Enabling Distributed Key-Value Stores with Low Latency-Impact Snapshot Support

Jordà Polo, Yolanda Becerra, David Carrera, Jordi Torres and Eduard Ayguadé
Barcelona Supercomputing Center (BSC) - Technical University of Catalonia (UPC)

Mike Spreitzer and Malgorzata Steinder
IBM T.J. Watson Research Center
Yorktown, NY

Abstract—**Current distributed key-value stores generally provide greater scalability at the expense of weaker consistency and isolation. However, additional isolation support is becoming increasingly important in the environments in which these stores are deployed, where different kinds of applications with different needs are executed, from transactional workloads to data analytics. While fully-fledged ACID support may not be feasible, it is still possible to take advantage of the design of these data stores, which often include the notion of multiversion concurrency control, to enable them with additional features at a much lower performance cost and maintaining its scalability and availability. In this paper we explore the effects that additional consistency guarantees and isolation capabilities may have on a state of the art key-value store: Apache Cassandra. We propose and implement a new multiversioned isolation level that provides stronger guarantees without compromising Cassandra's scalability and availability. As shown in our experiments, our version of Cassandra allows Snapshot Isolation-like transactions, preserving the overall performance and scalability of the system.**

Keywords*—*Data store, Distributed, Key-value, Cassandra, Isolation, Snapshot, Latency

I. Introduction

In recent years, the industry and research community have witnessed an extraordinary growth in research and development of data-analytic technologies. In addition to distributed, large-scale data processing with models like MapReduce, new distributed data stores have been introduced to deal with huge amounts of structured and semi-structured data: Google's BigTable [1], Amazon's Dynamo [2], and others often modeled after them. These key-value data stores were created out of need for highly reliable and scalable databases, and they have been extremely successful in introducing new ways to think about large-scale models and help solve problems that require dealing with huge amounts of data.

The emergence of these new data stores, along with its widespread and rapid adoption, is changing the way we think about storage. Only a few years ago, relational database systems used to be the only back-end storage solution, but its predominant and almost exclusive position is now being challenged. While scalable key-value stores are definitely not a replacement for RDBMs, which still provide a richer set of features and stronger semantics, they are marking an important shift in storage solutions. Instead of using a single database system on a high-end machine, many companies are now adopting a number of different and complementary technologies, from large-scale data processing frameworks to key-value stores to relational databases, often running on commodity hardware or cloud environments.

This new scenario is challenging since key-value stores are being adopted for uses that weren't initially considered, and data must sometimes be accessed and processed with a variety of tools as part of its dataflow. In this environment, distributed key-value stores are becoming one of the corner-stones as they become the central component of the back-end, interacting concurrently with multiple producers and consumers of data, and often serving different kinds of workloads at the same time: from responding to transactional queries to storing the output of long-running data analytics jobs.

Consistency and isolation become increasingly important as soon as multiple applications and workloads with different needs interact with each other. Providing strong semantics and fully-fledged transactions on top of distributed key-value stores often involves a significant penalty on the performance of the system since it is orthogonal to its goals. So, while fully-fledged ACID (a set of properties to describe transactions, and which stands for Atomicity, Consistency, Isolation and Durability) support may not be feasible, it is still possible to take advantage of the design of these data stores, which often include the notion of multiversion concurrency control, to enable them with additional features at a much lower performance cost and maintaining its scalability and availability.

This is the approach we are following in this paper. Our goal is to provide stronger isolation on top of a distributed key-value store in order to allow certain operations that would otherwise not be possible or require significant effort on the client side, but without compromising its performance. We implement this improved isolation level in the form of readable snapshots on top of Apache Cassandra, a state of the art distributed column-oriented key-value store.

The remaining sections of the paper describe our approach and implementation. We first present an overview of Cassandra in Section II, and then describe its isolation and consistency levels in Section III. Section IV describes how we have extended the level of isolation and how we implement it on top of Cassandra. An evaluation of our implementation is studied in Section V. Finally, we discuss related work in Section VI and conclude in Section VII.

II. Background

Apache Cassandra [3] is a distributed database management system initially developed by Facebook for internal usage and later released as an open source project. Cassandra inherits its data model from Google's BigTable [1], and its replication mechanism and distribution management from Amazon's Dynamo [2]. We use Cassandra as an example of a widely used key-value store known for its scalability and support for tunable consistency.

Cassandra's data model is schema-free, meaning there is no need to define the structure of the data in advance. Data is organized in *column families*, which are similar to tables in a relational database model. Each column family contains a set of *columns*, which are equivalent to attributes, and a set of related columns compose a *row*. Each row is identified by a *key*, which are provided by applications and are the main identifier used to locate data, and also to distribute data across nodes. Cassandra does not support relationships between column families, disregarding foreign keys and join operations. Knowing this, the best practice when designing a data model is to keep related data in the same column family, denormalizing it when required.

The architecture of Cassandra is completely decentralized and peer-to-peer, meaning all nodes in a Cassandra cluster are equivalent and provide the same functionality: receive read and write requests, or forward them to other nodes that are supposed to take care of the data according to how data is partitioned.

When a read request is issued to any target node, this node becomes the proxy of the operation, determines which nodes hold the data requested by the query, and performs further read requests to get the desired data directly from the nodes that hold the data. Cassandra implements automatic partitioning and replication mechanisms to decide which nodes are in charge of each replica. The user only needs to configure the number of replicas and the system assigns each replica to a node in the cluster. Data consistency is also tunable by the user when queries are performed, so depending on the desired level of consistency, operations can either return as soon as possible or wait until a majority or all nodes respond.

III. Isolation and Consistency Levels

The goal of current distributed key-value stores such as Cassandra [3] is to read and write data operations, exactly the same as any database system. However, while traditional databases provide strong consistency guarantees of replicated data by controlling the concurrent execution of transactions, Cassandra provides tunable consistency in order to favour scalability and availability. While there is no tight control of the execution of concurrent transactions, Cassandra still provides mechanisms to resolve conflicts and provide durability even in the presence of node failures.

Traditionally, database systems have provided different isolation levels that define how operations are visible to other concurrent operations. Standard ANSI SQL isolation levels have been criticized as too few [4], but in addition to standard ANSI SQL, other non-standard levels have been widely adopted by database systems. One such level is Snapshot Isolation, which provides almost the same guarantees as serializable transactions, but instead of avoiding concurrent updates, it simply allows transactions to see their own version of the data (a *snapshot*).

Cassandra, on the other hand, unlike traditional databases, doesn't provide any kind of server-side transaction or isolation support. For instance, if an application needs to insert related data to multiple tables, additional logic will be needed on the application (e.g. to manually roll-back the changes if one operation fails). Instead, Cassandra provides a tunable consistency mechanism that defines the state and behaviour of the system after executing an operation, and basically allows specifying how much consistency is required for each query.

Tables I and II show Cassandra's tunable read and write consistency levels, respectively.

TABLE I. Cassandra's read consistency levels.

Level	Description
One	Get data from first node to respond.
Quorum	Wait until majority of replicas respond.
All	Wait for all replicas to respond.

TABLE II. Cassandra's write consistency levels.

Level	Description
Zero	Return immediately, write value asynchronously.
Any	Write value or hint to at least one node.
One	Write value to log and memtable of at least one node.
Quorum	Write to majority of replicas.
All	Write to all replicas.

As it can be derived from their description, strong consistency can only be achieved when using Quorum and All consistency levels. More specifically, strong consistency can be guaranteed in Cassandra as long as equation 1 holds true.

$$\textit{Write replicas} + \textit{Read replicas} > \textit{Replication factor} \quad (1)$$

Operations that use weaker consistency levels, such as Zero, Any and One, aren't guaranteed to read the most recent data. However, this weaker consistency provides certain flexibility for applications that can benefit from better performance and don't have strong consistency needs.

A. Extending Cassandra's Isolation

While Cassandra's consistency is tunable, it doesn't offer a great deal of flexibility when compared to traditional databases and its support for transactions. Cassandra applications could benefit from extended isolation support, which would be specially helpful in the environments in which Cassandra is being used, and remove the burden of additional logic on the application side.

Lock-based approaches, used to implement true serializable transactions, aren't desirable due to the distributed and non-blocking nature of Cassandra, since locks would have a huge impact on the performance. But there are other approaches that seem more appropriate, such as multiversioned concurrency control. Cassandra, unlike other key-value or column stores, doesn't provide true multiversion capabilities and older

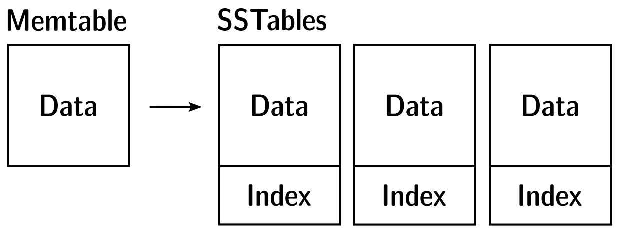

Fig. 1. Data is persisted in Cassandra by flushing a column family's memtable into an SSTable.

versions of the data aren't guaranteed to be available in the system, but its timestamps provide a basic notion of versions that can be the basis for multiversion-like capabilities.

Our goal is then to extend Cassandra to support an additional isolation level that will make it possible to provide stronger semantics using a multiversioned approach. In particular, we implement read-only transactions, guaranteeing that reads within a transaction are repeatable and exactly the same. This kind of transactions are specially relevant in the environments in which Cassandra is being adopted, where there's a continuous stream of new data and multiple consumers that sometimes need to operate on a consistent view of the database. Our proposal is similar to Snapshot Isolation in that it guarantees that all reads made in a transaction see the same *snapshot* of the data, but it isn't exactly the same since we aren't concerned with conflicting write operations. Hence from now on we call this new isolation level Snapshotted Reads.

IV. Implementing Snapshotted Reads

Implementing Snapshotted Reads requires multiple changes in different parts of Cassandra: first, the data store, to enable creating and maintaining versioned snapshots of the data, and second, the reading path, in order to read specific versions of the data.

A. Data Store

Cassandra nodes handles data for each column family using two structures: memtable and SSTable. A memtable is basically an in-memory write-back cache of data; once full, a memtable is flushed to disk as an SSTable. So, while there is a single active memtable per node and column family, there is usually a larger number of associated SSTables, as shown in Figure 1. Also, note that when memtables are persisted to disk as SSTables, an index and a bloom filter are also written along with the data, so as to make queries more efficient.

Once a memtable is flushed, its data is immutable and can't be changed by applications, so the only way to update a record in Cassandra is by textitappending data with a newer timestamp.

Our implementation of Snapshotted Reads takes advantage of the fact that SSTables are immutable to allow keeping multiple versions of the data and thus providing effective snapshots to different transactions. This mechanism is described in the following figures. Figure 2 shows the data for a column family stored in a particular node: there is data in memory as well as in three SSTables. Once we begin a Snapshotted Read transaction, a new snapshot of the data is created by 1) emptying the memtable, flushing its data into a new SSTable, and 2) assigning an identifier to all SSTables, as shown in Figure 3.

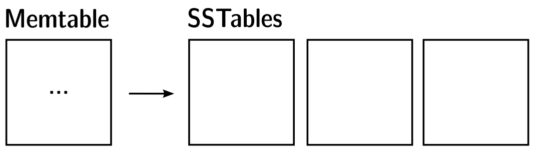

Fig. 2. State of a column family in a Cassandra node before starting a Snapshotted Read transaction.

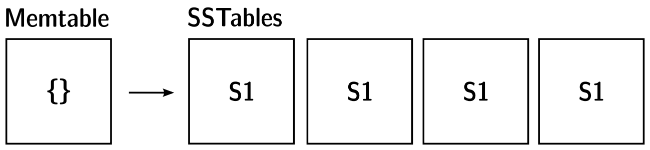

Fig. 3. State of a column family in a Cassandra node after starting a Snapshotted Read transaction and creating snapshot S1.

After the snapshot is created, the transaction will be able to read from it for as long as the transaction lasts, even if other transactions keep writing data to the column family. It is also possible for multiple transactions to keep their own snapshots of the data, as shown in the following figures. Figure 4 shows the state of column family when writes occur after a snapshot (S1). Writes continue to operate as expected, eventually creating new SSTables: in this particular example, there is new data in the memtable as well as in two SSTables. If a transaction were to begin a new Snapshotted Read and create new snapshot, the procedure would be the same: flush and assign identifiers to SSTables, as shown in Figure 5.

B. Reading and Compacting

Reading from a snapshot during a transaction is a matter of selecting data from the appropriate SSTables during the

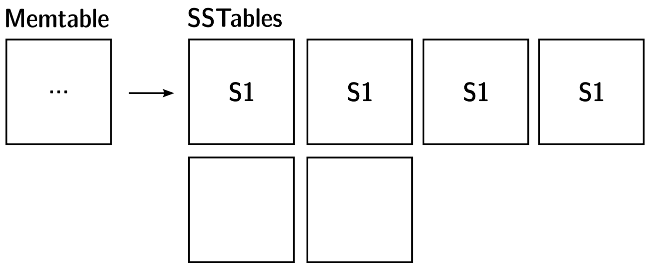

Fig. 4. State of a snapshotted column family in a Cassandra node after some additional writes.

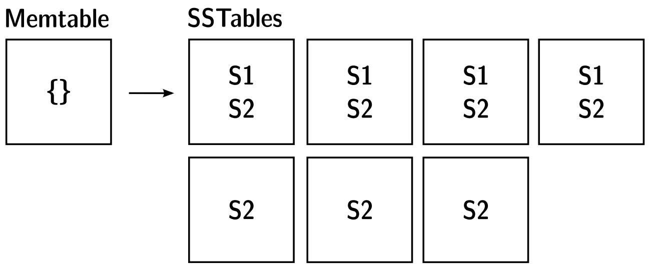

Fig. 5. State of a column family with two snapshots (S1, S2).

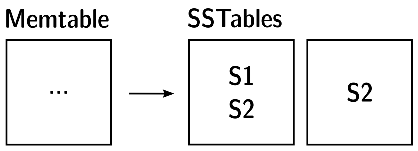

Fig. 6. State of a column family in a Cassandra node with two snapshots after a bounded compaction.

collation process, ignoring SSTables that aren't part of the snapshot.

However, multiple records may still be available even within a snapshot, and Cassandra provides mechanisms to address this kind of diverging results, such as read repair. Read repair simply pushes the most recent record to all replicas when multiple records are found during a read operation. Since there is no way to push new data to an older snapshot, read repair is disabled for Snapshotted Read transactions.

Compaction, on the other hand, is a background operation that merges SSTables, combining its columns with the most recent record and removing records that have been overwritten. Compaction removes duplication, freeing up disk space and optimizing the performance of future reads by reducing the number of seeks. The default compaction strategy is based on the size of SSTables, but doesn't consider snapshots, so it may delete relevant data for our transactions.

We have implemented a new compaction strategy that takes into account snapshots. The main idea behind the new strategy is to compact SSTables only within certain boundaries, namely snapshots. Using this bounded compaction strategy, only SSTables that share exactly the same set of snapshots are considered for compaction. For instance, continuing with the same example in Figure 5, compaction can only be applied to the older four SSTables (identified as S1 and S2) or the remaining three SSTables (identified as S2). One of the possible outcomes of a major compaction is shown in Figure 6.

C. *API Extension*

Finally, in order to support snapshots, some changes have been made to Cassandra's API, including 3 new operations: create snapshot, delete snapshot, and get data from a particular snapshot.

Operations to create or delete a snapshot take 2 arguments: first, the snapshot identifier, and then, optionally, the name of a column family. If no column family is specified, all column families in the current namespace are snapshotted. The operation to retrieve data from a snapshot resembles and works exactly like Cassandra's standard *get* operation with an additional argument to specify the snapshot identifier from which the data is to be retrieved.

V. Experiments

In this section we include results from three experiments that explore the performance of our implementation of Snapshotted Reads for Cassandra. Experiment 1 shows the overall performance of the system under different loads in order to compare the maximum throughput achievable with each version of Cassandra. In Experiment 2 we compare Cassandra with and without Snapshotted Read support using a synthetic benchmark in order to see what is the impact of keeping snapshots under different workloads trying to achieve maximum throughput. Finally, Experiment 3 studies how does our implementation perform and scale in the presence of multiple snapshots.

A. *Environment*

The following experiments have been executed on a Cassandra cluster consisting of 20 Quad-Core 2.13 GHz Intel Xeon machines with a single SATA disk and 12 GB of memory, connected with a gigabit ethernet network. The version of Cassandra used for all the experiments is 1.1.6.

In these experiments we run the synthetic workloads provided by the Yahoo! Cloud Serving Benchmark (YCSB) tool [5]. The workloads are defined as follows:

A: Update heavy.	Read/update ratio: 50%/50%. Application example: session store recording recent actions.
B: Read mostly.	Read/update ratio: 95%/5%. Application example: photo tagging; add a tag is an update, but most operations are to read tags.
C: Read, modify, write.	Read/read-modify-write ratio: 50%/50%. Application example: user database, where user records are read and modified by the user or to record user activity.
D: Read only.	Read/update ratio: 100%/0%. Application example: user profile cache, where profiles are constructed elsewhere (e.g. Hadoop).
E: Read latest.	Read/insert ratio: 95%/5%. Application example: user status updates, people want to read the latest.

The execution of each workload begins with the same initial dataset, which consists of 380,000,000 records (approximately 400 GB in total) stored across the 20 nodes of the cluster with a single replica, meaning each node stores approximately 20 GB of data, and thus exceeds the capacity of the memory. During each execution, a total of 15,000,000 read and/or write operations, depending on the workload, are executed from 5 clients on different nodes. Cassandra nodes are configured to run the default configuration for this system, consisting of 16 threads for read operations and 32 threads for writes.

The following tables and figures show the results of running the workloads with two different versions of Cassandra: the original version and our version of Cassandra with Snapshotted Read support. Note that for our version of Cassandra we also compare regular reads, which are equivalent to reading in the original Cassandra, regular reads in the presence of a snapshot, and finally snapshotted reads, which get data from one particular snapshot.

B. Experiment 1: Throughput

In this experiment we execute two workloads with different configurations in order to explore how does Cassandra perform reads under different loads (which are specified to the YCSB client as target throughputs). We first study the workload D, which only performs read operations, since our changes to Cassandra are focused on the read path. We then execute workload A in order to validate the results under update-intensive workloads.

Tables and figures in this section show the four different kinds of ways to read data from Cassandra that we compare in this experiment: the first one reading from the Original Cassandra, and the remaining three reading from Cassandra with Snapshotted Read support. In particular, for Cassandra with Snapshotted Read Support we evaluate performing regular reads (S/R), performing regular reads in the presence of a snapshot (S/RwS), and performing Snapshotted Reads (S/SR). The measured results are the average and corresponding standard deviation after running 5 executions of each configuration.

Table III and Figure 7 show the results of running workload D. As it can be observed, latency is similar under all configurations for each target throughput, and the same pattern can be observed in all executions: on the one hand the performance of regular reads is similar, independently of the version of Cassandra and the presence of the snapshot, and on the other hand reading from the snapshot is slightly slower with a slowdown around 10%.

TABLE III. AVERAGE READ LATENCY (MS) OF WORKLOAD D USING ORIGNAL CASSANDRA AND CASSANDRA WITH SNAPSHOTTED READ SUPPORT (S/R, S/RWS, SR)

Operations/s	Original	S/R	S/RwS	S/SR
1000	6.09	6.15	6.18	6.54
2000	7.44	7.46	7.64	7.96
3000	10.15	10.44	10.51	11.42
4000	13.18	13.33	13.45	14.06
5000	18.47	18.46	18.60	19.39

Figure 7 also shows the standard deviation of the executions, and the real throughput achieved with each target (shown as a black line). As it can be seen in this workload, the observed throughput grows linearly with the target throughput until its optimal level, which is approximately 4400 operations per second when running the Original Cassandra. After reaching the maximum throughput, latency simply degrades without any benefit in terms of throughput.

Similarly, Table IV and Figure 8 show the results of running workload A (50% read, 50% update) under different loads. The main difference in this workload compared to workload D (100% read) is the performance of reading from the snapshot, which is slightly faster. If more than one record is found when reading a key, Cassandra will merge the multiple records and select the right one. However, in this particular experiment, the snapshot is created under perfect conditions and SSTables are fully compacted. So, while regular reads may degrade slightly over time as new SSTables are created by updates and not yet compacted, the latency of snapshotted reads remains mostly the same since snapshots aren't updated.

Therefore, the differences in the performance when reading from a snapshot depending on the kind of workload can be

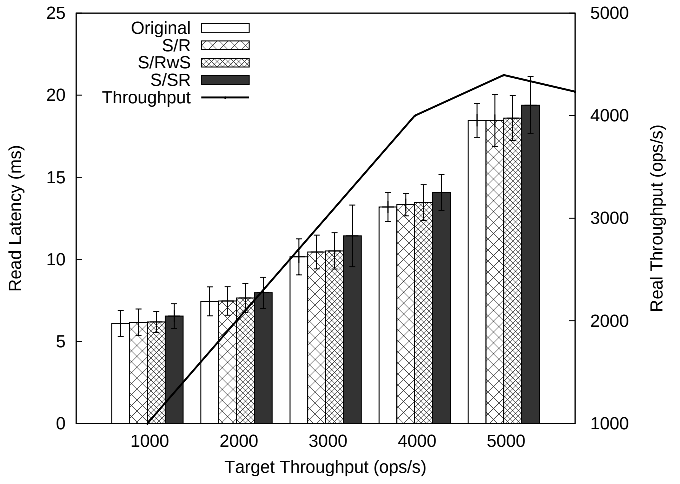

Fig. 7. Average read latency and observed thoughtput for varying targets of operations per second on Workload D

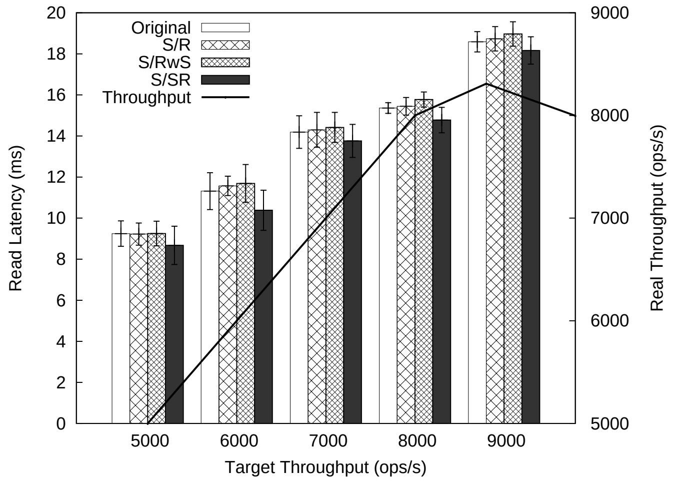

Fig. 8. Average read latency and observed thoughtput for varying targets of operations per second on Workload A

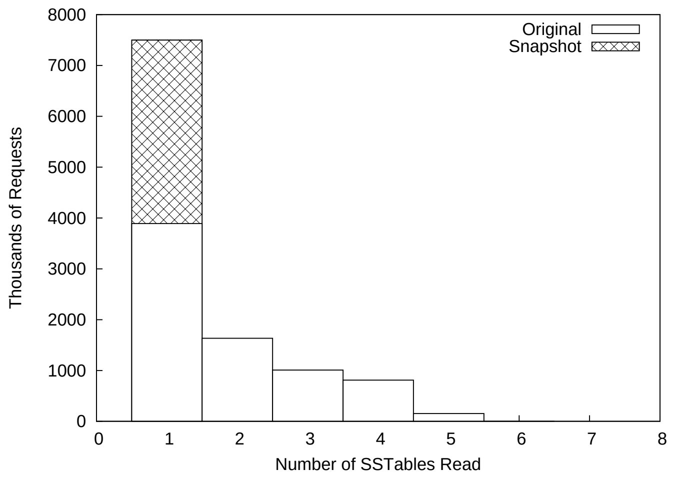

Fig. 9. Distribution of number of SSTables read for each read operation on workload A when performing regular and snapshotted reads

TABLE IV. AVERAGE READ LATENCY (MS) OF WORKLOAD A USING ORIGNAL CASSANDRA AND CASSANDRA WITH SNAPSHOTTED READ SUPPORT (S/R, S/RWS, SR)

Operations/s	Original	S/R	S/RwS	S/SR
4000	8.25	8.47	8.44	7.71
5000	9.25	9.25	9.22	8.68
6000	11.32	11.69	11.57	10.38
7000	14.19	14.42	14.30	13.76
8000	15.36	15.77	15.45	14.78
9000	18.59	18.73	18.97	18.17

explained by how Cassandra handles read and write operations, and the strategy used to compact SSTables. As described in Section II, Cassandra first writes data to a memtable, and once it is full it is flushed to disk as a new SSTable, which eventually will be compacted with other SSTables. Compaction is not only important to reclaim unused space, but also to limit the number of SSTables that must be checked when performing a read operation and thus its performance. Our version of Cassandra with Snapshotted Read support uses a custom compaction strategy, as described in Section IV-B. While our bounded compaction strategy is necessary to keep data from snapshots, it also makes compaction less likely since it won't allow compaction between snapshot boundaries.

The consequences of this behaviour for Cassandra with Snapshotted Read support are twofold: first, as observed, reading from a snapshot may be faster on workloads in which data is mostly updated (and snapshots eventually consolidated), and second, it may make regular reads slower when snapshots are present, due to the increased amount of SSTables caused by our compaction strategy.

Figure 9 shows the distribution of how many SSTables must be checked during read queries in workload A. As it can be observed, there is a significant difference between reading from a snapshot and performing a regular read. While snapshotted reads always get the data from a single SSTable, regular reads require checking two SSTables or more at least half of the time. Again, it should be noted that there is nothing that makes snapshot intrisincally less prone to spreading reads to multiple SSTables, it is simply their longer-term nature that helps consolidate and compact snapshots. Regular reads, on the other hand, need to deal with the latest updates, so they are more likely to be spread across multiple SSTables.

In order to compare how does the number of SSTables impact our version of Cassandra, we also executed the update-intensive workload A, and then increased the frequency at which the number of SSTables are generated, thus increasing the number of SSTables. As it can be observed in Figure 10, while performing regular reads with Original Cassandra becomes slower when we force a larger number of SSTables, reading from a snapshot remains mostly unchanged since it simply reads from the same subset of SSTables all the time.

C. Experiment 2: Read Latency

In this experiment we compare the latency of reading operations under different workloads in order to find out what's the impact of supporting snapshotted reads as well as what's the performance of reading from a snapshot compared to a regular read under a wider variety of scenarios.

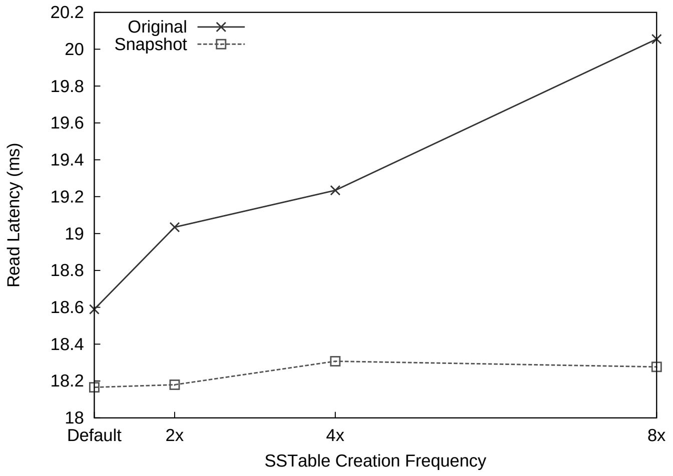

Fig. 10. Average read latency on Workload A, performing regular and snapshotted reads, and varying the frequency at which SSTables are created relative to to the default configuration

This experiment compares two different kinds of ways to read: the first one reading from the Original Cassandra, and the second one reading from a snapshot on our version of Cassandra with Snapshotted Read support. We omit here the results of regular reads on our version of Cassandra since they are similar to Original Cassandra. All workloads are executed 5 times, and the YCSB client is configured to run with the maximum number of operations per second.

These workloads show different behaviours. One the one hand, in *read-modify* workloads, including A and B, data is updated but the size of the data set (number of keys) remains the same. On the other hand, in *write-once* workloads, which include D and E (read-only and read-insert respectively), each record is only written once and it's not modified afterwards. Finally, workload C can be thought of as a special case since half of the operations are composed and perform a read followed by a write operation.

As shown in Table V and Figure 11, the latency of reading from a single snapshot is similar to performing regular reads. Generally speaking, reading from the original Cassandra is slightly faster, and the slower reads are from our version of Cassandra performing snapshotted reads on read-intensive workloads.

TABLE V. AVERAGE READ LATENCY (MS) USING ORIGNAL CASSANDRA AND CASSANDRA WITH SNAPSHOTTED READ SUPPORT (REGULAR, SNAPSHOT)

Workload	Original	Snapshot
A	18.59	18.17
B	18.66	18.75
C	19.08	20.04
D	18.47	19.39
E	12.07	12.73

As it can be observed, *read-modify* workloads (A, B) are the workloads that remain closer to each other, independently of the kind of read we are performing. Workload A remains faster when reading from a snapshot than when reading regularly, while snapshotted read under workload B is slightly slower since the amount of updates is relatively small and thus regular reads almost always involve a single SSTable. On

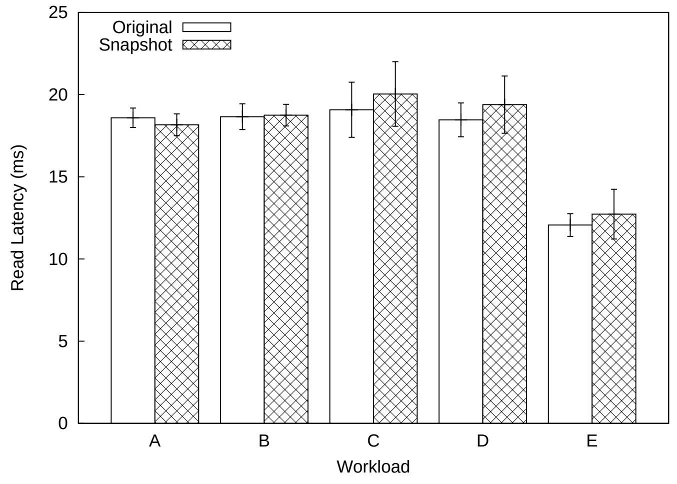

Fig. 11. Average read latency for each workload, comparing regular reads to reading from a snapshot

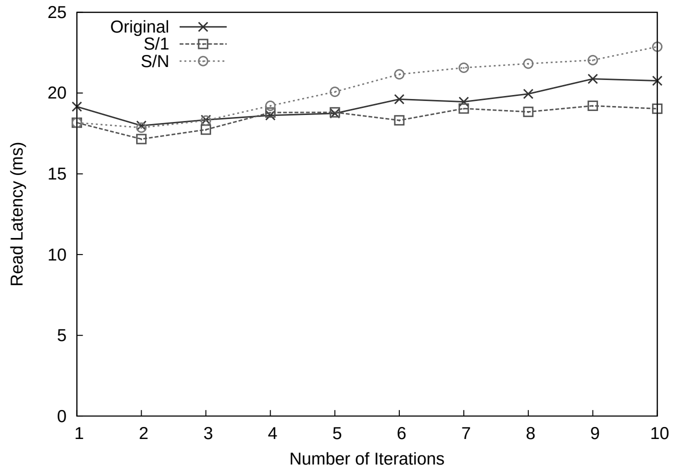

Fig. 12. Evolution of average read latency for 10 consecutive executions of Workload A

the other hand, both *write-once* workloads (D, E) display a much more noticeable difference between the two kinds of reads since data is only written once and so each operation reads from exactly one SSTable.

D. Experiment 3: Increasing the Number of Snapshots

While the previous experiments discuss the performance of different kinds of reads with a single fully-compacted snapshot, in this experiment we evaluate the evolution of the performance under a more realistic scenario in which multiple snapshots are created and read during its execution.

In order to test multiple snapshots and compare the results of previous experiments, we execute workload A ten times consecutively one after another. In particular, we execute 3 different versions in this experiment: first the original Cassandra performing regular reads, and then our version of Cassandra, either creating a single snapshot at the beginning (S/1), or creating a new snapshot for each iteration (S/N). Since workload A involves at least 50% of update operations, we ensure an increasing number of SSTables as new snapshots are created.

As shown in Figure 12, after the first few executions, the performance of read operations degrades slightly over time as new consecutive executions of workload A are completed, independently of the version of Cassandra we are running. Regular reads with Original Cassandra and snapshotted reads with our version of Cassandra and a single snapshot both become more stable after a few iterations and don't change too much afterwards. However, as it could be expected, departing from the initial scenario with fully compacted SSTables and keeping multiple snapshots becomes noticeably slower over time as shown in the Figure. When creating a new snapshot for each iteration (S/N), read latency goes from 18.17 ms during the first iteration to 22.87 after all iterations with 10 snapshots.

The varying performance can also be explained in terms of how the data is read and stored as SSTables. For instance, while with the original version of Cassandra there are 193 SSTables in the cluster after all executions, with our version of Cassandra creating a new snapshot for each iteration (S/N) there are as many as 532 SSTables. Figure 13 also shows the evolution of the distribution of SSTables read for each operation. As expected, at the beginning when there is only one snapshot and the data is still well compacted, all operations only read from a single SSTable. However, as soon as we introduce more snapshots (3 and 5 as shown in the Figure), the number of seeks to SSTables for each read operation increases as well, thus making read operations slower.

VI. Related Work

There have been many efforts to implement features usually available in relational databases on top of distributed data stores [6], [7], [8], and, as other have pointed out [9], [10], this further proves that some of their functionality is converging. Isolation and transactional support for distributed data stores is also a widely studied topic, and there has been some related work done, including support for lock-free transactions [11] and snapshot isolation [12] for distributed databases.

There has also been work more focused on stronger semantics for distributed key-value data stores. Google's Percolator [13] implements snapshot isolation semantics by extending Bigtable with multi-version timestamp ordering using a two-phase commit, while Spanner [10] and Megastore [14] also provide additional transactional support for Bigtable. In [15] and [16] the authors also implement snapshot isolation for HBase, allowing multi-row distributed transactions for this column-oriented database. While the former approach uses additional meta-data on top of standard HBase, the latter introduces a more advanced client to support snapshot isolation transactions.

There hasn't been much work done in the space of isolation for Cassandra in particular since improving it is orthogonal to its design, and other than the configurable consistency levels, there is basically no support for transactions. Cassandra currently only provides support to create *backup* snapshots, which are only meant to be used as a way to backing up and restoring data on disk. So, unlike our proposal, with *backup* snapshots it is only possible to read from one of these snapshots at at time and reading from a different snapshot involves reloading the database.

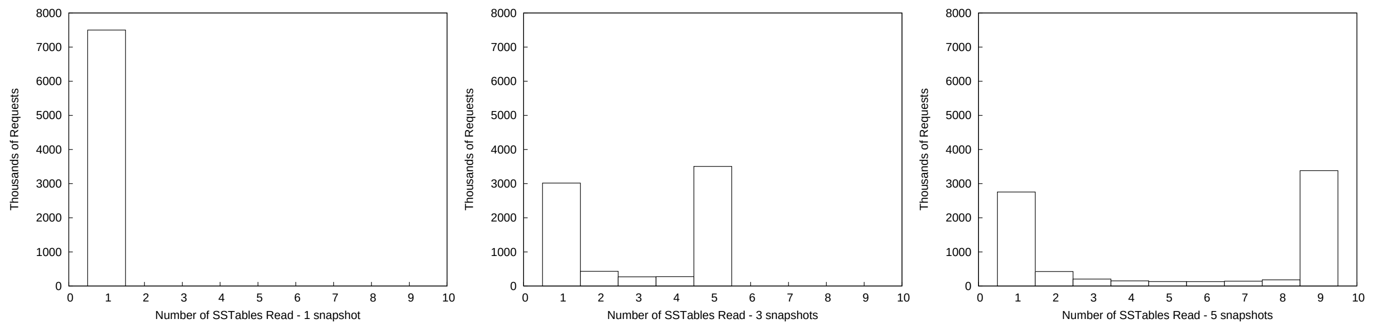

Fig. 13. Distribution of number of SSTables read for each read operation on workload A with multiple snapshots

VII. Conclusions

In this paper we have presented a technique to provide stronger isolation support on top of distributed key-value stores, and implemented it for Apache Cassandra.

Our approach takes advantage of the fact that one of the major structures used to persist data in this kind of stores are SSTables, which are immutable. Our proposal modifies Cassandra so as to keep SSTables when requested by concurrently running transactions, effectively allowing multi-versioned concurrency control for read operations on Cassandra in the form of snapshots. As shown in our evaluation, our version of Cassandra with Snapshotted Read support is able to read from snapshots with a low impact on read latency and the overall performance of the system. While regular reads are slightly slower on our version of Cassandra, operations that read from a snapshot are sometimes faster due to its limited scope.

We believe this approach to improve the isolation capabilities of distributed key-value stores without compromising its performance is specially interesting in the environments in which these stores are nowadays executed, which tend to involve a range of technologies on the back-end side instead of a single database solution, and different applications and workloads running at the same time sharing and processing the same data.

Acknowledgements

This work is partially supported by the Ministry of Science and Technology of Spain and the European Union's FEDER funds (TIN2007-60625, TIN2012-34557), by the Generalitat de Catalunya (2009-SGR-980), by the BSC-CNS Severo Ochoa program (SEV-2011-00067), by the HiPEAC European Network of Excellence (IST-004408, FP7-ICT-217068, FP7-ICT-287759), and by IBM through the 2008 and 2010 IBM Faculty Award program.

References

[1] 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," *ACM Trans. Comput. Syst.*, vol. 26, no. 2, pp. 4:1–4:26, Jun. 2008. [Online]. Available: http://doi.acm.org/10.1145/1365815.1365816

[2] 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 *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, ser. SOSP '07. NY, USA: ACM, 2007, pp. 205–220. [Online]. Available: http://doi.acm.org/10.1145/1294261.1294281

[3] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40, Apr. 2010. [Online]. Available: http://doi.acm.org/10.1145/1773912.1773922

[4] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil, "A critique of ANSI SQL isolation levels," in *Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, ser. SIGMOD '95. New York, NY, USA: ACM, 1995, pp. 1–10.

[5] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*, ser. SoCC '10. New York, NY, USA: ACM, 2010, pp. 143–154.

[6] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin, "Hadoopdb: an architectural hybrid of mapreduce and dbms technologies for analytical workloads," *Proc. VLDB Endow.*, vol. 2, no. 1, pp. 922–933, Aug. 2009.

[7] M. Armbrust, K. Curtis, T. Kraska, A. Fox, M. J. Franklin, and D. A. Patterson, "Piql: success-tolerant query processing in the cloud," *Proc. VLDB Endow.*, vol. 5, no. 3, pp. 181–192, Nov. 2011.

[8] L. Glendenning, I. Beschastnikh, A. Krishnamurthy, and T. Anderson, "Scalable consistency in scatter," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP '11. New York, NY, USA: ACM, 2011, pp. 15–28.

[9] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker, "A comparison of approaches to large-scale data analysis," in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, ser. SIGMOD '09. New York, NY, USA: ACM, 2009, pp. 165–178.

[10] J. C. et all, "Spanner: Google's globally-distributed database," in *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, ser. OSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 251–264.

[11] F. Junqueira, B. Reed, and M. Yabandeh, "Lock-free transactional support for large-scale storage systems," in *Dependable Systems and Networks Workshops (DSN-W), 2011 IEEE/IFIP 41st International Conference on*, june 2011, pp. 176 –181.

[12] D. Ramesh, A. K. Jain, and C. Kumar, "Implementation of atomicity and snapshot isolation for multi-row transactions on column oriented distributed databases using rdbms," in *Communications, Devices and Intelligent Systems, 2012 International Conference on*, dec. 2012, pp. 298 –301.

[13] D. Peng and F. Dabek, "Large-scale incremental processing using distributed transactions and notifications," in *9th USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[14] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh, "Megastore: Providing scalable, highly available storage for interactive services," in *Proceedings of the Conference on Innovative Data system Research*, 2011.

[15] C. Zhang and H. De Sterck, "Supporting multi-row distributed transactions with global snapshot isolation using bare-bones hbase," in *11th IEEE/ACM International Conference on Grid Computing*, oct. 2010, pp. 177 –184.

[16] C. Zhang and H. D. Sterck, "Hbasesi: Multi-row distributed transactions with global strong snapshot isolation on clouds," *Scalable Computing: Practice and Experience*, pp. –1–1, 2011.