U.P.B. Sci. Bull., Series C, Vol. 77, Iss. 2, 2015 ISSN 2286-3540

DISTRIBUTED DATABASE OPTIMIZATIONS WITH NoSQL MEMBERS

George Dan POPA[1]

Distributed database complexity, as well as wide usability area, raised diverse problems concerning data coherence, accessibility and performance. NoSQL (Not only SQL) databases provide solutions for some of these problems, completing or total replacing relational databases in specific situations.

This paper presents functional setups for two NoSQL databases, Infinispan and MongoDB, presenting an optimal architecture and obtained results.

Keywords: distributed databases, Infinispan, MongoDB, performance

1. Introduction

Distributed databases have conquered the world of data warehousing, due to the complex requirements of current day applications and increasing data quantity. However, there aren’t only advantages in using a distributed environment, but new challenges, too, in connecting all members, collaboration, data coherence, availability, performance and much more. Some of these issues have been addressed by additional data nodes, load balancers, proxy servers, but more problems remain unsolved due to the low scalability or data formats.

NoSQL (*Not only SQL*) databases represent the data storing systems that, unlike relational databases, don’t necessarily have relational properties between stored objects. Even if such properties may exist at certain levels, they don’t play key roles in the system management. One of the most important NoSQL database purpose is to obtain superior horizontal and vertical scalability, overcoming relational database limitations in storing large and complex information.

Storing mechanisms of NoSQL databases are as many as the number of the technologies used: object oriented, XML or JSON documents, grid, graph or key – value pairs. All these formats allow us to choose the best fit solution for each demand, in order to closely match the serving application data format.

Having these new technologies available, the challenge of building a consistent and efficient distributed database is system configuration. Through proper configuration, we can achieve a more scalable and usable database system for external applications, than using only relational database members.

[1] PhD student, Faculty of Automatic Control and Computers, University POLITEHNICA of Bucharest, Romania, e-mail: george.popa@yahoo.com

Due to the fact that each configuration is custom for each client application, after developing several specific database systems and analyzing their performance, the aim of this paper is to define an optimal architecture for every distributed environment containing NoSQL members.

2. Distributed database system and test data

Our working environment consists of a distributed database system set up for hosting Romanian-language Wikipedia database. Wikipedia encyclopedia is a collection of user provided articles, containing text, links and multimedia files. Each article's history and contributors information is saved into Wikipedia database, making it large enough to require special management server configurations.

Table 1

Main tables description of Wikipedia database

Table	Number of records	Size	Description
page	933,386	306 MB	Main page table, includes articles, templates and stubs.
revision	7,818,593	178 MB	Page history table, stores each of the page's revisions.
pagelinks	4,023,378	16 MB	Table containing redirects and external links.
text	7,818,593	6.4 GB	Text used in any of the page's revisions.
user	254,299	58 MB	Authorised users table.
image	27,516	5.4 GB	Multimedia files.

As listed in Table 1, some of the database's tables have a large number of rows, while others occupy large disk space. In the same time, some tables are read and write often, such as **user**, while others are read-only and rarely accessed, an example is **image** table. These tables have been separated on the two members of the distributed database, MySQL hosting the multimedia and other large tables, while Oracle database the frequently accessed tables [1]. This database segmentation was designed in order to benefit from each of the element's features and obtain optimum overall performance [2].

While this testing environment was setup using a centralized Java management application [3], other environments were also configured for performance comparison, build using commercial distributed database solutions: SyncML [4] and Sybase Replication Server [5].

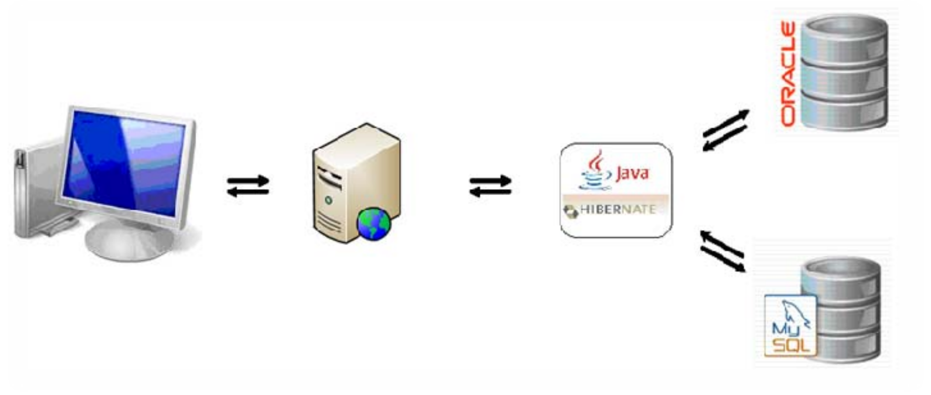

Fig. 1. Centralized database management system with Oracle and MySQL members.

For performance testing, we have used specialized automated test tools, such as *JMeter*, *autobench* and *httperf*, while *JProfiler* was used to investigate Java application behavior and monitor system resources [6]. The metrics used for comparison are:

- *Throughput* – number of successful responses returned to the clients;
- *Execution time* – medium execution time for each of the successful operation;
- *Concurrency* – number of active connections, representing the number of clients making simultaneous requests;
- *Utilization* – measures asset performance, determined by usage time reported at total available time.

Each individual test simulates usual user operation of a Wikipedia page, inserting a record in **page** table, three records in **pagelinks** table, five records in **revision** and **text** tables, all in the Oracle database. In the same time, three binary files, measuring 200 kB each, are inserted into MySQL database. These operations are doubled by reading the same amount of data, from the same tables.

After running series of 1000 automated tests, but under different concurrency level, starting from 1 and ending at 100 concurrent requests, the results were aggregated and visual displayed in chart, representing execution time by concurrency level, for each of the distributed system:

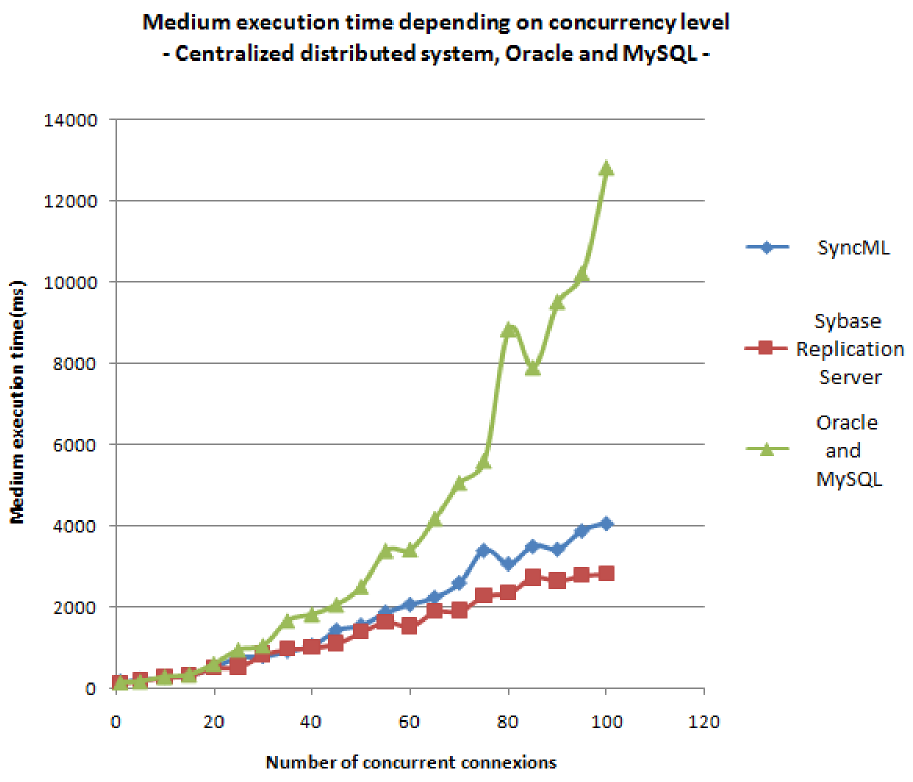

Fig. 2. Tests results for centralized Oracle and MySQL system, along with SyncML and Sybase Replication Server.

As shown in chart, out test environment with relational databases is showing major performance drawbacks, comparing to SyncML and Sybase Replication Server. More than this, the increase of the concurrency level leads to exponential increase of the execution time, showing both concurrent threads bottleneck and memory leak on Java management application.

3. Storing large tables in MongoDB

MySQL relational database has shown performance penalties for tables having millions of records, or large record size. Searching for a solution to allow better scalability, for both records count and size, we have found MongoDB, a NoSQL database [7]. It stores data in JSON or XML formats, with practical unlimited space usage. The good scalability feature is possible due to the lack of some relational features: stored objects don’t need to have the same data type or complexity, or joining data collections is not possible.

We have replaced MySQL database member from our testing distributed system with a MongoDB element, having the same data collections structure as MySQL tables. Running the automated tests described in the previous chapter, we got better results in this case, eliminating the exponential increasing execution time. However the test environment kept lower performance (in both execution time and throughput) and more optimizations are needed.

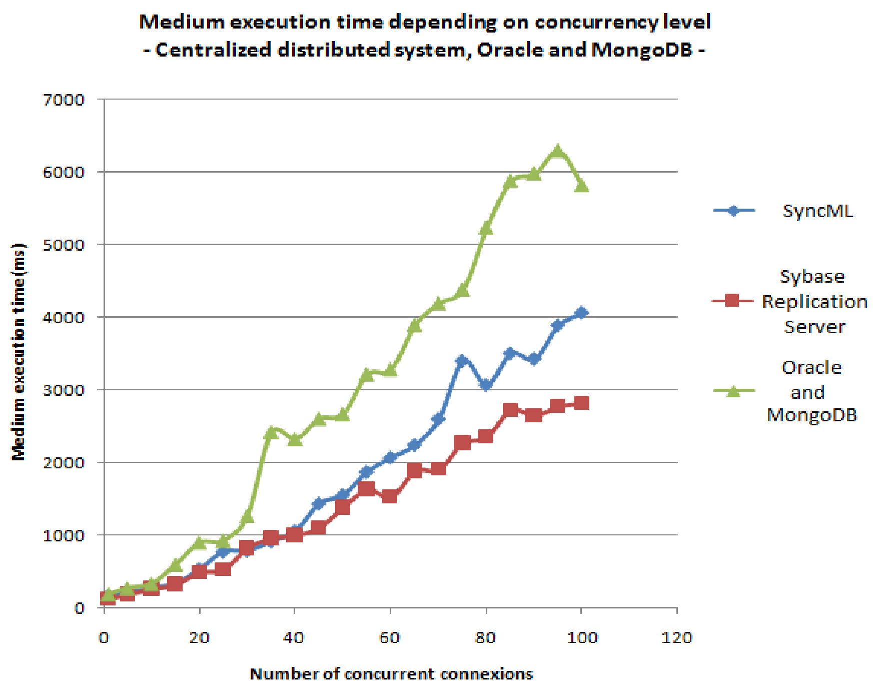

Fig. 3. Tests results for centralized Oracle and MongoDB system, along with SyncML and Sybase Replication Server.

Beyond its specific features and advantages, some of the MongoDB's missing relational characteristics may induce significant drawbacks in the system's functionality and performance. We are referring now at the *join* operations and *transactions*.

Hibernate OGM has achieved some levels of *join* operations, particularly unidirectional and bidirectional relationships for MongoDB associations. As a work around for *transactions*, Hibernate OGM offers transaction demarcations to trigger the flush operation on commit [8]. All the logical transaction operations are queued before written to disk and a user programmed flush to disk is triggered to simulate the transaction commit. Rollback cannot be achieved, as database previous state is not saved after changes are persisted.

Due to these inconveniences, we have moved any of the missing features implementation to the middle tier (Java management application). At application level, we are handling operation order and monitoring their result. While inside a logical transaction, consisting of several operations on multiple MongoDB collections, if one of the operation fails, then it is retried a limited number of times. If successful, we continue to execute the next operations. Otherwise, the next operations are canceled, but the previous operations changes remain persisted in database, leaving it in an inconsistent state. For example, we have stored an entry in the **page** collection, but associated **text** and **image** entries failed to be saved, breaking the logical transaction's atomicity. In that case we would have empty pages displayed to the user.

The order of the insert, update and delete operations can help us in solving these problems, in our example saving the associated **text** and **image** entries first, and then the **page** record. There are no limitations with foreign keys, as they are missing in MongoDB. In this way, any database inconsistency would be transparent to the end user, as the orphan **text** and **image** records cannot be accessed by the user.

Note that in our isolated environment, database inconsistencies did not occur, but some of the operations needed to be retried due to the timeouts induced by the highly concurrent requests.

The distributed environment requires *join* operations both inside each of the database member and at logical database level. Relational databases can handle joins for all their stored tables, retrieving in a single query data from multiple tables. At logical database level (distributed system) and inside NoSQL member MongoDB, joins are performed at Java application level. At least one query is necessary for each table / data collection involved, and then filter the results based on their identifiers and connection tables. The multiple queries and storing a large data quantity in Java heap memory for rows matching, have lead to some performance penalties using the *join* operation over NoSQL database members.

4. Decentralized system with Infinispan grid

The centralized architecture of the distributed system has proven its limitations in accepting highly concurrent requests. To overcome this bottleneck, we have distributed the Java management application, to run as an agent on every database member host machine. The Java agent is now handling notification requests from the Web server and can execute them if the necessary resources are on the local machine, else will redirect them to the proper agent.

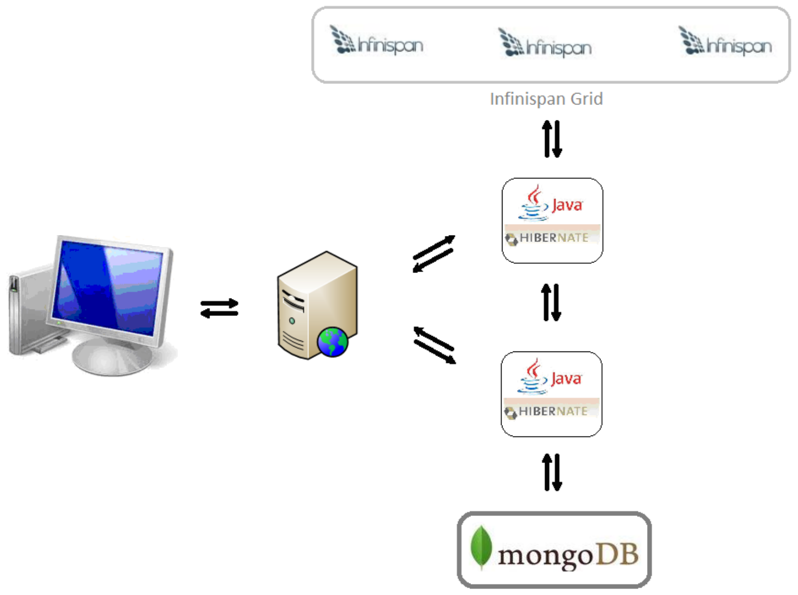

Fig. 4. Decentralized database management system with Infinispan and MongoDB members.

A new layer of communication is introduced between member machines, that may induce performance overhead, but the highly concurrent requests are split among them, thus increasing parallel computations.

Infinispan is a new NoSQL database, known for high scalability and availability [9]. Designed to work in system memory, rather than storing data on hard drive, it achieves very good access times for concurrent environment. Whenever new available space is required to work with larger stored data, a new segment can be added to the Infinispan grid, without size limitations. However, due to the fact that it operates in memory (recommended), the average cost per size unit is much higher than usual databases. Thus we have configured Infinispan grid to handle only the smallest database tables in size, but with most frequent accesses. In our test database, these tables are **page**, **revision**, **pagelinks** and **user**. The other two important tables, **text** and **image**, remained to be stored by MongoDB database member.

Infinispan Data Grid is very elastic in adding or removing grid nodes required by the application, without any data loss. Even if all nodes are removed from the grid, data is persisted on disk, using its persistent *CacheStore*, in order to be restored on future nodes. There are two main usages for clustering Infinispan nodes: data replication and distribution. We have used its data distributed feature, to extend its storage capacity for our needs.

As Infinispan works best if data is cached in memory, extending the Infinispan grid with additional nodes makes sense only if the nodes are distributed over separate physical machines. We have used the Infinispan *CacheManager* class to configure and synchronize the grid's members, thus making it transparent to the upper layers of our application, seen as a single data source. Each of the Infinispan grid's nodes is configured to store records of any of the tables assigned to Infinispan, implementing horizontal segmentation. We took advantage of the key – value pair data format Infinispan uses, in order to directly store Java objects into the database, without the need to use any serialization or persistence library such as Hibernate. Similar to other NoSQL databases, Infinispan does not provide the *join* functionality, so any complex query, involving multiple tables is split between distributed members. In those cases where the queries were independent, we achieved parallel computation in every grid's member, returning results to the Web server faster than a single database.

Our application needs required up to three separate Infinispan nodes, based on each of the machine's available memory, to ensure the fact that no data needs to be stored on disk. The Java management agent, located on one of the Infinispan nodes, connected the Infinispan grid with the rest of the distributed system and the Web server. Even if most of the application requests are sent over the local network, they are still served faster by the Infinispan grid cache, than usual disk based databases.

We have used Infinispan *TransactionManager* to implement transactional operations against the data store and achieve *ACID* features, thus ensuring database consistency.

Our automated test and diagnose tools showed that no memory leaks occurred when distributing the management system onto separate machines, or distributing the Infinispan grid. The Web server in front of the distributed system worked as a *load balancer*, dispatching new requests to the most responsive member, including the *server affinity* feature for successful results. Joining results operation, from all distributed database members was made at application level, in a similar manner as the centralized system.

Results presented in the chart below (Fig. 5) show that our test environment, consisting in an Infinispan grid and MongoDB members, performs comparable to the commercial systems SyncML and Sybase Replication Server. Even if for some concurrency levels it had the longest response time, for others it achieved the best results.

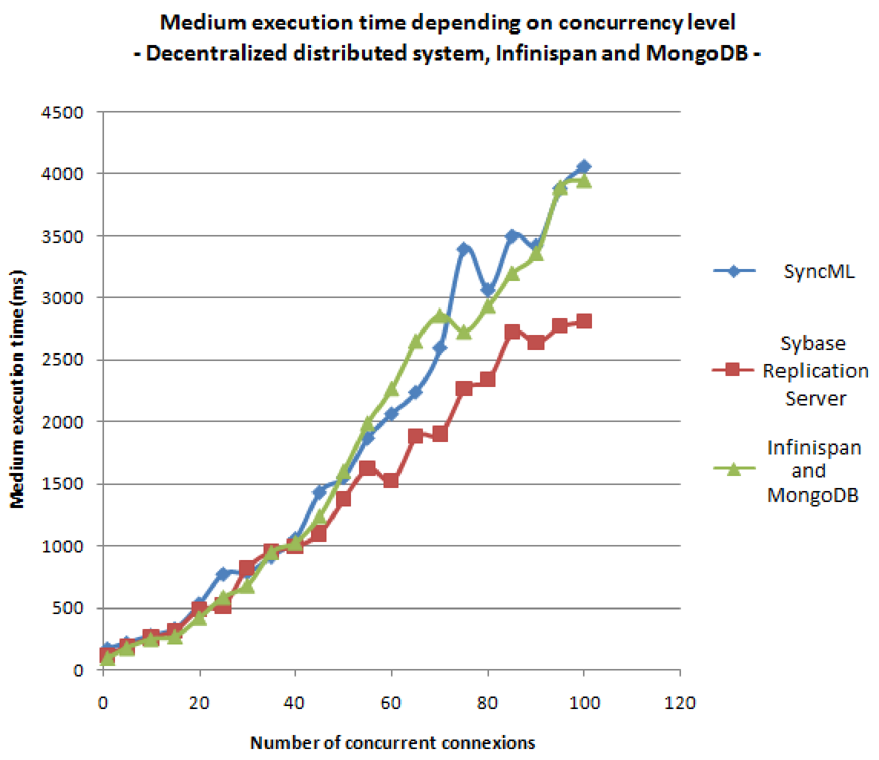

Fig. 5. Tests results for decentralized Infinispan and MongoDB system, along SyncML and Sybase Replication Server.

More optimizations are possible after investigating the user behavior in a real world environment, such as changing the table segmentation criteria, caching results for repetitive requests or finding new storing technology for specific usage [10].

6. Conclusions

The key to obtain the best results do not necessarily consist in getting the latest technology or the best available hardware, but in system configuration among the database members. As we tested various system combinations before reaching an optimal architecture as a competitive distributed database, some guidelines had been used:

- Database table segmentation must respect data usage, in order to avoid joins or data correlation on separate members.

- Horizontal segmentation, or *sharding*, is possible when necessary, but must be implemented using the same technology, such as Infinispan grid.
- Complex queries that involve multiple distributed members, must be split into sub-queries at application level (Web server), thus freeing the Java management applications from subsequent tasks and joining results.
- Use of technological heterogeneity (such as Infinispan grid or MongoDB) must take in consideration their specific features and application requirements (frequent accesses or large data sets, respectively).

We have demonstrated that using these principles and further tuning the database system, we can achieve comparable and better results than commercial expensive solutions, build for general use. NoSQL databases market share has a high potential and have been proved to be a good replacement for relational databases in specific situations.

R E F E R E N C E S

[1] *A. Boicea, A. Crivat, F. Radulescu, G. D. Popa,* Performance Evaluation and Tuning in an Oracle DBMS, DAAAM 2010, Vienna;

[2] *A. Boicea, C. Magdalina, D. C. Ionescu, F. Radulescu, G. D. Popa*, An Architecture for Distributed Databases on Workstations, DAAAM 2011, Vienna;

[3] *G. D. Popa*, Heterogeneous Database Integration Using Hibernate Mapping Files, DAAAM 2011, Vienna;

[4] *J. Suryanarayana*, Heterogeneous database replication with SyncML, IBM Technical Library, 2005;

[5] *R. H. Wiebener*, Synchronizing Data Among Heterogeneous Databases, Sybase Adaptive Server Library, 2010;

[6] *A. V. Pita, S. L. Roberts*, Performance of distributed database application models using Java RMI, IEEE International Computing and Communications, 1998;

[7] *** *MongoDB, Inc.*, The MongoDB 2.4 Manual, http://docs.mongodb.org/manual/, 2013;

[8] *** *JBoss Community*, Hibernate OGM 4.0 Documentation, http://docs.jboss.org/hibernate/ogm/4.0/reference/en-US/html/ogm-datastore-providers.html

[9] *** *Red Hat, Inc.*, Infinispan Documentation, http://infinispan.org/documentation/, 2013;

[10] *M. Shafique, M. S. Al-Shishtawi*, A methodology for integrating heterogeneous databases using a global schema, CIIT '07 The Sixth IASTED International Conference on Communications, Internet, and Information Technology, 2007.