Periodicals of Engineering and Natural Sciences ISSN 2303-4521
Vol.7, No.1, June 2019, pp.125-140
Available online at: http://pen.ius.edu.ba

Efficient read monotonic data aggregation across shards on the cloud

Narayanan Venkateswaran[1], Anurag Shekhar[2], Suvamoy Changder[3], Rajib Kar[4] and Narayan C. Debnath[5]
[1]Department of Computer Science and Engineering,National Institute of Technology Durgapur, India
[2]MySQLOracle India Pvt Ltd., India
[3]Department of Computer Science and Engineering,National Institute of Technology Durgapur, India
[4]Department of Electronics and Communication Engineering, National Institute of Technology Durgapur, India
[5]Department of Software Engineering,Eastern International University, Vietnam

Article Info

Received Nov 21st, 2018

Keyword:

Horizontal Scalability
Sharding
High Availability
Cloud
Consistency
Client-Centric Consistency
Monotonic Reads
Version Vectors
Vector Clocks

ABSTRACT

Client-centric consistency models define the view of the data storage expected by a client in relation to the operations done by a client within a session. Monotonic reads is a client-centric consistency model which ensures that if a process has seen a particular value for the object, any subsequent accesses will never return any previous values. Monotonic reads are used in several applications like news feeds and social networks to ensure that the user always has a forward moving view of the data.

The idea of Monotonic reads over multiple copies of the data and for lightly loaded systems is intuitive and easy to implement. For example, ensuring that a client session always fetches data from the same server automatically ensures that the user will never view old data.

However, such a simplistic setup will not work for large deployments on the cloud, where the data is sharded across multiple high availability setups and there are several million clients accessing data at the same time. In such a setup it becomes necessary to ensure that the data fetched from multiple shards are logically consistent with each other. The use of trivial implementations, like sticky sessions, causes severe performance degradation during peak loads.

This paper explores the challenges surrounding consistent monotonic reads over a sharded setup on the cloud and proposes an efficient architecture for the same. Performance of the proposed architecture is measured by implementing it on a cloud setup and measuring the response times for different shard counts. We show that the proposed solution scales with almost no change in performance as the number of shards increases.

Corresponding Author:

Suvamoy Changder,
Department of Computer Science and Engineering,
National Institute of Technology Durgapur,
Mahatma Gandhi Avenue, Durgapur 713209,
West Bengal, INDIA
Email: suvamoy.nitdgp@gmail.com

1.Introduction

In a replicated setup, accessing data items from different replica servers can result in different results for the same application operation.

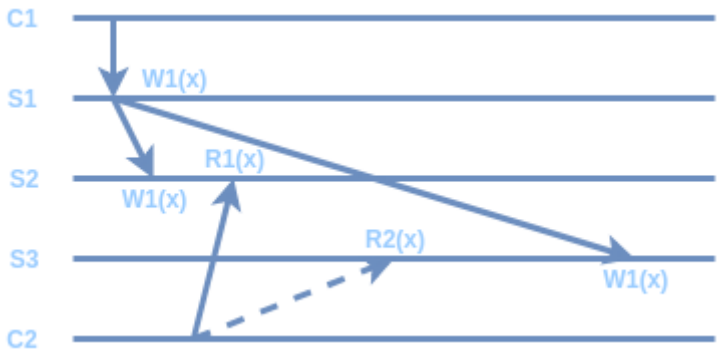

Figure 1: Reading from replicas

In Figure 1 the client C1 performs a write of data item X on server S1 that is replicated to servers S2 and S3. Now client C2 performs a read of data item X, which results in two cases:

- Read reaches server S2. Server S2 has already seen the write from server S1. C2 also has the write changes.
- Read reaches server S3. Server S3 has not seen the write from server S1. C2 does not have the write changes.

Thus, in the same session, client C2's read operation can have different results based on the replica server from which the read is performed [22] [23] [2]. This causes inconsistency in the client view within the same session.

Client-centric views define the view of the data storage expected by a client in relation to the operations done by a client within a session (over a single replica set). This helps describe the application consistency and requirement in read and write operations performed on the servers of that replica set. [6] [24].

Client-centric consistency is defined keeping a replicated dataset in mind [9] [19]. That is, clarity can be obtained about which server within a replica needs to be accessed to perform a client operation within a session. However, when the dataset is sharded across multiple replica sets [20] [18], the client-centric views do not clearly define the behaviour of queries accessing multiple shards (replica sets) within the same session.

This paper focuses on monotonic reads, one type of client-centric consistency, and shows that the existing definition of monotonic reads has shortcomings when extended to the sharding use case. The focus, specifically, is on maintaining consistency between data fetched from multiple shards (or replica sets), and an efficient architecture for achieving the same is proposed.

Section 2 explores the formal definitions and representations for monotonic reads and shows the use cases in which they falter. Section 3 proposes extensions to the existing theory on monotonic reads and describes an efficient architecture for implementing the same. Section 4 evaluates the performance of the proposed architecture on a cloud deployment and measures its performance for various shard configurations and update rates.

2. Background and Related Work

This section explores the existing theory around client-centric consistency and monotonic reads. The shortcomings of applying the existing theory on a sharding topology are also explained here.

2.1. Background

Client-centric consistency and the theory surrounding it are built around the existence of multiple copies of the same data item in one replica of a database. The theory and formal representations surrounding client-

centric consistencies [24] help predict what the clients see within a given session. The client session is often assumed to deal with multiple servers of the same replica set. Databases that popularized the notion of relaxed consistency like MongoDB [10], Riak [17], Cassandra [4] and Dynamo [7] addressed client-centric consistency with this restrictive notion of a session that operates over a single replica set.

In a truly distributed database, however, a client session has to deal with multiple replica sets, each containing a shard (partition) of the database in question. The existing theory and formal representations become significantly complex when a given query can run over multiple replica sets and consistency needs to be ensured across these replicas.

Sections 2.2 and 2.3 explore the existing theory and formal representations surrounding client-centric consistency.

2.2. Session

Eventual consistency and the client-centric consistencies that derive from it are defined within the boundaries of a session. A session represents a contracting unit of system service to clients (and their applications) and is composed of a denumerable sequence of read and write operations (performed by them). Sessions are intended to present individual clients with a view of the data storage that is consistent with their own actions in the system servicing unit. Within a session, a client can inquire and change the data storage under some promised service properties, e.g., consistency guarantee.

A session is usually defined within the boundaries of a server and a sequence of read and write operations performed by the client on the server. Thus, a session can simply be represented by equation 1:

$S_t = R[x], W[x] \mid x \text{ is a data item}$ (1)

For a session S_i with n operations, the j^{th} operation op_j and the $op_{(j+1)}{}^{th}$ operation are related by the equation 2, (where op is a read or write operation as seen in equation 1) [24]:

$op_j <_i op_{j+1}$ where $j \varepsilon [1, n]$ (2)

2.3. Client-centric Consistency

Four client-centric consistencies exist based on the different expectations of data states on session operations [6]:

- Read-your-writes - The effect of a write operation in a session is always seen by a later read operation within the same session.
- Monotonic reads - A read in a session always reads the same or more recent version of a data item that was previously read within the same session.
- Write-follows-reads - A write in a session always happens on the version of a data item read in a session or a more recent version.
- Monotonic writes - A write operation in a session is completed on a data item before any subsequent write operation on the same data item by the same session.

This paper will focus on monotonic reads consistency [24].

2.4. Client-centric consistency and Sharding

Using the formal definitions in section 2.3, it is possible to understand the behaviour of a query in a client session, when the data resides on a replication topology. However, when the data is partitioned across multiple shards, the same query operates across several replicas, each storing a different partition of the data. Trying to

extend the semantics directly from the definition over server replicas can lead to inconsistency between the data from the partitions. This cascades into an inability to view the partitioned data as a consistent dataset.
The next section attempts to list use cases that result in inconsistency when the existing semantics of monotonic reads is applied on shards. The decision to explore monotonic reads versus the broader area of client-centric consistency was taken to reduce scope, redundancy and clutter, while offering more possibilities for detailed analysis.

3. Monotonic reads in a sharded setup

This section provides formal representations for monotonic reads on a sharded setup and explains the challenges in performing consistent reads in such a setup. An architecture for overcoming these challenges and performing efficient monotonic reads over a sharded setup is also proposed.

3.1. Local vs Global Operations

In a sharded setup, the database schema is partitioned across multiple servers. A database operation (DDL, DML or Query) performed by the user happens,

- On a single shard
- On multiple shards

An operation that is performed on multiple shards is executed across multiple servers hosting the shards. This and the following sections refer to operations that span multiple shards as global operations. Since the paper primarily focuses on client-centric consistency, the term Global Operations is used to refer to any client operation that requires modifying or fetching data from multiple shards.

3.2. Formal Representations

In this section, the definition of read monotonicity in section 2.3 is extrapolated to sharding and it is shown that this can result in inconsistency. The section further refines this definition to include the changes for performing consistent reads across shards.

A session that contains reads and writes from multiple shards can be represented by equation 3:

$$S_t = R_{shk}[x], W_{shk}[x] \mid sh_k \varepsilon sh_1 , sh_2 \ldots sh_n \quad (3)$$

In the absence of global writes, monotonic reads in section 2.3 can be extended with the following definition,
A set of reads in a session from multiple shards satisfies read monotonicity, if the reads from the high availability cluster of each shard are read monotonic.

The definition above implies that equation 4 holds for each shard, where x_a, x_b, x_c are replicas of the data item x in the high availability setup on shard sh_k .

$$w_{shk}<R_{shk}(1)(x_a) \rightarrow w_{shk}<R_{shk}(2)(x_b) \quad (4a)$$
$$w_{shk}<R_{shk}(1)(x_a) \rightarrow w_{shk}<R_{shk}(3)(x_c) \quad (4b)$$

Equation 4 explains read monotonicity with respect to the local writes that happen on shard sh_k .

- Equation 4a states that the set of writes that happen on x_a before $R_{shk}(1)$ also happen on the copy x_b before $R_{shk}(2)$.
- Equation 4b states that the set of writes that happen on x_a before $R_{shk}(1)$ also happen on the copy x_c before $R_{shk}(3)$.

Note: The reads $R_{shk}(1)$, $R_{shk}(2)$ and $R_{shk}(3)$ are reads that happen on different replicas for a given shard.

This explains the impact of writes localized to the shards. However, this does not explain the impact of writes that are global across all the shards. The need for global writes and their impact on read monotonicity are explained in the next section.

3.3. Reading Consistent Data

Equation 4 talks about the relationship between writes and reads localized to a shard. In the event of a global write, however, the impact of the write is seen across all the shards in the topology.

In a sharded system, a user query might often need to access data from multiple shards. The data set corresponding to such a query will contain data from multiple shards. In order to be consistent, the dataset should take into account the impact of the global write operation across all the shards.

Thus, in the presence of global writes, equation 4 might ensure a read monotonic dataset, but it will not ensure a consistent dataset.

$W_g(x), R_{sh1}(1)(x), R_{sh2}(1)(x), R_{sh3}(1)(x)$ (5)

In the equation 5, the write $W_g(x)$ influences all the three shards sh_1, sh_2, sh_3. Thus, the consistency of the data represented by the read set $R_{sh1}(1)(x)$,

$R_{sh2}(1)(x)$, $R_{sh3}(1)(x)$ would depend on the result of the global write $W_g(x)$ being reflected in all the shards. This can be illustrated with the following example:

Consider a write $W_g(x)$ that occurs before a projection $\pi_{ck}(R)$ and adds the column ck to the schemas in shards sh_1, sh_2, sh_3. Assume that the write has not yet been reflected in the shard sh_2. If the projection runs on shard sh_2, it will fail because the column has still not been created in shard sh_2.

This data can still be presented to the user without violating the definition of read monotonicity. However, depending on the nature of the information, the absence of the data from a shard would make the dataset illogical to a user. For example, if each of the shards represented information from a country, the result of a query that summarizes information across all countries would look inconsistent with the information from a country completely missing.

3.4. Global State

The problem of presenting inconsistent data in section 3.3 happens because it becomes difficult to verify if all the shards are at the same global state. This section explains what global state means, creates formal representations for the global state and includes it in the representations of monotonic reads from the previous sections.

Sharding horizontally partitions a database across a given set of servers. Although the goal of sharding is to partition the database into independent elements, the user is oblivious to the underlying partitioning. The requirement of presenting the user with a unified picture of the underlying store forces the system designer to maintain some aspects that are common across all the shards, e.g., Schema, Global Tables, etc.

The global state represents the state of the aspects that are common across the shards. The schema of the sharded tables is one example of a global state. The user should see the same schema, irrespective of the shard queried. A change in schema needs to be reflected across all the shards. Similarly, the system designer could decide to keep a few tables common to all the shards. Pincode table is one common example of a global table.

It is also important to remember that not all aspects of the global schema are relevant to every query that is run. For example, if the sharded database contains the relations R_1, R_2, R_3 ... R_n and the global relations R_{G1}, R_{G2} … R_{Gn} , a projection $\pi_{Rcol>val}$ R_1 cares only about the global state of the relation R_1. Thus, performing consistent monotonic reads can be defined as follows,

A set of reads in a session from multiple shards satisfies monotonic reads and is consistent, if the reads from the high availability cluster of each shard satisfy monotonic reads and the shards have the same state for those aspects of the global schema that are relevant to the query being run.

The next section describes an efficient architecture for performing consistent monotonic reads across shards in the presence of global updates.

3.5. Proposed architecture for consistent reads in the presence of global updates

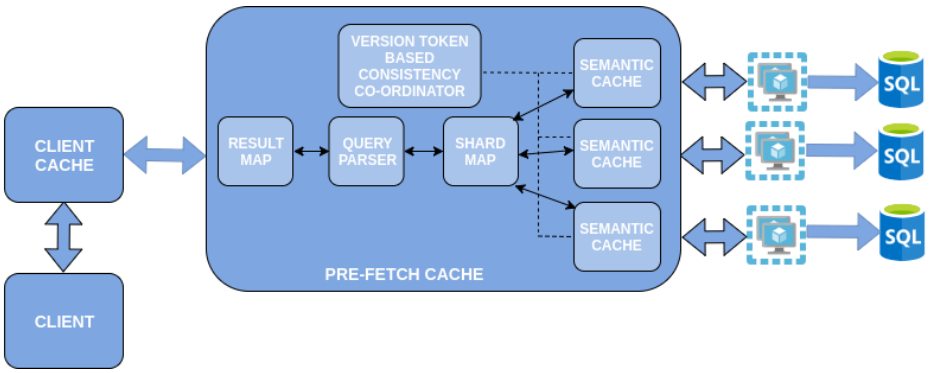

Figure 2: Version token aware prefetch cache

The proposed architecture extends a semantic cache [16] [21] to add version token awareness. The sections 3.5.1 and 3.5.2 give a brief introduction to semantic caching and version tokens respectively, before diving into the nuances of the proposed methodology in section 3.5.3.

3.5.1. Semantic caching of data in shards

Semantic caching [16] [21] allows a client to maintain both the semantic description and the result of a query in the cache. For a sharded setup, each query is decomposed into the constituent queries that need to be fired on each shard. The results of each of these queries are cached alongside the semantic descriptions.

The join EMP $\triangleright\triangleleft_{emp\_id}$ SAL $\triangleright\triangleleft_{dept_id}$DEPT, for example, is fired separately in each of the shards, for the appropriate emp ID ranges of these shards. That is, if the range of values in each of these shards is,

- $emp\_id_1 < emp\_id < emp\_id_2$
- $emp\_id_2 < emp\_id < emp\_id_3$
- $emp\_id_3 > emp\_id$

the join is decomposed into the following three joins,

- EMP $\triangleright\triangleleft_{emp\_id1 < emp\_id < emp\_id2}$ SAL $\triangleright\triangleleft_{dept_id}$DEPT
- EMP $\triangleright\triangleleft_{emp\_id2 < emp\_id < emp\_id3}$ SAL $\triangleright\triangleleft_{dept\_id}$ DEPT
- EMP $\triangleright\triangleleft_{emp\_id > emp\_id3}$ SAL $\triangleright\triangleleft_{dept\_id}$ DEPT

Each of the above joins is executed on each shard and the results are cached against the respective query.

3.5.2. Version tokens for cross-shard consistency

A version vector [14] [1] is often used as a mechanism for tracking the state of data in a distributed system. This section describes how version vectors can be used to track the state of the global aspects of the sharded system.

Each shard in the topology contains one vector element for each global aspect of the sharding topology. For example, if the sharded database contains the relations R_1, R_2, R_3, R_4 and the global relations R_{G1}, R_{G2}, the vector elements in a shard can be represented by the following;

R_1	R_2	R_3	R_4	R_{G1}	R_{G2}	
0	0	0	0	0	0	S_1

The vector above contains 0 as the value for each of the vector elements. This represents the initial state of the global aspects when the topology has not changed. When the state of a particular global aspect changes, the corresponding vector element is increased by 1. When the schema of the relation R_2 is changed, the vector is updated as follows:

R_1	R_2	R_3	R_4	R_{G1}	R_{G2}	
0	1	0	0	0	0	S_1

One more schema update of R_2 and an insert into a global table R_{G1} result in the following version vector:

R_1	R_2	R_3	R_4	R_{G1}	R_{G2}	
0	2	0	0	1	0	S_1

Thus, a change in the global aspect causes a change in the corresponding vector element.

The vectors from each of the servers can be consolidated as a matrix of vectors, such as the one given below:

R_1	R_2	R_3	R_4	R_{G1}	R_{G2}	
0	0	0	0	0	0	S_1
0	0	0	0	0	0	S_2
0	0	0	0	0	0	S_3

When a query that runs across multiple shards is executed, the aspects relevant to the query are extracted from the above matrix and this subset matrix is used for generating a consistent result.

For example, when a query that runs with relevant information from relations R_1, R_2 and the global relation R_{G1} is executed, the following matrix can be used to verify if the data is consistent:

R_1	R_2	R_{G1}	
0	0	0	S_1
0	0	0	S_2
0	0	0	S_3

A consistent query is executed when all the shards have the same state for the global aspects relevant to a query. Thus, all the values in a column should be the same, for consistent query execution. Therefore, in the above matrix the following conditions should be satisfied:

- (0, 0) = (1, 0) = (2, 0)
- (0, 1) = (1, 1) = (2, 1)
- (0, 2) = (1, 2) = (2, 2)

3.5.3. The Architecture

The basic idea of the architecture is shown in the figure 2. Each component of the architecture is explained in greater detail in this section.

Client The client is agnostic to the presence of shards, and fires a query as it would on an unsharded setup. For example, a client fires the join, EMP $\triangleright\triangleleft_{emp_id}SAL\triangleright\triangleleft_{dept_id}$DEPT, oblivious to the fact that the data in the tables is distributed across multiple shard servers. The client receives a consolidated result set containing the results of firing the query on all the shards.

Client Cache The client cache is a semantic cache that stores consolidated results from the prefetch cache. A query to be executed is sent from the client to the client cache. The client cache maps a query to its consolidated result. If a mapping is found, the results of the query are returned to the client immediately. A pipeline of consistent results is formed by the client cache by combining with the prefetch cache.

The client cache can be designed to be part of the same process as the client or be deployed in a different process, e.g., as a cluster of memcached servers.

Prefetch cache The prefetch cache is used for the amalgamation of a consistent dataset from the shards. It contains the logic for parsing a query and extracting the shard key range it operates over. This sharding key range is used to decide the shards the query should be executed on. The query is then passed to the semantic cache front-ends and the results obtained are combined into a consistent dataset before being presented to the client cache.

```
Result: Fetch consistent data from the shards
sh = 1;
sync = false;
iteration = 1;
max_iterations = 5;

while !sync and iteration≤ max_iterations do
        Refresh the semantic caches and update their version tokens;
        Lock the result map and update its contents;
        sync = true;
        Unlock the result map;
        if version_token of all the shards is not the samethen
           sync = false;
        end
        iteration = iteration + 1;
end


if !sync then
        Lock the arbitrator for updates;
        Refresh the semantic caches and update their version tokens;
        Lock the result map and update its contents;
        Unlock the result map;
        Unlock shards for updates;
end
```

Algorithm 1: Performing Efficient Monotonic Reads

The prefetch cache fetches the version tokens associated with the datasets along with the datasets themselves. The version tokens are used to ensure that the datasets fetched from the shards are consistent with each other. The prefetch cache also contains the logic necessary to compare the fetched version tokens and trigger a refetchin the event that an inconsistency is detected.

The prefetch cache does not necessarily need to run separately from the client cache. The choice of prefetch cache location is implementation dependent.

Result Map The result map performs the basic functionality of the semantic cache, mapping the query to its consolidated result set. If the result set is not available, a fetch is triggered from the shards.
Query Parser The query parser extracts the value of the shard key from the query that is fired on the cache. The value of the shard key is used for mapping the incoming query to the shards.

Shard Map The shard map caches the metadata of the sharding topology. It is responsible for splitting the incumbent query into the constituent queries on the shards. Given an incoming query EMP $\triangleright\triangleleft_{emp_id}SAL\triangleright\triangleleft_{dept_id}$ DEPT, the shard map splits the query into the following queries on three shards:

- EMP $\triangleright\triangleleft_{emp\_id1 <emp\_id< emp\_id2}$ SAL $\triangleright\triangleleft_{dept_id}$DEPT
- EMP $\triangleright\triangleleft_{emp\_id2 <emp\_id< emp\_id3}$ SAL $\triangleright\triangleleft_{dept\_id}$ DEPT
- EMP $\triangleright\triangleleft_{emp\_id>emp\_id3}$ SAL $\triangleright\triangleleft_{dept\_id}$ DEPT

Semantic Caches There is one semantic cache for each shard. The constituent queries are fired on the semantic cache of the respective shard.

The version tokens fetched are used to check if the data from the shards are consistent with each other. If an inconsistency is detected, the maximum of the fetched version tokens is used as a baseline and a refetch is triggered on that semantic cache.

Version Token Based Consistency Coordinator The version token based consistency coordinator maintains the consistency between the contents of the semantic caches. It maintains the version vector matrix for the global data in the shards, as explained in section 3.5.2, and ensures that the data fetched from the shards belong to the same global state in the shards.

For example, assume a setup containing sharded relations R_1, R_2 and a global relation R_{G1} . If a fetch from the shards results in the following version vector matrix,

R_1	R_2	R_{G1}	
0	0	0	S_1
0	0	0	S_2
0	0	0	S_3

it can be seen that there is a version token mismatch for relation R_1. The maximum of the fetched version tokens is 1. Since the version token for the shard in server S_3 is 0, a re-fetch happens from this server.
For the same setup, the following version vector matrix will trigger a re-fetch from all the shards:

R_1	R_2	R_{G1}	
1	0	0	S_1
1	0	0	S_2
0	0	1	S_3

When the global update rate is very high, there could be a situation where the prefetch cache is playing catch up and consensus cannot be reached. Although such a continuous set of global updates is illogical and signifies a failure of the sharding scheme, there are multiple techniques for attaining consensus, e.g., Atomic commitment protocols [11] [15], Distributed Locking [3] [8], Central Arbitration, etc.

In this paper, a central arbitrator is used for routing updates across all the shards. This is shown in figure 3. A client who wants to update all the shards sends the updates to the central arbitrator. The arbitrator takes care of reliably routing the updates to all the shards.

When the update rate becomes very high, the central arbitrator is used to lock updates for a brief interval, allowing the shards to reach consensus. Algorithm 1 formally states this refresh algorithm.

Version Token Enabled Data Stores Several data stores support the creation of persistent version tokens in databases [13]. These version tokens can be queried to enquire for the state of the data in the specific shard. The architecture represented in figure 2 assumes support for version tokens in the datastores.
In the next section, the results of deploying the architecture on cloud servers are presented.

4. Quantitative Analysis

4.1. Cloud Setup and Sharding Schema

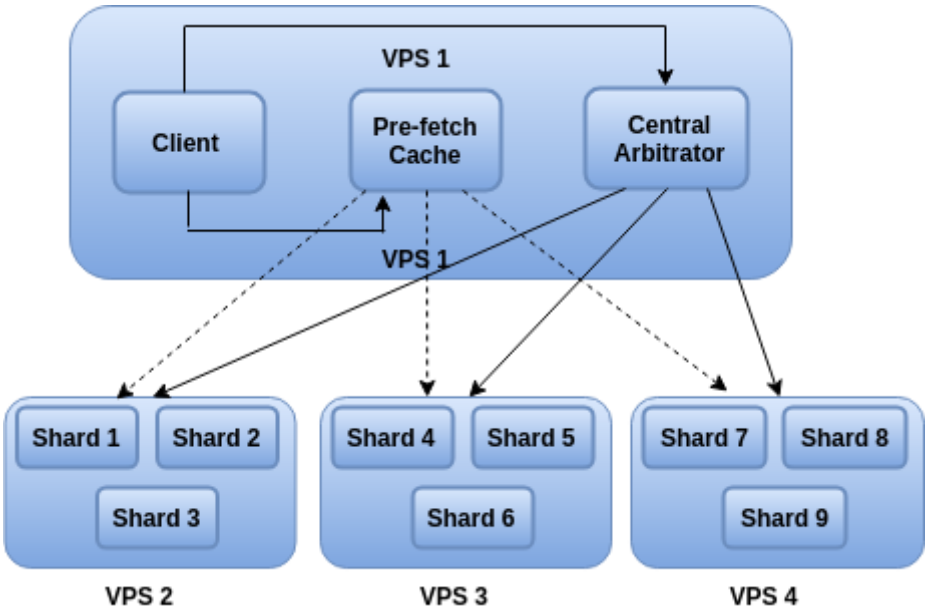

Figure 3: Basic setup

The setup used to obtain the quantitative metrics was built on virtual private servers (VPS) obtained from the cloud service provider - cloudatcost [5]. This is shown in figure 3, which also depicts the flow of updates through the setup. Four VPS were used to deploy the components of the system. The VPS were Ubuntu based, with 6 GB RAM and 100 GB of hard disk each. The client and the prefetch cache were written in Python, whereas the central arbitrator and the shards were MySQL 5.7 servers.

The client, the prefetch cache and the central arbitrator were deployed on the same VPS. The central arbitrator in this case was a database server (MySQL) in an asynchronous replication setup with the servers that were part of the shards. If the central arbitrator were locked for updates, it would also stop the flow of updates to the shards. Queries from the client were routed through the prefetch cache. If the prefetch cache contained the results of the query, it is returned to the clients directly. Otherwise, the pre-fetch cache fetches the results from the servers and returns them to the client.

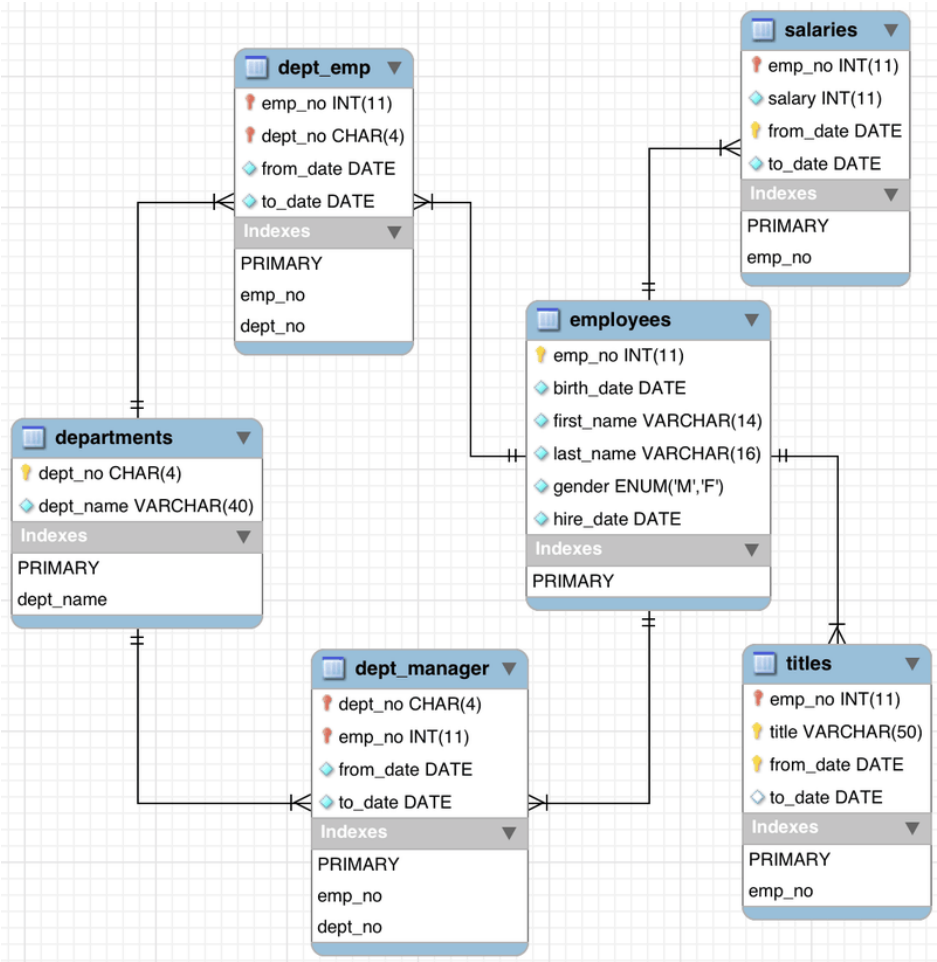

Figure 4: MySQL Employee Schema

Figure 4 contains the Entity Relationship diagram of the schema used to store the data that was sharded. The schema represents the popular Employees Sample Database [12], which helps provide a large base of data spread over six separate tables. This structure is simple and easy to visualize, while at the same time being comprehensive. The employee database was sharded using the emp_no as the sharding key.

The schema was populated with 100 first names and 1000 last names. A full join of these two generated 100000 unique names. The emp_no was changed to an auto-generated column to generate primary keys. For each employee a birth_date and hire_datewere randomly generated. An insert trigger was created on the employees table for generating the salary history of an employee starting from the joining date to the current date. Each employee was assigned a department from one of the 9 available departments.

A simple range based partitioning scheme was used to distribute the data across the shards. In the range based scheme employees with emp_no 0 – 9999 were placed in shard 1, emp_no 10000 - 19999 were placed in shard 2 and so on. An increment of the salary of all the employees by 1 was used as the global update operation. The global update was performed at random intervals through the central arbitrator.

Listing 1: Table Scan Query

```
SELECT COUNT( * ) FROM (
  SELECT S.emp_no, E.first_name, E.last_name, MAX(S.salary)
  FROM Salaries AS S, Employees AS E,
  WHERE S.emp_no = E.emp_no
      AND S.emp_no< %(max)s
      AND S.emp_no> %(min)s
```

```
  GROUP BY S.emp_no
)
```

The query in listing 1 was used to generate load and also for latency measurements. The query was designed to perform a full scan of the data in the shard.

4.2. Quantitative Analysis

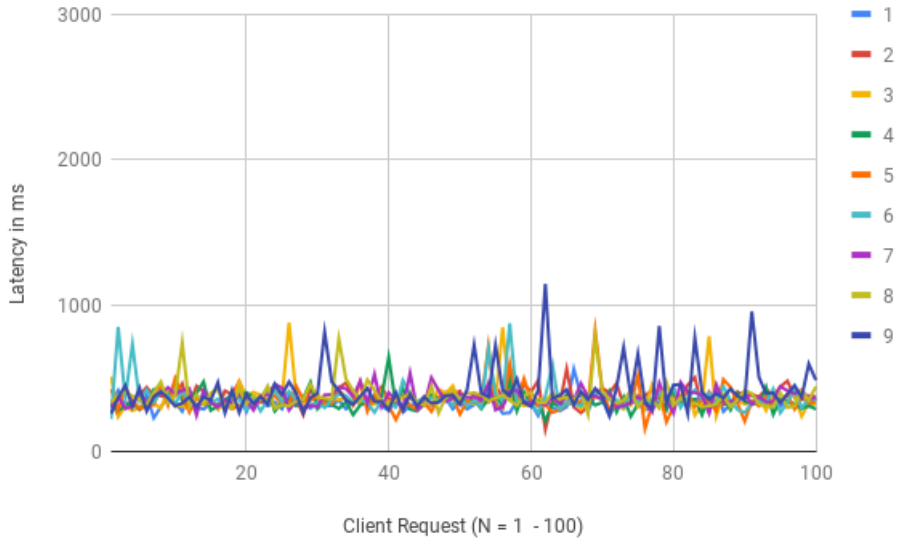

Figure 5: Latency (in ms) for 100 requests in each shard N (1 - 9)

Figure 5 plots the latency of 100 requests for each shard setup. It should be noted that the latencies are in milliseconds. It can be seen from the figure that barring a few spikes that can be accounted for by unreliable performance of the cloud machines and the network, the latency numbers obtained are uniform across the shards. From the graph it is safe to conclude that the proposed architecture performs extremely well as the number of shards increases.

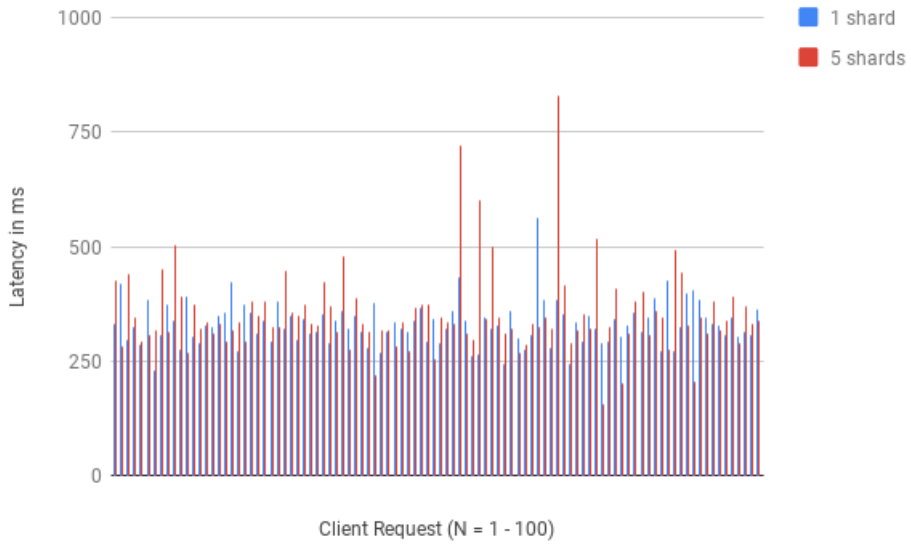

Figure 6: Latency (in ms) for 100 requests for shard counts 1 and 5

Figure 6 plots the latency of the requests in the unsharded setup (N = 1) against the setup having 5 shards. The impact of the unreliable cloud machines and the network can be seen in the unsharded setup, too. Analysis of

the reason for the prominent spikes in the sharded setup revealed that they coincided with the times when the result map was locked for updating from the semantic caches.

It can be observed that the spiking latencies are not clustered and that the latency in the unsharded case is very similar to the latency of the requests in the sharded case. This shows that with the proposed architecture the performance of monotonic reads is not impacted by sharding. Thus, the architecture leverages the reduced consistency guarantees very efficiently.

Table 1: Mean and Median latency (in ms) of 100 requests in each shard N (1- 9)

Shards (N)	Mean	Median
1	331.66	327.05
2	518.83	379.05
3	361.06	346.57
4	349.00	344.22
5	352.11	332.51
6	372.76	351.10
7	370.88	371.76
8	377.50	360.17
9	418.49	381.65

The mean and median of the obtained latencies over the 100 observations were also calculated to find the average performance of the proposed architecture. The obtained measures of central tendency are presented in table 1 and plotted in figure 7.

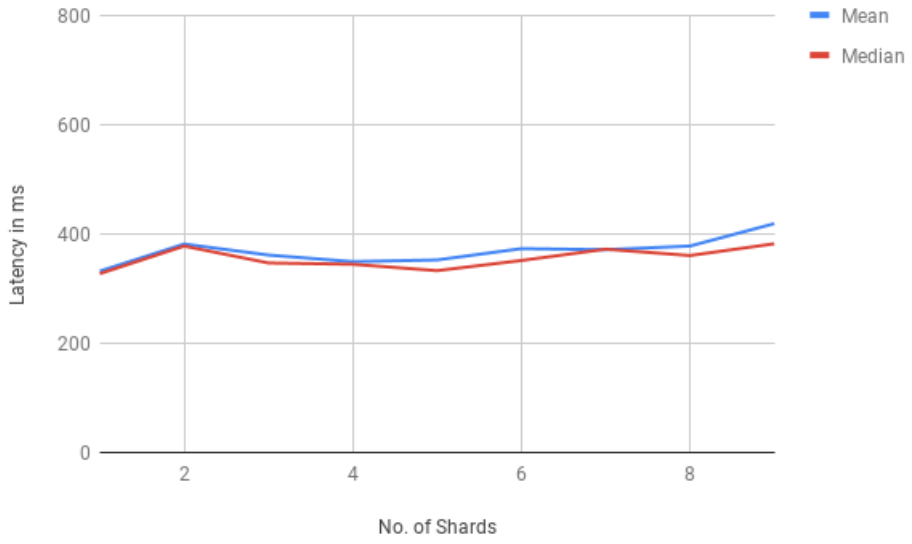

Figure 7: Mean and Median latency (in ms) of 100 requests in each shard N (1- 9)

It can be seen that, in spite of the spike caused due to cloud server and network performance, the observations are still close to each other. This indicates that the average latency is not influenced by the change in partitioning. The architecture thus shows excellent scaling for the central tendency measures across increasing shards.

A point worth noting is that averages are prone to being heavy influenced by the outliers. The median, on the other hand, is relatively free of spikes. However, for the given observations both the mean and the median are within acceptable boundaries.

Thus, it can be concluded from the quantitative analysis that the architecture scales with almost no change in performance as the number of shards increases. Hence it provides a very efficient solution for consistent monotonic reads in the presence of global updates.

5. Conclusion and Future Work

This paper looks in detail at the existing theory behind client-centric consistency, and monotonic reads in particular. The shortcomings of the existing theory surrounding read monotonicity when applied to a sharding / partitioning setup are discussed. It is shown that directly extending the existing theory to a partitioned setup can result in inconsistency. A novel method is then proposed that can help avoid inconsistency without affecting the performance the user will see from monotonic reads. This method is implemented on the cloud, and as shown the method performs well even as the number of shards increases.

It should be noted that the paper deliberately focuses on read monotonicity for the sake of simplicity and to emphasize the issues that arise when migrating to a sharded setup. Each of the client-centric models offers additional complexity and challenges that should be addressed on a case-by-case basis. Analysis of these models on a sharded setup would offer considerable scope for research.

Acknowledgments

I would like to express my deepest gratitude to Dr SuvamoyChangder and Dr Narayan C. Debnathwithout whose guidance and encouragement, the paper would not have taken shape.

References

[1] P. S. Almeida, C. Baquero, and V. Fonte. “Version stamps-decentralized version vectors”. In: Proceedings 22nd International Conference on Distributed Computing Systems. 2002, pp. 544–551.
[2] D. Barbara and H. Garcia-Molina. “The case for controlled inconsistency in replicated data”. In: [1990] Proceedings. Workshop on the Management of Replicated Data. Nov. 1990, pp. 35–38.
[3] Mike Burrows. “The Chubby Lock Service for Loosely-coupled Distributed Systems”. In: Proceedings of the 7th Symposium on Operating Systems Design and Implementation. OSDI ’06. Seattle, Washington: USENIX Association, 2006, pp. 335–350.
[4] Cassandra Architecture.
https://docs.datastax.com/en/cassandra/2.0/cassandra/architecture/architectureIntro_c.html.
Accessed:2016-02-16.
[5] Cloud At Cost. https://www.cloudatcost.com/. Accessed: 2018-07-29.
[6] W. Golab et al. “Client-Centric Benchmarking of Eventual Consistency for Cloud Storage Systems”. In: 2014 IEEE 34th International Conference on Distributed Computing Systems. June 2014, pp. 493–502.
[7] Deniz Hastorun et al. “Dynamo: amazon’s highly available key-value store”. In: In Proc. SOSP. 2007, pp. 205–220.
[8] Patrick Hunt et al. “ZooKeeper: Wait-free Coordination for Internet-scale Systems”. In: Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference. USENIXATC’10. Boston, MA: USENIX Association, 2010, pp. 11–11.
[9] BeomHeyn Kim, Sukwon Oh, and David Lie. “Consistency Oracles: Towards an Interactive and Flexible Consistency Model Specification”. In: Proceedings of the 16th Workshop on Hot Topics in Operating Systems. HotOS ’17. Whistler, BC, Canada: ACM, 2017, pp. 82–87.
[10] Y. Liu, Y. Wang, and Y. Jin. “Research on the improvement of MongoDB Auto-Sharding in cloud environment”. In: 2012 7th International Conference on Computer Science Education (ICCSE). July 2012, pp. 851–854.
[11] C. Mohan, B. Lindsay, and R. Obermarck. “Transaction Management in the R* Distributed Database Management System”. In: ACM Trans. Database Syst. 11.4 (Dec. 1986), pp. 378–396.
[12]MySQL Employee Sample Database. https://dev.mysql.com/doc/employee/en/sakila-structure.html. Accessed: June, 2018.
[13] MySQL Version Tokens. https://dev.mysql.com/doc/refman/5.7/en/version-tokens.html. Accessed: 2018-07-29.

[14] D. S. Parker et al. “Detection of Mutual Inconsistency in Distributed Systems”. In: IEEE Trans. Softw. Eng. 9.3 (May 1983), pp. 240–247.
[15] Yoav Raz. “The Dynamic Two Phase Commitment (D2PC) protocol”. In: Database Theory — ICDT ’95. Ed. by Georg Gottlob and Moshe Y. Vardi. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 162–176.
[16] Qun Ren, M. H. Dunham, and V. Kumar. “Semantic caching and query processing”. In: IEEE Transactions on Knowledge and Data Engineering 15.1 (Jan. 2003), pp. 192–210.
[17] Riak Architecture. http://docs.basho.com/riak/latest/ops/mdc/v3/architecture/. Accessed: 2016-02-16.
[18] L. Saino, I. Psaras, and G. Pavlou. “Understanding sharded caching systems”. In: IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications. Apr. 2016, pp. 1–9.
[19] Rashed Salem, Safa’a S. Saleh, and Hatem Abd elkader. “Scalable Data-Oriented Replication with Flexible Consistency in Real-Time Data Systems”. In: 15 (Mar. 2016).
[20] Charles Bell; Mats Kindahl; Lars Thalmann. MySQL High Availability, 2nd Edition. O’Reilly Media, Inc., Apr. 2014.
[21] Z. Yang, K. Ma, and J. Zhong. “Toward a Semantic Cache Supporting Version-Based Consistency”. In: 2016 10th International Conference on Complex, Intelligent, and Software Intensive Systems (CISIS). July 2016, pp. 367–372.
[22] Chi Zhang and Zheng Zhang. “Trading replication consistency for performance and availability: an adaptive approach”. In: 23rd International Conference on Distributed Computing Systems, 2003. Proceedings. May 2003, pp. 687–695.
[23] Irene Zhang et al. “Building Consistent Transactions with Inconsistent Replication”. In: Proceedings of the 25th Symposium on Operating Systems Principles. SOSP ’15. Monterey, California: ACM, 2015, pp. 263–278.
[24] Yuqing Zhu and Jianmin Wang. “Client-centric Consistency Formalization and Verification for System with Large-scale Distributed Data Storage”. In: Future Gener. Comput. Syst. 26.8 (Oct. 2010), pp. 1180–1188.

Authors

	**Narayanan Venkateswaran** is a researcher in the Computer Science and Engineering Department of National Institute of Technology Durgapur. He has over 13 years of experience in the industry having worked in Enterprise DB, Oracle, Amazon, Microsoft and Sun Microsystems. He is an open source committer in the Apache Derby project and has contributed extensively to several databases, including JavaDB and MySQL.
	**Anurag Shekhar** is a Senior Principal Member Technical Staff at Oracle India Pvt Ltd. He has over 22 years of experience in the Software industry, having worked in Sun Microsystems, IBM and Oracle. He has contributed heavily to several databases including, DB2, MySQL and JavaDB. He has mentored several junior engineers and has worked extensively in the area of database internals and file systems.
	**Dr SuvamoyChangder** is an Assistant professor in the Department of Computer Science and Engineering of National Institute of Technology Durgapur. He completed his PhD in Information Security, with specialization in Steganography and watermarking, from NIT Durgapur. He has over 15 years of teaching experience in addition to several years in the information technology industry before that. Some of the subjects that Dr Suvamoy handles include Data Structures, Design and Analysis of Algorithms, Information and Coding Theory, Database Management Systems.
	**Dr Rajib Kar**passed B. E. degree in Electronics and Communication Engineering, from Regional Engineering College, Durgapur, West Bengal, India in the year 2001. He received the M. Tech and Ph. D. degrees from National Institute of Technology, Durgapur, West Bengal, India in the year 2008 and 2011, respectively. Presently, he is attached with National Institute of Technology, Durgapur, West Bengal, India, as Assistant Professor in the Department of Electronics and Communication Engineering. His research interest includes VLSI circuit optimization, Signal Processing via Evolutionary Computing Techniques. He has published more than 350 research papers in International Journals and Conferences.
	**Dr. Narayan C. Debnath** has been a Full Professor of Computer Science since 1989 and currently the Chairman of Computer Science at Department of Software Engineering, Eastern International University, Vietnam. He is also serving as the Director and Past-President of the International Society for Computers and Their Applications (ISCA). Dr. Debnath is a recipient of a Doctorate degree in Computer Science and a Doctorate degree in Applied Physics (Electrical Engineering). In the past, he served as the President, Vice President, and Conference Coordinator of the International Society for Computers and Their Applications (ISCA), and has been a member of the ISCA Board of Directors since 2001. He served as the Acting Chairman of the Department of Computer Science at Winona State University and received numerous Honors and Awards. During 1986-1989, Dr. Debnath was a faculty of Computer Science at the University of Wisconsin-River Falls, USA, where he was nominated for the National Science Foundation Presidential Young Investigator Award in 1989.