Achieving Consistent Storage for Scalable MMORPG Environments

Ziqiang Diao
Otto-von-Guericke-University of Magdeburg
Building 29, Universitätsplatz 2
39106 Magdeburg, Germany
diao@iti.cs.uni-magdeburg.de

Pengfei Zhao
Otto-von-Guericke-University of Magdeburg
Building 29, Universitätsplatz 2
39106 Magdeburg, Germany
pzhao@iti.cs.uni-magdeburg.de

Eike Schallehn
Otto-von-Guericke-University of Magdeburg
Building 29, Universitätsplatz 2
39106 Magdeburg, Germany
eike@iti.cs.uni-magdeburg.de

Siba Mohammad
Otto-von-Guericke-University of Magdeburg
Building 29, Universitätsplatz 2
39106 Magdeburg, Germany
siba.mohammad@iti.cs.uni-magdeburg.de

ABSTRACT

As outlined for instance by the CAP theorem, achieving consistency guarantees within a 100% available and fault-tolerant distributed system is impossible. Nevertheless, in real-life applications actual properties are neither black nor white and the degree of fulfilment of requirements depends on the likelihood of failures and communication parameters of distributed systems. While typical Cloud-based applications weaken consistency in accordance with less strict applications requirements, strong consistency can also be achieved, for instance by tunable consistency. This, however, often comes with a strong degradation of scalability (performance of growing clusters) and availability. Based on a project investigating the usefulness of Cloud DBMS for Massively Multi-player Online Role-Playing Games (MMORPGs) we describe how strong consistency can be provided for such a scenario, by still proving a high-level of availability and performance suitable for this specific application. For this purpose we implement a lightweight mechanism to detect failures based on timestamps and only react accordingly if required.

CCS Concepts

•Information systems → Parallel and distributed DBMSs; Database performance evaluation; Massively multiplayer online games; *Database transaction processing;*

IDEAS '15, July 13 - 15, 2015, Yokohama, Japan
© 2015 ACM. ISBN 978-1-4503-3414-3/15/07…$15.00
DOI: http://dx.doi.org/10.1145/2790755.2790758

Keywords

data consistency, system availability, NoSQL databases, online game

1. INTRODUCTION

Using highly distributed and parallel data management solutions appears as an attractive choice in many Cloud or Web applications because of the support for huge and flexible data volumes and numbers of users. Cloud data management systems such as BigTable, HBase, MongoDB, or Riak address these specific requirements such as scalability and availability [17], while often only offering limited support for classical DBMS features such as query languages and concurrency control. The CAP-theorem [14], describing the iron triangle of consistency, availability, and partition tolerance of distributed systems, has shown that these conflicting requirements remain as a problem for which only application specific trade-offs can be found.

In the currently running *CloudCraft* project we investigate the usability of Cloud data management systems for Massively Multi-player Online Role-Playing Games (MMORPGs), which represent applications of growing popularity and commercial importance and have recently gained interest from the database community [6, 27]. In our project, we found that the solutions offered ad hoc are suitable for several data sets, e.g. logs and in-game chats requiring only weak consistency or user and login services that may accept lowered levels of availability to achieve higher consistency. Nevertheless, for game state data, e.g. an avatar's properties and position in a persistent game world, players expect high consistency and availability. A performance suitable to fulfill the real-time constraints of a computer game is a further necessary requirement to grant a continuous game experience [21, 26, 13, 8].

To address these problems, we present an approach developed within the *CloudCraft* project, which takes advantage of application specific properties. We suggested and implemented a two-level data management architecture, where the game data is initially stored in a main-memory DBMS (in our case H2) for local game servers and propagated via configurable checkpoint mechanisms to an underlying scalable storage layer using the Cassandra Cloud data management system. Typical properties of this scenario are frequent bulk writes and rare read operations. To grant strong consistency we apply a timestamp-based mechanism and information about storage locations, which in combination allow detecting possible inconsistencies and dealing with them according to application requirements.

The presentation in this paper is structured as follows: in Section 2 we introduce the reason why we propose to use Cassandra in this project. Cassandra cannot support game consistency efficiently, so we highlight this issue in Section 3. In Section 4 we introduce a concept to achieve the required consistency level for our application as outlined before. Its implementation and an evaluation of the runtime performance as well as the achieved consistency are presented in Section 5.

2. USE OF CASSANDRA IN MMORPGS

Data in an MMORG could be classified into four data sets, namely account data, game data, state data, and log data [12]. These different classes vary widely regarding their requirements and have to be managed accordingly [12]. The most demanding data set is state data, which includes an avatar's properties, inventories, and position as well as states of game objects. All these may be modified constantly.

According to the architecture currently applied in MMORPGs [22, 19], modifications of state data are executed by an in-memory database in real-time [5, 23], so that these changes can be synchronously propagated to the relevant players within an acceptable period of time [10]. They are then backed up to the disk resident database periodically (checkpointing) [6, 23]. The checkpoint is created for two purposes: one is to recover the in-memory database in the case of a system failure; another one is to restore the state data to the in-memory database when a player restarts the game (state data will be removed from the in-memory database when a player logs out of the game in order to reduce memory consumption). Since state data of an avatar are only fetched from the disk resident database once during the game, a write-intensive database is required to persist them. Beyond that, it is intolerable that players realize that their last game records are lost when they restart the game. Accordingly, strong or at least a Read-Your-Writes consistency [24] is required for such data. Furthermore, the database must be scalable because the number of players in an MMORPG could be up to millions [16, 18].

Currently, the popular MMOPRGs mainly apply a distributed RDBMS to manage data (e.g., Second Life and World of Warcraft both run on MySQL [2], Guild Wars runs on Microsoft SQL Server [1]). However, the existing RDBMS cannot fully satisfy all these requirements simultaneously [26, 7]. In consideration of these factors, we chose to apply Cassandra[1] in the *CloudCraft* project [12]. The advantages of using Cassandra are as follows: Cassandra is always writeable because there are no single points of failure (checkpointing of state data will always be available); its write performance is high, and is more efficient than the read performance, which is different to other Cloud storage systems [20]; Cassandra provides a flexible data model (comparing with RDBMS), so all state data of one player can be stored together, which decreases the response time to query them (no join operations); Cassandra supplies user specified data consistency, which guarantees different levels of data consistency (e.g., Read-Your-Writes consistency for state data); furthermore, it is easy to scale out.

[1]Apache Cassandra project: http://cassandra.apache.org/ (accessed 09.02.2015)

3. DATA CONSISTENCY IN CASSANDRA

3.1 Issues cased by Guaranteeing High Level Consistency

Cassandra takes a decentralized structure. That means, there is no master/primary copy for a data object. On the one hand, a failure of one node will not affect the system availability (no single points of failure). On the other hand, to guarantee a high level consistency is expensive. Because in Cassandra a query could be executed by any replica, to guarantee a high level consistency, such as strong consistency, consistency level (e.g., *ONE*, *TWO*, and *ALL*) of a write operation and its subsequent read operation should meet a prerequisite:

$$W + R > N, with\ W, R \in \{1, 2, \ldots, N\} \tag{1}$$

In this formula, N refers to the total number of replicas (replication factor), while W and R represent the consistency level of write and read, respectively. This formula states that only if the total number of replicas responded write and its subsequent read exceeds the replication factor, Cassandra could ensure strong consistency. This is because only in this case at least one of the replicas that respond to the query contains the up-to-date data. As a result, more than half of the replicas have to participate in the process of updating and fetching data, which reduces the system performance. In the case of only one available replica (replication factor is more than one), write or/and read cannot be successful because the prerequisite is not met, thereby effecting the system availability.

3.2 Eventual Consistency in Cassandra.

Performing write *ONE* and read *ONE* cannot ensure that the up-to-date data will be fetched because data are eventually consistent on all replicas in Cassandra (*ONE* means, if one of the replicas has responded the request, this query will be considered complete.). However, the time gap between the write and a following read is a vital parameter [4], which affects the accuracy. We have carried out some experiments on Cassandra to evaluate its efficiency of data propagation. The experimental setup here is the same as it described in Section 5.

Detection of Inconsistent Data..

In experiments we quantified the effect of eventual consistency, when the consistency level of both write and read is specified to *ONE* (replication factor is three). The result is shown in Table 1.

During the first test, all five nodes in the cluster are alive. Clients send a total of 10000 write requests to Cassandra. As soon as a write request is successful, the client sends a read request regarding that modification (so called eager detection). In theory, 66.7% of data returned by Cassandra

Table 1: Detection of Inconsistent Data

Operation	Number of nodes	Write requests	Read requests	Detection method	Inconsistent data
WORO	5	10000	10000	Eager	10.43%
WORO	5	10000	10000	Lazy	0%
WORO	$3 \rightarrow 5$	10000	10000	Lazy	4.09%
WORO	$3 \rightarrow 5$	200000	10000	Lazy	22.16%

could be stale. In our practical experiment, however, only an average of 10.43% of the results are stale. If the read requests are sent after all (10000) write requests are successful (lazy detection), no stale data have been detected (see test two). The reason is that, no matter which consistency level is specified by the client, Cassandra actually forwards the write request to all available replicas at the same time. In other words, if all replicas are available, the modification will be synchronized to all of them.

For the next test we have simulated a bad case: two nodes are failed while updating data, but all five nodes are alive while reading data (if more than two nodes fail, for some data objects all three replicas will be unavailable. Thus, write operations could not be performed). In this case, reading from the two temporarily unavailable nodes should fetch stale data. However, the result (test three) shows that only an average of 4.09% of data is stale. We increase the total number of write requests to 200000 so that there could be more inconsistent data (about 300 MB) in the cluster, and Cassandra needs more time to replay writes. The experimental result shows that although the number of stale data is enlarged (about 22.16%), it is not directly proportional with the increase of the number of write requests (see test four). Furthermore, the inconsistency window in this situation is about 942296ms (15m 42s), which is very short.

3.3 Conclusion and Prospect.

The Cassandra cluster in our experiment is deployed in an Intranet, which reduces network latency. However, from the experiment result, we can still conclude that Cassandra actually offers a higher data consistency level than it promises. Although there could be some inconsistent data in the case of node failure, they only exist for a short period of time after the node is restarted. In MMORPGS a player typically restarts the game after a long time. During this time gap, modifications of state data have sufficient time to be propagated to all available replicas. Therefore, in most cases the up-to-date data could actually be fetched by specifying consistency level of writes and reads to *ONE*. Unfortunately, a client could not specify it like that only because there might be an unpredictable node failure during the update, and consequently stale data would be returned later. To address this issue, the system should be able to detect stale data automatically. Only when stale data are returned, the system should increase the consistency level of read and execute it again. To achieve this goal, the existing Cloud storage system needs to be extended [3, 15, 25, 11].

4. TIMESTAMP-BASED FAILURE DETECTION

This section will give a brief introduction of the application-specific scenario, the proposed concept, as well as an optimization method. In previous work, we proposed a timestamp-based model (TSModel) for Cloud-based MMORPGs to solve the issue discussed in the last subsection [11]. In this work, we aim at implementing and improving this concept based on Cassandra, where the timestamp can be applied as a version ID for each checkpoint, which will then be used as a query criterion to fetch the current data [3].

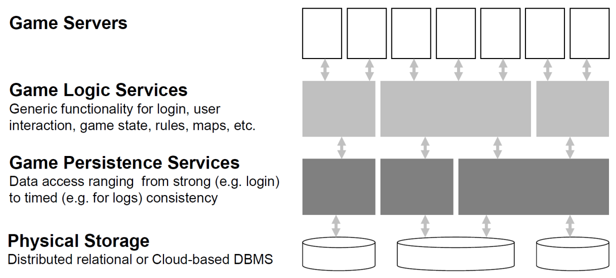

Game Servers
Game Logic Services
Generic functionality for login, user interaction, game state, rules, maps, etc.
Game Persistence Services
Data access ranging from strong (e.g. login) to timed (e.g. for logs) consistency
Physical Storage
Distributed relational or Cloud-based DBMS

Figure 1: *CloudCraft* architecture: reusable services for MMORPGs.

4.1 Integration with MMORPG Application Scenario.

In order to guarantee game-specific consistency, a game persistence layer is applied between the game logic layer and the physical storage layer in the new Cloud-based game architecture (see Figure 1). This layer consists of data access servers (DAS) holding the timestamp tables (TST). A DAS is responsible for creating consistent checkpoints from the in-memory database, flushing them to Cassandra, fetching data from Cassandra regarding game-specific consistency, and playing the role of a counter (generation of monotonically increasing timestamps). The structure of a TST is very simple, containing only four attributes, namely an avatar's or game object's ID (Id), the last checkpointing time (TS), the host address of the last checkpoint in the Cassandra cluster (IP), and a player's log status for avatar data.

A time asynchronisation among DAS's less than the frequency of checkpointing is acceptable. In a typical client-server-based MMORPG, unless a player has changed the zone server or a DAS has failed, checkpointing of game state data is handled by a fixed DAS. For this reason, an accurate global time synchronization is not necessary. The system clock on each DAS could be synchronized by applying the network time protocol (NTP).

4.2 Checkpointing and Data Recovery with the TSModel.

To describe the timestamp-based detection model, let us first outline the process of checkpointing game state data. The DAS creates a consistent snapshot of game state from the in-memory database periodically. The system's current time of the DAS will be used as a unique monotonically increasing version ID (also called TS) for each checkpoint.

```
Data: snapshot of game state
Result: back up to Cassandra
begin
    $TS \longleftarrow$ system current time
    $CL \longleftarrow ONE$
    $Cassandra.put(Id, TS, data)$ with CL
    $TST.put(Id, TS)$
end
```

Algorithm 1: Checkpointing

The DAS executes a bulk write to Cassandra with consistency level (CL) *ONE*. Cassandra divides the message into several write requests based on Id. The current state of an object and the TS are persisted together in one row. When the DAS receives a success acknowledgement, it will use the same TS to update the TST accordingly. When a player has quitted the game and the state data of her/his avatar have been backed up to Cassandra, the log status will be modified to "Logout". Then, the DAS sends a delete request to the in-memory database to remove the respective state data from it.

When a player restarts the game, the DAS first checks the player's log status in the TST. If the value is "Login", that means the previous checkpointing is not yet completed, so the up-to-date state data of her/his avatar is still hosted in the in-memory database. In this case, it is not necessary to recover the in-memory database. If the value is "Logout", the DAS gets the timestamp from the TST, and then uses TS and Id as query criteria to retrieve the up-to-date data with CL *ONE*. When a replica in Cassandra receives the request, it compares the TS with its own TS. If they match, the state data will be returned. Otherwise, a null value will be sent back. In this case, the DAS has to increase the CL and send the read request again until the up-to-date data are found or all available replicas have been retrieved. If the expected version still has not been found, the latest version (but stale) in Cassandra has to be used for recovery. At last, the player's log status in the TST will be modified from "Logout" to "Login".

```
Data: avatar/object Id
Result: avatar/object's state data
begin
    $TS \longleftarrow TST.getTS(Id)$
    $CL \longleftarrow ONE$
    $data \longleftarrow null$
    while $data == null$ and $CL \leq$ number of available replicas do
        $data \longleftarrow Cassandra.get(Id, TS)$ with CL
        if $data == null$ then
            CL++
        end
    end
    if $data == null$ then
        $CL \longleftarrow$ number of available replicas
        $dataSet \longleftarrow Cassandra.get(Id)$ with CL
        for $d \in dataSet$ do
            if $data.TS < d.TS$ then
                $data \longleftarrow d$
            end
        end
    end
    $return(data)$
end
```

Algorithm 2: Data Recovery

4.3 Optimisation using a Node-aware Policy.

From the description above, we can conclude that if the first attempt of retrieval fails, the read request has to be executed again, which increases the response time. Therefore, the success rate determines the read performance. The reason for receiving stale data is that the read request is executed by a node that does not host the up-to-date replica. To optimise this model, we propose to sacrifice a part of database transparency in exchange for the success rate. In other words, the IP address of a node, which hosts the current replica of an avatar's or game object's state, will also be recorded in the TST. For subsequent read requests on this state data, the DAS will connect to this node directly. In this case, the success rate will be increased if the host is still available. The new proposal will not affect the system availability. The checkpoint could still be flushed to any replica as before; if the host fails, a read request could be executed by the other nodes. In this paper, we name refer to this strategy as *NodeAwarePolicy*.

4.4 System Reliability

In order to prevent a single TST from becoming a bottleneck of the system, several TSTs on different nodes need to work in parallel. The checkpoiting information (e.g., TS) is assigned to different TSTs depending on the location(game world/map) of avatars/objects. If one of the TSTs is down, a new/another TST should take place its work immediately. A TST failure will not affect the gameplay. The lost data (e.g., TS) could be replaced by accepting new checkpoints (for active avatars/objects), or recovered by fetching relevant information (such as get the biggest version ID of the checkpoints of one avatar/object) from Cassandra cluster(for inactive avatars). During the recovery, a player could perform a read *ALL* to get the up-to-date state data of her/his avatar.

5. IMPLEMENTATION AND EVALUATION

5.1 Overview of the Implementation.

To use a timestamp as a query criterion in Cassandra, we have to either create a secondary index on the TS column, or set it as part of the compound primary key. Since the TS is modified frequently in our application scenario, maintaining such an index will seriously affect the write performance (checkpointing). For this reason, we propose to use the avatar/object ID and TS columns as compound primary key. Hence, the checkpointing is converted to an insert operation (not update). The disadvantage is that, the stale data cannot be removed automatically because Cassandra does not know if they have already gone out of use. Thus, an additional process is required to detect and clean up them, as discussed below.

For the TST on the server side, we have applied H2[2], which is a lightweight in-memory RDBMS. A query in a Cassandra cluster is first sent to a coordinator (a random node), which will then forward the request to nodes that

[2]H2 website: http://www.h2database.com/html/main.html (accessed 09.02.2015)

Table 2: Experimental Setup

Computer System	virtual machine
CPU	Intel(R) Xeon(R) E5620 2.40 GHz
RAM	8 GB (28.3% ∼ 34.5% used)
Disk	90.18 GB, 7200RPM
Operating System	Ubuntu 13.04 (64 bit)
Java version	1.7.0_25
Network	100MBit/s
Cassandra Version	1.2.13
Number of Nodes	5
Replication Factor	3
Number of Rows	30 million
Number of Columns	110 ∼ 160
Data Size	⩾ 120 GB
Client Driver	DataStax Java Driver 1.0
Writes : Reads	9 : 1

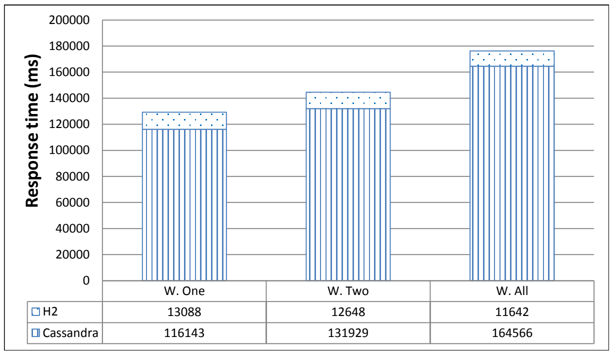
host relevant replicas. That means, only the coordinator decides which replica will respond the query. Therefore, we have implemented a new user-defined load balance policy in the Cassandra client driver in order to specify the coordinator for each query. The *NodeAwarePolicy* strategy in our project functions like this: at the checkpointing phase, a write request will be sent to a node (coordinator), which hosts an relevant available replica. So we can ensure that this node will contain the current checkpoint. The IP address of this node will be recorded in the TST; at the recovery phase, the same node will be used as coordinator. Since this node already contains the (up-tp-date) replica, it could execute the read locally, and does not need to forward the request to any other nodes (if CL is *ONE*). In this way, the query performance as well as the accuracy could be improved.

5.2 Experimental Setup.

Table 2 shows the setup of the testbed. We have deployed a five-node Cassandra cluster. Although the number of nodes is less than that in a practical game database, it is enough for our evaluations because we can already perform tests with different consistency levels, simulate a node failure, and get inconsistent data (see Table 1); 30 million rows have been pre-inserted in Cassandra cluster as well as the TST to simulate a real number of registered players in an MMORPG; In practice, there will be multiple TSTs in parallel. We have only deployed one in order to evaluate its performance under heavy workload; Each row in Cassandra contains a flexible number of columns from 110 to 160, which simulates the different number of properties that an object has; Column name is string type, and column value is integer type; Cassandra is mainly used in this scenario to back up data, so writes are significantly more than reads. During the experiment, we have executed 10 processes in parallel to communicate with Cassandra, with nine for writing and one for reading; If there is no special statement, the response time showed in following figures is the total time of executing 10000 write/read operations.

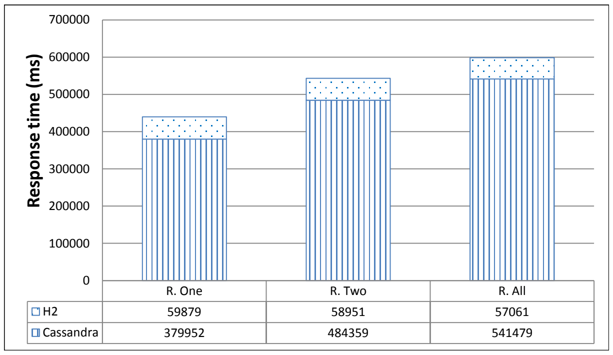
5.3 Effect of Accessing the TST.

As outlined before, the TST has a simple structure (only four columns). Compared to the query in a distributed disk resident database with data replication, its effect is negligible. Figure 2a and 2b present the response time of writes



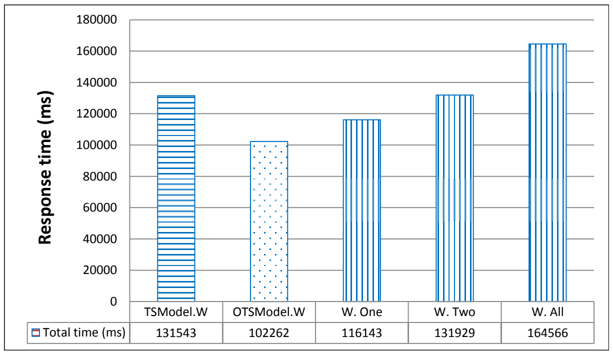
	W. One	W. Two	W. All
□ H2	13088	12648	11642
▥ Cassandra	116143	131929	164566

(a) Basic write performance.



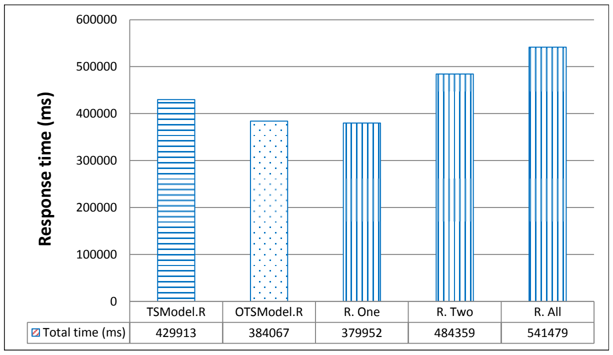
	R. One	R. Two	R. All
□ H2	59879	58951	57061
▥ Cassandra	379952	484359	541479

(b) Basic read performance.



	TSModel.W	OTSModel.W	W. One	W. Two	W. All
▤ Total time (ms)	131543	102262	116143	131929	164566

(c) Write performance with new strategies.

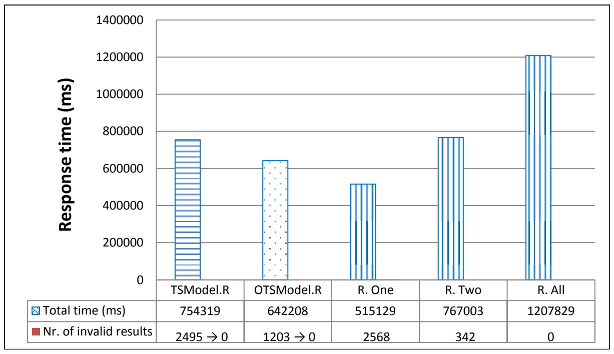


	TSModel.R	OTSModel.R	R. One	R. Two	R. All
▨ Total time (ms)	429913	384067	379952	484359	541479

(d) Read performance with new strategies.

Figure 2: Query performance in the new architecture.

and reads with different CLs. The response time of query-



	TSModel.R	OTSModel.R	R. One	R. Two	R. All
▨ Total time (ms)	754319	642208	515129	767003	1207829
■ Nr. of invalid results	2495 → 0	1203 → 0	2568	342	0

Figure 3: Read performance under node failure.

ing the TST only occupies about 9% in writes and about 12% in reads. Moreover, even if the response time of H2 is calculated, the total time of querying with a low CL is still shorter than the response time of querying Cassandra with a high CL. For instance, the total time of write *ONE* is 129231 ms, which is still shorter than performing write *TWO* in Cassandra (131929 ms).

5.4 Query Performance with new strategies.

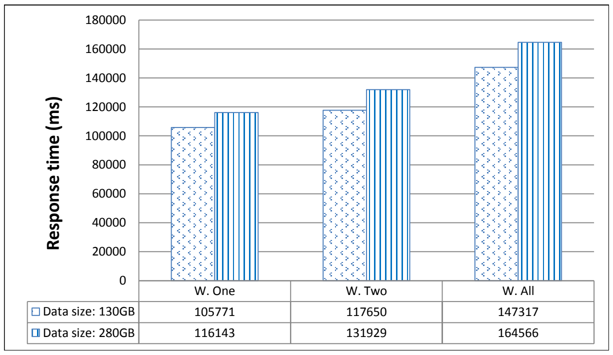
Figure 2c and 2d compares the write/read performance with different strategies and CLs when all five nodes are available. We name our timestamp-based model as *TSModel* and the optimised strategy (*NodeAwarePolicy*) as *OTSModel*. The total time of these two strategies also includes the response time of H2, which is not calculated in the other three methods. It is apparent from the figures that the query performance of *TSModel* is between write/read *ONE* and *TWO*. The query performance of the *OTSModel* is similar to or even better than write/read *ONE*. The reason is that there is less communication among nodes in the cluster (coordinator is one of the replicas).

5.5 Read Performance with node failures.

Similar to test four presented in Table 1, we have simulated the temporary node failure again. Figure 3 leads us to the conclusion that only read *ALL* could ensure to fetch the up-to-date data by just retrieving once. However, its performance is the worst. Although the response time of read *ONE* and read *TWO* is relatively shorter, both of them cannot guarantee data currency. The performance of *TSModel* and *OTSModel* is between read *ONE* and read *TWO*, whereas it can ensure to fetch all up-to-date data eventually. In practice, even though we have used the *NodeAwarePolicy*, invalid data (null value) could still be returned. Through tracing the query, we found that the coordinator did not always execute the request locally, sometimes it forwarded the request to another replica, which might be just recovered from a node failure, and consequently does not host the up-to-date data. The reason could be that the workload of this coordinator is too heavy, so the other replicas could process the request faster. But we can still state that the invalid data are halved, and consequently the read performance is much closer to read *ONE*.

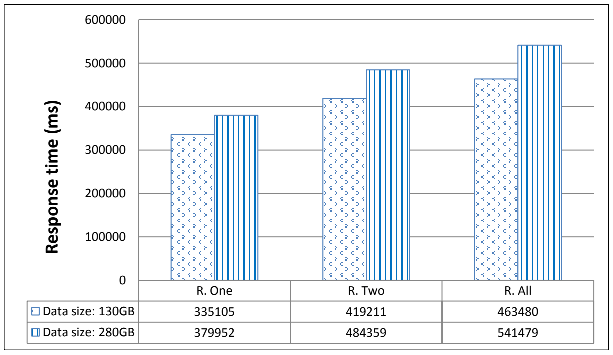
5.6 Effect of Data Size.

As discussed in Section 5, the data size of the cluster will increase obviously because of the stale data. Figure 4



	W. One	W. Two	W. All
▢ Data size: 130GB	105771	117650	147317
▥ Data size: 280GB	116143	131929	164566

(a) Write performance.



	R. One	R. Two	R. All
▢ Data size: 130GB	335105	419211	463480
▥ Data size: 280GB	379952	484359	541479

(b) Read performance.

Figure 4: Effect of data size.

describes the system performance under different data size (130 GB and 280 GB). Both write and read performance have been affected in this process. The time is wasted by retrieving a large number of files (SSTable) from disk. Therefore, we conclude that it is imperative to clean up all stale data timely.

The strategy of deleting stale data could be classified into two groups, namely, eager deletion and lazy deletion. Eager deletion refers to deleting the stale data instantly after flushing a new checkpoint; Lazy deletion describes that stale data will be deleted together asynchronously during a garbage collection at a specific time or under a certain condition (e.g., when the cluster is idle). Cassandra does not yet support a range query on the second compound primary key (TS). That means, the stale data could only be deleted one by one, thereby spending the same time, whichever strategy is chosen. Lazy deletion prevents bringing extra workload during peak hours. However, we have to record the timestamp of each checkpoint on the server side, or get it by executing an expensive read *ALL* in the cluster and use it to detect stale data. Overall, we need to choose the strategy based on the actual scenario.

6. RELATED WORK

The problem tackled in this paper was addressed before. E.g., [9] proposes to solve the game consistency issue by extending an existing Cloud storage system. The authors propose to support strong consistency, which will affect the system performance. Our timestamp-based approach could guarantee the data currency in an eventually consistent environment.

The possibility of using timestamps as version ID to identify current data in a replicated database has already been studied. However, these solutions either focus on the other storage systems or are designed for other application scenarios. For instance, [3] proposed a new key-based timestamping service to generate monotonically increasing timestamps, which is designed for distributed hash tables with a P2P structure. The main objective is to synchronize timestamps in a P2P environment and to scale up to large numbers of peers. The testbed in this work takes a client/server structure, where these are not problematic. Furthermore, our proposal is specific to Cassandra; the efficiency and accuracy of retrieval and garbage collection are our objective.

Similarly, in [15], authors also record the version ID got from a Cloud storage system for data currency. However, they suggested that if a read request fetched an old version ID from the Cloud, the system should wait for some milliseconds and try it again. Consequently, this approach blocks the read operation and increases unnecessary response time, which is not suitable for MMORPGs.

7. CONCLUSIONS

We presented an approach to use Cassandra for MMORPGs, which are a specific class of applications with high requirements regarding scalability. Nevertheless, the implied tradeoff between consistency and runtime performance and/or availability cannot be solved easily, as we demonstrated. For this purpose, we introduced several concepts to manage consistency in a multi-layered architecture. A key ingredient is a simple timestamp-based approach, which detects inconsistencies on the fly and only mends these if they occur. As the check itself is quite light-weight and, as also shown, situations triggering the inconsistency are very unlikely in our scenario, the new approach provides excellent runtime performance compared to strongly consistent operations as provided by Cassandra itself.

8. REFERENCES

[1] The database technology of guild wars. http://www.dbms2.com/2007/06/09/the-database-technology-of-guild-wars/.

[2] Mmo games are still screwed up in their database technology. http://www.dbms2.com/2009/06/14/mmo-rpggames-database-technology/.

[3] R. Akbarinia, E. Pacitti, and P. Valduriez. Data currency in replicated DHTs. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of data (SIGMOD 2007)*, pages 211–222, June 2007.

[4] P. Bailis and S. Venkataraman. Probabilistically Bounded Staleness for Practical Partial Quorums. *Proceedings of the VLDB Endowment*, 5(8):776–787, April 2012.

[5] T. Cao, B. Sowell, M. V. Salles, A. Demers, and J. Gehrke. BRRL : A Recovery Library for Main-Memory Applications in the Cloud Categories and Subject Descriptors. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data (SIGMOD 2011)*, pages 1233–1235, June 2011.

[6] T. Cao, M. Vaz Salles, B. Sowell, Y. Yue, A. Demers, J. Gehrke, and W. White. Fast Checkpoint Recovery Algorithms for Frequently Consistent Applications. In *Proceedings of the 2011 International Conference on Management of Data (SIGMOD 2011)*, pages 265–276, June 2011.

[7] R. Cattell. Scalable SQL and NoSQL Data Stores. *ACM SIGMOD Record*, 39(4):12–27, December 2010.

[8] J. Chen, M. Delap, and H. Lu. Locality Aware Dynamic Load Management for Massively Multiplayer Games. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP 2005)*, pages 289–300, June 2005.

[9] S. Das, D. Agrawal, and A. E. Abbadi. G-store: A Scalable Data Store for Transactional Multi Key Access in the Cloud. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC 2010)*, pages 163–174, June 2010.

[10] A. Demers, J. Gehrke, C. Koch, B. Sowell, and W. White. Database Research in Computer Games. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data (SIGMOD 2009)*, pages 1011–1014, June 2009.

[11] Z. Diao. Consistency Models for Cloud-based Online Games: the Storage System's Perspective. In *25th GI-Workshop on Foundations of Databases (Grundlagen von Datenbanken)*, pages 16–21, May 2013.

[12] Z. Diao, E. Schallehn, S. Wang, and S. Mohammad. Cloud Data Management for Online Games: Potentials and Open Issues. *Datenbank-Spektrum*, 13(3):179–188, October 2013.

[13] T. Fischer, M. Daum, F. Irmert, C. Neumann, and R. Lenz. Exploitation of event-semantics for distributed publish/subscribe systems in massively multiuser virtual environments. In *Proceedings of the Fourteenth International Database Engineering & Applications Symposium (IDEAS 2010)*, pages 90–97, August 2010.

[14] S. Gilbert and N. Lynch. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services. *ACM Special Interest Group on Algorithms and Computation Theory (SIGACT 2002)*, 33(2):51–59, June 2002.

[15] F. Gropengieß er, S. Baumann, and K.-U. Sattler. Cloudy Transactions Cooperative XML Authoring on Amazon S3. In *Datenbanksysteme für Business, Technologie und Web (BTW 2011)*, pages 307–326, March 2011.

[16] N. Gupta, A. Demers, and J. Gehrke. SEMMO : A Scalable Engine for Massively Multiplayer Online Games. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD 2008)*, pages 1234–1238, June 2008.

[17] N. Gupta, A. Demers, J. Gehrke, P. Unterbrunner, and W. White. Scalability for Virtual Worlds. In *Proceedings of the 25th International Conference on Data Engineering (ICDE 2009)*, pages 1311–1314, March 2009.

[18] T. Iimura, H. Hazeyama, and Y. Kadobayashi. Zoned Federation of Game Servers : a Peer-to-peer Approach

to Scalable Multi-player Online Games. In *Proceedings of 3rd ACM SIGCOMM Workshop on Network and System Support for Games (NETGAMES 2004)*, pages 116–120, September 2004.

[19] J. Kienzle, C. Verbrugge, B. Kemme, A. Denault, and M. Hawker. Mammoth A Massively Multiplayer Game Research Framework. In *Proceedings of the 4th International Conference on Foundations of Digital Games (FDG 2009)*, pages 308–315, April 2009.

[20] A. Lakshman and P. Malik. Cassandra - A Decentralized Structured Storage System. *Operating Systems Review*, 44(2):35–40, April 2010.

[21] F. W. Li, L. W. Li, and R. W. Lau. Supporting Continuous Consistency in Multiplayer Online Games. In *Proceedings of the 12th Annual ACM International conference on Multimedia (ACM Multimedia 2004)*, pages 388–391, October 2004.

[22] Y. Lin, B. Kemme, M. Patino-Martinez, and R. Jimenez-Peris. Applying Database Replication to Multi-player Online Games. In *Proceedings of 5th ACM SIGCOMM Workshop on Network and System Support for Games (NetGames 2006)*, page Article No. 15, 2006.

[23] M. Vaz Salles, T. Cao, B. Sowell, A. Demers, J. Gehrke, C. Koch, and W. White. An Evaluation of Checkpoint Recovery for Massively Multiplayer Online Games. *Proceedings of the VLDB Endowment*, 2(1):1258–1269, August 2009.

[24] W. Vogels. Eventually Consistent. *Communications of the ACM (CACM)*, 52(1):40–44, January 2009.

[25] Z. Wei, G. Pierre, and C.-H. Chi. Scalable Transactions for Web Applications in the Cloud. In *15th International Euro-Par Conference (Euro-Par 2009)*, pages 442–453, Augustune 2009.

[26] W. White, C. Koch, N. Gupta, J. Gehrke, and A. Demers. Database research opportunities in Computer Games. *ACM SIGMOD Record*, 36(3):7–13, September 2007.

[27] A. Yahyavi and B. Kemme. Peer-to-Peer Architectures for Massively Multiplayer Online Games: A Survey. *ACM Computing Surveys (CSUR)*, 46(1):Article No. 9, October 2013.