G-Store: A Scalable Data Store for Transactional Multi key Access in the Cloud

Sudipto Das Divyakant Agrawal Amr El Abbadi
Department of Computer Science
University of California, Santa Barbara
Santa Barbara, CA 93106-5110, USA
{sudipto, agrawal, amr}@cs.ucsb.edu

ABSTRACT

Cloud computing has emerged as a preferred platform for deploying scalable web-applications. With the growing scale of these applications and the data associated with them, scalable data management systems form a crucial part of the cloud infrastructure. *Key-Value* stores – such as Bigtable, PNUTS, Dynamo, and their open source analogues– have been the preferred data stores for applications in the cloud. In these systems, data is represented as *Key-Value* pairs, and atomic access is provided only at the granularity of *single keys*. While these properties work well for current applications, they are insufficient for the next generation web applications – such as online gaming, social networks, collaborative editing, and many more – which emphasize collaboration. Since collaboration by definition requires consistent access to *groups* of keys, scalable and consistent *multi key* access is critical for such applications. We propose the Key Group abstraction that defines a relationship between a group of *keys* and is the granule for on-demand transactional access. This abstraction allows the Key Grouping protocol to collocate control for the keys in the group to allow efficient access to the group of keys. Using the Key Grouping protocol, we design and implement G-Store which uses a *key-value* store as an underlying substrate to provide efficient, scalable, and transactional *multi key* access. Our implementation using a standard *key-value* store and experiments using a cluster of commodity machines show that G-Store preserves the desired properties of *key-value* stores, while providing *multi key* access functionality at a very low overhead.

Categories and Subject Descriptors

H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*Distributed systems*; H.2.4 [**Database Management**]: Systems—*Concurrency, Transaction processing*

General Terms

Algorithms, Design, Performance.

Keywords

Cloud computing, *Key-Value* stores, Consistency, Multi key access.

SoCC'10, June 10–11, 2010, Indianapolis, Indiana, USA.
Copyright 2010 ACM 978-1-4503-0036-0/10/06 ...$10.00.

1. INTRODUCTION

The widespread popularity of Cloud computing as a preferred platform for the deployment of web applications has resulted in an enormous number of applications moving to the cloud, and the huge success of cloud service providers. Due to the increasing number of web applications being hosted in the cloud, and the growing scale of data which these applications store, process, and serve – scalable data management systems form a critical part of cloud infrastructures.

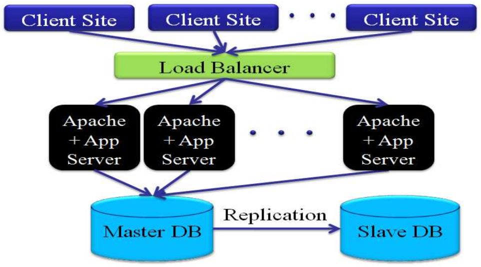


Figure 1: Typical Software Stack for Web Applications.

One of the major uses of the cloud is for *application hosting*, *content delivery*, *e-commerce*, and *web hosting*; and data management systems form an important part of the software stack for these applications. Figure 1 depicts the software stack typically used for web-applications. As user traffic increases, the application and web servers can be easily scaled out by adding new instances, but the database server often becomes the scalability bottleneck [29, 33]. Even though relational database technologies have been extremely successful in the traditional enterprise setting, data management in the cloud must possess additional key features which traditional relational databases were not designed to support.

The pay-per-use model, and virtually infinite scalability in the cloud have broken the infrastructure barrier for new application ideas to be tried out. As a result, deployment of modern application ideas which were not economically feasible in the traditional enterprise setting, have become possible due to the cloud computing paradigm. Additionally, these applications often have unpredictable and varying load patterns depending on their popularity [20, 31]. Therefore, the data management systems for these applications must adapt to these varying load patterns while being highly scalable. Furthermore, due to the large scale of cloud infrastructures, failures are common, and hence, fault-tolerance, high availability, and ease of administration are essential features of data management systems in the cloud. In summary, data management systems in the cloud should be able to scale out using commodity servers, be fault-tolerant, highly available, and easy to administer –

features which are inherent to *key-value* stores, thus making them the preferred data management solutions in the cloud. As a result, applications in the cloud favor scalable and fault-tolerant *key-value* stores (also referred to as *row stores*) such as Bigtable [8], PNUTS [9], Dynamo [12], and their open-source analogues [17].

The various *key-value* stores differ in terms of data model, availability, and consistency guarantees – but the property common to all systems is the *Key-Value* abstraction where data is viewed as *key-value* pairs and atomic access is supported only at the granularity of *single keys*. This *single key* atomic access semantics naturally allows efficient horizontal data partitioning, and provides the basis for scalability and availability in these systems. Even though a majority of present web applications have *single key* access patterns [12], many current applications, and a large number of Web 2.0 applications (such as those based on collaboration) go beyond the semantics of *single key* access, and foray into the space of *multi key* accesses [3]. Present scalable data management systems therefore cannot directly cater to the requirements of these modern applications, and these applications either have to fall back to traditional databases, or to rely on various ad-hoc solutions.

An Application Scenario. Consider an online multi-player casino application. Each player has a profile, an expertise level, and some virtual (or actual) money, in addition to various other attributes. Each player's profile can be represented as a *key-value* pair in a *Key-Value* store, and can be accessed by a unique id which serves as the *key*. A number of players come together to play in a game instance. During the course of the game, as different players play their turns, each player's rating or score will change, and so will their account balance; and these changes have to be transactional across the players involved in the game[1]. Similarly, there can be multiple concurrent transactions involving the participants of a game as different players are playing their turns. Since present *Key-Value* stores only provide guarantees for *single key* accesses, using a *key-value* store for this casino application would require application level workflows for handling *multi key* accesses, and providing the atomicity and consistency guarantees required for the correct operation of the game [28]. Many similar applications relying on collaboration require *multi key* access guarantees which present generation *key-value* stores fail to provide.

As noted by many practitioners [22,28], the reduced consistency guarantees and *single key* access granularity supported by the *key-value* stores often places huge burden on the application programmers who now have to reason about failures, concurrency, and inconsistent data in order to ensure correctness. An alternative to avoiding this complexity is to use traditional database technologies which are often considered unsuitable for the scale of applications in the cloud [29]. As a result, there is a huge chasm when it comes to selecting an appropriate data store for these applications. Google's data store for AppEngine, referred to as MegaStore [22], takes a step beyond *single key* access patterns by supporting transactional access for groups of keys referred to as *entity groups*, which are formed using *keys* with a specific hierarchical structure. Since *keys* cannot be updated in place, once a key is created as a part of a group, it has to be in the group for the rest of its lifetime. This static nature of *entity groups*, in addition to the requirement that *keys* be contiguous in sort order, are in many cases insufficient and restrictive. For instance, in case of the casino application, *multi key* access guarantees are needed only during the course of a game. Once a game finishes, different users can move to different game instances thereby requiring guarantees on dynamic groups of keys – a feature not currently supported by MegaStore.

G-Store. We present G-Store, a scalable data store providing transactional *multi key* access guarantees over *dynamic, non-overlapping groups of keys* using a *key-value* store as an underlying substrate, and therefore inheriting its scalability, fault-tolerance, and high availability. The basic innovation that allows scalable *multi key* access is the Key Group abstraction which defines a granule of on-demand transactional access. The Key Grouping protocol uses the Key Group abstraction to transfer *ownership*[2] for all *keys* in a group to a single node which then efficiently executes the operations on the Key Group. This design is suitable for applications that require transactional access to groups of keys that are transient in nature, but live long enough to amortize the cost of group formation. Our assumption is that the number of *keys* in a group is small enough to be *owned* by a single node. Considering the size and capacity of present commodity hardware, groups with thousands to hundreds of thousands of *keys* can be efficiently supported. Furthermore, the system can scale-out from tens to hundreds of commodity nodes to support millions of Key Groups. G-Store inherits the data model as well as the set of operations from the underlying *Key-Value* store; the only addition being that the notions of atomicity and consistency are extended from a single key to a group of keys.

Contributions.

- We propose the Key Group abstraction which defines a granule for on-demand transactional access over a dynamically selected set of keys. This abstraction allows Key Groups to be dynamically created and dissolved, while providing application clients the ability to arbitrarily select keys to form a Key Group.
- We propose the Key Grouping protocol which collocates *ownership* of all the *keys* in a group at a single node, thereby allowing efficient *multi key* access.
- We describe two implementations for G-Store, a data store layered on top of a *key-value* store, which supports transactional *multi key* access using a *key-value* store as an underlying data store. The experimental results, using a cluster of machines, show the scalability of group creation, as well as the low overhead maintenance of groups while providing a richer functionality when compared to the *Key-Value* store used as a substrate.

Organization. Section 2 provides a survey of the state-of-the-art in scalable data management systems. Section 3 introduces the Key Group abstraction which allows collocation of *keys* of a Key Group, and Section 4 describes the Key Grouping protocol which transfers ownership of the members of a Key Group to a single node. Section 5 provides two alternative implementation designs of G-Store using a *key-value* store as a substrate. Section 6 provides experimental evaluations of the proposed system on a cluster of machines, and Section 7 concludes the paper.

2. RELATED WORK

Scalable and distributed data management has been the vision of the database research community for more than two decades. This thrust towards scaling out to multiple nodes resulted in two different types of systems: distributed database systems such as R^* [25] and SSD-1 [30], which were designed for update intensive workloads; and parallel database systems such as Gamma [13] and Grace [15], which allowed updates but were predominantly used

[1] We use the term transactional to refer to the ACID properties attributed to database transactions [32].

[2] Ownership refers to exclusive read/write access to *keys*.

for analytical workloads. These systems were aimed at providing the functionality of a centralized database server, while scaling out to a number of nodes. Even though parallel database systems have been extremely successful, distributed databases have remained as research prototypes primarily due to the overhead of distributed transactions, and the fact that guaranteeing transactional properties in the presence of various kinds of failures limiting scalability and availability of these systems. As a result, the database industry resorted to scaling up of database servers rather than scaling out.[3] Based on more than two decades of experience with such systems, the community realized that scaling out while providing consistency of transactions, and guaranteeing availability in the presence of different failures was not feasible [1, 19]. In addition, changes in data access patterns resulting from a new generation of web-applications called for systems with high scalability and availability at the expense of weaker consistency guarantees. As a result, the concept of *Key-Value* stores, and accesses at the granularity of *single keys* was put forward as the sole means to attain high scalability, and availability [19]. Based on these principles, a number of *Key-Value* stores [8, 9, 12, 17] were designed and successfully implemented. Even though the distributed database paradigm was not successful, a lot of important concepts developed for those systems have been effectively used in modern scalable systems. For example, Sinfonia [2] proposes the use of a constrained version of two phase commit (2PC) [16] in developing the *minitransaction* primitive that allows scalable distributed synchronization. These systems demonstrate that prudent and effective use of certain principles from earlier distributed systems can lead to scalable designs.

There has been a plethora of scalable data management system proposals for the cloud. Brantner et al. [7] propose a design of a database which utilizes a cloud based storage service like Amazon S3. Kraska et al. [23] build on this design to evaluate the impact of the choice of consistency of data when compared to the operational cost in terms of money and performance. They proposes a framework where consistency guarantees can be specified on data rather than on transactions. Lomet et al. [26, 27] propose a radically different approach towards scaling databases in the cloud by "unbundling" the database into two components, the transaction component and the data component. Das et al. [10, 11] propose a design of an elastic and scalable data management system for the cloud which is designed using the principle of *key-value* stores, but uses a partitioned database design for elasticity and transactional semantics. In the design of H-Store [21], the authors show how partitioning a main memory database, and replicating partitions at a number of nodes can be used to scale to large amounts of data in an enterprise cluster with huge memories and fast network.

Modern application requirements often require novel designs for data management systems. Yang et al. [33] propose a novel design of a data management system for supporting a large number of small applications in the context of various social networking sites that allow third party applications in their sites. The goal of the system is to provide RDBMS like functionality to the applications while catering to a large number of such applications, each of which is small enough to fit in a single node. Armbrust et al. [4] propose a scale independent data storage layer for social computing applications which is dynamically scalable and supports declarative consistency, while providing a scale aware query language. Evidently, different systems are targeted towards different application scopes and explore various design choices. Our design of G-Store is targeted to applications whose data has an inherent *key-value* schema, and requires scalable and transactional access to groups of *keys* which are formed dynamically by the applications.

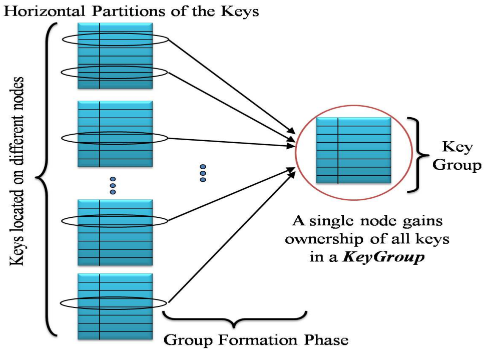

Horizontal Partitions of the Keys
Keys located on different nodes
Key Group
A single node gains ownership of all keys in a *KeyGroup*
Group Formation Phase

Figure 2: The Key Group abstraction allowing collocation of keys for efficient *multi key* accesses.

3. THE KEYGROUP ABSTRACTION

The Key Group abstraction provides a powerful yet flexible means for efficient *multi key* access and defines the granule for transactional access. This abstraction allows applications to select members of a group from any set of *keys* in the data store and dynamically create (and dissolve) groups on the fly, while allowing the data store to provide efficient, scalable, and transactional access to these groups of keys. At any instant of time, a given *key* can participate in a single group, but during its lifetime, a *key* can be a member of multiple groups. For example, consider the online casino application: at any instant of time, a player can participate in a single game instance, but over the course of time, a player can be part of multiple game instances with different players in different instances. *Multi-key* accesses are allowed only for *keys* that are part of a group, and only during the lifetime of the group. The system does not mandate that all *keys* in the data store be part of groups. At any instant, there can be multiple *keys* that are not part of any group, and can be accessed by the application using the *single key* semantics supported by the *Key-Value* store. Conceptually, they can also be viewed as groups with only a single member. For every formed group, there is a **leader** key (also referred to as the leader of the group) which is a *key* selected from amongst the members of the group, and the remaining members are referred to as the **follower** keys. The leader is part of the group's identity, but from an application's perspective, the semantics of operations on the leader is no different from that on the followers. In present *key-value* stores, even though multiple *keys* may be collocated at a particular node, their collocation is by *chance* and not by *choice* and hence are treated as independent entities. MegaStore [22] uses the *entity group* abstraction for providing transactional guarantees over the collocated keys which form a contiguous part of the key range. But as noted earlier, the static nature of *entity groups* as well as the contiguity requirement are often insufficient. The Key Group abstraction provides more flexibility to the application designers when compared to *Key-Value* stores or Google's MegaStore, while allowing efficient and scalable accesses.

Keys in a Key Group do not have any ordering restrictions, and hence the members of a *group* can be on different nodes. Thus, using a *key-value* store as-is would require distributed synchronization (such as 2PC [16]) to provide atomic and consistent accesses to these *keys*. The Key Group abstraction captures the relationship among the member keys of a group. Once this relationship is es-

[3] Scale up refers to using a more powerful server, also known as vertical scaling. Scale out refers to adding more servers and is also known as horizontal scaling.

tablished, *ownership* of the *keys* in a Key Group is collocated at a single node by *choice* in order to support efficient *multi key* access. Figure 2 provides an illustration of how the Key Group abstraction allows collocation of *ownership*. Each Key Group has an associated *leader*, and the node which *owns* the *leader* is assigned ownership of the group, and the nodes corresponding to the *followers yield* ownership to the *leader*, i.e., the node owning the *leader* key gains exclusive read and write access to all the *followers*. Once this transfer of ownership is complete, all read/write accesses for the members of a group is served by the leader which can guarantee consistent access to the keys without the need for any distributed synchronization. This process of *yielding* ownership to the *leader* during group formation, and obtaining the ownership back after a group is disbanded can be complicated in the presence of failures or other dynamic changes of the underlying system. We therefore design the Key Grouping protocol to guarantee correctness in the presence of failure. The group formation phase introduces some minimal overhead in the system. This design is therefore suited for applications which perform a large number of operations during the lifetime of the group such that the efficiency of the *multi key* accesses amortizes the group formation cost.

4. THE GROUPING PROTOCOL

Intuitively, the goal of the proposed Key Grouping protocol is to transfer key ownership safely from the *followers* to the *leader* during group formation, and from the *leader* to the *followers* during group deletion. **Safety** or **correctness** requires that there should never be an instance where more than one node claims ownership of an item. **Liveness**, on the other hand, requires that in the absence of repeated failures, no data item is without an owner indefinitely.

Group creation is initiated by an application client (henceforth referred to as client) sending a *Group Create request* with the *Group Id* and the members. Group formation can either be *atomic*, where the group is either created completely with all the keys joining or it fails with none of them joining; or *best effort*, where the group creation succeeds with whatever keys that joined. The Key Grouping protocol involves the *leader* of the group, which acts as the coordinator, and the *followers*, which are the *owners* of the keys in the group.[4] The *leader* key can either be specified by the client, or is automatically selected by the system. The group create request is routed to the node which *owns* the *leader* key. The leader *logs* the member list, and sends a *Join Request* ⟨J⟩ to all the followers (i.e., each node that *owns* a *follower* key). Once the group creation phase terminates successfully, the client can issue operations on the group. When the client wants to disband the group, it initiates the *group deletion* phase with a *Group Delete request*. In the steady state, transferring ownership of *keys* from followers to the leader of the group is similar to acquiring "locks", and the reverse process is similar to releasing the "locks". The Key Grouping protocol is thus reminiscent of the 2PL protocol [14] for concurrency control in transaction management. In the rest of the section, we discuss two variants of the Key Grouping protocol assuming different message delivery guarantees and show how group creation and deletion can be safely done in the presence of various failure and error scenarios. In our initial description of the protocol, we assume no failures. We then describe protocol operation in the presence of message failures and node failures. Node failure is dealt with using write ahead logging as discussed later, and we do not consider malicious node behavior.

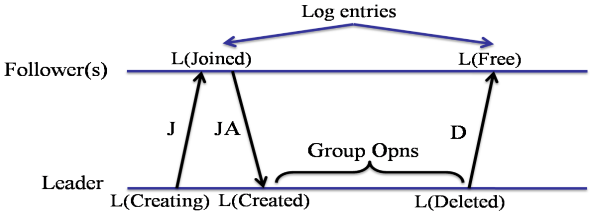


Figure 3: Key Grouping protocol with reliable messaging.

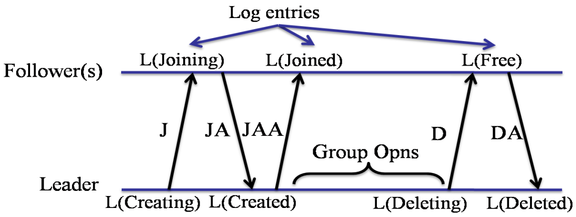


Figure 4: Protocol with unreliable messaging.

4.1 Protocol with Reliable Messaging

We first describe the protocol assuming reliable messaging such as TCP which ensures that messages are never lost, duplicated, or reordered. The leader requests for key *ownership* from the followers (*Join Request* ⟨J⟩), and depending on availability, ownership is either transferred to the leader, or the request is rejected (*Join Ack* ⟨JA⟩). The ⟨JA⟩ message indicates whether the follower key joined the group. Depending on the application's requirements, the leader can inform the application as members join the group, or when all the members have joined and the *group creation* phase has terminated. The group deletion phase is straightforward and involves notifying the followers with a *Delete Request* ⟨D⟩. Figure 3 provides an illustration of the protocol in a failure free scenario. If the client requested the group to be formed atomically, then if a ⟨JA⟩ message arrives where a follower did not join the group, the leader initiates group deletion.

4.2 Protocol with Unreliable Messaging

The Key Grouping protocol requires the transfer of a small number of short messages, and in the absence of failures, the protocol resembles a handshake protocol similar to a TCP connection setup. Therefore, using a protocol like TCP would be an overkill. We therefore develop a protocol assuming unreliable message delivery. The basics of the protocol still remain similar to that described in Section 4.1, but incorporate additional messages to deal with the possibility of message related failures.

Figure 4 illustrates the protocol with unreliable messaging which, in the steady state, results in two additional messages, one during creation and one during deletion. During group creation, the ⟨JA⟩ message, in addition to notifying whether a key is free or part of a group, acts as an acknowledgement for the ⟨J⟩ request. On receipt of a ⟨JA⟩, the leader sends a *Join Ack Ack* ⟨JAA⟩ to the follower, the receipt of which finishes the group creation phase for that follower. This group creation phase is two phase, and is similar to the 2PC protocol [16] for transaction commit. During group dissolution, the leader sends a *Delete Request* ⟨D⟩ to the followers. On receipt of a ⟨D⟩ the follower regains ownership of the key, and then responds to the leader with a *Delete Ack* ⟨DA⟩. The receipt of ⟨DA⟩ from all the followers ensures that the group has been disbanded.

Though extremely simple in the steady state, group formation

[4] In the rest of the paper, we will use the terms **leader** and **follower** to refer to the *keys* in the data store as well as the nodes where the *keys* reside.

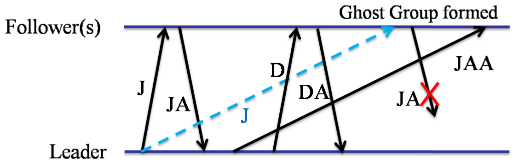


Figure 5: Complications arising from various types of message failures resulting in formation of ghost groups.

can become complicated in the presence of reordering and duplication of messages. For example, consider the pathological scenario in Figure 5 which has message failure, duplication, and reordering in a protocol instance where the *group delete phase* overlaps with the *group create phase*. The leader sends a ⟨J⟩ message to a follower and this ⟨J⟩ message is duplicated in the network due to some error (the duplicate message is depicted by a dashed line). The follower receives a copy of the ⟨J⟩ message, and replies with the ⟨JA⟩ message. On receipt of a ⟨JA⟩, the leader sends a ⟨JAA⟩ which is delayed in the network and not immediately delivered to the follower. In the mean time, the client requests deletion of the group, and the leader sends out the ⟨D⟩ message. The follower, before receiving the ⟨JAA⟩, responds to the ⟨D⟩ message with a ⟨DA⟩ message, and disbands the group. After a while, the follower receives the duplicated ⟨J⟩ message for the deleted group, and sends out a ⟨JA⟩ for the group which is lost in the network. The follower then receives the delayed ⟨JAA⟩ and its receipt completes a group creation phase for the follower. In this pathological scenario, the follower has yielded control to a non-existent group, or a **Ghost Group**. Formation of *ghost groups* can be a severe problem for the *liveness* of the system, since a *key*, which is now part of a *ghost group*, becomes unavailable. Therefore, the protocol must have some mechanism for detecting and dealing with *duplicated* and *stale* messages.

We now describe the Key Grouping protocol in detail, and then explain how various failure scenarios such as message loss, reordering, and duplication are handled. The protocol is designed to be oblivious of the dynamics of the underlying *key-value* store such as node failures or *key* reassignment. This is achieved by identifying the followers using the *keys* rather than the physical identity of the nodes. The *key-value* store maintains a persistent mapping of the *keys* to the nodes *owning* them, and the protocol uses this information to communicate with the follower nodes. We now describe the actions taken by the nodes on receipt of the different protocol messages, and describe the handling of lost, reordered, and duplicated messages later in the section.

Group creation phase

Group create request. On receipt of a *group create request* from the client, the leader verifies the request for a unique *group id*. The leader *logs* this request and the list of members in the group, and sends a ⟨J⟩ request to each of the participants. This ⟨J⟩ request is retried until the leader receives a ⟨JA⟩ from all the followers.

Join request ⟨J⟩. On receipt of a ⟨J⟩ request at a follower, the follower ascertains the freshness and uniqueness of the message to deal with duplication of messages. If the message is a duplicate, then the follower sends a ⟨JA⟩ without any *logging* of the action. If the follower key is not part of any active group, this message is *logged*, and the follower replies with a ⟨JA⟩ message notifying its intent to *yield*. To deal with spurious ⟨JAA⟩ messages and eliminate the problem of *ghost groups*, the follower should be able to link the ⟨JAA⟩ to the corresponding ⟨JA⟩. This is achieved by using a sequence number generated by the follower referred to as the *Yield Id*. This *yield id* is copied into the ⟨JA⟩ message along with the *group id*. The ⟨JA⟩ message is retried until the follower receives the ⟨JAA⟩ message, and this retry deals with the loss of ⟨JA⟩ and ⟨JAA⟩ messages as well as deleting *ghost groups*.

Join Ack ⟨JA⟩. On receipt of a ⟨JA⟩ message, the leader checks the *group id*. If it does not match the ids of any of the currently active groups, then the leader sends a ⟨D⟩ message, and does not *log* this action or retry this message. Occurrence of this event is possible when the message was a delayed message, or the follower yielded to a delayed ⟨J⟩. In either case, a ⟨D⟩ message would be sufficient. If the *group id* matches a current group, then the leader sends a ⟨JAA⟩ message copying the *yield id* from the ⟨JA⟩ to the ⟨JAA⟩ irrespective of whether the ⟨JA⟩ is a duplicate. If this is the first ⟨JA⟩ received from that follower for this group, a log entry is added to indicate that the follower has joined the group. The ⟨JAA⟩ message is never retried, and the loss of ⟨JAA⟩ messages is handled by the retries of the ⟨JA⟩ message. The receipt of ⟨JA⟩ messages from all the followers terminates the group creation phase at the leader.

Join Ack Ack ⟨JAA⟩. On receipt of a ⟨JAA⟩ message, the follower checks the *group id* and *yield id* to determine freshness and uniqueness of the message. If the *yield id* in the message does not match the expected *yield id*, then this ⟨JAA⟩ is treated as a spurious message and is ignored. This prevents the formation of *ghost groups*. In the scenario shown in Figure 5, the delayed ⟨JAA⟩ will have a different *yield id* which will allow the follower to reject it as a spurious message. If the message is detected to be unique and fresh, then the follower marks the *key*'s state as *yielded*, *logs* this event, and completes the group creation process for the follower.

Group deletion phase

Group delete request. When the leader receives the *group delete* request from the application client, it *logs* a delete entry and initiates the process of *yielding* ownership back to the followers. It then sends a ⟨D⟩ message to each of its followers in the group. The ⟨D⟩ messages are retried until all ⟨DA⟩ messages are received. At this point, the group has been marked for deletion, and the leader will reject any future accesses of the *keys* in this group.

Delete request ⟨D⟩. When the follower receives a ⟨D⟩ request, it validates this message, and on successful validation *logs* this message signifying that it has regained ownership of the key. Irrespective of whether this ⟨D⟩ message was duplicate, stale, spurious, or valid – the follower will respond to a ⟨D⟩ message with a ⟨DA⟩ message, and this ⟨DA⟩ message is not retried.

Delete ack ⟨DA⟩. On receipt of a ⟨DA⟩ message from any follower, the leader again checks for the validity of the message. If this is the first message from that follower for this group, and the *group id* corresponds to an active group, then this operation is *logged* indicating that the ownership of the *key* has been successfully transferred back to the follower. Once the leader has received a ⟨DA⟩ from all the followers, the group deletion phase terminates.

Message loss, reordering, or duplication

These errors can occur due to a variety of reasons: network errors, partitions, and node failures to name a few. As a result, any protocol message can be lost, delayed, or duplicated, and the Key Grouping protocol is designed to be resilient to these errors. Lost messages are handled by associating timers with message transmissions, and retrying a message if an acknowledgement is not received before the timeout. For messages which do not have an associated acknowledgement (for example the ⟨JAA⟩ and ⟨DA⟩), the sender relies on the recipient of the message to retry having not received the message, since those messages themselves are acknowledgements. As an example, for the ⟨JAA⟩ message, the follower is expecting the ⟨JAA⟩ as an acknowledgement of the ⟨JA⟩ message. Therefore, if a $\langle$JAA$\rangle$ message is lost, the follower will retry the $\langle$JA$\rangle$ message, on receipt of which, the leader will resend the $\langle$JAA$\rangle$. The loss of $\langle$DA$\rangle$ is handled in a similar way.

The Key Grouping protocol uses the following techniques to deal with reordering and duplication of messages. *First*, the protocol uses *Unique Ids* for all messages to detect duplicate and reordered messages. All the messages of the group have the *group id*, and the ordering of the messages within the group creation phase is achieved using a *yield id* which is unique within a group. Referring back to the scenario of Figure 5 where *ghost groups* are formed due to stale and duplicate messages, the follower uses the *yield id* to detect that the $\langle$JAA$\rangle$ is stale, and rejects this message. As a result, it will retry the $\langle$JA$\rangle$ message after a timeout. On receipt of the $\langle$JA$\rangle$ at the leader, it determines that this message is from a deleted or stale group, and replies with a $\langle$D$\rangle$ message, which in turn *frees* the key that was part of the *ghost group*. It is intuitive that the loss of messages does not hurt since the messages will be retried by the follower. Therefore, *group id* and *yield id* together empower the protocol to deal with duplicate and reordered messages.

Concurrent group creates and deletes

Based on the requirements of the application, or as a result of repeated failures resulting in an application timeout, a *leader* might receive a *Group Delete request* from the application before *group creation* has terminated. The Key Grouping protocol also handles such a scenario. For the *followers* that had received the $\langle$J$\rangle$ and/or $\langle$JAA$\rangle$, this is equivalent to a normal delete and is handled the same way a $\langle$D$\rangle$ is handled during normal operation. Special handling is needed to deal with the followers which have not received the $\langle$J$\rangle$ message yet as a result of message loss or reordering. For those followers, the $\langle$D$\rangle$ message is for a group about which the follower has no knowledge. If the follower rejects this message, the leader might be blocked. To prevent this, the follower sends a $\langle$DA$\rangle$ to the leader, but since it is not aware of the group, it cannot maintain any state associated with this message (maintaining this information in the log prevents garbage collection of this entry). Now when the follower receives the $\langle$J$\rangle$ request, it has no means to determine that this is a stale message, and thus accepts it as a normal message and replies with a $\langle$JA$\rangle$. This scenario is equivalent to the formation of *ghost groups* and is handled in a similar manner.

4.3 Recovery from Node Failures

G-Store is designed to operate on a cluster of commodity nodes, and hence, it is designed to be resilient to node failures common in such infrastructures. Similar to relational databases, *write ahead logging* [32] of critical protocol actions is used to deal with node failures, and to aid recovery and guarantee durability. A node *logs* all critical group related operations (such as status of groups for which it is the leader, status of its *keys*, and so on). During recovery from a failure, this *log* is replayed to recreate the state of the node prior to the crash, and resume the disrupted group operations. This includes restoring the state of the groups which were successfully formed, as well as restoring the state of the protocol for the groups whose creation or deletion was in progress when the node crashed. The write ahead log is maintained as a part of the protocol state, and is separate from any log used by the underlying *key-value* store.

In addition to the state of the protocol and the groups, maintaining the unique identifiers such as the *group id* and the *yield id* are also critical for protocol operation, and hence this state should also be recovered after a failure. During normal operation, the protocol ensures that the *yield id* is monotonically incremented, and the highest known *yield id* is logged with the last $\langle$JA$\rangle$ message sent. Therefore, after recovery, the node starts at a *yield id* larger than that obtained from the log. The *group id* is a combination of a unique id and the leader key, and this information is also logged during normal operation of the protocol and hence is recovered. Hence, after recovery is complete, the entire state of the protocol is restored to the point before the crash, and the node can resume normal operation.

The Key Grouping protocol ensures *group safety* in the presence of arbitrary failures. But there are various availability tradeoffs in the presence of arbitrary failures. For example, let us consider the case when a follower has yielded control of a *key* k to a *ghost group* due to the receipt of a delayed $\langle$J$\rangle$ request. Once a *key* k joins a *ghost group*, even though k should be available to be accessed independently or to join another group (since the group to which k joined is already deleted), k cannot be accessed since k's status indicates that it is part of a group. It is only when the $\langle$D$\rangle$ message is received from the *leader* of the ghost group that k becomes available. Therefore, deletion of *ghost groups* is dependent on the availability of the *leader*. If the *leader* crashes or cannot communicate with the node owning k, then k becomes unavailable for an extended period of time, even though the node on which k resides is available. k becomes available again once the leader's state is recovered and the ghost group deleted. This scenario would not be possible in a simple *Key-Value* store.

The Key Group abstraction also allows for extended availability in the presence of certain types of failures. For example, if we consider a *key* k' which is part of a group G. As a result of group formation, the *ownership* of a *key* is migrated to the *leader* which is responsible for all accesses to the *key* while caching the recent updates. Therefore, after group formation is completed, if the node responsible for a follower crashes, the *leader* can still continue serving k' until G is deleted. Clearly, in a normal *Key-Value* store, the failure of the node on which k' resides would make k' unavailable, while in G-Store, k' is available as long as G is alive.

5. IMPLEMENTATION OF G-STORE

In this section, we explain the implementation of G-Store using the Key Grouping protocol and a *key-value* store as a substrate. In this section, we use the two variants of the Key Grouping protocol described in the previous section to provide two implementation designs of G-Store. The first approach uses the reliable messaging variant for a client based implementation where the grouping protocol is executed as a layer outside the data store, and all accesses to the data store are handled by this layer. We refer to this design as a *client based implementation* since the grouping layer is a client to the *Key-Value* store. This approach is suitable as a pluggable grouping library which can be made compatible with multiple underlying *key-value* stores, or in the case where the *key-value* store is part of an infrastructure and cannot be modified. The second design involves extending a *key-value* store to implement the Key Grouping protocol as a *middleware* inside the *Key-Value* store, with the middleware handling the grouping specific operations and the *Key-Value* store handling the data storage. This approach is suited for cases where the infrastructure provider is implementing a data store which supports *multi key* accesses and has access to the code base for the *key-value* store, and both variants of Key Grouping protocol can be used to this design.

The operations supported by G-Store are derived from the operations supported by the underlying *Key-Value* store's operations which generally amount to reads/scans, writes, or read/modify/update of a single row of data. In addition to these operations, G-Store introduces the concept of transactions involving one or more operations. Any combination of read, write, or read/modify/update operations on keys of a group can be enclosed in a transaction, and G-Store ensures the ACID properties.

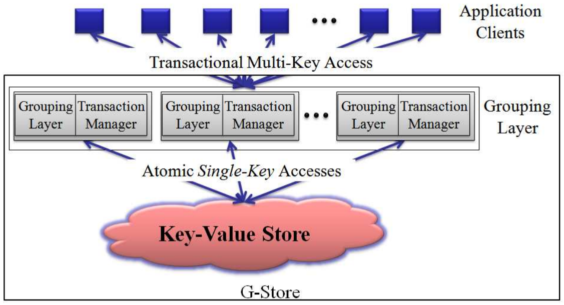


Figure 6: A client based implementation of G-Store.

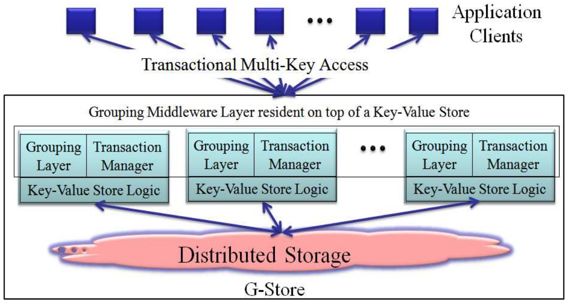


Figure 7: A middleware based implementation of G-Store.

In our implementation of G-Store, we consider *Key-Value* stores such as Bigtable [8] and HBase [17] where replication of data is not visible at the data store level, and hence the data store provides strong consistency guarantees on the data. Therefore, the transactions on a group can also guarantee strong consistency on data accesses. In data stores such as PNUTS [9] and Dynamo [12], the *key-value* pairs are replicated with replication handled by the data store, and can be configured to support weaker consistence of reads. The interplay of strong consistency guarantees of transactions on the group with weaker consistency guarantees of the data store can result in weaker consistency and isolation guarantees of transactions on a group – and we leave this analysis as future work. In the rest of this section, we first describe these two implementation designs, and then describe some implementation issues which are common to both these designs.

5.1 Client based Implementation

This implementation consists of a client layer resident outside the *key-value* store which provides *multi key* access guarantees using the *single key* accesses supported by the underlying *key-value* store. Figure 6 provides an illustration of this design. The application clients interact with the *grouping layer*, which provides a view of a data store supporting transactional *multi key* accesses. The grouping layer treats the *Key-Value* store as a black box, and interacts with it using the client APIs supported by the store. Very little state about the groups is maintained in the middleware, so that failures in the grouping layer need minimal recovery. Since the client API is invariably based on TCP, this design uses the variant of Key Grouping protocol which leverages reliable messaging.

In order to allow a scalable design, as depicted in Figure 6, the grouping layer can be potentially distributed across a cluster of nodes. Therefore, the grouping layer must execute the Key Grouping protocol so that the *ownership* of the *keys* is located at a single node. In case when the grouping layer is distributed, the *key* space is horizontally partitioned. Our implementation uses range partitioning, but other partitioning schemes such as hash partitioning can also be supported. On receipt of a *group create* request from the application client, the request is forwarded to the node in the grouping layer which is responsible for the *leader* key in the group. That node now executes the Key Grouping protocol and acquires ownership for accesses to the group. In this design, the *followers* are the actual *keys* which are resident in the *Key-Value* store, and the ⟨J⟩ request is equivalent to requesting a "lock" on the *key*. The locking of the *key* is performed by introducing a new column in the table which stores the locking information, and the ⟨J⟩ request is a *test and set* operation on the locking attribute to test availability. [5] If the *key* is available, then the *test and set* operation atomically obtains the lock and stores the group id. Since the *test and set* operation is on a *single key*, all *Key-Value* stores support this operation. Once the join request has been sent to all the *keys* in the group, the group creation phase terminates.

Once group creation has successfully completed, the grouping layer caches the contents of the *keys* and executes transactions on the group. More details about transaction execution, efficient log management, and update propagation can be found later on in the section. In an application deployment platform like Google AppEngine, the application code is executed on a per invocation basis and is stateless, and thus all the state is stored in the data store. In such a design, the benefits of caching cannot be leveraged, and the application code has to execute the transaction and propagate the updates immediately to the data store. But in other platforms like Amazon EC2 clients using SimpleDB as a service, the client layer deployed in EC2 can benefit from caching.

5.2 Middleware based Implementation

In this section, we provide a design of G-Store where the Key Grouping protocol and the group related operations are implemented as a middleware layer conceptually resident on top of a *Key-Value* store. In this implementation, we select a *Key-Value* store design similar to that of Bigtable [8] and use its open-source clone HBase [17] where the entire key space is horizontally range-partitioned amongst the nodes in the system, referred to as a *tablet server* in Bigtable terminology, and each server is responsible for certain key ranges. The grouping middleware logic is conceptually collocated with the *tablet server* logic, and the layer is responsible for all the key ranges which the collocated *tablet server* is serving. Data replication for fault-tolerance is handled in the distributed storage level, and hence replication is not visible at the data store level. Figure 7 provides a high-level overview of G-Store in this middleware based design. The *grouping layer* is part of the *middleware* and executes the Key Grouping protocol. This allows the protocol messages to be batched for all the collocated keys. This layer provides the abstraction of Key Groups, and using this abstraction, a *transaction manager* provides access to the *keys* while guaranteeing different levels of consistency, isolation, and functionality. The client interface of G-Store consists of a library which exposes a *group* based access interface, and an access request from the application is marshalled to the appropriate physical node that is the *leader* of the group.

In this design, the grouping layer maintains the state of the group, and the *key-value* store does not need to know about the existence of the groups. All the protocol logging operations are done in the grouping layer, and the grouping layer is also responsible for recov-

[5] Some *Key-Value* stores (such as HBase [17]) provide a client API for acquiring *row locks*. We do not use this API since it is a blocking operation in some implementations, and is also a data store specific solution.

ery of the group states in the presence of failures. In this design, both variants of the Key Grouping protocol can be used for group formation, and the choice is dependent on the designer of the system. Once the group creation request arrives at the *leader* node, it sends the ⟨J⟩ message to the *followers* which are the grouping layer peers on the *tablet servers* serving the follower *keys*. The grouping layer logic at each node keeps track of the *keys* which it is responsible for and are part of existing groups. This information is updated with every group creation and deletion request. On termination of the Key Grouping protocol and successful creation of the group, the transaction manager maintains a cache of the contents of the group members so that transactions on the group can be executed locally. More details about transaction management, efficient log management, and update propagation can be found later in the section. Since the G-Store design does not mandate all keys to be part of groups, there can be accesses to *keys* that are not part of any group. All such accesses are *single key* accesses, and are delegated to the *Key-Value* store. The middleware based design is also minimally intrusive to the design of the *Key-Value* store, and requires very little modification to the code of the *Key-Value* stores.

5.3 Transaction Management

The Key Grouping protocol ensures that the *ownership* of the members of a group reside with a single node. Therefore, implementing transaction management in a group does not require any distributed synchronization and is similar to transaction management in single node relational databases. This allows us to leverage matured and proven techniques from more than two decades of database research [6, 32] for transaction management on a group. Since transactions are guaranteed to be limited to the boundaries of a single node, this design follows the principles of scalable system design [1, 19] and allows G-Store to provide efficient and scalable transactional access within a group.

In our implementation of G-Store, we use Optimistic Concurrency Control (OCC) [24] for guaranteeing serializable transactions. OCC obviates the requirements of acquiring locks and is preferred in scenarios where there is less contention for the resources. But as noted earlier, since transaction management in G-Store is no different than that supported in RDBMS, other techniques such as Two Phase Locking [14] and its variants [6, 32], or Snapshot Isolation [5] can also be implemented, and we leave this as future work. Atomicity and durability of transactions is achieved through *write ahead logging* similar to that used in databases [32]. Since the leader *owns* access to the group members, the updates are *asynchronously* applied to the underlying *key-value* store. Note that since the Key Group abstraction does not define a relationship between two groups, groups are independent of each other, and the transactions on a group guarantee consistency only within the confines of a group.

The Key Group abstraction allows efficient execution of ACID transactions on a group while allowing the system to scale efficiently through horizontal partitioning, a design which forms the core of scalable *Key-Value* stores. G-Store is analogous to RDBMSs which rely on atomic disk reads and disk writes at the granularity of pages to implement transactions spanning higher level logical entities such as tables, while G-Store uses atomic access at a *single key* granularity provided by the underlying *Key-Value* store. The only difference is that in G-Store, transactions are limited to smaller logical entities called *groups*. If the application does not require transactional access, and only needs atomicity of the updates on a group of keys, then instead of a transaction manager, an *atomic updater*, which guarantees only atomicity and durability of updates, can be used along with the grouping layer.

5.4 Efficient Logging and Update propagation

During the lifetime of a group, all accesses to the members of the groups are guaranteed to be through the leader, and the leader has exclusive access to the members of the group. The leader therefore caches the contents of the group members as well as the updates made as a result of the transactions on the group. This obviates the need for expensive distributed commitment protocols like 2PC [16] when a transaction on a group is being executed, even though the *keys* in a group can be physically located on multiple nodes. This results in efficient transaction execution and reduction in execution latency. But the updates need to be asynchronously propagated to the actual data store. G-Store deals with this using an **update shipper** which is responsible for asynchronously propagating the updates to the participating *keys*.

All updates to a group are appended to a *write ahead log* which is similar to a commit log in a database. Conceptually, each group can have a separate *logical* log, but for performance as well as efficiency considerations, all the groups collocated on the same server share the same physical log. The *leader* executes the group operations, logs them, and updates its local cache which stores the data corresponding to the members of the group. Once the updates are logged, the application client's request is complete and is acknowledged, even though the update has not been applied to the actual data store. In our implementation, the write ahead log is maintained in a distributed file system (DFS) [18] which guarantees persistence of the log entries beyond a failed *leader*. To ensure that updates from the operations on a group are visible to independent accesses after a group has been deleted, before a *leader* processes a *group deletion request* and sends out the ⟨D⟩ message, it should ensure that the *update shipper* has shipped all cached updates of the group to the followers, and the data in the store is up-to-date.

Since the write ahead log is stored in a DFS (Hadoop DFS [18] in our implementation), individual log writes are expensive. Therefore for efficiency, the *log shipper* leverages from batched updates similar to group commits in a database [16, 32]. Commits and hence log writes are processed in batches, and this results in significant throughput enhancement of the logging operation. This further justifies the design of combining the logical logs per group into a single physical logs per server which allows for more operations per batch and hence results in higher throughput. Additionally, as observed in Bigtable [8], when writing to a DFS, a single logger thread might become a performance bottleneck. We therefore use a design where multiple logging threads writing to independent files are used to improve the performance of the logger.

6. EXPERIMENTAL EVALUATION

We now experimentally evaluate the performance of the Key Grouping protocol and G-Store on a cluster of machines in the Amazon Web Services Elastic Compute Cloud (EC2) infrastructure. Both the Key Grouping protocol as well as G-Store have been implemented in Java. We use HBase version 0.20.2 [17] as the *Key-Value* store in both the client based and middleware based designs. We evaluate the performance of the system using an application benchmark which resembles the type of workloads for which the system has been designed. We first describe the benchmark used for the experiments and the experimental set up used for evaluating the system, and then provide results from the experiments.

6.1 An Application Benchmark

The application benchmark used is based on an online multi-player gaming application. The participants in the game have a profile which is represented as a *key-value* pair, where a unique id corresponding to a user is the *key*. The benchmark consists

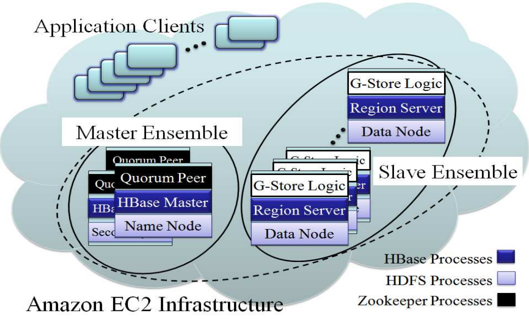


Figure 8: Cluster Setup for the experiments.

of a single *Players* table that contains information corresponding to all the players in the game. Note that even though the benchmark has only a single table, G-Store is designed to support *multi key* operations over multiple tables resident in the *Key-Value* store. Since *Key-Value* stores do not support joins or foreign key relations, from the protocol's perspective, having a single table is no different from multiple tables. In this benchmark, the number of concurrent groups and the average number of *keys* in a group are used for scaling the size of the *Players* table. If there are $\mathcal{G}$ concurrent groups with an average of $\mathcal{K}$ *keys* per group, then the *Players* table must contain at least $n \times \mathcal{G} \times \mathcal{K}$ keys, where n is the scaling factor used to determine the number of keys which are not part of any group. The application clients submit group create requests which contain the members of the group. The members of a group are chosen arbitrarily, and the number of members in the group are chosen from a normal distribution with $\mathcal{K}$ as the mean and $\sigma_{\mathcal{K}}$ as the standard deviation. Once group creation succeeds, the client issues operations on the group. The number of operations on a group is chosen from a normal distribution with mean $\mathcal{N}$ and standard deviation $\sigma_{\mathcal{N}}$. Each client submits an operation on a group, and once the operation is completed, the client waits for a random time between δ and 2δ before submitting the next operation. On completion of all operations, the group is deleted, and the client starts with a new group create request. A client can have concurrent *independent* requests to multiple groups, and a group can also have multiple clients. Since G-Store provides isolation against concurrent operations on a group, clients need not be aware of the existence of concurrent clients.

The parameters in the benchmark – the number of concurrent groups $\mathcal{G}$, the average number of keys $\mathcal{K}$, the scaling factor n. the average number of operations on a group $\mathcal{N}$, and the time interval between operations δ– are used to evaluate various aspects of the system's performance. The first three parameters control the scale of the system, while $\mathcal{N}$ controls the amortization effect which the group operations have on the group formation and deletion overhead. On the other hand, the parameter δ allows the clients to control the stress on the grouping layer for performing concurrent operations.

6.2 Experimental Setup

Experiments were performed on a cluster of commodity machines in Amazon EC2. All machines in the cluster were "High CPU Extra Large Instances" (`c1.xlarge`) with 7 GB of memory, 20 EC2 Compute Units (8 virtual cores with 2.5 EC2 Compute Units each), 1690 GB of local instance storage, and a 64-bit platform. Figure 8 provides an illustration of the experimental cluster setup. HBase uses HDFS [18] as a fault tolerant distributed file system, and we use HDFS version 0.20.1 for our experiments. HBase also uses Zookeeper (`http://hadoop.apache.org/zookeeper`) for distributed lease management. In our experiments, we used Zookeeper version 3.2.2. Since the health of Zookeeper is critical for the proper functioning of HBase, an ensemble of servers is used for running Zookeeper so that Zookeeper's state can be replicated on multiple nodes. In an ensemble of $(2n + 1)$ nodes, Zookeeper can tolerate up to n node failures. In our configuration, we used a 3 node Zookeeper ensemble; we refer to these machines as the Master Ensemble. The nodes in the Zookeeper ensemble also host the Master of the HBase installation, as well as the NameNode and Secondary Namenode of the HDFS installation. The HBase Master and HDFS Namenode are responsible for the metadata operations, and hence are not in the data path and therefore are lightly loaded. A separate ensemble of nodes, referred to as the Slave Ensemble, is used to host the slave processes of HBase and HDFS: namely the Region Servers of HBase which serve the regions or partitions of the tables, and the Data Nodes of HDFS which store the filesystem data. The Slave Ensemble also hosts the G-Store logic for the implementation of the Key Grouping protocol and the operations on the group. The number of nodes in the slave ensemble was set to 10, which amounts to about 70 GB of main memory, 16.15 TB of disk space, and 80 virtual CPU cores. Together the Master and the Slave Ensembles constitute the data management infrastructure. Application client processes were executed on a different set of nodes, but under the same infrastructure.

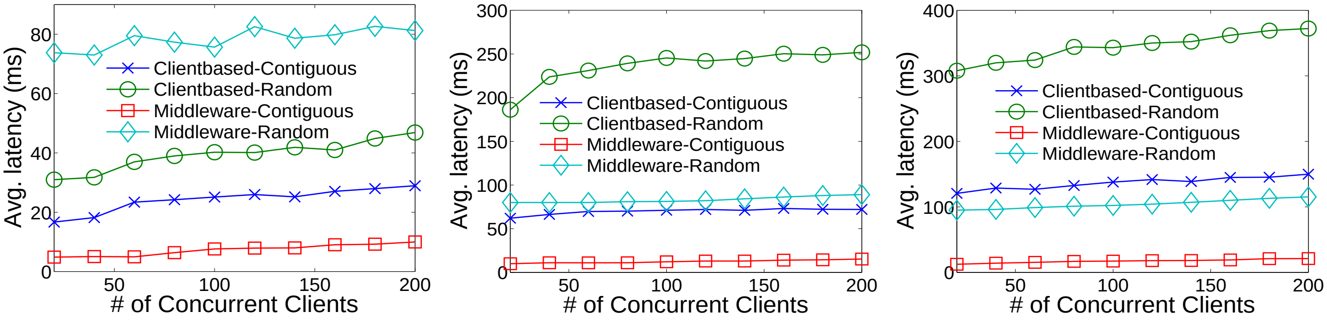
The application benchmark used for our experiments consists of a single table (*Players*) which stores the profile information of the players participating in the game. Each row is identified by a unique key which corresponds to a player's id. The *Players* table has 25 column families (see the data model of tables in Bigtable [8] or HBase [17]) which stores information corresponding to various profile related information such as: *Name, Age, Sex, Rating, Account Balance* etc. In our experiments, each column family contains a single column. The table was loaded with about 1 billion rows of randomly generated data, and each cell (a specific column family in a row) contains about 20–40 bytes of data, and the size of the table on disk without any compression was close to a terabyte (with 3× replication in the distributed storage layer). After the insertion of data, the cluster is allowed to repartition and load balance the data before any group related operations are performed. This allows an even distribution of the workload across all nodes in the cluster. During the progress of the group operations, the table is updated, but no bulk insertions or deletions are performed. Our implementation of Key Grouping protocol is *non-blocking*, i.e., if a key is part of a group, any concurrent $\langle \mathtt{J} \rangle$ request for that key will be rejected. Furthermore, group formation was set to *best effort* and succeeds when a response was received from all the followers irrespective of whether they joined. Most of the default configurations of HBase and HDFS were used for our experiments. Overridden and additional configurations are provided in Table 1.

6.3 Evaluating Group Creation

In this section, we experimentally evaluate the latency and throughput of group formation under various types of loads. This will provide a measure of the scalability of the system in dealing with hundreds of thousands of group create requests. The group formation only workload is simulated by setting the number of operations on the group ($\mathcal{N}$) to 0 so that the application client creates the group, and then deletes the group once group creation succeeds. We evaluate the two variants of the design described in Section 5: the client based design uses the reliable messaging variant of Key Grouping protocol while the middleware based design is based on the variant using unreliable messaging. In addition, we also evaluate two

Parameter	Description	Value
`hbase.regionserver.handler.count`	Number of handlers in the region server	25
`hbase.regionserver.flushlogentries`	Number of log entries to accumulate before flushing	100
`group.numprotocolworkers`	Number of worker threads in the grouping layer	25
`dfs.support.append`	Support appends to HDFS files	true
`JAVA_HEAP_SIZE`	The maximum size (MB) of the heap of each process	2000

Table 1: Experimental Configuration.



(a) 10 keys per group.

(b) 50 keys per group.

(c) 100 keys per group.

Figure 9: Latency of group create requests.

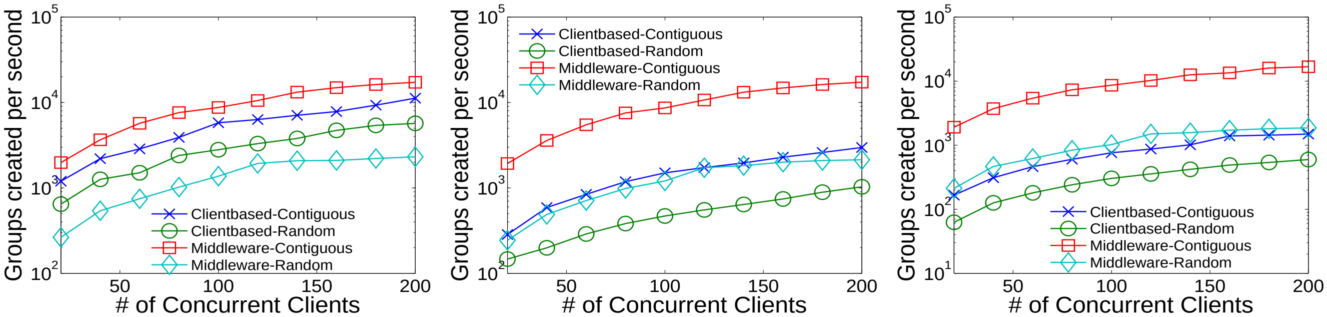
different ways of selecting keys that form a group: a *contiguous* selection where the keys in a group are selected such that they form a range of keys; and *random* selection, where the keys are selected in any arbitrary order. The *contiguous* selection is reminiscent of Google MegaStore's *entity group* abstraction [22] which mandates that keys in an entity group are from a contiguous key space. The *random* selection demonstrates the flexibility of the Key Group abstraction that allows the keys in a group to be arbitrarily selected.

Figure 9 plots the latency of group create requests as the number of concurrent clients submitting requests increases. The latency of group creation is measured as the time elapsed since the client issued the group create request and obtained the group creation notification from the grouping layer. The latency (in ms) is plotted along the y-axis, and the number of concurrent clients is plotted along the x-axis. The different lines in the graphs correspond to the variants of the protocol implementation and the different choices of key selection for a group, and the different plots correspond to different sizes of the groups (varying parameter $\mathcal{K}$). It is evident from the figures that for considerable group sizes, the middleware based approach outperforms the client based approach. Additionally, when the keys in a Key Group are from a contiguous space, the performance of the protocol is considerably improved. The middleware based approach performs better owing to the benefits of batching the protocol messages directed to the same node. In the case where the keys are contiguous, group creation often results in a single $\langle \mathtt{J} \rangle$ message. On the contrary, in the client based design, irrespective of whether the keys are contiguous or not, multiple messages are sent across the network resulting in higher latency. This is because the *Key-Value* store supports only single key *test and set* operation which is used to implement the "lock" or followers *yielding* control to the leader. Another interesting observation is that the middleware based design is not very sensitive to the number of keys in a group. This is again due to the batching of the requests, which results in fewer messages; if there are n nodes in the cluster, there can be at most n messages irrespective of the size of the group. On the other hand, the performance of the client based design is dependent on the number of keys in the group. Therefore, for most practical applications, the middleware based implementation is the design of choice.

Figure 10 provides more evidence of the superiority of the middleware based design. In this figure, we plot the throughput of group create requests served per second, aggregated over the entire cluster. Along y-axis is the number of group create requests processed per second (drawn in a logarithmic scale), and along the x-axis is the number of concurrent clients. Figure 10 also demonstrates the almost linear scalability in terms of the group create requests being served – as the number of clients increase from 20 to 200, an almost 10 times increase in group creation throughput is observed. Furthermore, a 10 node G-Store cluster is able to processes tens of thousands of concurrent groups.

6.4 Evaluating Operations on a Group

In this section, we evaluate the performance of G-Store for a realistic workload where operations are performed on a group once the group creation succeeds. We use the application benchmark described earlier in this section to simulate load of an online multiplayer gaming application. Once a group has been formed, execution of operations on a group will be similar in both the client based as well as middleware based designs. In this section, we evaluate the performance of both the alternative system designs. In these experiments, we compare the performance of G-Store to that of HBase in terms of the average latency of the operations. For G-Store, the reported average latency includes the group formation and deletion times, in addition to the time taken by the operations. Since HBase does not have any notion of groups, and no transactional guarantees on *multi-key* accesses; HBase performance numbers provide a baseline to measure the overhead of introducing transactional guarantees based on the Key Group abstraction when compared to accesses without guarantees. When using G-Store, updates are associated to a group, and G-Store will ensure the transactional guarantees for these accesses. For HBase, the updates to a group of keys are applied as a batch of updates to the rows which are part of the group, and no transactional guarantees are provided for these accesses. We experimentally evaluate the performance of

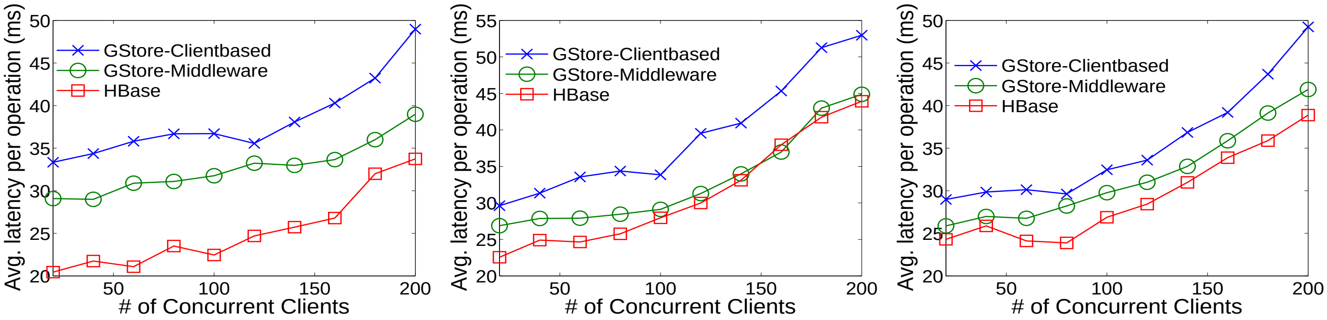


(a) 10 keys per group.

(b) 50 keys per group.

(c) 100 keys per group.

Figure 10: Throughput of group create requests.



(a) 10 operations per group created.

(b) 50 operations per group created.

(c) 100 operations per group created.

Figure 11: Average latency per operation on a group. The HBase workload simulates grouping operations by issuing operations only limited to a group of keys – neither are groups formed actually, nor are there any guarantees on the group operations in HBase.

the system for different values of $\mathcal{N}$ – the number of operations on a group. We set the number of keys in a group ($\mathcal{K}$) to 50, and the sleep interval δ between consecutive operations to 10 ms.

Figure 11 plots the average latency of each operation, as the number of operations on a group are varied. In all the experiments, the number of concurrent clients are varied from 20 to 200. The x-axis plots the number of concurrent clients, while the y-axis plots the average latency per operation (in ms). The average latency of operations in GStore includes the latency of group creates and is computed as:

$$\frac{\text{Group Create Latency} + \text{Latency of operations}}{\text{Number of operations}}$$

while for HBase, the latency is the average over all operations. As is evident from Figure 11, the grouping layer introduces very little overhead (10–30%) when compared to the baseline HBase implementation while providing ACID guarantees on *multi-key* accesses which the baseline implementation does not provide. The overhead is considerably reduced as the number of operations per group increases. This reduction in overhead is primarily due to the benefits of batching of the updates which the Key Group abstraction enables, which in turn amortizes the cost of group formation and maintenance of grouping specific meta information and transaction management. Even though updates in G-Store are first processed by the *grouping layer* and then transmitted to the data store (compared to direct application of updates in case of HBase), the asynchronous propagation of updates from the grouping layer to the data store (see Section 5.4) does not affect the latency of the updates. Therefore, for a very low overhead, G-Store provides transactional *multi key* accesses using a *Key-Value* store as a substrate.

7. CONCLUSIONS AND FUTURE WORK

A plethora of modern data intensive applications pose novel challenges to large scale data management systems for cloud computing infrastructures. We proposed a novel Key Group abstraction and the Key Grouping protocol that allows extending scalable *Key-Value* stores to efficiently support transactional *multi key* accesses using only the *single key* access guarantees supported by *Key-Value* stores. We demonstrated how the proposed Key Grouping protocol can be used for the design of G-Store. G-Store is suitable for applications that require access to non-disjoint groups of *keys* which are dynamic in nature. We proposed two design variants and experimentally evaluated the designs in a cloud computing infrastructure using an application benchmark that resembles the type of applications for which G-Store has been designed. Our evaluation showed that G-Store preserves the scalability of the underlying *Key-Value* store, handles thousands of concurrent group creations, and provides a low overhead means to supporting transactional *multi key* operations.

In the future, we would like to explore the implications of the Key Grouping protocol in the presence of analytical workloads and index structures built on *Key-Value* stores. We would also like to explore the feasibility of the design of G-Store using *Key-Value* stores such a Dynamo and PNUTS where the data store spans multiple data centers and geographical regions, and supports replication and weaker consistency guarantees of reads, and evaluate the ramifications of the weaker consistency guarantees of the data store on the consistency and isolation guarantees of transactions on groups.

Acknowledgements

The authors would like to thank Shyam Antony for the initial discussions during the design of the Key Grouping protocol, the anonymous reviewers, and Pamela Bhattacharya for their insightful comments on the earlier versions of the paper which has helped in improving this paper. This work is partially supported by NSF Grants IIS-0744539 and IIS-0847925, and an AWS in Education grant.

8. REFERENCES

[1] D. Agrawal, A. El Abbadi, S. Antony, and S. Das. Data Management Challenges in Cloud Computing Infrastructures. In *DNIS*, 2010.

[2] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *SOSP*, pages 159–174, 2007.

[3] S. Amer-Yahia, V. Markl, A. Halevy, A. Doan, G. Alonso, D. Kossmann, and G. Weikum. Databases and Web 2.0 panel at VLDB 2007. *SIGMOD Rec.*, 37(1):49–52, 2008.

[4] M. Armbrust, A. Fox, D. A. Patterson, N. Lanham, B. Trushkowsky, J. Trutna, and H. Oh. SCADS: Scale Independent Storage for Social Computing Applications. In *CIDR Perspectives*, 2009.

[5] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *SIGMOD*, pages 1–10, 1995.

[6] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, Reading, Massachusetts, 1987.

[7] M. Brantner, D. Florescu, D. Graf, D. Kossmann, and T. Kraska. Building a database on S3. In *SIGMOD*, pages 251–264, 2008.

[8] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *OSDI*, pages 205–218, 2006.

[9] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, 2008.

[10] S. Das, S. Agarwal, D. Agrawal, and A. El Abbadi. ElasTraS: An Elastic, Scalable, and Self Managing Transactional Database for the Cloud. Technical Report 2010-04, CS, UCSB, 2010. `http://www.cs.ucsb.edu/research/tech_reports/`.

[11] S. Das, D. Agrawal, and A. El Abbadi. ElasTraS: An Elastic Transactional Data Store in the Cloud. In *USENIX HotCloud*, 2009.

[12] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *SOSP*, pages 205–220, 2007.

[13] D. J. DeWitt, R. H. Gerber, G. Graefe, M. L. Heytens, K. B. Kumar, and M. Muralikrishna. GAMMA - A High Performance Dataflow Database Machine. In *VLDB*, pages 228–237, 1986.

[14] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19(11):624–633, 1976.

[15] S. Fushimi, M. Kitsuregawa, and H. Tanaka. An Overview of The System Software of A Parallel Relational Database Machine GRACE. In *VLDB*, pages 209–219, 1986.

[16] J. Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, pages 393–481, London, UK, 1978. Springer-Verlag.

[17] HBase: Bigtable-like structured storage for Hadoop HDFS, 2009. `http://hadoop.apache.org/hbase/`.

[18] HDFS: A distributed file system that provides high throughput access to application data, 2009. `http://hadoop.apache.org/hdfs/`.

[19] P. Helland. Life beyond Distributed Transactions: An Apostate's Opinion. In *CIDR*, pages 132–141, 2007.

[20] A. Hirsch. Cool Facebook Application Game – Scrabulous – Facebook's Scrabble. `http://www.makeuseof.com/tag/best-facebook-application-game-scrabulous-facebooks-scrabble/`, 2007.

[21] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. B. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-store: a high-performance, distributed main memory transaction processing system. *PVLDB*, 1(2):1496–1499, 2008.

[22] J. Karlsson and P. Zeyliger. Megastore – Scalable Data System for User-facing Apps (Invited Talk). In *SIGMOD*, 2008.

[23] T. Kraska, M. Hentschel, G. Alonso, and D. Kossmann. Consistency Rationing in the Cloud: Pay only when it matters. *PVLDB*, 2(1):253–264, 2009.

[24] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, 1981.

[25] B. G. Lindsay, L. M. Haas, C. Mohan, P. F. Wilms, and R. A. Yost. Computation and communication in R*: A distributed database manager. *ACM Trans. Comput. Syst.*, 2(1):24–38, 1984.

[26] D. B. Lomet, A. Fekete, G. Weikum, and M. J. Zwilling. Unbundling transaction services in the cloud. In *CIDR Perspectives*, 2009.

[27] D. B. Lomet and M. F. Mokbel. Locking Key Ranges with Unbundled Transaction Services. *PVLDB*, 2(1):265–276, 2009.

[28] D. Obasanjo. When databases lie: Consistency vs. availability in distributed systems. `http://www.25hoursaday.com/weblog/2007/10/10/WhenDatabasesLieConsistencyVsAvailabilityInDistributedSystems.aspx`, 2009.

[29] R. Rawson and J. Gray. HBase at Hadoop World NYC. `http://www.docstoc.com/docs/12426408/HBase-at-Hadoop-World-NYC/`, 2009.

[30] J. B. Rothnie Jr., P. A. Bernstein, S. Fox, N. Goodman, M. Hammer, T. A. Landers, C. L. Reeve, D. W. Shipman, and E. Wong. Introduction to a System for Distributed Databases (SDD-1). *ACM Trans. Database Syst.*, 5(1):1–17, 1980.

[31] T. von Eicken. Righscale Blog: Animoto's Facebook Scale-up. `http://blog.rightscale.com/2008/04/23/animoto-facebook-scale-up/`, April 2008.

[32] G. Weikum and G. Vossen. *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery*. Morgan Kaufmann Publishers Inc., 2001.

[33] F. Yang, J. Shanmugasundaram, and R. Yerneni. A scalable data platform for a large number of small applications. In *CIDR*, 2009.