systems (or KVSs), while the persistent data remains unmoved in the shared storage. A centralised controller is also used for metadata management. Although this approach minimises the cost of data migration, it is not applicable to decentralised shared-nothing KVSs.

2.2 Data Placement

The data placement problem has been extensively studied in literature. The common approach in state-of-the-art KVSs to data placement is to manage the data through coarse grain structures such as buckets or virtual nodes rather than identifying an optimal placement strategy at the granularity of single data items [14].

[1] Voldemort: `http://www.project-voldemort.com/voldemort`
[2] Apache HBase: `http://hbase.apache.org`

Consistent hashing-based KVSs [10,15] have typically adopted a random placement strategy, in which a random hash function is used to assign groups of data items (i.e. buckets or virtual nodes) to nodes. This allows key lookups to be performed locally, in a very efficient manner [10]. Other KVSs [3,7,5] rely on dedicated directory services that provide flexible mapping from virtual nodes to physical nodes. Essentially, this approach also uses random placement strategy. The advantages of this strategy are simplicity and the effectiveness of load-balancing [10,3].

There is also extensive work of finding optimal data placement strategies. Ursa [21] and Schism [8] rely on centralised components to compute the placement and to maintain a location map, which is not applicable to our system. Others research efforts [16,22] have proposed distributed replica placement algorithms. However, these efforts only consider the placement of read-only replicas, while we discuss the ownership management of virtual nodes to support both read and write operations.

In this paper, we extend the **virtual-node** approach to confine each transferable partition into a bounded size, that is enforced by auto-sharding, to avoid data skew. The novelty lies in executing auto-sharding using decentralised coordination. We also describe a random placement scheme to compliment the auto-sharding mechanism, in order to achieve fast bootstrap time and load-balancing.

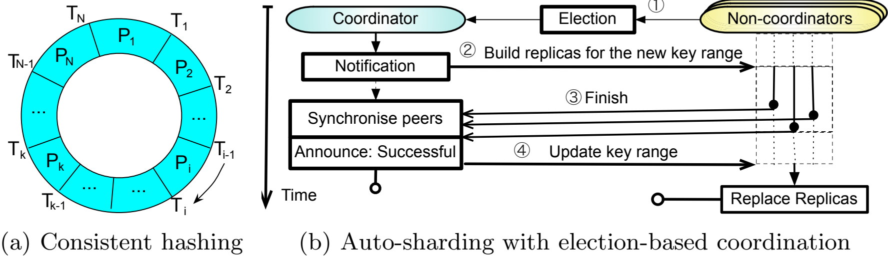
3 Design

Our system follows the typical decentralised shared-nothing architecture. The key space of a database is hashed into multiple partitions. Each partition, denoted as P_i, is replicated to multiple nodes for high availability. The data is stored in a separate persistent storage volume, each attached to an individual node. Each node (denoted as n_i) serves many partitions for load balancing purposes. The nodes are organised as peer-to-peer (P2P), similar to DHTs [17,19]. Each node maintains enough routing information locally so as to route a request to the appropriate node directly, i.e. in *0-hop* [12]. Thus, clients can connect to any node for query execution.

We have designed a middleware layer that sits between the key space of a database and the storage of nodes. This middleware was implemented on a KVS that already has gossip-based [20] membership protocol and failure detection, hinted handoff [10] to handle node failures, and timestamp based reconciliation to ensure eventual consistency. This section describes the synthesis of decentralised auto-sharding and replica placement algorithms in this layer that improves the efficiency of node bootstrapping.

3.1 Data Partitioning for Building Transferable Replicas

Our partitioning algorithm builds on consistent hashing [13], in which the largest hash value is wrapped around to the smallest to form a ring of key space.



(a) Consistent hashing (b) Auto-sharding with election-based coordination

Fig. 2. Data Partitioning

As shown in Figure 2(a), when a database is created, a number of tokens, $\{T_i : 0 < i \leq N\}$, are generated to segment the key space of the database into N consecutive equal-size key ranges, where N is configurable by the KVS administrators. Each key range defines one partition of data. Therefore, each partition P_i can be associated with the token T_i, which defines the upper bound of P_i. The lower bound is determined by the predecessor T_{i-1}.

Sharding Operations. The aim of auto-sharding is to confine the actual volume of data in each partition. Building on the **virtual-node** approach described in Section 2, we propose to shard the partitions online to address the problem of data skew. Let $Size(P_i)$ be the data volume of partition P_i. The maximum size Θ_{max} and the minimum size Θ_{min} are defined as in Equation 1 and Equation 2, respectively.

$$\forall i \in [1, N], Size(P_i) \leq \Theta_{max} \tag{1}$$

$$\forall i \in [1, N-1], Size(P_i) + Size(P_{i+1}) \geq \Theta_{min} \tag{2}$$

The partition P_i is *split* when $Size(P_i)$ exceeds Θ_{max}. A new token T_{new} is inserted between T_{i-1} and T_i, such that the resulting sub-ranges $(T_{i-1}, T_{new}]$ and $(T_{new}, T_i]$ contain roughly equal volumes of data. In contrast, two adjacent partitions (e.g. P_i and P_{i+1}) are *merge*d, if their total size is below Θ_{min}. To merge P_i and P_{i+1}, the token T_i that sets the boundary of these two partitions is removed. Thus, the merged key range is $(T_{i-1}, T_{i+1}]$.

The challenge of sharding, either split or merge, lies in the consolidation of each partition replica as a single transferable unit for better performance of data migration. To consolidate a replica, key-value pairs belonging to different partitions, are stored in separated files. To execute a sharding, new replicas (i.e. data files) are created, to store *every* (and *only*) key-value pair belonging to the sharded partition. Hence, *rebuilding replicas* of the affected partition is the key operation in sharding.

In addition, we discuss the need for merging partitions. In practice, there is less harm in retaining “sparse” partitions that contain small volume of data, rather than merging them aggressively, since a small-sized partition replica is easy to move. Nevertheless, in order to reduce the number of sparse partitions for better performance of query processing, we attempt to merge partitions when

applicable. The extra conditions for the *merge* operation are as follows. Firstly, if two adjacent partitions are not stored on the same *set* of nodes, then they are not merged. Secondly, we try to maintain a minimum number of partitions in each key space. If the actual number of partitions is no greater than the predefined value N, then the merge operation will not be triggered. Lastly, to avoid oscillation of split and merge, we set $\Theta_{max} \geq 2\Theta_{min}$. Therefore, the size of a newly-merged partition is not greater than $\Theta_{max}/2$, which is not big enough to trigger a split. Also, each sub-partition of a newly split partition is not less than Θ_{min}, which is not small enough to trigger a merge.

Coordinating Sharding. The key operation of sharding is *rebuilding replicas*. However, since each partition is replicated to multiple nodes, the operation of rebuilding each local replica is executed by different nodes asynchronously. Thus, coordination is required to ensure the consistency of the key space and persistent data across the nodes that participate in sharding. As shown in Figure 2(b), sharding in a distributed KVS is coordinated in four steps:

Step 1: Election. When the data volume of a partition replica reaches the boundary (Θ_{max} or Θ_{min}), the node that serves the partition initiates the sharding. A coordinator is elected with a distributed consensus policy. In this paper, we have leveraged the Chubby implementation [2] for electing the coordinator. According to Chubby, the coordinator must obtain votes from a majority of the participating nodes, plus promises that those participating nodes will not elect a different coordinator for a time interval known as the *master lease*, which is periodically renewed. In our implementation, the node that initiates the sharding retrieves the complete list of nodes that store the partition. The list is sorted by certain criteria, and the node on top is voted as the *coordinator* (for this single operation only). The other participating nodes also vote for the node on top of the list. Since every node maintains the complete partition-node mapping locally, the sorted list is unique. The node on top wins a majority of the vote.

Step 2: Notification. There is a prerequisite before launching the sharding. In the case of split, the coordinator calculates the splitting token T_{new} that will be used by all the participating nodes. In the case of merge, the coordinator examines whether the extra conditions for merging are satisfied. Once the prerequisite is met, the coordinator notifies that a sharding should be launched. Then, all the participating nodes start to shard their own replica simultaneously.

Step 3: Synchronisation. The operation of *rebuilding replicas* is executed and completed asynchronously by different nodes. When a node finishes, it notifies the coordinator and then waits for further announcement. The coordinator synchronises this operation until all the participating nodes have finished.

Step 4: Announcement. Once the coordinator has received the notification of *Finish* from all the participating nodes, it announces globally that the key range of the affected partition should be updated to the new range. On receiving this final announcement, every node in the KVS updates the query routing information, and each participating node replaces the old replicas with the newly-built replicas asynchronously. In this way, a sharding operation is completed.

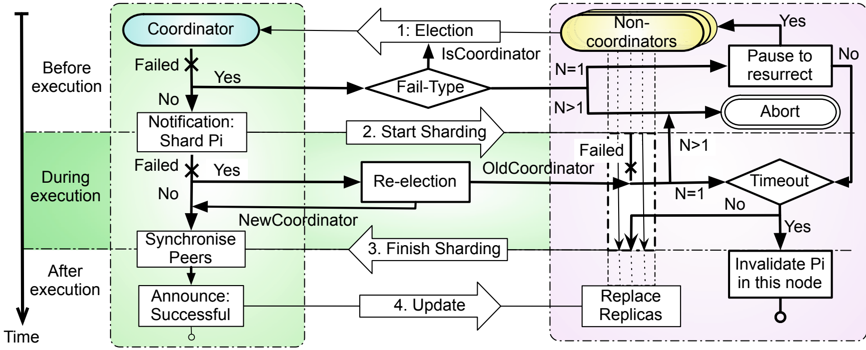


Fig. 3. Failure recovery in the election-based coordination

Failover During Sharding. Based on the four-step coordination as described, we discuss how to handle node failures during a sharding operation. Figure 3, extended from Figure 2(b), depicts the failure recovery.

The failure detection in our system is gossip-based [20]. We assume detection error exists, since a failure detector is not always completely accurate. A detection error, when caused by message loss, is *false-positive*, in which case a node is not dead, but detected as dead. In contrast, in a *false-negative* detection error, due to the delay in detection, a node is actually dead but considered as still alive. In our design, the communication between a coordinator and a non-coordinator follows the typical handshake policy. Hence, the false-negative error can be easily corrected. Therefore, we focus on addressing the false-positive detection error.

This failover scheme focuses on two scenarios: i) if only one participating node fails during the process, the sharding can succeed with or without the failed node's resurrection; ii) if more than one participating node fail, the sharding can be aborted and rolled back without data loss. We discuss the failover when a participating node is detected as failed (by gossip) *before*, *during*, or *after* the execution of rebuilding replicas of P_i. In the following, T_{pause} is defined as a time period that is longer than twice the end-to-end gossip broadcast delay.

Before the execution, if a participating node is detected as failed, the sharding procedure is paused for T_{pause}, to await whether the failed node can resurrect (e.g., due to false detection). If it is the coordinator node that fails, a different coordinator should be elected following the pause, even when the previous coordinator resurrects. After the pause, the sharding continues if at most one node fails, or is aborted (by any participating node) if there are more than one node failures. Nodes resurrecting after T_{pause} can no longer serve P_i.

During the execution, every participating node maintains the complete list of nodes that are sharding P_i. Whenever more than one participating nodes are detected as failed, and remain dead for a period of T_{pause}, any participating node that detects this event can abort the sharding via broadcast. Otherwise, if only one node (even the coordinator) fails, every other participating node continues the operation of rebuilding replicas regardlessly.

After all the living nodes have finished the execution, if there is one failed node, the sharding procedure is paused for T_{pause} to await the node's resurrection. If the node resurrects within T_{pause}, the other nodes should await until the resurrected node finishes the sharding. Otherwise, the failed node is announced dead and then removed. In addition, if the dead node is the coordinator, a new coordinator is elected amongst the living nodes. Finally, the (new) coordinator announces that the sharding is successful. Similarly, nodes resurrecting after T_{pause} cannot serve P_i, so their replicas of P_i are invalidated. Nevertheless, these nodes can replicate the partition from the other successful nodes.

The sharding can be aborted, whenever more than one node failures are detected, or any other unexpected events occur. Such abortion does not incur any data loss, since the original partition replicas are in use before the announcement of success. The details of maintaining data consistency are discussed in Section 4.1. The aborted sharding will be reinitiated after a long pause.

Thus, our partitioning algorithm consolidates partitions that are suitable for efficient data migration. Based on such consolidated replicas, we discuss the data placement strategy for node bootstrapping (and decommissioning).

3.2 Selecting Partition Replicas for Bootstrapping

The aim of replica placement is to achieve load balancing and quick node bootstrapping. When a new empty node is to be bootstrapped, it selects and pulls a list of partition replicas from the existing nodes based on a set of rules.

Rule 1: Complexity Reduction. Partition reallocation and sharding are mutually exclusive. That is, partitions that are being sharded will not be selected for replication, and partitions that are being reallocated will not be sharded. Hence, we reduce the complexity of coordinating data reallocation and sharding.

Rule 2: High Availability. Each partition P_i has ν_i replicas allocated in ν_i different nodes. We defined the replication number K, such that $\forall i \in [1, N], \nu_i \geq K$, wherein N is the number of partitions. If a partition has less than K replicas (e.g. due to node failure), a replica is duplicated to the new node.

Rule 3: Load Balancing. The nodes with higher workloads have higher priority to offer replicas. Hence, heavily loaded nodes have the priority to move out more replicas (thus shifting the workload) to the new node.

Rule 4: Data Balancing. Since each partition replica is confined into bounded sizes by sharding, balancing the number of partition replicas can result in balancing the volume of data stored in each node. Let $\overline{R}$ be the average number of replicas each node has. Before bootstrapping, the new node recalculates $\overline{R} = \sum_{i=1}^{N} \nu_i/(n+1)$, in which n is the number of existing nodes. The new node can obtain no more than $\overline{R}$ replicas, while an existing node can offer (i.e. move out) replicas as long as it has more than $\overline{R}$ replicas.

To achieve quick bootstrapping, we propose a two-phase data migration strategy. In the *pre-bootstrapping* phase, the new node aims at maintaining high availability (referring to Rule 2) and alleviating the nodes that are under heavy workloads (Rule 3). Each heavily loaded node is requested to move out a small portion, e.g. 10%, of its replicas. Once the new node receives these replicas, it

completes bootstrapping and starts serving queries as a member of the KVS immediately. In our implementation, we used the CPU usage to estimate the workload each node undertakes. The CPU usage is piggybacked on the heartbeat gossip message, sent by each living node periodically and cached by every other node. Therefore, a new node can download the complete workload information from any existing node. A node is marked (by the new node) as *heavily loaded*, if its CPU usage is over 50% and reasonably (e.g. 20%) greater than the average of all the nodes. The threshold for identifying a heavily-load node is configurable by the system administrators.

In the *post-bootstrapping* phase, as long as Rule 4 is satisfied, the newly joined node continues to pull in more replicas from a list of nodes sorted according to Rule 3. This process is run in a background thread, with data transfer rate throttled, such that the side-effects towards front-end query processing are minimised. In this two-phase procedure, the new node receives the majority of its replicas in the *post-bootstrapping* phase, since in the *pre-bootstrapping* phase there are limited number of *heavily loaded* nodes, each offering only a small number of replicas. Therefore, the new node completes the *pre-bootstrapping* phase in a timely manner, i.e. quick bootstrapping.

There are also considerations on how to select partition replicas when an existing node is requested (e.g. by the new node) to offer data. Each node maintains an exponential moving average (EMA) of the *local* hit count for each replica, which is updated periodically as in Equation 3. The moving average hit count of partition P_i at time t is denoted as $H_{i,t}$, and the actual hit count of P_i between time $t-1$ and t is denoted as $h_{i,t-1}$. The coefficient α represents the degree of weighting decrease.

$$H_{i,t} = \alpha h_{i,t-1} + (1-\alpha) H_{i,t-1} \tag{3}$$

To select a replica to move out, the node sorts its own replicas by the EMA of hit count. We avoid the greedy heuristic (i.e. move the hottest or coldest replica), since it may destabilise the system by causing more data movement. Instead, the node traverses the list starting from the middle, until it finds the first replica that does not exist in the destination node.

3.3 Node Decommissioning

We have also designed a replica placement scheme for node decommissioning. There are circumstances when node decommissioning is necessary: i) a living node is misbehaving, e.g. it is failing more often than it should or its performance is noticeably slow. ii) there are redundant compute resources, e.g. none of the living nodes is *heavily loaded* and there exists nodes that receive less queries than expected. In any case, the decision to decommission a node is made by the KVS administrators. In this paper we only discuss how to reallocate the replicas when node decommissioning is requested.

The node to be decommissioned moves out its replicas *one by one* to the other living nodes. It can safely leave the KVS when there is no more replica under its ownership. To choose a destination node for a replica (e.g. P_i), the node retrieves a list of living nodes that do not own P_i. Then it selects the least loaded node

from the list as the destination. We prefer to balance the query workload (i.e. CPU usage) rather than the data volume, since storage is rarely the bottleneck in the cloud. Note that the workload information is gossiped periodically. Each time when the node attempts to move out a replica, it may choose a different node as the destination. In this way, the decommissioning node distributes its own replicas to the peers, and then leaves the KVS without data loss.

4 Data Recovery and Consistency

As our implementation builds on the Apache Cassandra project, we have leveraged hinted handoff [10] implemented in Cassandra to recover the data for node failure. When a replica node for the key is down, a hint is written to the coordinator node of the related partition. The coordinator node is chosen with the same election policy as in auto-sharding. However, unlike Cassandra, wherein users can define a consistency level for each individual query, we proposed to enforce each write to be saved in every replica of the targeted partition. Therefore, our consistency strategy caters for read-intensive workloads. The consistency issues during partition sharding and replica movement are discussed as follows.

4.1 Data Consistency During Sharding

While the replicas of a partition is being rebuilt during sharding, there are two sets of replicas (i.e. data files) coexisting in each participating node. One set belongs to the original partition, and the other set of data files belongs to the future partition (i.e. after sharding). Reads and writes of a key-value pair are treated differently to maintain data consistency.

Writes (i.e. update or delete) are saved in the data files of the future partition. If the sharding is completed successfully, the files of the original partition can be abandoned safely. Otherwise, if the operation fails, the key-value pairs written to the future partition are merged back to the original partition. In the extreme case where the sharding fails very often, writes are enforced in "dual play", i.e. saved in both the original and future partitions. Thereby, the data files of the future partition can be safely discarded whenever a sharding fails.

Reads are dealt with depending on how writes are processed. If writes are enforced in "dual play", the data value can be retrieved directly from the replicas of the original partition. Otherwise, if writes are saved to the replicas of the future partition only, the node retrieves the data value from both the original and future partitions. We have leveraged timestamp-based reconciliation in Dynamo [10] to allow multiple versions of an object to be present in the system. The timestamps of multiple collided values are compared, and the latest value "wins". Thereby, we maintain eventual consistency across different sets of replicas.

At the end of sharding, after the operation is announced successful, the file handlers of the partition replica is replaced in an atomic operation within each participating node, independently and asynchronously from other nodes. Thus, data files of the original partitions are deleted safely from all the nodes.

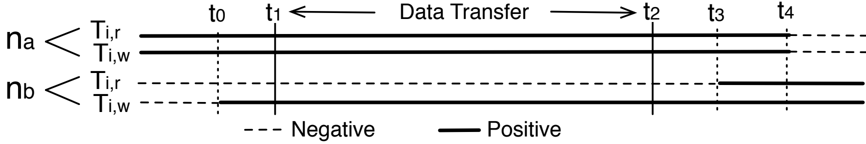


Fig. 4. Switching token values during replica migration for data consistency

4.2 Data Consistency When Moving Replicas

We have implemented a token ownership policy to ensure data consistency for partition replicas that are being moved or duplicated. As discussed, a list of tokens $\{T_i\}$ split the key space into consecutive key ranges. Each partition P_i is associated with one token T_i. For query-execution purpose, every node maintains two boolean values for each P_i: one readable token $T_{i,r}$, and one writable token $T_{i,w}$. The nodes that own the *positive* value of $T_{i,r}$ or $T_{i,w}$ are entitled to serve reads or writes from P_i, respectively.

Figure 4 depicts when and how to switch the values of $T_{i,r}$ and $T_{i,w}$, when a replica of the partition P_i is moved from n_a to n_b. The time intervals between (t_0, t_1), (t_2, t_3) and (t_3, t_4) are longer than the end-to-end gossip broadcast delay, so that every updated value is well propagated. Before and during data migration, the source node n_a, which is serving P_i, owns the *positive* $T_{i,r}$ and $T_{i,w}$. The destination n_b, which does not serve P_i initially, owns the *negative* tokens. Before the data is transferred, n_b switches its $T_{i,w}$ to *positive* at t_0, so that n_b is entitled to receive the latest updates destined for P_i. After the data is transferred, n_b switches its $T_{i,r}$ to *positive* at t_3, since n_b is now eligible to serve reads from P_i. After n_b has taken over P_i, n_a resets both its tokens to *negative* at t_4. In the end, the replica of P_i in n_a is discarded.

The operation of duplicating replicas (e.g. from n_a to n_b) is very similar to moving replicas. The only difference is that, in duplication, the source node n_a owns the positive $T_{i,r}$ and $T_{i,w}$ at all times. Thus, n_a neither resets tokens to *negative* at t_4, nor deletes replicas at the end of data transfer.

5 Evaluation

We have evaluated ElasCass against Apache Cassandra (version 1.0.5) that uses **split-move** as discussed in Section 2. Thus, this section evaluates the efficiency of the proposed approach against the split-move approach for node bootstrapping. Hence, the experimental results of Cassandra are labeled as **split-move**.

5.1 Experimental Setup

The experiments were conducted on Amazon EC2. Each VM runs as one node of the KVS. All of the VM instances are based off a common Linux image. The

Table 1. Compute capacity of VMs in experiments

Name	Value
OS	Ubuntu 12.04, 3.2.0-29-virtual, x86_64
File system	ext3
Instance Type	m1.large
Memory	7.5 GB
CPU	2 virtual cores with 2 EC2 Compute Units each
Storage	2 ephemeral storage with 420GB each
Disk I/O	High

Table 2. Parameters configured in YCSB

Name	Value
records	size = 1KB, count = 100 million
insert order	hashed with 64-bit FNV
read/update ratio	50/50 for write-intensive, 95/5 for read-intensive
request distribution	zipfian (constant = 0.99)
	hotspot (80% of requests targeting at 20% of data)
ConsistencyLevel	write: *ALL*; read: *ONE*

compute capacity is shown in Table 1. For performance reasons, the persistent data of the KVS was stored on the 400 GB ephemeral storage of the VM, rather than on an Elastic Block Storage (EBS) volume. This is consistent with known production deployments of Cassandra on EC2 [4]. The I/O performance of this ephemeral storage is categorised as "High". According to Amazon[3], *High* I/O instances can deliver in excess of 100,000 random read IOPS and as many as 80,000 random write IOPS.

The YCSB benchmark (version 0.1.4) [6] was used in this experiment with parameters configured as shown in Table 2. The dataset is generated by the YCSB client in the *loading* section. The total size is approximately 100GB. The inserted keys are hashed with the 64-bit FNV function[4], so the hotspot data is scattered onto many partitions. Both write-intensive and read-intensive workloads were generated using YCSB. Each workload was generated with two different request distributions, i.e. zipfian and hotspot. The consistency level[5] is set as *ALL* for write operations, and *ONE* for read operations. This parameter specifies how many replicas must respond before a result is returned to the client. It tunes response time versus data accuracy, but does not affect the eventual consistency in key-value stores.

To evaluate load balancing, an *imbalance* index I_L is defined to indicate the imbalance in load $\{L_i\}_{i=1}^{n}$ across a group of n nodes. Let $I_L = \sigma_L/\overline{L}$, where $\overline{L}$

[3] http://aws.amazon.com/ec2/instance-types/

[4] FowlerNollVo is a non-cryptographic hash function created by Glenn Fowler, Landon Curt Noll, and Phong Vo.

[5] http://www.datastax.com/docs/1.0/dml/data_consistency

is the average value of all the loads $\{L_i\}_{i=1}^n$, and σ_L is the standard deviation of $\{L_i\}_{i=1}^n$. This index shows the proportion of the variation (or dispersion) from the average. A smaller value of I_L indicates better load balancing. We have evaluated the balancing of both the data volume and the query workload.

In addition, the average CPU utilisation is used to quantify the workload per node. We monitored the CPU usage periodically using the linux command "*sar -u 5 2*", which reports the average CPU usage every 10 seconds.

5.2 Node Bootstrapping

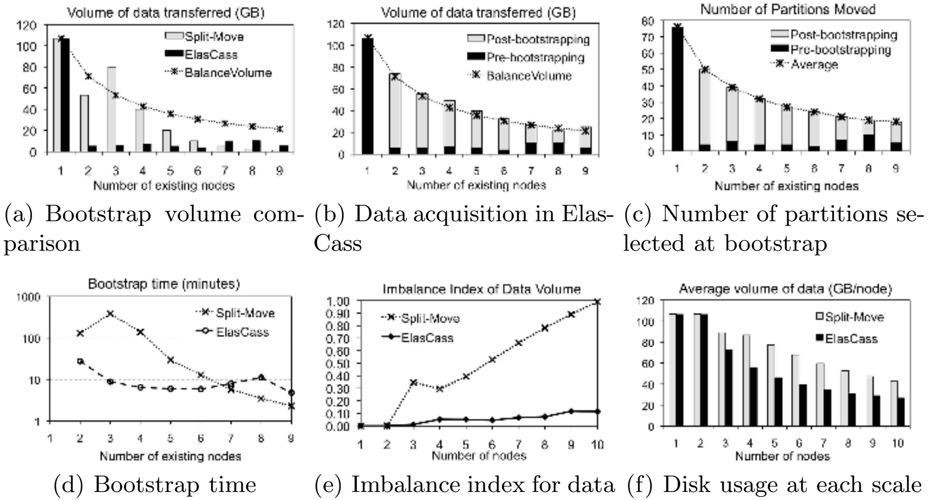
In this experiment, we demonstrate the effects of bootstrapping nodes one after another, in a relatively short time, in each KVS. Apart from the bootstrap time, we measure the volume of data acquired by a node at bootstrap (*bootstrap volume*). Ideally, the ith node should share $1/i$ of the total volume of data in the system (*BalanceVolume*). However, this is affected by the partitioning and placement strategies employed. Therefore, we also measure the imbalance index of data distribution across the nodes.

Both ElasCass and the original Apache Cassandra were initialised with one node. The 100GB data was loaded on to the first node, with Θ_{max}=2GB and Θ_{min}=1GB in ElasCass. The replication level of both systems is configured as $K = 2$. Therefore, when the next node was added, the 100GB data was automatically replicated to the second node fully. From two nodes onwards, one empty node was added at each time. The data was reallocated according to different strategies in these two systems.

During the whole process of node bootstrapping, both systems were subjected to a read-intensive background workload that followed the hotspot distribution (Table 2). Each time before a new node was initiated, we made sure that every existing node had been serving queries for at least 15 minutes as a normal member of the KVS. Then, we tuned the number of threads in the YCSB client, such that the CPU usage of the most loaded node was less than 80%, while the average CPU usage of all the existing nodes fluctuated around 50%. Therefore, both systems were *moderately* loaded before a new node was added.

Figure 5(d) shows the bootstrap times for ElasCass and original Cassandra with increasing number of nodes. The bootstrap time for ElasCass is bounded while Cassandra, following the split-move approach, starts with a high bootstrap time (over 100 minutes) that reduces with the number of nodes. This can be explained by the bootstrap volume as shown in Figure 5(a). At all scales from three nodes onwards, ElasCass managed to transfer less than 10GB of data constantly at bootstrap, as the remaining replicas were migrated in the post-bootstrapping phase. In contrast, with the split-move approach, the volume of data migrated decreases from over 80GB to merely 1GB. Specifically, from seven nodes onwards, the split-move approach did not migrate enough data as ElasCass did, thus requiring less time for bootstrapping. The penalty is that split-move suffered from load imbalance issue, revealed in Figure 5(e).

Figure 5(b) depicts the volume of data transferred in ElasCass during and after bootstrap. As shown, the total volume of data transferred in both phases



(a) Bootstrap volume comparison (b) Data acquisition in ElasCass (c) Number of partitions selected at bootstrap

(d) Bootstrap time (e) Imbalance index for data (f) Disk usage at each scale

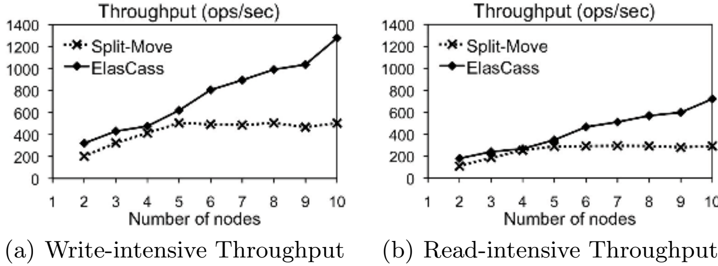
Fig. 5. Performance of bootstrapping one node with different nodes

is roughly equal to *BalanceVolume* at each scale. Figure 5(c) demonstrates the same result in the form of numbers of partition replicas moved. The total number of replicas moved is exactly equal to the average number of replicas each node owns. This means that the data distribution in ElasCass is closer to the ideal than that produced by the split-move approach.

With the split-move approach, the data volume drops exponentially as the system scales up. The reason is that, when a key range of data is moved to the new node, the persistent data is retained in (but no longer served by) the source node, until the files are re-*compact*ed, so as to avoid the loss of data and the overhead of deleting large amounts of individual key-value pairs. However, since file *compaction* is a heavyweight operation, it is rarely triggered when serving read-intensive workloads. As a result, during the evaluation, all the new nodes chose the same node (i.e. the node with the most data on disk) as the data source. Even worse, each time the chosen node had to offer half of its remaining key range, which was reduced exponentially as new nodes were added.

Figure 5(e) backs this up by comparing the imbalance indices for both ElasCass and the split-move approach. The imbalance index of ElasCass is low, which indicates that the data is evenly distributed. In contrast, in the split-move approach, the data is less balanced when more new nodes are added. Figure 5(f) shows the average volume of data stored on disk in all the nodes. The split-move approach occupies more storage because the source node that offers data at bootstrap tends to retain the invalid data on disk.

Overall, this experiment demonstrates that, as the system scales, ElasCass is able to: i) bootstrap an empty node within a relatively short time (i.e. 10 minutes), by limiting the number of partition replicas transferred pre-bootstrapping; ii) distributes the data more evenly amongst the nodes in the post-bootstrapping phase; and iii) occupy less storage than the split-move approach.



(a) Write-intensive Throughput (b) Read-intensive Throughput

Fig. 6. The performance of query processing with zipfian distribution

5.3 Performance of Query Processing

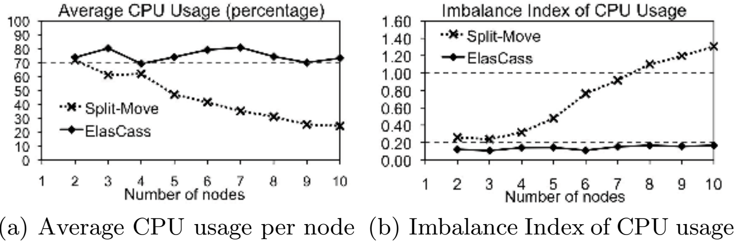
We focussed on the improvement of workload throughput as the system scaled. In order to measure the steady-state throughput, we set an upper-bound for the average read latency as 100 milliseconds. Before each test, we tuned the number of threads in the YCSB client, such that the average read latency is one-step below this bound. Based on this latency, we tuned the operation count (i.e. number of requests), so that the test can last long enough (at least 1000 seconds). Therefore, in all the tests presented, the average read latency is slightly less than 100 ms, and each run lasts at least 1000 seconds.

Apart from workload throughput, we also measure the CPU utilisation to quantify the workload each node undertakes. We calculate the average CPU usage per node to indicate resource utilisation, and the imbalance index (defined in Subsection 5.1) of the CPU usage of all the nodes to evaluate load balancing in the system. The experimental results for the workload following the zipfian distribution are extremely similar to the results for the hotspot distribution. Due to space limits, we present the experimental results for the zipfian distribution only in this section.

Figure 6 depicts the throughputs of query processing in Cassandra (using split-move) and ElasCass against increasing number of nodes. As can be seen in Figure 6(a), when the system is subject to write-intensive workloads, the throughput of the KVS using split-move stops improving after adding the 5th node, while in ElasCass, the throughput increases linearly with the number of nodes. Figure 6(b) depicts a similar trend. ElasCass continues to demonstrate better scalability than split-move under read-intensive workloads.

Moreover, if we compare Figure 6(b) with 6(a), it can be seen that both systems have higher throughputs under write-intensive workloads than read-intensive. This is because write operations are buffered in memory and written in batch mode, while read operations require random disk I/Os, which are confined by the I/O performance.

The reason why ElasCass outperformed the KVS using split-move by such an extent is due to the imbalance in data distribution in the latter (Figure 5(e)). Due to the lack of data moved to the new nodes, the split-move approach was not able to scale properly. In practice, a *compaction* can be launched manually



(a) Average CPU usage per node (b) Imbalance Index of CPU usage

Fig. 7. How the workload is balanced across nodes under read-intensive workloads following the zipfian distribution

before bootstrapping a node. Compaction will update the information about the data volume on each node, so that a new node can choose source nodes more appropriately. However, this evaluation was designed to demonstrate how the system will behave without human intervention. The KVS using split-move was not able to complete a file *compaction* during the evaluation, which makes it unadaptable in the scenario where new nodes are added one after another in a relatively short time. In contrast, ElasCass is able to move a small portion of partitions from heavily loaded nodes during the pre-bootstrapping phase for quick bootstrapping, and then retrieves a large volume of data in the post-bootstrapping phase to achieve the balancing of workload and data volume.

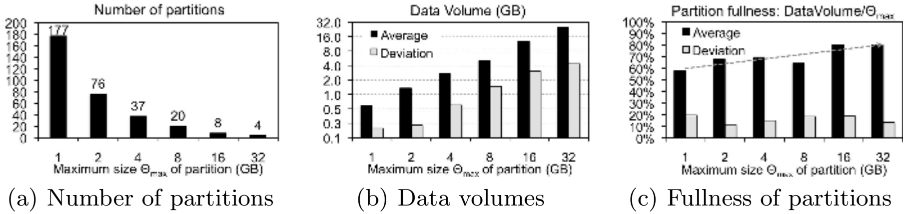
Figure 7 shows the average CPU usage of all the nodes during the tests under the read-intensive workloads. The results for the write-intensive workloads show a similar trend, so they are not presented due to page limit. As seen, the average CPU usage of ElasCass remains above 70%. However, in the KVS using split-move, the CPU usage declines gradually as the system scales up. The results indicate that ElasCass is able to fully utilise the provisioned compute resources at different scales for serving queries, while in Cassandra the newly added nodes were not efficiently incorporated into query processing.

Figure 7(b) presents the imbalance index of the CPU usage. With the split-move approach, the imbalance index climbs up as the system scales. From the scale of eight nodes onwards, the index even goes beyond 1.0, which means that the standard deviation of the CPU usages is even greater than the average usage. The results indicate that some nodes are heavily loaded, while the others remain idle. The workload was not balanced with split-move. However, this index in ElasCass remains below 0.2 in all the tests. A small value of imbalance index indicates that the workload is well balanced in ElasCass.

Overall, this set of experiments demonstrates that, due to better balancing of data volume in ElasCass, it outperforms the KVS using split-move in query processing by a large extent in terms of scalability and load balancing.

5.4 Data Partitioning

In this experiment, we demonstrate how the maximum size Θ_{max} and the minimum size Θ_{min} (defined in Equation 1 and 2) can affect the partitioning results



(a) Number of partitions (b) Data volumes (c) Fullness of partitions

Fig. 8. Partitioning 100GB of data under different sharding threshold

with our sharding strategy. We used the same setting for the YCSB client to generate the 100GB dataset (Table 2), but the dataset was loaded independently in each test with different values of Θ_{max} and Θ_{min}. There are six tests, in which the maximum size Θ_{max} increases from 1GB to 32GB exponentially. The minimum size is set as one half of the upper, i.e. $\Theta_{min} = \Theta_{max}/2$.

Given different values of Θ_{max}, Figure 8(a) shows the total number of partitions generated, while Figure 8(b) shows the volume of data stored in the partitions. As can be seen, as the value of Θ_{max} increases, the resulted number of partitions decreases inversely, whilst the average volume of data in each partition grows linearly with Θ_{max}.

Different settings of Θ_{max} can affect the system's performance. If Θ_{max} is too small, partitions are sharded very frequently, which increases the overhead of building replicas. In addition, small Θ_{max} results in a large number of partitions, which increases the complexity of partition reallocation. On the other hand, if Θ_{max} is too large, the resulted partitions will contain a large volume of data. Moving a large-size partition replica may end up in overwhelming the node that takes it over. Moreover, it takes substantially long time to reallocate a large-size replica, which is not efficient. Therefore, in the remaining evaluations, we set Θ_{max}=2GB and Θ_{min}=1GB.

In Figure 8(c), we use the term "fullness" to compare the data volume shown in Figure 8(b). The value of fullness is calculated as the data volume of the partition divided by Θ_{max}. In other words, the fullness indicates how full the partition is before it reaches the maximum capacity. Figure 8(c) shows that the average fullness of the partition ranges between 60% and 80%, and the standard deviation of fullness is consistently below 20% given different Θ_{max}. The results indicate that the dataset is effectively segmented into a list of partitions that are of roughly equal sizes, without sparse partitions (i.e. having little data) generated. In addition, there is a trend that larger values of Θ_{max} tend to result in greater fullness. This is because smaller upper bounds increase the frequency of partition splitting. Note that when a partition is split, the data volume of the resulting partitions is only half of Θ_{max}, i.e. the fullness is 50%.

6 Conclusion

Efficient node bootstrapping is an important feature for distributed KVSs running on IaaS. We have presented a decentralised scheme of data partitioning

and placement to efficiently bootstrap nodes in shared nothing KVSs. Our auto-sharding scheme, extended from the virtual-node based data management, improves the efficiency of data movement at bootstrap, by consolidating each partition into single transferable units without data skew. Using a two-phase placement strategy, we minimise the data movement during bootstrap to achieve fast bootstrapping, while populating the newly-added nodes after bootstrap to achieve well balanced workload and data distribution.

We have implemented the proposed scheme in Apache Cassandra [15] that follows the split-move strategy, to present ElasCass. We evaluated our scheme against the split-move approach, by experimentally evaluating ElasCass against Cassandra using YCSB on public IaaS. We demonstrated that ElasCass was capable of incorporating new empty nodes consecutively in a relatively short time, with ideally balanced data distribution and much better balanced workload than Cassandra. As a result, ElasCass exhibited better resource utilisation in compute and storage, and outperformed Cassandra in scalability by a large extent under the biased workloads. We also demonstrated the capability of our auto-sharding scheme to confine each partition into a bounded size, without data skew or sparse partitions.

In the future, we plan to augment this scheme with the control logic that determines when and how many nodes should be bootstrapped or decommissioned. This control logic along with the ability to add and remove nodes efficiently, form the basis for autonomous elasticity in shared nothing KVSs on IaaS platforms.

Acknowledgments. The authors would like to thank Smart Services CRC Pty Ltd for the grant of Services Aggregation project that made this work possible.

References

1. Aberer, K.: Peer-to-peer data management. Synthesis Lectures on Data Management 3(2), 1–150 (2011)
2. Burrows, M.: The chubby lock service for loosely-coupled distributed systems. In: Proceedings of the 7th Symposium on Operating Systems Design and Implementation, pp. 335–350. USENIX Association (2006)
3. Chang, F., Dean, J., Ghemawat, S., Hsieh, W., Wallach, D., Burrows, M., Chandra, T., Fikes, A., Gruber, R.: Bigtable: A distributed storage system for structured data. ACM Transactions on Computer Systems (TOCS) 26(2), 1–26 (2008)
4. Cockroft, A.: Netflix goes global. In: Proc. 14th International Workshop on High Performance Transaction Systems (HPTS). USENIX (2011)
5. Cooper, B., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P., Jacobsen, H., Puz, N., Weaver, D., Yerneni, R.: Pnuts: Yahoo!'s hosted data serving platform. Proceedings of the VLDB Endowment 1(2), 1277–1288 (2008)
6. Cooper, B., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with ycsb. In: Proceedings of the 1st ACM Symposium on Cloud Computing, pp. 143–154. ACM (2010)
7. Corbett, J.C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., et al.: Spanner: Googles globally-distributed database. In: Proceedings of OSDI, vol. 1 (2012)

8. Curino, C., Jones, E., Zhang, Y., Madden, S.: Schism: a workload-driven approach to database replication and partitioning. Proceedings of the VLDB Endowment 3(1-2), 48–57 (2010)
9. Das, S., Nishimura, S., Agrawal, D., El Abbadi, A.: Albatross: lightweight elasticity in shared storage databases for the cloud using live data migration. Proceedings of the VLDB Endowment 4(8), 494–505 (2011)
10. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: amazon's highly available key-value store. In: SOSP, vol. 7, pp. 205–220 (2007)
11. Ghemawat, S., Gobioff, H., Leung, S.: The Google file system. In: ACM SIGOPS Operating Systems Review, vol. 37, pp. 29–43. ACM (2003)
12. Gupta, A., Liskov, B., Rodrigues, R.: One hop lookups for peer-to-peer overlays. In: HotOS, pp. 7–12 (2003)
13. Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M., Lewin, D.: Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In: Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing, pp. 654–663. ACM (1997)
14. Krishnan, P., Raz, D., Shavitt, Y.: The cache location problem. IEEE/ACM Transactions on Networking (TON) 8(5), 568–582 (2000)
15. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. ACM SIGOPS Operating Systems Review 44(2), 35–40 (2010)
16. Laoutaris, N., Telelis, O., Zissimopoulos, V., Stavrakakis, I.: Distributed selfish replication. IEEE Transactions on Parallel and Distributed Systems 17(12), 1401–1413 (2006)
17. Rowstron, A., Druschel, P.: Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In: Guerraoui, R. (ed.) Middleware 2001. LNCS, vol. 2218, pp. 329–350. Springer, Heidelberg (2001)
18. Shvachko, K., Kuang, H., Radia, S., Chansler, R.: The hadoop distributed file system. In: 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), pp. 1–10. IEEE (2010)
19. Stoica, I., Morris, R., Karger, D., Kaashoek, M., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications. ACM SIGCOMM Computer Communication Review 31(4), 149–160 (2001)
20. Van Renesse, R., Minsky, Y., Hayden, M.: A gossip-style failure detection service. In: Middleware 1998, pp. 55–70. Springer (1998)
21. You, G.-W., Hwang, S.-W., Jain, N.: Scalable Load Balancing in Cluster Storage Systems. In: Kon, F., Kermarrec, A.-M. (eds.) Middleware 2011. LNCS, vol. 7049, pp. 101–122. Springer, Heidelberg (2011)
22. Zaman, S., Grosu, D.: A distributed algorithm for the replica placement problem. IEEE Transactions on Parallel and Distributed Systems 22(9), 1455–1468 (2011)