Testing the Suitability of Cassandra for Cloud Computing Environments
Consistency, Availability and Partition Tolerance

Felix Beyer, Arne Koschel,
Christian Schulz, Michael Schäfer
Faculty IV, Department for Computer Science
Applied University of Sciences and Arts
Hannover, Germany
{felix.beyer, christian.schulz2, michael.schaefer}@stud.fh-hannover.de, arne.koschel@fh-hannover.de
Stella Gatziu Grivas, Marc Schaaf
Institute for Information Systems
University of Applied Sciences Northwestern Switzerland
Olten, Switzerland
{stella.gatziugrivas, marc.schaaf}@fhnw.ch

Irina Astrove
Institute of Cybernetics
Tallinn University of Technology
Tallinn, Estonia
irina@cs.ioc.ee
Alexander Reich
BeEvolution GmbH
Hannover, Germany
alexander.reich@beevolution.de

Abstract—Since relational database management systems (DBMSs) are ill-suited to cloud computing environments, multiple efforts are now underway to offer a viable alternative to relational DBMSs. These efforts have led to the rise of a new kind of DBMS called NoSQL. One of the most visible products in this rise is Cassandra. Cassandra is a NoSQL DBMS, which can also be used as a clustered file system. Cassandra was claimed to be particularly well suited for cloud computing environments. Our goal in this paper was to confirm or deny that claim. Towards this goal, we conducted tests on Cassandra to determine what levels of consistency, availability and partition tolerance can be achieved and if these can be achieved without sacrificing performance.

Keywords—Cloud computing, Cassandra, consistency, availability, partition tolerance, experiments.

I. INTRODUCTION

Consistency, availability and partition tolerance are of great importance to cloud computing environments. These can be achieved by using relational or NoSQL database management systems (DBMSs). Since NoSQL DBMSs are still a new research area, various definitions exist that may even contradict each other. For this paper, we have chosen the following definition: NoSQL is a movement grouping all efforts, which intend to provide a viable alternative to (SQL-based) relational databases for storing and processing data [1].

Relational DBMSs [3] are 30 years old. They have been the dominant storage technology behind websites. The past few years have seen the emergence of cloud computing environments, which are going to be an increasingly common backbone for websites. But cloud computing environments and relational DBMSs do not fit well together [10]. In particular, relational databases can scale, but usually only when this scaling happens on a single node (i.e., vertical scaling). When the capacity of that single node is reached, relational databases need to scale horizontally and be distributed across multiple nodes over a network. This is when the suitability of relational DBMSs for cloud computing environments is reduced.

A. Consistency

Consistency guarantees that every node in the cluster has the same view on data. So once one node has written some data, all other nodes in the cluster will see those data.

The importance of consistency for cloud computing environments is perhaps best explained by example. Consider an airline company that provides a booking website. Assume that the airline company’s database is distributed over a network, so data can be accessed from different nodes. Consistency is endangered now because one node may change data without knowing about the changes have been made by other nodes. In particular, assume that a customer opens a session on the booking website and a last available seat for the selected flight is displayed to the customer. This seat has already been booked, but the node serving the customer’s session does not know about it yet. The result is that the customer can still book the last seat. Next time when the nodes synchronize each other, inconsistency shows up as there will be two bookings for one and the same seat.

To avoid a situation like the above, NoSQL DBMSs should provide consistency. Relational DBMSs typically use ACID (Atomicity Consistency Isolation Durability) transactions for this purpose. But ACID transactions are not distributed-system friendly. Therefore, NoSQL DBMSs typically either skip them entirely or comply with BASE (Basically Available Soft-state Eventual Consistency).

Compliance with BASE means that the latest version of data on one node might not match that on other nodes; so every node in the cluster is only guaranteed to see writes eventually. As a result, NoSQL DBMSs might not handle long running business processes [6] like booking flights, where the current state of data, e.g., seats availability on the plane, should be shown to all other customers while one
customer, who is booking a flight, has not finished the booking yet.

B. Availability

Availability guarantees that if one node fails, there will still be some copies of data on other nodes in the cluster, so the availability of the whole cluster will not be endangered by that node failure.

Continuing the previous example, assume that the node serving the customer’s session experiences a failure during which the customer cannot book the last seat anymore.

To avoid a situation like the above, NoSQL DBMSs should provide availability. Relational DBMSs typically use replication for this purpose. The same technique is used by NoSQL DBMSs.

C. Partition Tolerance

Partition tolerance guarantees that the cluster remains operational even when communication between nodes in the cluster is lost.

Continuing the previous example, assume that the airline company’s database is running on multiple nodes across a network. Also, assume that a network connection with the node serving the customer’s session is lost due to a network failure. The database is now partitioned. If the database is tolerant of it, then the cluster can still perform read and write operations, i.e., the customer can still book the last seat. If not, the cluster will be completely inaccessible.

To avoid a situation like the above, NoSQL DBMSs should provide partition tolerance— they typically use quorum for this purpose. Being single-node, relational databases cannot be partitioned.

II. CONTRIBUTION

In this paper, we deal with using NoSQL DBMSs in cloud computing environments. Unlike many other papers, we do not focus on traditional approaches that use clustered file systems like Gluster [2] or relational DBMSs like MySQL and Oracle. Rather, we introduce a novel approach that uses Cassandra.

Cassandra [5] was claimed to be particularly well suited for cloud computing environments. Our goal was to confirm or deny that claim. For this purpose, we experimented with Cassandra. In particular, we built a test setup, developed a test application and conducted tests on Cassandra using this application.

III. CASSANDRA

Cassandra is a recently upcoming NoSQL DBMS that can also be used as a clustered file system [4]. It was originally developed as an open source by Facebook in 2007 to horizontally scale their internal application; viz. Inbox Search. Later in 2009 Facebook released Cassandra to Apache. This allowed Cassandra to move forward in the direction that is more general to the public than just to Facebook’s in-house needs.

Recently, Cassandra has acquired great popularity and showed high potentials for cloud computing. This is because Cassandra offers a variety of possibilities to provide the desired levels of consistency, availability and partition tolerance.

A. Consistency

In Cassandra, every operation is assigned a consistency level, so that it can be decided whether the consistency should be guaranteed among all nodes in the cluster or it is acceptable if some node might not contain the latest version of data, e.g., in case of a node failure. In particular, Cassandra supports the following consistency levels:

- ANY: \( W + R > N \)
- ONE: \( W = 1 \) or \( R = 1 \)
- QUORUM: \( W = Q \) or \( R = Q \)
- ALL: \( W = N \) or \( R = N \)

where \( R \) is the number of records to read (i.e., the number of reads on a replica), \( W \) is the number of records to write (i.e., the number of writes on a replica), \( N \) is a replication factor and \( Q = N / 2 + 1 \).

Even though Cassandra complies with BASE, it is still possible to have ACID transactional consistency guarantees using ZooKeeper [7], a coordination service for distributed systems. For short running business processes, single path locking can be used (classes ZkReadLock and ZkWriteLock). However, in distributed systems with many interactions, the use of single path locking is not recommended since it often results in deadlocks. It is better to use multi-path locking (a class ZkMultiLock) since this class contains methods, which check for deadlocks and handle them before they occur. A downside of multi-path locking is decreased performance. For simple applications, both single and multi-path locking is sufficient to ensure consistency. More complex applications, however, require the use of a class ZkTransaction. This class works in conjunction with ZkMultiLock. It provides a simplified Thrift API, which allows for specification of a series of data mutation operations to be performed by a transaction. After the transaction has been specified, a method commit is executed with an instance ZkMultiLock passed it as a parameter. At this point, cages will add a reference to a transaction node, which is created by ZooKeeper. Next, the transaction can read the current values of the data, which are to be updated. At this point, the original state will be written into the transaction node [8]. Once this has been done, the data mutations will be performed. After that, all references to the transaction node from within the locks will be removed. The transaction node gets deleted and the transaction itself has been committed.

If the node fails during the execution of a sequence of individual data mutations, the cages will immediately be unlocked. The transaction, which has already been executed, will be rolled back to the “written before” state in the transaction node. So the state of the database will be identical to the original state before the node has performed its operations. This guarantees consistency of the database and complies with so-called relaxed ACID since changes one node makes during a long running business process will be seen by other nodes in the cluster [9].
B. Availability

In Cassandra, availability is achieved through replication. Every node in the cluster that needs access to data has its own replica, so a failure of one node will not make all replicas unavailable at the same time.

C. Partition Tolerance

In Cassandra, partition tolerance is achieved through quorum (e.g., if one node is separated from the other two nodes in the cluster, it stops processing).

IV. Test Setup

The test setup consisted of a cluster having two nodes: primary and secondary. Writes are directed at both nodes, while reads are directed to just one of the nodes, which is known as the primary node. Because the other node is kept updated, it is known as a secondary node. It is always ready to take over. If the primary node should fail or become inaccessible for any reason, Cassandra will redirect reads to the secondary node and processing will continue uninterrupted. Before the failed node comes back on line, any interim updates will be applied to synchronize it with the other node.

A. Cluster Infrastructure

To configure the first node, we adjusted some variables in the configuration file. In particular, we set both ThriftAddress and ListenAddress to the IP address of the first node to enable intra-cluster communication and data access. (The database was accessed using Thrift API.) Also, we set ReplicationFactor to a value that was equal to the number of nodes in the cluster (i.e., 2) to ensure that a failure of one of the nodes would not make both replicas unavailable at the same time. (In general, the cluster can be configured with more than two replicas, depending on the probability of failures and the requirements for availability.)

For the second node, we set both ThriftAddress and ListenAddress to the IP address of the second node. In addition, we set Seed to the IP address of the first node so that the second node would know to which server it had to connect for getting data when it was added to the cluster. Finally, we set AutoBootstrap to true. This resulted in the second node being added to the cluster automatically. (If a new node is added, only seed nodes in the cluster need to be configured, instead of adjusting all node configurations.)

After the cluster configuration had been completed, we checked if the two nodes would connect to each other. We did it by using a command ring, which returned a list of all available nodes. Although this check showed that the two nodes were available in the cluster, we analyzed entries in the log file generated by Cassandra to see if the cluster remained operational over some period of time.

The following listing shows an excerpt from the resulting log file:

```
INFO 16:50:25,966 Starting up server gossip
INFO 16:50:26,045 Binding thrift service to 192.168.5.132:9160
```

As can be seen, the second node (192.168.5.134) was added to the cluster, and a synchronization process called hinted handoff was started and finished.

B. Test Database Schema

Cassandra supports a data model that is based on column families. A column family is a container for columns, analogous to a table in relational DBMSs; it holds the columns as an ordered list (a column family row), which can be referenced by the column name. There are two kinds of column families: simple and super. Simple column families consist of columns, which are grouped. Super column families can be viewed as a column family within another column family.

In Cassandra, a database is a distributed multi-dimensional map, which is indexed by a key. The top dimension is referred to as a key space and under this key space, column families follow. The key space is divided up by a cluster into ranges delimited by tokens.

In Cassandra, a database schema is flexible, meaning that we do not have to decide what columns we need in the records ahead of time. Rather, we can just add or delete columns on the fly. This is by contrast to relational DBMSs, where a database schema is fixed and pre-defined.

In the test setup, we used a simple database schema Address. There was only one key space Keyspace1 containing a column family Standard2, which in its turn contained the following columns: firstname, lastname, street, housenumber, zip, city, and country. To populate the column family with data sets, we used the following statements:

```
setKeyspace1.Standard2["1"]['"firstname"]="MyFirstname"
setKeyspace1.Standard2["1"]['"lastname"]="MyLastname"
setKeyspace1.Standard2["1"]['"street"]="MyStreet"
setKeyspace1.Standard2["1"]['"housenumber"]="MyHouseNumber"
setKeyspace1.Standard2["1"]['"zip"]="MyZip"
setKeyspace1.Standard2["1"]['"city"]="MyCity"
setKeyspace1.Standard2["1"]['"country"]="MyCountry"
```

In this listing, the key value was set to 1. However, for any next data sets, this value was increased by one in order to differentiate the data sets from each other.
V. Test Application

To experiment with Cassandra, we developed a test application in Java. This application took the following arguments as input: a node IP, a Cassandra port, a command to be performed (viz., put, delete or get), data for the command and optionally a key ID of the data. The test application consisted of the following classes.

A. SelectClient

This class was used to determine the time periods for every method execution.

B. CassandraClient

This class was used to open and close a connection to the database.

C. PutCassandraData

This class was used to insert data into the database. The class had a method putDataIntoCassandra, which defines the column names, generates new records and adds them to the database. The record generation was performed by a random generator, which combines data from the specified lists, and could be repeated any number of times using a loop.

D. GetCassandraData

This class was used to retrieve records from the database. Retrieving records was performed by the following methods:

- getKeyList, which sets a range for the specified key space and gets a key range from Cassandra.
- getData, which reads all records in the specified key range and returns the result.
- getDataByKey, which defines a slice range, reads one specific record identified by its key ID and returns the result.
- printData, which displays on the shell all records in the specified maximum range.
- printDataByKey, which displays on the shell one specific record identified by its key ID.

E. DeleteCassandraData

This class was used to remove records from the database. Removing records was performed by the following methods:

- deleteCassandraData, which creates a key range and deletes all records in the specified key range.
- deleteCassandraDataByKey, which deletes one specific record identified by its key ID.

VI. Experiments

After setting up the cluster infrastructure, we performed the following test cases using the test application. After each test case, we analyzed the log file entries generated by Cassandra.

A. Test Case 1: Putting Data to Database

In this test case, we checked if records could be inserted into the database. For this purpose, we tried to add data to the first node.

The following listing shows an excerpt from the resulting log file for the first node:

```
DEBUG 16:52:47,373 insert
DEBUG 16:52:47,381 insert writing local key 1
DEBUG 16:52:47,383 insert writing key 1 to 432@192.168.5.134
DEBUG 16:52:47,391 Processing response on a callback from 432@192.168.5.134
```

At first, an insert was executed, following by a local write. Then a remote write was executed, following by a response from the second node (192.168.5.134) to check if this node had received the data.

B. Test Case 2: Getting Data from Database

In this test case, we checked if records could be removed from the database. For this purpose, we tried to read data from the first node.

The following listing shows an excerpt from the resulting log file for the first node:

```
DEBUG 16:53:42,116 range slice
DEBUG 16:53:42,117 RangeSliceCommand[{'keyspace'='Keyspace1', 'columnfamily'='Standard2', 'supercolumn'=null, 'predicate'=SlicePredicate({'columnnames':[[B@1b7c76]], 'range'=[0,0], 'maxkeys'=1})}<somerangesliceoutput>
DEBUG 16:53:42,191 get slice <somegetsliceoutput>
DEBUG 16:53:42,203 Reading consistency digest for 1 from 606@[192.168.5.134,192.168.5.132]
```

At first, a range slice was executed; it set the key space, the column family and the range. It was followed by a get slice, which collected the requested data. An entry reading consistency digest in the log file indicated that the database was checked for consistency.

C. Test Case 3: Deleting Data from Database

In this test case, we checked if records could be removed from the database. For this purpose, we tried to delete data from the first node.

The following listing shows an excerpt from the resulting log file for the first node:

```
DEBUG 16:54:04,475 remove
DEBUG 16:54:04,476 insert writing local key 1
DEBUG 16:54:04,477 insert writing key 1 to 676@192.168.5.134
DEBUG 16:54:04,480 Processing response on a callback from 676@192.168.5.134
```

At first, a remove was executed, following by a local write, which set the data values to null. Then a remote write was executed, following by a response from the second node (192.168.5.134) to check if this node set the data to null. Thus, deleting data was somehow similar to adding data.
D. Test Case 4: Consistency

In this test case, we checked if all nodes in the cluster had the same view on data even in the presence of updates. For this purpose, we added some data to the first node and tried to read the data back from the second node.

The following listing shows an excerpt from the resulting log file for the first node:

```
DEBUG 18:09:55,489 Adding hint for 192.168.5.134 <some row mutation operation which adds new data on the first node>
INFO 18:11:29,284 Node 192.168.5.134 state normal, token 115100908670755235738753006493737225538 <some row mutation operation which adds the changed data of the first node>
INFO 18:11:29,284 Node 192.168.5.134 state jump to normal
INFO 18:11:29,284 Will not change my token ownership to 192.168.5.134
INFO 18:11:29,284 Started hinted handoff for endpoint 2192.168.5.134 <some data mutation operation>
INFO 18:11:29,385 Finished hinted handoff of 2 rows to endpoint 192.168.5.134
```

At first, some data mutation was performed. Then a token was sent to the second node, following by starting and finishing a synchronization process with the second node (192.168.5.134) as the endpoint.

The following listing shows an excerpt from the resulting log file for the second node:

```
DEBUG 16:58:13,064 Node 192.168.5.132 state normal, token 115100908670755235738753006493737225538 <some row mutation operation which adds new data on the first node>
INFO 16:58:13,344 Started hinted handoff for endpoint 192.168.5.132
INFO 16:58:13,351 Finished hinted handoff of 0 rows to endpoint 192.168.5.132
```

At first, the token was received from the first node. Then some data mutation was performed, following by starting and finishing another synchronization process with the first node (192.168.5.132) as the endpoint. After the synchronization process had finished, the data on the second node were one and the same as on the first node, thus indicating that the database was in a consistent state.

It should be noted that since we wrote data with a consistency level of ONE and wanted to get the same data back while reading, we read the data with a consistency level of ALL.

E. Test Case 5: Availability

In this test case, we checked if the database was available even in the presence of node failures. For this purpose, we disconnected the first node to simulate its failure and tried to read data from the second node to see if some copy of the data was still available.

Since data were replicated within a single cluster, they were available even after the first node had been disconnected. The performance for a read operation became half as fast as before. But this was fine for a two-node cluster.

F. Test Case 6: Partition Tolerance

In this test case, we checked if the database was tolerant to partitions in the presence of network failures. For this purpose, we disconnected the second node to simulate a loss of a network connection between the two nodes and tried to write data with a consistency level of ONE to the first node to see if that node could still process the write (even knowing that data on the second node could not be updated immediately).

The following listing shows an excerpt from the resulting log file for the first node:

```
DEBUG 18:11:29,116 range slice
DEBUG 18:11:29,117 RangeSliceCommand(keyspace =’Keyspace1’, columnfamily=’Standard2’, columnnames:=[[B@1b7c76]], supercolumn=null, predicate=SlicePredicate( range=[0,0], maxkeys=1)<somerangeslicefield>
DEBUG 18:11:29,191 get slice <somerangeslicefield>
DEBUG 18:11:29,460 Processing response on an async result from 5678@192.168.5.134
```

As can be seen, the first node performed a write operation, thus favoring availability over consistency. An entry async result in the log file indicated that the second node would not know about interim updates until the network connection was restored.

In our next step, we repeated the same test but with a consistency level of QUORUM. Since the first node could not communicate with the second node to inform it about interim updates, the first node stopped processing the write, thus favoring consistency over availability. The cluster became read-only.

G. Test Case 7: Performance

In this test case, we checked if consistency could be achieved without sacrificing performance. For this purpose, we ran Test Case 1, Test Case 2 and Test Case 3 with 100, 1000, 10000 and 100000 data iterations.

We also experimented with different consistency levels to gain extra speed for read or write operations. For example, when we ran the tests with 10000 and 100000 data iterations, we were more concerned about write performance than read performance. Therefore, we wrote data with a consistency level of ONE (W=1) and read data with a consistency level of ALL (R=N). As a result, each read had to access all copies of data to determine which of them contained the latest version of data, whereas each write had to update only one copy of data. This time when we ran the tests with 100 and 1000 data iterations, we were more concerned about read performance than write performance. Therefore, we wrote data with a consistency level of ALL (W=N) and read data with a consistency level of ONE (R=1).

Figure 1 shows the result of our tests. As can be seen, consistency was achieved at expense of performance because...
of the need for starting and finishing a synchronization process every time when the database was updated.

VII. CONCLUSION

During many years clustered file systems like Gluster and (SQL-based) relational DBMSs like MySQL and Oracle have been the dominant technologies for providing an efficient and reliable data store in cloud computing environments. However, with the trend towards cloud computing, these systems get new competitors – NoSQL DBMSs. One of them is Cassandra, which was evaluated in this paper.

Cassandra was claimed to be particularly well suited for cloud computing environments. Our goal was to confirm or deny that claim. Towards this goal, we experimented with Cassandra. Our experiments showed that Cassandra did offer an efficient and reliable data store in cloud computing environments, either while favoring availability and partition tolerance over consistency or while favoring consistency and partition tolerance over availability.

The result of our experiments was in agreement with the CAP (Consistency, Availability and Partition tolerance) theorem [11]. This theorem simply states that out of consistency, availability and partition tolerance, a distributed system can choose to provide two but never three at the same time, as shown in Figure 2. For example, relational DBMSs typically provide both consistency and availability, but not partition tolerance. By contrast, NoSQL DBMSs typically provide both availability and partition tolerance, but not consistency.

VIII. FUTURE WORK

In the future, we are going to increase a number of nodes in the cluster. Eventually applying the results of our tests to real-world applications is also part of our future work.

ACKNOWLEDGMENT

Irina Astrova’s work was supported by the Estonian Centre of Excellence in Computer Science (EXCS) funded mainly by the European Regional Development Fund (ERDF).

REFERENCES