DEGREE PROJECT, IN COMPUTER SCIENCE, DD143X , FIRST LEVEL
STOCKHOLM, SWEDEN 2015

ACID and BASE in PostgreSQL

DISTRIBUTED DATABASE PERFORMANCE

PONTUS FALK AND IGNACIO QUEZADA

KTH ROYAL INSTITUTE OF TECHNOLOGY

CSC SCHOOL

KTH Computer Science
and Communication

ACID and BASE in PostgreSQL

Distributed Database Performance

Group 68

PONTUS FALK pontusfa@kth.se
IGNACIO QUEZADA CRUZ piqc@kth.se

Royal Institute of Technology
School of Computer Science and Communication
DD143X - Degree Project in Computer Science, First Cycle
Supervisor: Michael Minock (minock@kth.se)
Examiner: Örjan Ekeberg (ekeberg@csc.kth.se)

May 8, 2015

ABSTRACT

This thesis studies the performance impact of the Two Phase Commit mechanism used in order to achieve consistency in a distributed PostgreSQL system. This impact is compared to the performance gained by removing Two Phase Commit and abandoning consistency. The tests are run on different number of nodes in the system to find a link between the performance hit and the size of the system. Performance is measured using the unit Queries per Second.

Firstly, a short introduction is given to demonstrate why and when distributed database systems is necessary and the difficulties it poses. Thereafter follows a description of the system constructed to conduct the experiment, together with the results of this experiment. The thesis is concluded with a discussion regarding the results and thoughts on future variations of the experiment.

The conclusion of the experiment is that Two Phase Commit puts a low maximum performance on the system, especially when compared to the alternative of abandoning consistency. The number of nodes did not significantly affect the performance of the system using Two Phase Commit, however the alternative had a clear negative correlation between numbers of nodes and Queries per Second.

SAMMANFATTNING

Detta arbete studerar hur *Two Phase Commit*, som används för att tillhandahålla konsistens, påverkar prestandan i ett distribuerat PostgreSQL-system. Denna påverkan jämförs med prestandapåverkan som ges av att utesluta *Two Phase Commit* från systemet och överge konsistens-kravet. Testerna utförs med olika antal noder i systemet för att försöka finna en länk mellan prestandapåverkan och storleken på systemet. Prestanda mäts med enheten *Queries per Second*.

Först ges en kort introduktion för att demonstrera varför och när distribuerade databassystem är nödvändiga och vilka problem de medför. Därefter följer en beskrivning av systemet som konstruerats för experimentet tillsammans med dess resultat. Arbetet avslutas med en diskussion över resultaten och tankar om framtida variationer av experimentet.

Slutsatsen av experimentet är att *Two Phase Commit* sätter en låg maximal prestanda på systemet, speciellt när det jämförs med alternativet då konsistens-kravet överges. Antalet noder ger ingen signifikant skillnad på prestandan på systemet med *Two Phase Commit*, däremot så har alternativet en klar negativ korrelation mellan antalet noder och mängden *Queries per Second*.

CONTENTS

1 INTRODUCTION 2
1.1 the Curse of Success 2
1.2 Problem Statement 2
2 BACKGROUND 3
2.1 Acid and Base 3
2.2 PostgreSQL extensions for distributed systems 4
3 METHOD 6
3.1 Experiment Setup 6
3.1.1 Business Application 6
3.1.2 Soft- and Hardware Specification 7
3.1.3 Constants and Variables 8
3.2 Experiment execution 8
4 RESULT 9
4.1 Raw data 9
4.1.1 Tests with 1 node 9
4.1.2 Tests with 2 nodes and ACID 12
4.1.3 Tests with 2 nodes and BASE 14
4.1.4 Tests with 4 nodes and ACID 17
4.1.5 Tests with 4 nodes and BASE 19
4.2 Data Translation 22
5 DISCUSSION 24
5.1 The throughput 24
5.2 Abandoning ACID 24
5.3 Number of Nodes 24
5.4 Reliability and Validity 25
6 CONCLUSION 26
6.1 Answering the Problem 26
6.2 Where to Go From Here 26
7 REFERENCES 27

ABBREVIATIONS

2PC	Two phase Commit
ACID	Atomicity, Consistency, Isolation, Durability
BASE	Basically Available, Soft State, Eventual Consistency
CAP	Consistency, Availability, Partition Fault Tolerance
GNU	Gnu's Not Unix
QPS	Queries Per Second
RRD	Round Robin Database
SSH	Secure Shell

1 INTRODUCTION

1.1 THE CURSE OF SUCCESS

When a database system needs to grow due to increasing workload there are two general ways to approach the problem: vertical or horizontal scaling. In the vertical case, resources such as memory and CPU are added or upgraded to the already existing system. Horizontally scaling is done by adding database nodes to form or extend a distributed database system.

With today's technology, there is a hard limit on vertical scaling as there is a peak performance on all hardware resources. As a result there has been a shift in focus to scaling databases horizontally. Different approaches have emerged, such as data replication with a load-balancer as well as dividing rows or tables between nodes, also known as sharding. There are, however, a multitude of problems that arise due to these approaches, one such example is data inconsistency.

The topic on how to successfully and correctly scale databases horizontally is interesting. The problem and possible solutions lie in the theoretical domain but the implications have a very practical impact: many business models are dependant on well functioning data access and storage.

1.2 PROBLEM STATEMENT

This thesis aims to study problems that are inherent to horizontally scaling databases with regard to one of the important ACID properties, the consistency. How high throughput can a fully ACID distributed database system achieve, with regards to queries per second? How much do the number of nodes in the system affect the throughput? Is there a significant improvement when allowing consistency to be violated?

2

BACKGROUND

2.1 ACID AND BASE

ACID has stayed relevant for modern relational database transactions since it was formally introduced in 1983 [1]. *Atomicity, Consistency, Isolation, Durability* are properties found in all major modern relational database management systems [2, 3, 4].

However, distributed databases are affected by the well established CAP theorem [5] that states that distributed data systems can only ever achieve two out of the three properties *Consistency, Availability, Partition fault tolerance*. In practice, a distributed database system wants to stay operational even if one of its nodes go down, which leaves the system to choose between either Consistency or Availability.

For a database system where Consistency is important, a *two-phase commit(2PC)* can be used [6]. A transaction using 2PC begins with the node that wants to perform a transaction requesting a vote by the system's other nodes if they can commit a transaction. The node initiating a vote is called the master node, and the voting nodes are called slave nodes. This master/slave relation is on a per-transaction scope and any node can begin a vote.

If all slave nodes votes yes, the master node issues the command to do the commit to all slave nodes and awaits the confirmation that all nodes have committed the transaction. If one or more slave nodes vote no, the master node issues the command to rollback the transaction.

This ensures consistency but is vulnerable to availability problems. For a transaction to either commit or rollback, all the nodes must be active and replying [7]. Using a timeout for a vote is critical to avoid a permanent system lock.

Availability can for some purposes be a more valuable property than Consistency, which means giving up ACID in favor of BASE, *Basically Available, Soft State, Eventual Consistency* [7]. Basically available is the A in the CAP theorem, promising availability. Soft State is the property where a node's state may change at times without user input, due to alter statements coming from other nodes or a master server. Eventual Consistency promises consistency in the distributed database system given enough time, but offers no upper time limit.

2.2 POSTGRESQL EXTENSIONS FOR DISTRIBUTED SYSTEMS

There are many implementations for creating distributed PostgreSQL databases, where some focus on availability while others focus on consistency. PostgreSQL version 9.0 [8] offers the use of a replication mechanism for load balanced database systems to increase the system's availability. A replication system providing consistency based on embedded 2PC transaction method was implemented in PostgreSQL version 9.1 [9].

Apart from the official implementations, there are 3rd party plug-ins used to deploy distributed databases with different properties. Some of these properties can be master-master replication consisting of the possibility of starting a 2PC transaction from any node, as opposed to PostgreSQL's built-in master-slave which centralizes the initiation point of transactions to one node.

Some of the plug-ins providing ACID replication are:

- Postgres-XC aims at read-and-write scalability with an architecture divided in three components: A Global Transaction Manager providing consistent transaction management for the rest of the components. A Coordinator acting as a server for the applications. Data nodes is the component that stores the data [10].
- PgPool-II is a set of tools that function as a middleware between the business application and the database system. The tools are deployed on a central server and they load-balance the select queries amongst the database nodes and send the update statements to all nodes [11]. Load-balancing is a mechanism used to distribute the connections between the nodes, so the node with least load will be prioritized instead of a node with already high load [12].

There are two more plug-ins that deserve to be mentioned although they are not under active development:

- PGCluster is an extension of PostgreSQL version 8 whose architecture consists of three parts: the load-balancer, the cluster of database nodes, and the replication server. The replication server discards database nodes that have failed on updates and makes them available again after executing the required queries to get them up to date [13].

- Postgres-R is an extension of PostgreSQL version 6 whose replication system is based on Group Communication System. It is used as an alternative to 2PC where every member of a group, consisting of the database nodes, will inform the master node after an update broadcast if a commit failed. These nodes then become flagged as outdated while executing a recovery process [14]. The majority of the nodes are required to stay updated to ensure an automatic recovery process, otherwise the distributed database halts waiting for manual intervention [15].

3 METHOD

The posed problems are of a kind where answers may be found using quantitative analysis from data collected through experiments. Therefore a system is constructed to enable experiments to be run and easily collect the necessary data.

3.1 EXPERIMENT SETUP

The experiment is carried out by constructing nodes consisting of three parts:

- Relational Database
- Business Application
- Soft- and Hardware Specification

PostgreSQL database is chosen as relational database as it is commonly used in both academia and industry. The nodes will share the same database schema and content, the distribution of data is replicated and not partitioned. This ensures Partition fault tolerance.

3.1.1 *Business Application*

The custom-built application is programmed to fulfill a precise purpose. The purpose is to ensure minimal overhead together with a full set of features needed for the experiment and no superfluous features. The application is used to simulate a generic shopping company performing everyday tasks, shoppers buying items, items being shipped et cetera.

This application handles the communication between the nodes and they act as master/slaves where any node can initiate a transaction. There are two operation modes for transactions:

- ACID, where all transactions are using 2PC. This mode ensures Consistency.
- BASE, where all transactions commit as they are received. This mode ensures Availability.

The rationale for creating a custom-built application rather than relying on existent 3rd party plug-ins is that the operations needed for the experiment are very precise and niche. As such, no one plug-in offers the required features. The application implements relevant features from many plug-ins:

- The application relies on a set of tools that act as a middleware application, same as the PgPool-II plug-in works. Instead of a centralized load-balancer, the distributed system balances the incoming load to any node that are select queries.
- The application has a simple master-master replication system. It was originally an idea of the implementation of Postgres-R's Group Communication System.
- For the BASE operation mode, availability is ensured by not locking table rows on all nodes' databases before committing. Similar is done with the PGCluster plug-in.
- For the ACID operation mode, consistency is ensured with PostgreSQL's built-in 2PC transaction implementation.

3.1.2 *Soft- and Hardware Specification*

Hardware:	4 Raspberry Pi 2 model B
Network Connection:	100 MB/s
Operating System:	Arch Linux Snapshot 2015-04-08
Python:	3.2.2
Timing, Graph Plotting:	rrd tool 1.5.0

Table 1: Specification of the experiment's soft- and hardware setup.

Each node is hosted on a Raspberry Pi connected to the same local network. All nodes use the same image containing all software used in the experiment. This setup ensures that each node is equal with regards to hard- and software performance, configuration and near-zero network latency between the nodes. t

3.1.3 *Constants and Variables*

All tests share these constants:

- Number of select queries: 4800. The test ends when all select queries are executed.
- Threads running select queries: 8. The select queries are evenly divided between all threads.
- Threads running alters statements: 8. Each alter thread executes randomly constructed alter statements for the duration of the test.

The variables used to test the limits of the system:

- Number of nodes. Varies between 1 and 4.
- Time between alter queries. Varies between 100 ms and 2 s.

3.2 EXPERIMENT EXECUTION

A network switch of 5 ports is used to connect the four Raspberry Pis and a computer to control them via SSH. A minimal configuration is needed on the nodes to allow incoming connection to PostgreSQL via network. The deployment of the application is straightforward using python virtual environments and the pip utility to install all the dependencies needed in this sandbox.

Having all connected and in idle state, the IP address of each node is saved to a configuration file used by the main application. The main application is then invoked with the parameters for the test run. An example invokation of the main application is:

```
$ ./main.py -st 8 -at 8 -n 4 -s 2.0 –tpc
```

Where *st* stands for select threads, *at* stands for alter threads, *n* for the number of nodes, *s* for the sleep time and *tpc* stands for two phase commit. Omitting *tpc* flag instructs the application to alter the database using BASE instead of ACID.

To run this test command 25 times with different values, a bash script is used to make a batch run of all the tests at once rather than one by one.

At the end of each test run, a short summary with the data is printed to the terminal. The result graphs are automatically rendered at the end of each test run into a graph directory, where all 25 graphs can be fetched once the batch run had completed.

4 RESULT

4.1 RAW DATA

The results are divided and presented in sections based on the number of nodes, since this was the parameter that had the most significant impact on the results of the tests.

There are a total of 25 tests in sets of five, with an output for every test run in graph format and data.

4.1.1 *Tests with 1 node*

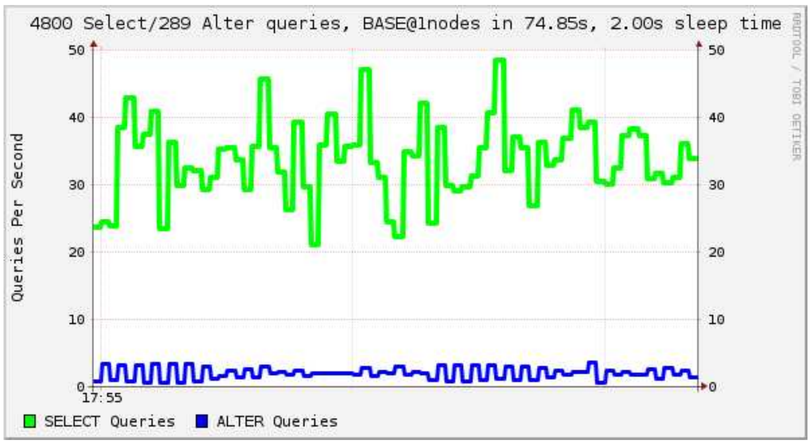


Figure 1: One node, 2.00 seconds sleep time

Figure 1 shows a steady interval of 2 seconds between alter queries.

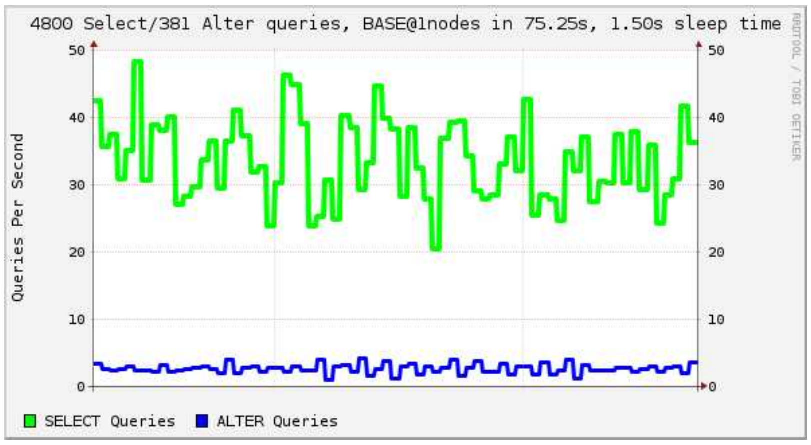


Figure 2: One node, 1.50 seconds sleep time

Figure 2 shows a more stable alter thread without significantly impacting of the amount of select queries per second.

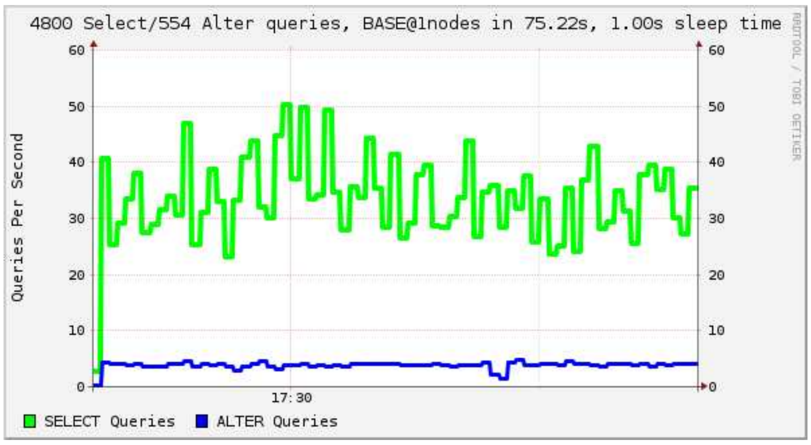


Figure 3: One node, 1.00 seconds sleep time

Figure 3 shows an incident where the alter thread aligned and locked the select and alter threads.

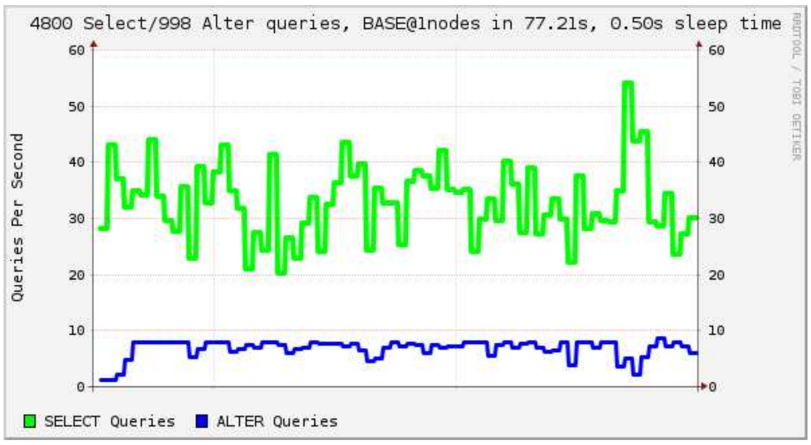


Figure 4: One node, 0.5 seconds sleep time

Figure 4 shows a higher alter statement rate without a significant impact on the select thread.

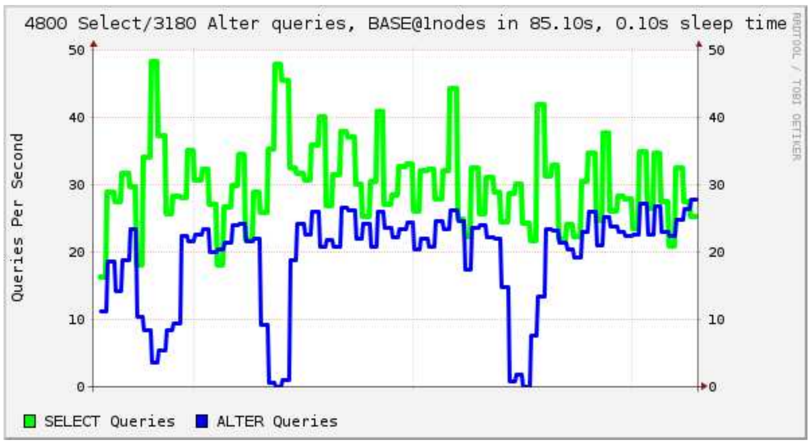


Figure 5: One node, 0.1 seconds sleep time

Figure 5 shows an alignment of alter queries which causes a lock hurting the alter thread but allowing the select threads to execute more queries. Also shown is the average of select queries is lower but the alter statements are higher.

4.1.2 *Tests with 2 nodes and ACID*

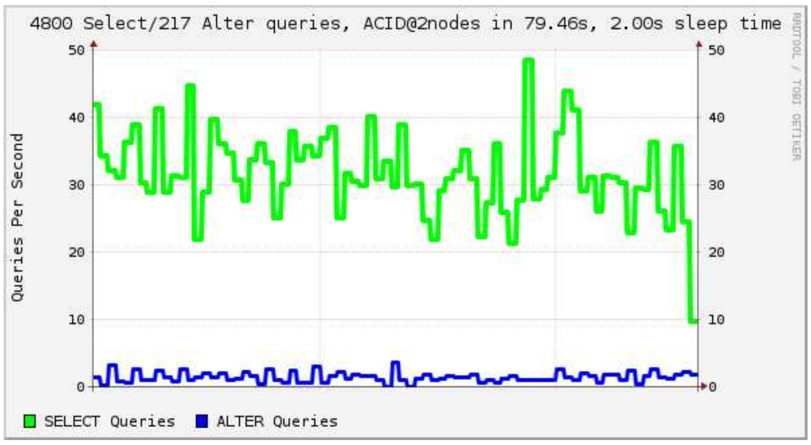


Figure 6: Two nodes, 2.00 seconds sleep time

Figure 6 shows how the latency of using multiple nodes affects the system, the alter thread becomes slower, impacting the speed of the select thread and the amount of alter statements executed.

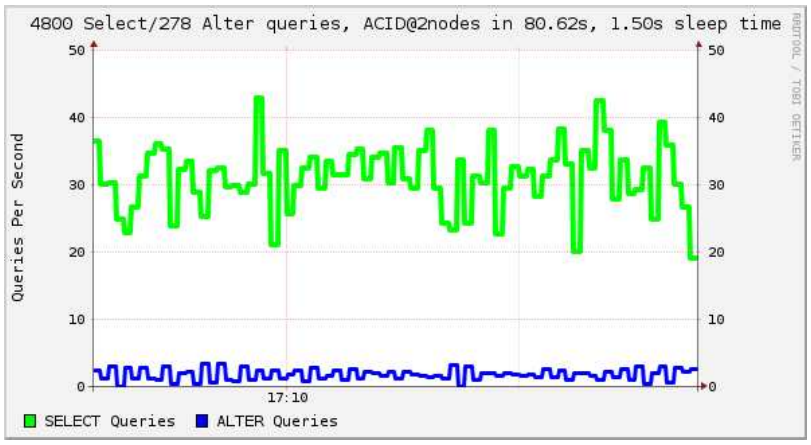


Figure 7: Two nodes, ACID, 1.50 seconds sleep time

Figure 7 shows how the test run becomes slower.

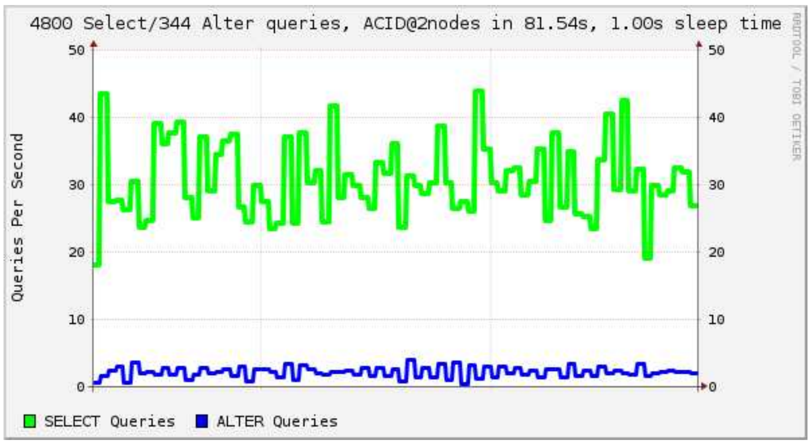


Figure 8: Two nodes, ACID, 1.00 seconds sleep time

Figure 8 shows a more stable alter thread compared to the graph before.

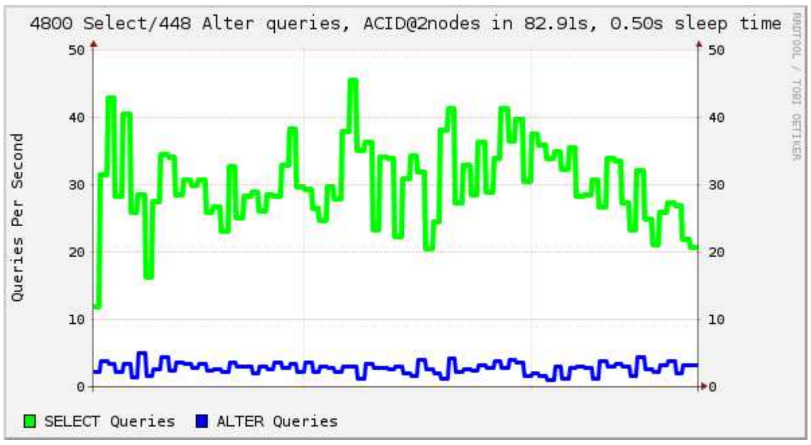


Figure 9: Two nodes, ACID, 0.5 seconds sleep time

Figure 9 shows a similarly steady alter thread.

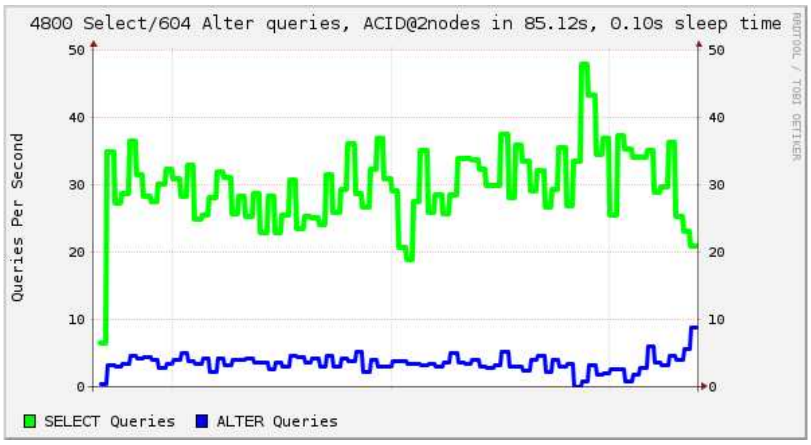


Figure 10: Two node, ACID, 0.1 seconds sleep time

Figure 10 shows a slight speedup in the alter thread, resulting in the test running slower.

4.1.3 *Tests with 2 nodes and BASE*

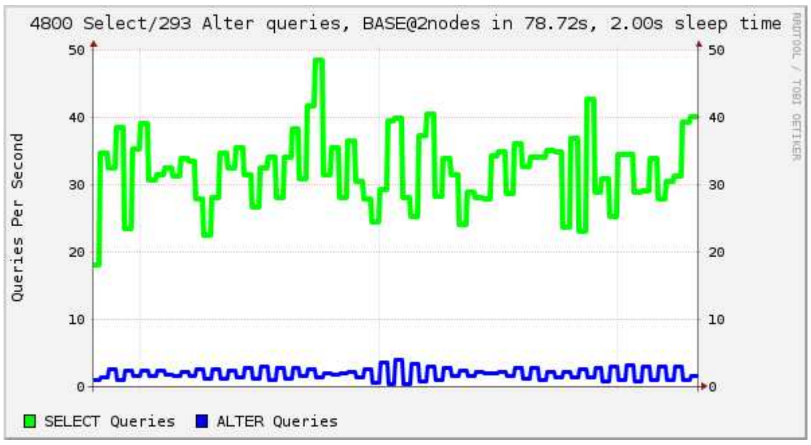


Figure 11: Two nodes, BASE, 2.00 seconds sleep time and BASE

Figure 11 shows a steady 2 seconds alter thread with a fast select thread.

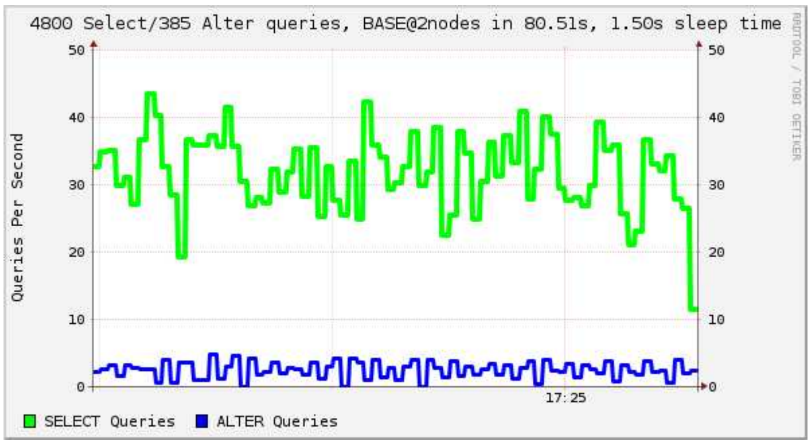


Figure 12: Two nodes, BASE, 1.50 seconds sleep time and BASE

Figure 12 shows a faster run with more alter queries.

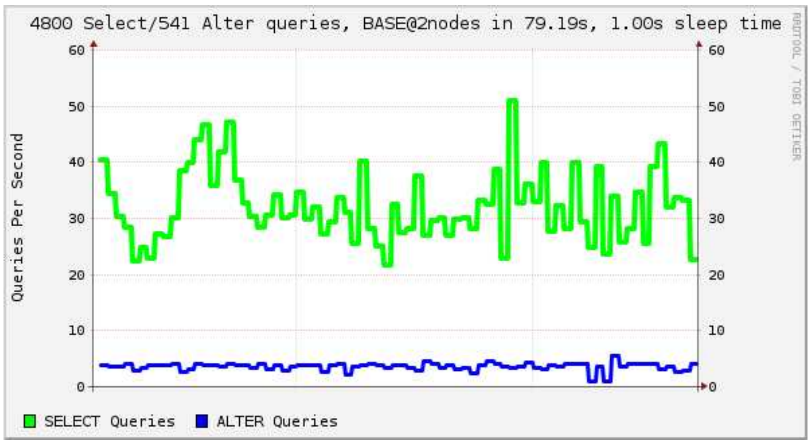
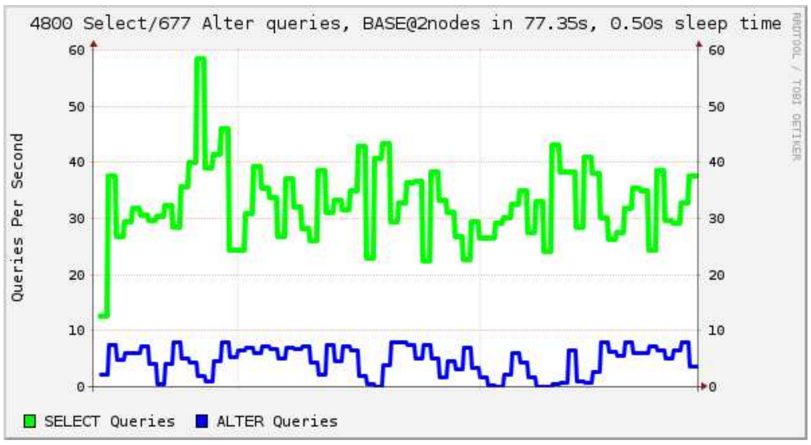


Figure 13: Two nodes, BASE, 1.00 seconds sleep time and BASE

Figure 13 shows more alter statements without impacting the total runtime.



t

Figure 14: Two nodes, BASE, 0.5 seconds sleep time and BASE

Figure 14 shows a test run with more alter statements.

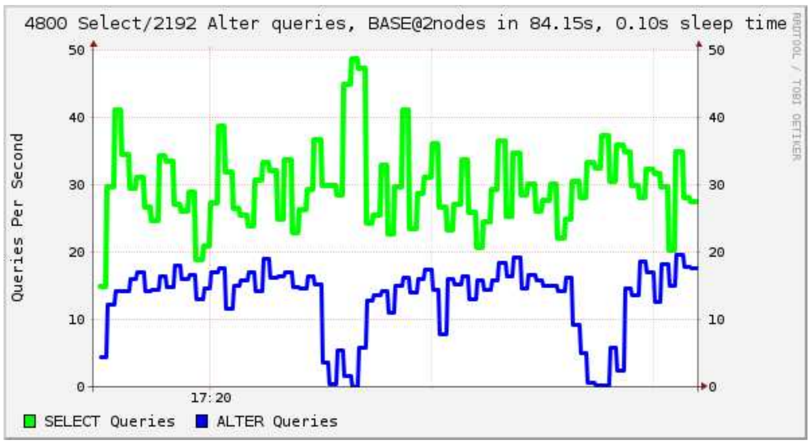


Figure 15: Two nodes, BASE, 0.1 seconds sleep time and BASE

Figure 15 shows how the locking system has a strong impact on the queries.

4.1.4 *Tests with 4 nodes and ACID*

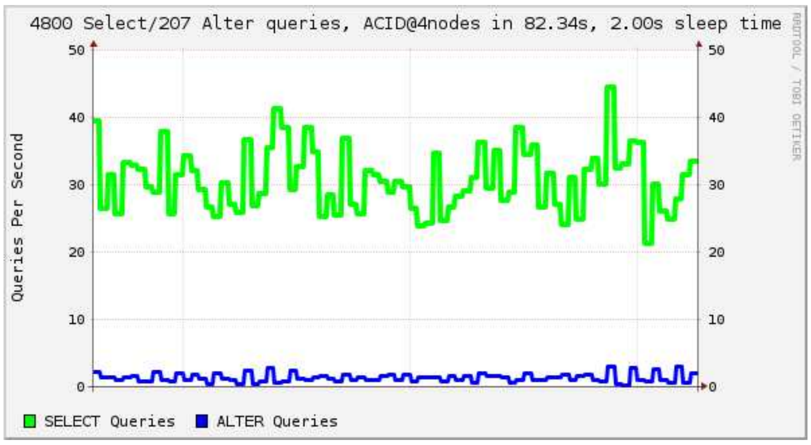


Figure 16: Two nodes, 2.00 seconds sleep time

Figure 16 shows a low alter statement curve.

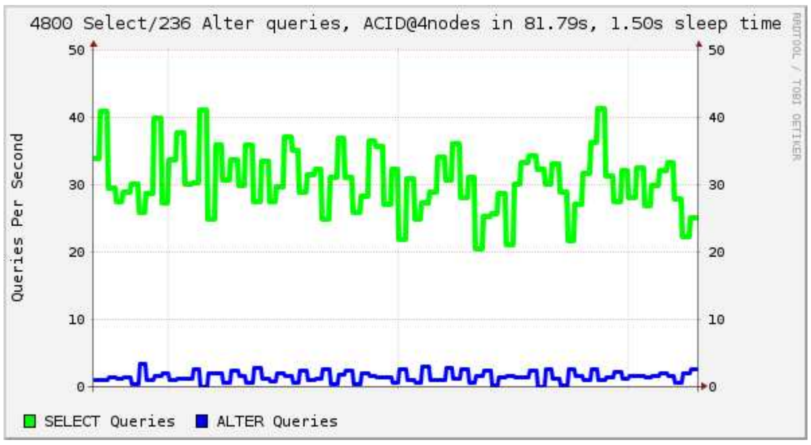


Figure 17: Four nodes, ACID, 1.50 seconds sleep time

Figure 17 shows an insignificant difference compared to figure 16.

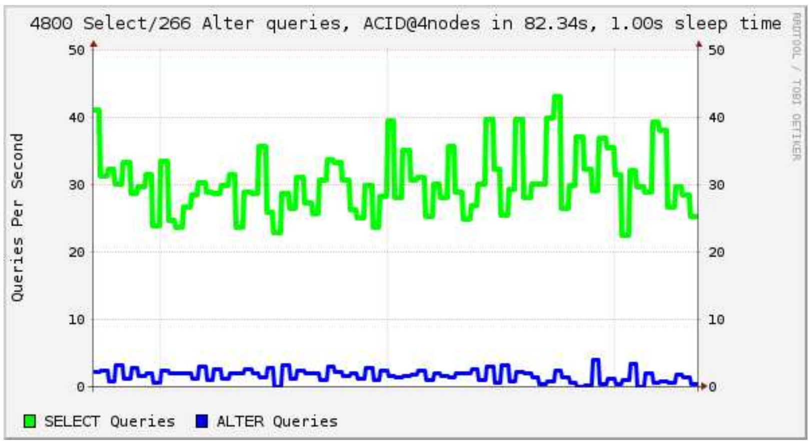


Figure 18: Two nodes, ACID, 1.00 seconds sleep time

Figure 18 shows a relatively stable graph for select queries, but a volatile alter statement.

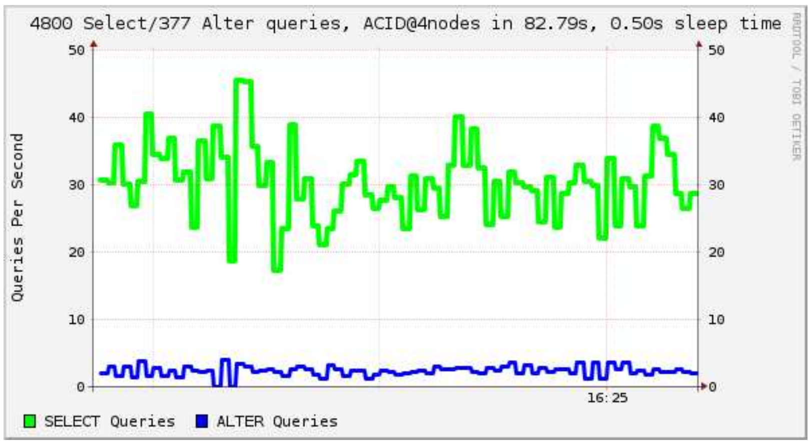


Figure 19: Four nodes, ACID, 0.5 seconds sleep time

Figure 19 shows a stable alter statement flow.

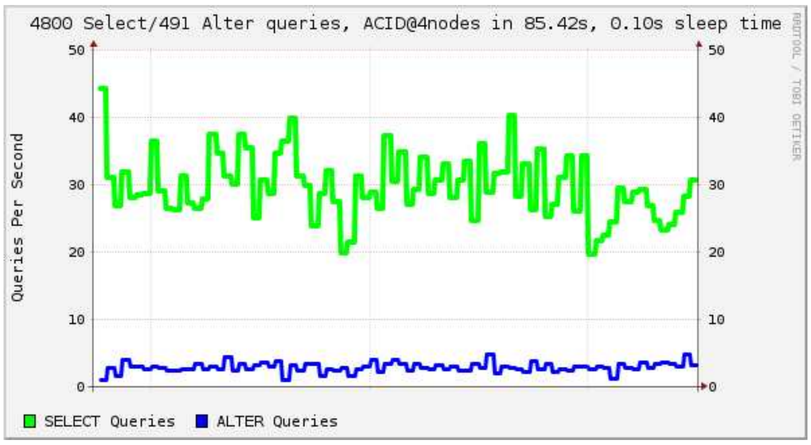


Figure 20: Four nodes, ACID, 0.1 seconds sleep time

Figure 20 shows an unusual select queries graph.

4.1.5 *Tests with 4 nodes and BASE*

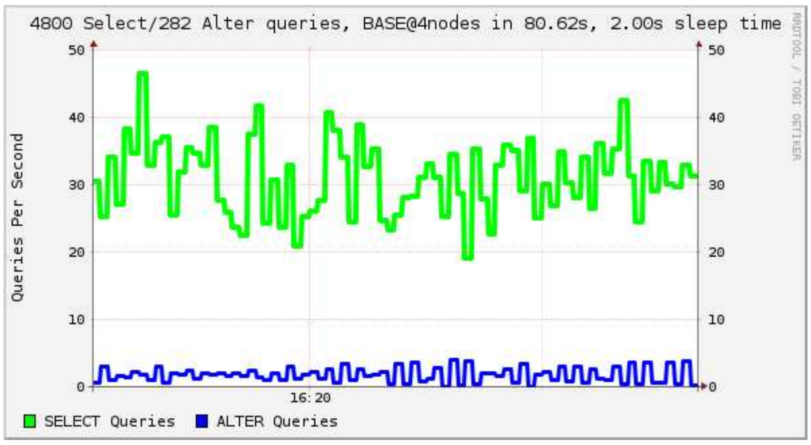


Figure 21: Four nodes, BASE, 2.00 seconds sleep time and BASE

Figure 21 shows alter statements finishing fast, leaving the threads to idle for periods of time.

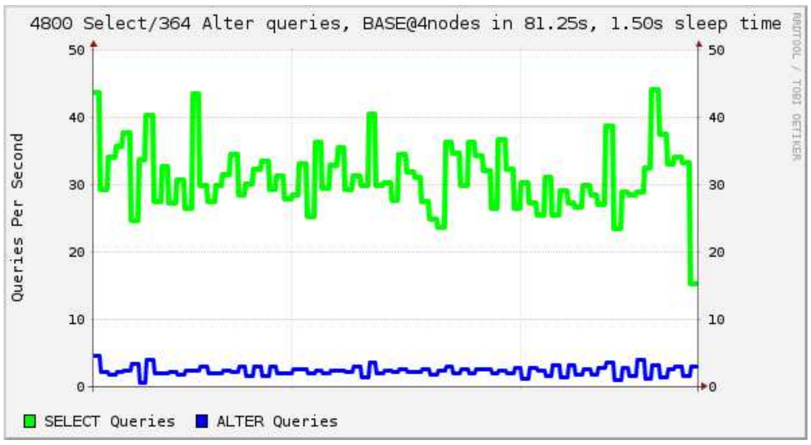


Figure 22: Four nodes, BASE, 1.50 seconds sleep time and BASE

Figure 22 shows a steady alter thread.

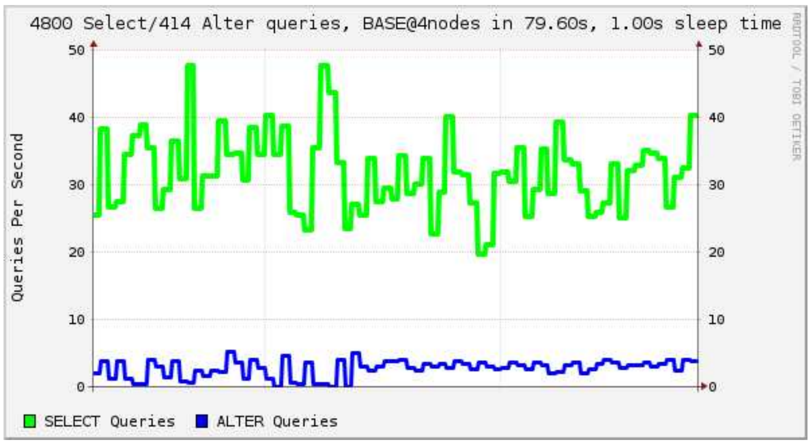
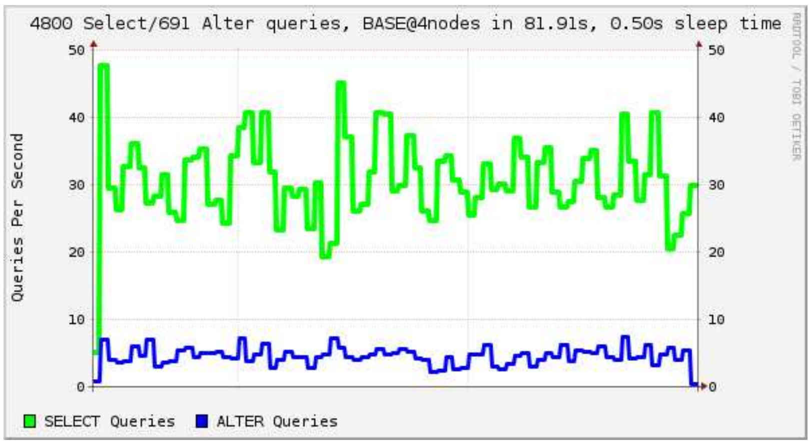


Figure 23: Four nodes, BASE, 1.00 seconds sleep time and BASE

Figure 23 shows a run with more alters statements.



t

Figure 24: Four nodes, BASE, 0.5 seconds sleep time and BASE

Figure 24 shows an increase in alter threads without significantly impacting the total runtime.

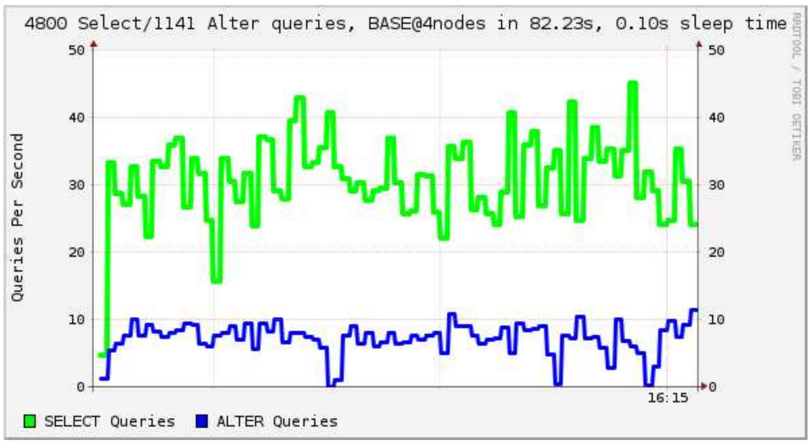


Figure 25: Four nodes, BASE, 0.1 seconds sleep time and BASE

Figure 25 shows multiple declines in alter statements.

Figures 21-25 all follow the trends seen in previous figures.

4.2 DATA TRANSLATION

The data retrieved from the tests is assembled in table 1. The Queries Per Second is calculated from the sum of the amount of alter statements and select queries divided by the total time needed to run the test.

QPS	Time (s)
289	74,85
381	75,25
554	75,22
998	77,21
3180	85,10

Table 2: Table with raw data for tests with one node, QPS is Queries Per Second, and Time the time needed in seconds to run the test.

QPS ACID	Time (s) ACID	QPS BASE	Time (s) BASE
217	79,46	293	78,72
278	80,62	385	80,51
344	81,54	541	79,19
448	82,91	677	77,35
604	85,12	2192	84,15

Table 3: Table with raw data for tests with two nodes, QPS is Queries Per Second, and Time the time needed in seconds to run the test.

QPS ACID	Time (s) ACID	QPS BASE	Time (s) BASE
207	82,34	282	80,62
236	81,79	364	81,25
266	82,34	414	79,60
377	82,79	691	81,91
491	85,42	1141	82,23

Table 4: Table with raw data for tests with four nodes, QPS is Queries Per Second, and Time the time needed in seconds to run the test.

Tables 1-4 collect the data from the 25 test runs separating them into three different tables by the number of nodes used on each test. For a better understanding of the values in the three tables, a chart is drawn with the same values in figure 26. It reflects the results of all tests.

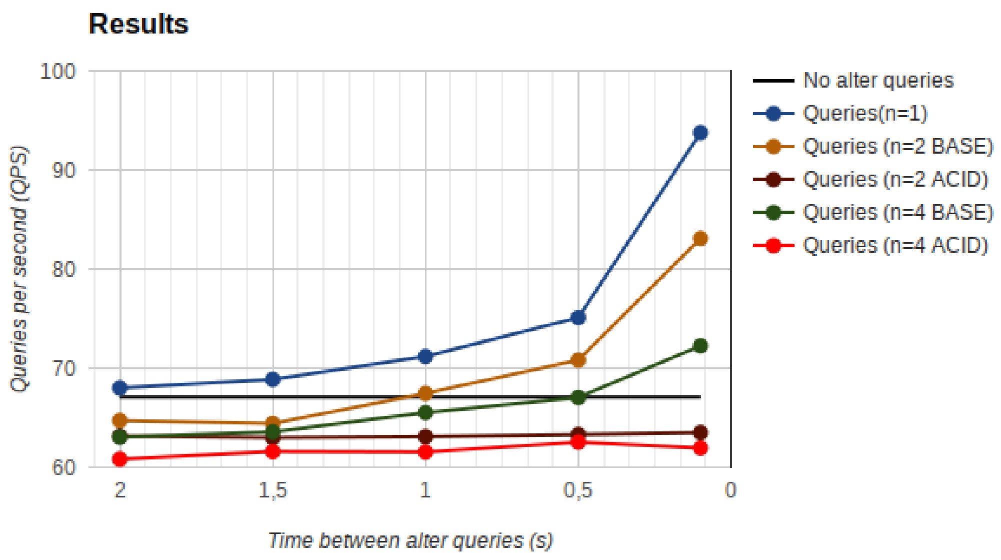


Figure 26: Queries per second through different sleep times, n is the number of nodes.

Figure 26 shows that while the test runs using only one node have the best performance with a higher amount of queries per second, the BASE tests running with two and four nodes show a clear advantage when consistency is ignored. The ACID test run does not show an improvement in queries per second even if the continuity between alter statements is higher. The test run which shows a constant line in the graph is a test run with only select queries which was used for reference purposes only.

5 DISCUSSION

The discussion is suitable to be divided between the three questions posed in the problem statements, and reason about the results with each question in focus.

5.1 THE THROUGHPUT

Throughput for ACID in figure 26 detailing QPS is underwhelming. The two lines for ACID in two and four nodes are the lowest for all different sleep times and unlike the other results they do not improve but have a near-constant QPS.

Lowering sleep time between alter statement executions allows for higher QPS, but the locking mechanism from 2PC is a bottleneck and provides what to appears to be an upper limit for ACID QPS.

5.2 ABANDONING ACID

For situations where consistency is not of outmost importance, figure 26 clearly proofs that BASE with its eventual consistency is superior by a wide margin. Lowering the sleep time between alter statements yields a QPS growth of what appears to be quadratic or better.

5.3 NUMBER OF NODES

Figure 26 seems to show that there is no real performance gain or loss QPS-wise between two and four nodes for the ACID case. A probable explanation to this is the parallel nature of 2PC coupled with the fact that the experiment was setup to minimize and equalize latency between all nodes.

The same figure also shows that there is a significant difference between two and four nodes for BASE QPS. Not only does the four nodes test start with lower QPS, the growth compared to the two nodes test is slower, suggesting the difference will grow even larger if sleep time would be lowered further.

5.4 RELIABILITY AND VALIDITY

The aim of this thesis was not to attempt to optimize the business application implementation to achieve a high number of QPS for each test result. Rather, the aim was to use an implementation using sound and well proven techniques to give justice to both modes, ACID and BASE. The important parts of the results were not the actual numbers but the trends they provided for the various parameters.

Care was taken to ensure no hidden bottleneck would disrupt or infer with any of the results. The implementation was run through a performance monitor to catch any resource hogs. A profiler confirmed a vast majority of the time used to run the tests was used inside the PostgreSQL driver and not in the implementation.

The amount and kind of select queries and alter statements used in the experiments were written to realistically simulate a real life scenario for a business application. A large amount of statements were executed to minimize the impact a few slow ones would have on the results in any significant way.

These precautions lead to the conclusion that the results are reliable and valid, any and all conclusions drawn from them are therefore sound.

6 CONCLUSION

To conclude this thesis, an answer to the problem statement is given as well as thoughts on where to go from here.

6.1 answering the problem

The questions posed in the problem statement were three-fold:

- Performance throughput in an ACID system
- Change in performance replacing ACID with BASE
- How the number of nodes affects the performance

The results showed a constant limit for the ACID system reaching full potential far below that of the BASE system. The number of nodes affected the BASE system more significantly than it did for the ACID system.

6.2 where to go from here

This thesis has laid the foundation for a host of variations on the parameters in the test. A few interesting examples could be:

- Add artificial latency between the nodes
- Vary the kind and size of data in the database
- Add more nodes to the system
- Measure more sleep times to better approximate a function for the curves

This field grows more and more important and more research is needed to prepare for the future data storage needs.

7 REFERENCES

[1] Andreas Reuter Theo Haerder. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, December 1983.

[2] Oracle. *MySQL 5.6 Reference Manual: MySQL and the ACID Model*, 2015. https://dev.mysql.com/doc/refman/5.6/en/mysql-acid.html.

[3] Microsoft. *Microsoft SQL Server Transaction (Database Engine)*, 2009. https://technet.microsoft.com/en-us/library/ms190612%28v=sql.105%29.aspx.

[4] Tom Lane. *Transactions Processing in PostgreSQL*. Microsoft, 2000. http://www.postgresql.org/files/developer/transactions.pdf.

[5] ACM. *Towards robust distributed systems*, 2000. Symposium on Principles of Distributed Computing.

[6] Patrick Valduriez M. Tamer Özsu. *Principles of distributed database systems*. Springer, 233 Spring Street, New York, 2011.

[7] Dan Pritchett. Base: An acid alternative. *Queue - Object-Relational Mapping*, 2008.

[8] PostgreSQL Developers. *PostgreSQL Wiki: Replication, Clustering and Connection Pooling*. https://wiki.postgresql.org/wiki/Replication,_Clustering,_and_Connection_Pooling.

[9] PostgreSQL Developers. *PostgreSQL 9.1.15 Documentation*, 2011. http://www.postgresql.org/docs/9.1/static/warm-standby.html#SYNCHRONOUS-REPLICATION.

[10] The Postgres-XC Development Group. *Postgres-XC 1.1 Documentation*, 2013. http://postgres-xc.sourceforge.net/docs/1_1/xc-overview-components.html.

[11] PgPool Developers. *pgpool Wiki*, 2015. http://www.pgpool.net/mediawiki/index.php/Main_Page.

[12] PostgreSQL Developers. *PostgreSQL 9.1.15 Documentation*, 2011. http://www.postgresql.org/docs/9.1/static/high-availability.html.

[13] PGCluster Developers. *PGCluster Feature,* 2005. http://pgcluster.projects.pgfoundry.org/feature.html.

[14] *Postgres-R(SI): Combining Replica Control with Concurrency Control based on Snapshot Isolation*, April 2005. Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on; 2005.

[15] Markus Wanner. *Postgres-R (8) Architecture*, January 2009. http://www.postgres-r.org/downloads/concept.pdf.