An Object-oriented Approach for Extending MySQL into NoSQL with Enhanced Performance and Scalability

H. Shim, Y. Sohn, Y. Sung, Y. Kang, I. Kim, and O. Kwon
Samsung Electronics Co., Ltd
{hyunju67.shim, ycsohn, yw.sung, ygace.kang, ij00.kim, ohoon.kwon}@samsung.com

***Abstract*—This paper introduces an object-oriented approach for extending MySQL into Not-only SQL (NoSQL) for enhanced performance and scalability. Main goal of our system is to provide a database management system that handles a huge amount of data in a fast and reliable way. In designing database management Application Programming Interface (API), we adopted an Object-Relational Mapping (ORM) approach that provides a mapping between objects in user code with tables in database. This mapping allows developers to handle data in database tables by manipulating the objects that are mapped to. To provide a flexible and efficient distribution of large amount of data among multiple database nodes, we designed a unique ID scheme which is optimized for the primary key lookup operations by encoding the shard key information into its data ID. In this way, queries predicated with data ID can always go directly to the target node no matter which key the data is distributed on. With our ORM approach, query predicates are composed as a combination of Java method calls and no query parsing is necessary at database layer. Having no query to be parsed, we leveraged HandlerSocket, a MySQL plugin, which bypasses the upper layer of MySQL hence improving overall performance. To evaluate our system, we performed a set of tests and proved that our system provides improved performance and linear scalability compared to the traditional MySQL approach.**

Keywords-performance, robustness, scalability, object-relational mapping, distributed query processing

I. INTRODUCTION

Recently, due to the huge success in social networking services such as Facebook and Twitter, the amount of user data to be handled by a single service has exponentially grown. To support a service where a huge number of users generate tons of data at every second, its database management system must not only be fast and robust but also be highly scalable and available. In database system, scalability is defined as an ability to increase the total throughputs linearly as database storages are added. With highly scalable database system, simply adding more database nodes to the system will handle ever increasing user data without performance degradation. For highly available system, its database solution must eliminate any single-point-of-failure in system and guarantee its service level agreement.

Traditional database management systems (DBMS) designed to serve complex queries with Atomicity, Consistency, Isolation, and Durability (ACID) properties have demonstrated its architectural limitations in handling the large amount of data that recent social services generate. To remedy this situation, researchers investigated in alternative database management systems that can handle large amount of data with high performance and scalability. As a result of those efforts, a new type of database management systems called NoSQL (Not only SQL) were introduced including Amazon's Dynamo [1] and Google's Bigtable [2]. Unlike data in traditional database systems were modeled and handled as relational tables, data models in NoSQL are characterized into several categories such as document-based [3], column-oriented [4], key-value pairs [5], graph-based [6], and etcetera.

Although NoSQL provides a fast access to vast amount of data with variety of data models, being recent technology, they are not robust enough compared to the traditional DBMS such as Oracle [7] and MySQL [8]. Also, being distributed database system, NoSQL has an innate trade-off between high-availability and data consistency [9] and recent trend is that developers choose NoSQL or RDBMS for specific needs depending upon the nature of their data and services [10]. In this paper, we introduce a data access framework we developed on top of MySQL to provide a fast and robust data access with high scalability and availability.

Section II describes about the overall architecture of our system. Section III and Section introduce our approach to database sharding and programming model. Section V describes our experimental results and, finally, Section VI concludes this paper and explains about our future works.

II. OVERALL ARCHITECTURE

Main goal of our system is to provide a fast and robust data access framework with high scalability and availability as followings:

- For high scalability, we horizontally partitioned tables into multiple database nodes.
- For high availability, we automated the procedures for master failover and provided an online tool for shard rebalancing that redistributes data from one shard to another.
- For robust data management, we leveraged the MySQL storage engine, which has been used for many commercial services for a long time.
- For fast data access among multiple data nodes, we designed a unique ID scheme which is optimized for the ID lookup operation no matter which key the data is distributed on. This is done by encoding the shard key information into the data ID.

Copyright (c) IARIA, 2014. ISBN: 978-1-61208-334-6

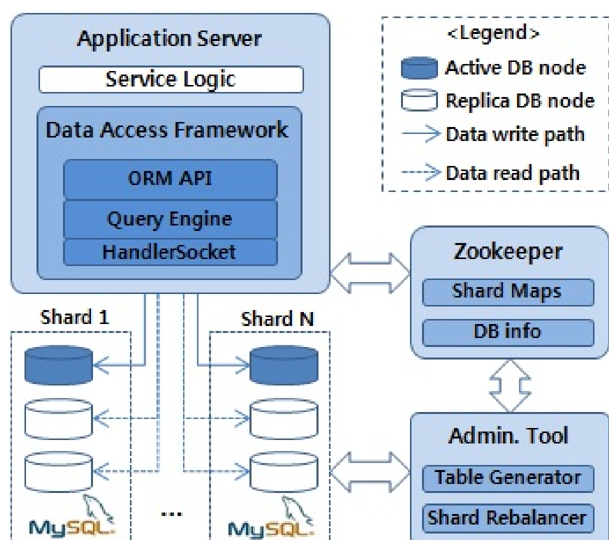

Figure 1: Overall Architecture of Our System.

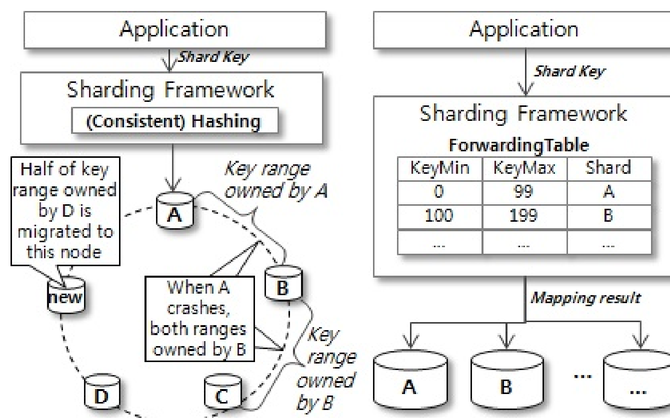

Figure 2: Concepts of Hash Sharding (left) and Rule Sharding(right).

- For fast data access in a single data node, we adopted HandlerSocket plug-in [11] and made read requests be routed to replica nodes.

As shown in Figure 1, our system is mainly composed of four parts: Data Access Framework in client side, which builds query conditions from user code and distributes queries among multiple MySQL nodes, Zookeeper which coordinates shard maps that resides in client nodes, MySQL database nodes with Active-Standby replications, and Administration Tools which provide command line interfaces for shard creation and online rebalancing. Among many features of our system, data distribution, programming API, performance, and scalability aspects of our system are discussed in the following sections.

III. Horizontal Partitioning of Data

A. Basic Concept of Database Sharding

In our system, we achieved database scalability by horizontally partitioning a single table into multiple data nodes where rows of a table are held separately based on the values of a certain key. This vertical partitioning of tables is called sharding and the key used to divide value range of data is called shard key. Two famous approaches for sharding are hash-based sharding and range-based sharding. In the hash-based sharding, data is simply distributed based on the hash values of its shard key hence data are evenly distributed among multiple data nodes. However, since data are scattered among multiple shards - data nodes, it is inefficient to perform queries with range conditions e.g., smaller than, larger than, and etcetera. For those range queries, every shard needs to be accessed to check if the node has the data that meet the range conditions. Also, with a simple hash, when a new data node is added into the system to increase the total capacity of storage, data from every node must be relocated based on the new hash function.

To remedy this data migration of every shard in a simple hash-sharding, consistent hashing was introduced [12]. In consistent hashing, hash key space is mapped to the points on a ring representation and each point is mapped to a data node. Thus, each data node covers a certain range of hash key space represented on the ring. When the number of data nodes changes due to crashing of any node or addition of a new node, only K/N keys need to be reallocated on average where K is the number of keys and N is the number of points or data node on the ring. To reduce data migration in case of node crash, data in the nodes are replicated in such a way that each node in the ring keeps the master copy of its own key ranges and replicas of adjacent nodes. In the left image of Figure 2, data being replicated to its adjacent nodes, when the node A crashes, requests made to A are then routed to node B hence node B became the master for the data whose hash key space is mapped to A and B. When a new node is added, a hash key range owned by D is divided into two and half of them are migrated to the newly added node letting the newly added node become a new master node of the migrated half.

Unlike hash-based sharding, rule-based sharding groups shard key ranges and maps the groups to a set of data nodes. The file that keeps this mapping information is normally referred to as a forwarding table. Advantages of rule-based sharding are flexible data distribution and efficient support for ranged-queries. With range-based sharding, database administrators can make sharding rules specific to their applications and data with similar shard key values are stored in the same data node hence supporting range queries efficiently. One of main disadvantages of range-based sharding is uneven distribution of data. If the sharding rule does not reflect the actual distribution of data then some of data nodes might get congested while others are sparse. To make data nodes evenly distributed, frequent data migrations and shard rule updates might be required. Right image of Figure 2 illustrates the main concept of rule-based sharding. In our system, we designed a modified rule-based sharding with doubly mapped forwarding table and fixed length of ID. Following section explains details about the sharding in our system.

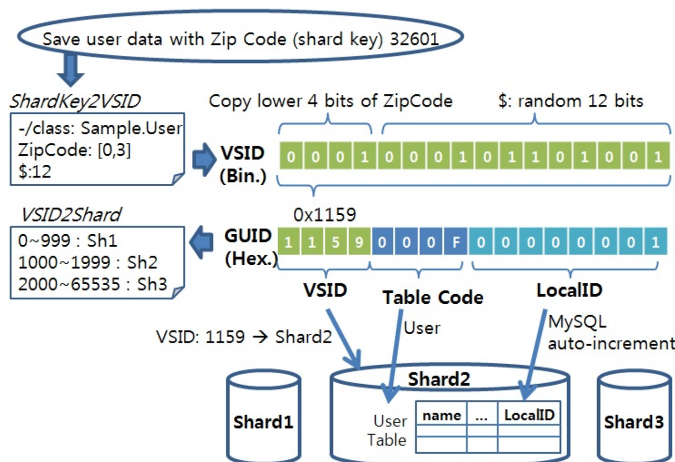

Figure 3: How Sharding Works in Our System.

B. Doubly Mapped Range-based Sharding

When distributing data among multiple nodes, they must be grouped based on the values of a certain key, which is called a shard key. For efficient distribution of data, it is important to select a shard key that is common for most of queries in the system. Because, once data is distributed among multiple shards based on their shard key values, queries predicated with a shard key can be directly sent to the target shard by referring to the forwarding tables while others must be sent to every shards to see if the data to be queried reside in there.

For an efficient sharding of data where most of their queries are ID lookup operations, we designed a 64 bits ID scheme in such a way that the shard key values are encoded within it. With this approach, data is distributed based on any key while the ID lookup operations can always go directly to its destination shard by referring to the shard rule files. In the ID scheme we designed, the first 16 bits represent the virtual shard ID (VSID), which is mapped to a physical database node, the middle 16 bits represent a database table and the remaining 32 bits represent a local ID which is unique within the table.

For data read and write operations, our system maintains two mapping files: ShardKey2VSID and VSID2Shard the ShardKey2VSID file maps the shard key value ranges to 2^{16} VSIDs and the VSID2Shard file maps the 2^{16} VSIDs to the physical database nodes in system. For data writing, a VSID is first assigned based on its shard key value by referring to the ShardKey2VSID rule file. Once a VSID is assigned, a target database is determined by referring to the VSID2Shard rule file. Once the physical shard determined, data is stored in the data types table with the local ID provided by the MySQLs auto-increment functionality. Figure 3 shows how ID is generated for data and stored in our system. For data reading with ID, our system looks at the first 16 bits of the ID, which is the VSID part, and finds the target shard by referring to the VSID2Shard file. For data queries including shard keys values can also be directly sent to the destination shards by comparing the shard key values with the ShardKey2VSID and VSID2Shard files. Like other sharding systems, queries without shardkeys must be sent to every shard.

One of the main design goals of our VSID is to support flexible sharding of data with multiple shard keys while encoding the shard key values into the fixed length of ID. Using our system, a database administrator can set rules for each 16 bits to distribute data according to the characteristics of their applications. For example, sharding a user data according to the birthday and zip code in such a way that the upper 8 bits of VSID are set based on the birthday and the lower 8 bits are set based on the zip code of users will result that user data with similar birthday and zip code will be stored in the same database. If related data are gathered in a single shard, operations such as transactions and joins can be performed locally using SQL in user code hence providing better performance. Note that transaction and join operations among distributed data are very complex and time consuming tasks. Also note that, being Java API, users can use our API and standard SQL statements together in their code for simple data look up operations and transaction and join operations, respectively.

IV. Programming Model

In relational database systems, the Structured Query Language (SQL) provides a standard way to access and manipulate data in database tables. To access data in RDBM tables from an application, developers normally compose SQL statements in a string representation and request/execute the query using the database client API. Figure 4 shows the example code for creating a database table and inserting/querying data into/from the table using SQL in Java.

Although SQL provides a standard and structured way to store and retrieve data in tables, when used in Object-Oriented (OO) programming languages where data to be manipulated is represented as properties of object, there exists a gap between the representations of data in the programming code side and in the database side. That is, data in database tables are represented as a set of columns with scalar values while the data in OO programs are represented as properties with associated get/set methods. While developers can manually map the objects in their code into the rows of tables, some Object-Relation Mapping (ORM) frameworks have been introduced to free developers from this manual mapping [13][14]. Using ORM API developers can save and retrieve objects into and from the relational database tables as if they are manipulating their objects using standard OO methods. In our system, we designed and developed a set of ORM API which is highly optimized for building and requesting complex queries to database tables which are physically distributed in a fast way. Following sections describe details about accessing and manipulating data in tables from Java code using our ORM API set.

A. Data Definition

While relational database tables are normally created using the Data Definition Language (DDL) statements of SQL as shown in Figure 4, with the ORM approaches, tables can also

Copyright (c) IARIA, 2014. ISBN: 978-1-61208-334-6

```
//Create Table
  conn = DriverManager.getConnection();
  st = conn.createStatement();
  String sql = "CREATE TABLE User (
       U_Id int NOT NULL, name varchar(8) NOT NULL,
       email varchar(8), addr varchar(8),
       PRIMARY KEY (U_Id),
       FOREIGN KEY (P_Id) REFERENCES Persons(P_Id)) ";
  st.execute(sql);

//Create Row
connection = DriverManager.getConnection();
  preparedStatement =
  prepareStatement(connection, "INSERT INTO Authors
       (firstname, lastname) VALUES (? , ?)");
  int affectedRows = preparedStatement.executeUpdate();

//Query
  connection = DriverManager.getConnection();
  preparedStatement = prepareStatement(connection,
  "SELECT id, email FROM User WHERE name=?", false,values);
  resultSet = preparedStatement.executeQuery();
  if (resultSet.next()) { user = map(resultSet);
```

Figure 4: Example Code of SQL Statement in Java Code.

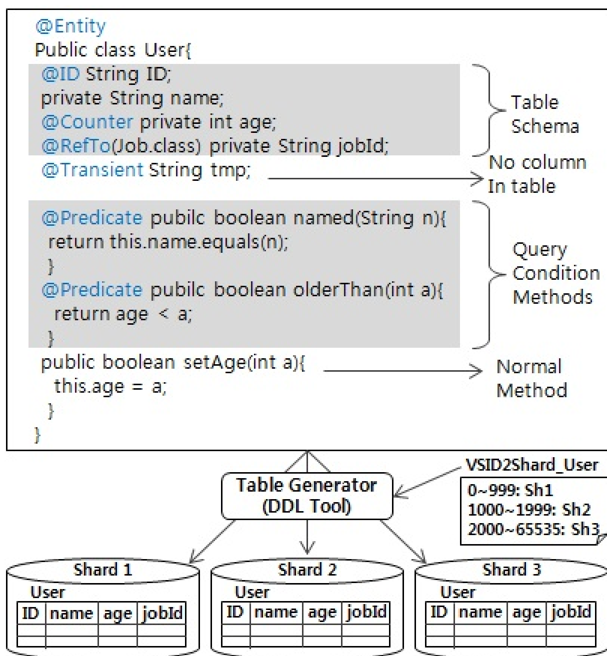

Figure 5: Data Definition and Query Condition in Our System.

be created from the annotations in the source code or from XML documents that specify mapping between the objects and database tables [13][14]. In our system, for the definition of the structure and schema of database tables across the distributed database tables, we first designed a set of Java annotations and then developed a tool that reads the annotations in a Java code and creates the tables in the sharded databases according to

```
Session s = Loom.newsession();

//Insert a user whose name is John and 40 years old
User newUser = new user();
newUser.setName("John");
newUser.setAge(40);
s.save(newuUser);

//Update users' age to 32 whose name is "Jane"
User jane = s.build(UserQuery.class).named("Jane").fetch();
jane.setAge(32);
s.save(jane);

// Delete users whose name is Tom
User tom = s.build(UserQuery.class).named("Tom").fetch();
s.delete(tom);
s.commit();

//Query user whose name is "John" and age is older than 19
UserQuery q =
s.build(UserQuery.class).olderThan(19).named("John");
List<User> users = q.fetchlist();
```

Figure 6: Composing a Query and Fetching Data.

the pre-defined sharding rules.

Using our system, developers can create tables with a primary key, a foreign key, an auto-increment column using our annotations such as *@Entity*, *@ID*, *@RefTo*, and *@Counter* annotations, respectively. For every class with *@Entity* annotation, a corresponding table is created in database nodes. Class variables with no annotations will be created as a simple column with its data type defined in the code. For the class variables with *@Transient* annotation, no matching column will be created in the table. Figure 5 shows the example code for the creation of a table in our system and basic concepts about how the tables are created in multiple shards by the DDL tool of our system. In Figure 5, the User table is created in three shards according to the rules in VSID2Shard file.

We also provide a predicate annotation for methods where developers can specify query predicates to be made for the correspondent table. Middle part of the Figure 5 shows the example *@Predicate* methods which compare the method parameters with the values of the name and age of the entity object and return the comparison results. This *@Predicate* annotated method is a building block for complex queries to the database table that corresponds to the current entity object. In the application code, developers can compose a complex query to a database table by calling the *@Predicate* annotated methods of the corresponding entity object. Following section explains how to build a complex query by leveraging this *@Predicate* annotated methods in detail.

B. Data Manipulation

With our ORM approach, developers can manage data in a table as they manage them with object. To insert data into a table, developers simply create an object with values to its member variables and provide the object as a parameter of our *save()* API. To update data in a table, developers first fetch the row of the table to be updated using our *fetch()* API

note that our *fetch()* API fetches data from database tables and maps them to the object upon which the data request is made. Once the row is fetched as an object, update can be made to the object using the *set()* method and save the object into the database table again using our *save()* API.

In our system, complex queries can be built by combining the *@Predicate* annotated methods that were pre-defined in the *@Entity* class. For example, using the predicate methods in Figure 5, the *named* and *olderThan* methods, we can build a query for selecting users whose name is John and whose age is older than 19 as in the last example in Figure 6. Note that composition of a query is achieved by sequentially calling the *@Predicate* methods of the Entity class. Once the query condition is built, developers can fetch data that meet the query conditions by calling the *fetch()* or *fetchlist()* API at the end of the query conditions built. For the fetch or *fetchlist()* methods, our query engine takes the user-built query and fetches the data that meet the conditions from the multiple shards.

Advantages of our ORM approach can be summarized in two points. First one is that since class variables and methods are mapped to a database tables and query predicates, application developers can manage data in database transparently - that is the application developers even do not need to aware the database behind the application and manage data as if they are managing objects in their application. This satisfies the goal of ORM, which is eliminating the conceptual gap between the objects in OO programs and the relational tables. Also, since there is no string manipulation for composition SQL statements, many typo errors can also be eliminated.

Second advantage of our approach is that since the query conditions are defined as class methods, developers can specify query conditions using the operators allowed in SQL WHERE clause such as LIKE, IN, MAX, etc., along with the language operators, such as bitwise operators and variety of string operators. With SQL approach, developers apply additional language operators to the SQL query results to meet the certain requirements of the application. In our system, the query condition defined using language operator is called a client query and the query condition that can be mapped to the SQL WHERE clause is called a DB query. Next section describes details about the query engine of our system.

C. Query Engine Internals

Query engine in our system is responsible for constructing a database query from the query predicates in application code and making request to database tables in multiple shards and, finally, returning the query results to the application as an object. Figure 7 illustrates the internals of our query engine. Once the *build()* API is called, our query engine starts building an empty query object that correspondent to the query class specified in its argument. In Figure 6, this correspondence to *s.build(UserQuery.class)*. Once an empty query object is ready, the query engine adds the query conditions to the query object as it is specified in the application. In Figure 6, this correspondence to *olderThan(19)* and *named(John)*.

Once the query object is built with the given query conditions, our query engine separates the database query conditions

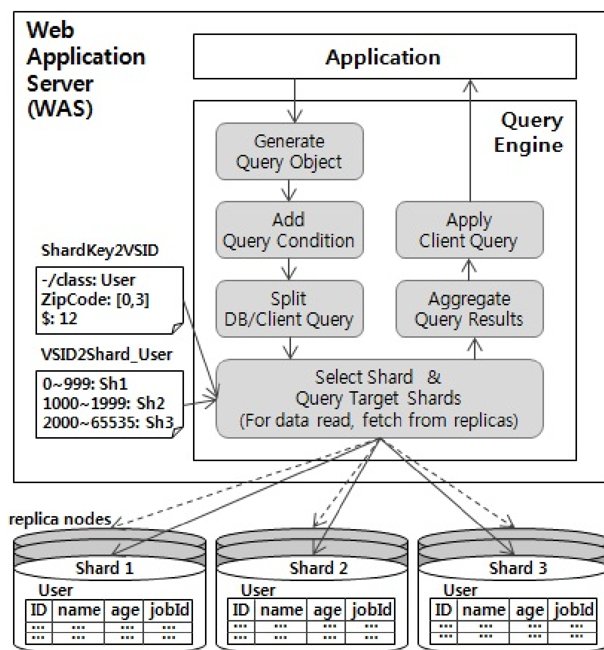

Figure 7: Internals of Query Engine.

from the client query conditions. Previously, it was mentioned that, the query condition defined using language operator is called a client query and the query condition that can be mapped to the SQL WHERE clause is called a DB query. As the database queries are filtered out, our query engine converts the database queries into a format of HandlerSocket protocol. Finally, the query engine refers to the sharding rule files to identify the target shards where the query must be sent and makes database request to them.

As the database query results are sent back from the target shards, the query engine aggregates the results and applies the previously filtered-out client query conditions to the results and, finally, returns the result to the application as an object. Note that, at the bottom of Figure 6, the result of *fetchlist()* is directly assigned to the list of User objects.

V. Performance and Scalability

A. Performance Enhancement

To process SQL statements in the server-side, MySQL upper layer includes a parser and optimizer that builds a parse tree and makes a plan for an optimized execution path for the given query, respectively. Although this parser and optimizer handles complex SQL queries in an efficient way, they can also be a burden when the query is very simple so that there is not much thing to do for parsing and optimizing. For fast data access with simple query conditions such as ID lookups and index searches, HandlerSocket [11] was introduced which bypasses the upper layer of MySQL hence improving read and write performances. In our system, we implemented our ORM API using HandlerSocket protocols that communicate

with HandlerSocket plug-in in server-side and directly access data in MySQL's storage engine, InnoDB.

One more performance enhancement we made to our system is leveraging replica nodes to serve for read requests. MySQL provides a built-in master-slave replication where read/write operations go to the master node while the data updates are copied to the replica nodes asynchronously [15]. Although the main purpose of this replication is keeping extra copies of data for high availability, we can also achieve an improved performance by routing read requests to one of replicas and reducing bottleneck in the master node. In our system, we designed our query engine to read data from replica nodes in a round-robin fashion. If a certain replica node is not responding then one of the other replica nodes in the same shard is read and the failed replica node is tried after some time.

B. Scalability Enhancement

Earlier in this paper it was mentioned that, linear scalability of database is defined as an ability to increase the total throughput linearly as database nodes are added. Being a range-based sharding solution, adding a new data node and redistributing data in our system is achieved in following steps by our shard rebalancing tool:

1) Data migration - migrate the data of a shard whose volume reaches its threshold. This involves copying data from a congested node to the newly added node and deleting the copied data from the congested node. This way, the congested node and the newly added node shares data load.
2) Sharding rule update - Update the sharding rule files accordingly and synchronize the updated rule files in every WAS node so that the requests for the migrated data go to the newly added node.

Figure 8 illustrates the concept of the shard rebalancing in our system. In our system, Apache Zookeeper [16] is used for the automatic synchronization of the mapping rule files reside in every WAS nodes.

C. Evaluation Results

To measure the performance and scalability of our system, we performed a set of tests. Firstly, to compare the performance of our approach with the traditional MySQL approach, we wrote two sets of test codes that access tables in database - one using JDBC API and one using our ORM API. Then we set up 1 database node and increased the total number of application nodes to measure the saturated throughput per second (TPS) and average latency for two different approaches. Note that we made each application node spawn 100 threads that run our test codes and to get the saturated throughput for 1 database node we should increase the total number of application nodes up to 28. The test results demonstrated that our system outperforms the MySQL approach for 34 and 13 times in terms of throughput and saves 68% and 74% in terms of latency for read and write operations, respectively. Detailed test results are summarized in Table I and Figure 9.

To measure the scalability of our system, we increased the total number of database nodes from 1 to 8 and measured the aggregated TPS of all 28 application nodes. As a result of this test, we proved that our system linearly scales out as the number of database nodes gets increased. Table II and Figure 10 show our test results in detail.

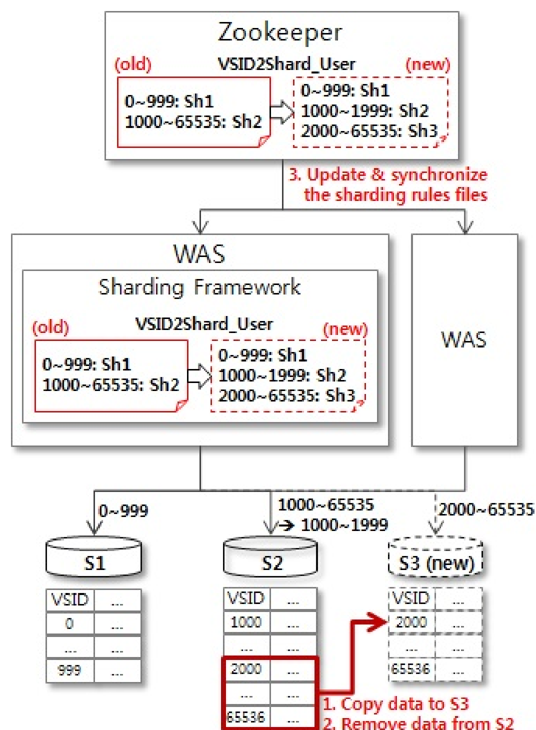

Figure 8: Shard Rebalancing.

TABLE I: Result of Performance Test

	Throughput		Latency (ms)	
	SQL-JDBC	ORM-HS	SQL-JDBC	ORM-HS
Data Read	5365	179922	2.0	0.64
Data Write	3902	49627	2.9	0.74

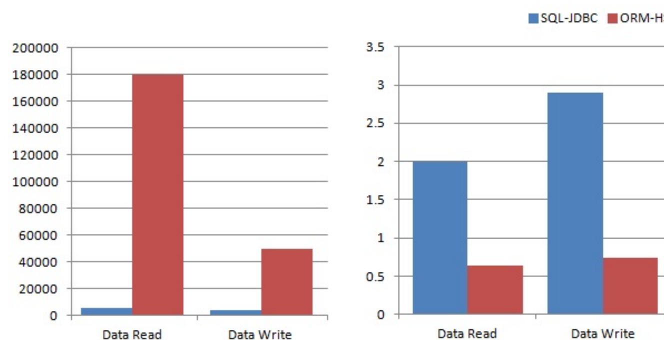

Figure 9: Result Graphs for Throughput Test(Left) and Latency Test(Right).

TABLE II: Result of Scalability Test

	1 DB node	4 DB nodes	8 DB nodes
Data Read	179922	579368	843768
Data Write	49627	169031	303800

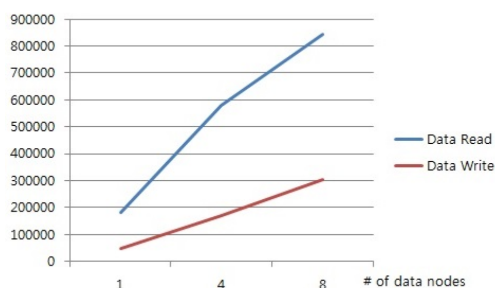

Figure 10: Result Graph for Scalability Test.

VI. CONCLUSION AND FUTURE WORK

In this paper, we introduced our database sharding framework with enhanced performance and scalability. For a flexible and efficient distribution of data among multiple database nodes, we designed a unique ID scheme which is optimized for the primary key lookup operations by encoding the target database node information as a part of ID. We also designed a set of ORM API that maps the conditioned objects in application codes to the database table and queries. For fast data access, we implemented our API using MySQL HandlerSocket plug-in which bypasses the upper layer of MySQL. We also leveraged replica nodes to serve for read requests for an enhanced performance by distributing requests to a master node.

To evaluate performance and scalability of our system, we performed a set of tests and proved that our system provides improved performance and linear scalability compared to the traditional MySQL solutions. As our future work, we firstly plan to provide zero-downtime availability in case of a master node failure by implementing an active-active replication with the quorum based algorithm [17]. We also plan to support transactions and map-reduce style queries such as order-by and join in the distributed environment.

REFERENCES

[1] G. DeCandia et al., Dynamo: Amazons Highly Available Key-value Store," In Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles, ACM New York, NY, USA, Oct. 2007, pp. 205-220

[2] F. Chang et al., "Bigtable: A distributed storage system for structured data," ACM Transactions on Computer Systems (TOCS), ACM New York, NY, Jun. 2008, pp. 1-26.

[3] K. Banker, 2011, MongoDB in Action, Manning Publications

[4] E. Hewitt , Cassandra: The Definitive Guide, OReily Media Inc.

[5] T. Macedo and F. Oliveira, Redis Cookbook: Practical Techniques for Fast Data Manipulation, O'Reilly Media, 2011.

[6] Neo4j, the Graph Database - Learn, Develop, Participate. [Online]. Available: http://www.neo4j.org/ [retrieved: 02, 2014]

[7] K. Loney, Oracle Database 11g The Complete Reference, Oracle Press, 2009.

[8] B. Schwartz, P. Zaitsev, and V. Tkachenko, High Performance MySQL: Optimization, Backups, and Replication, O'Reilly Media, November 2011.

[9] W. Vogels, All Things Distributed: Eventually Consistent, [Online]. Available: http://www.allthingsdistributed.com/2007/12/eventually_consistent.html/ [retrieved: 02, 2014]

[10] N. Leavitt, "Will NoSQL Databases Live Up to Their Promise?," Computer, volume 43, number 2, Feb. 2010, pp. 12-14.

[11] HandlerSocket. [Online]. Available: http://yoshinorimatsunobu.blogspot.kr/search/label/handlersocket/ [retrieved: 02 2014]

[12] D. Karger et al., "Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web," In ACM Symposium on Theory of Computing, Philadelphia, Pennsylvania, USA, May. 1997. pp. 654-663.

[13] Hibernate JBoss Community. [Online]. Available: http://www.hibernate.org/ [retrieved: 02, 2014]

[14] Java Persistent API. [Online]. Available: http://www.oracle.com/technetwork/java/javaee/tech/persistence-jsp-140049.html/ [retrieved: 02, 2014]

[15] S. Pachev, Understanding MySQL Internals, O'Reilly Media, 2007.

[16] Apache Zookeeper. [Online]. Available: http://zookeeper.apache.org/ [retrieved: 02, 2014]

[17] D. Agrawal and A. E. Abbadi, The tree quorum protocol: an efficient approach for managing replicated data, in Proceedings of the sixteenth international conference on Very large databases, Brisbane, Australia, Sep. 1990, pp.243-254.